User Guide

Exported from JBoss Community Documentation Editor at 2013-01-28 05:48:14 EST

Table of Contents

1 New in Infinispan 4.1.0

1. Grid File System using on Infinispan
2. Server modules
    1. Hot Rod
        1. Protocol specification (version 1)
        2. Using the Hot Rod server module
        3. Java Hot Rod client
        4. Consistent concurrent updates with Hot Rod versioned operations
        5. Multiple layers of caches
        6. Interacting With Hot Rod Server From Within Same JVM
    2. Memcached
        1. Using Infinispan Memcached server
            1. Talking to Memcached from non-Java clients
    3. WebSocket
        1. Infinispan Websocket server
    4. Load testing server modules
    5. Command line options
3. Key affinity service
4. Configuration Reference
5. JMX attributes and operations
6. API docs (javadocs)

2 New in Infinispan 4.2.0

1. Distribution
    1. Server Hinting
2. Starting a new project with Infinispan - check out Infinispan Maven Archetypes
3. Configuration Reference
4. JMX attributes and operations
5. API docs (javadocs)

3 New in Infinispan 5.0.0

1. Generating keys mapped to specific cluster nodes
2. Portable Serialization for Hot Rod with Apache Avro
3. Plugging Infinispan with user defined Externalizers
4. Distributed Executors Framework
5. Fluent Programmatic Configuration
6. Configuration Reference
7. JMX attributes and operations
8. API docs (javadocs)

4 New in Infinispan 5.2.0

1. Cross-site replication

5 Integration with other frameworks

6 Querying Infinispan

6.1 The infinispan-query module

This module adds querying capabilities to Infinispan. It uses Hibernate Search and Apache Lucene to index and search objects in the cache. It allows users to obtain objects within the cache without needing to know the keys to each object that they want to obtain, so you can search your objects basing on some of it's properties, for example to retrieve all red cars (exact metadata match), or all books about a specific topic (full text search and relevance scoring).

6.1.1 Usage with Infinispan 5

Indexing must be enabled in the configuration (as explained in XML Configuration or Programmatic configuration); then you interact with the Search capabilities via a *SearchManager* which exposes methods to perform the queries.

6.2 Simple example

We're going to store *Book* instances in Infinispan; each *Book* will be defined as in the following example; we have to choose which properties are indexed, and for each property we can optionally choose advanced indexing options using the annotations defined in the Hibernate Search project.

```
// example values stored in the cache and indexed:
import org.hibernate.search.annotations.*;

//Values you want to index need to be annotated with @Indexed, then you pick which fields and
how they are to be indexed:
@Indexed
public class Book {
   @Field String title;
   @Field String description;
   @Field @DateBridge(resolution=Resolution.YEAR) Date publicationYear;
   @IndexedEmbedded Set<Author> authors = new HashSet<Author>();
}

public class Author {
   @Field String name;
   @Field String surname;
   // hashCode() and equals() omitted
}
```

Now assuming we stored several *Book* instances in our Infinispan *Cache*, we can search them for any matching field as in the following example.

```
// get the search manager from the cache:
SearchManager searchManager = org.infinispan.query.Search.getSearchManager(cache);

// create any standard Lucene query, via Lucene's QueryParser or any other means:
org.apache.lucene.search.Query fullTextQuery = //any Apache Lucene Query

// convert the Lucene query to a CacheQuery:
CacheQuery cacheQuery = searchManager.getQuery( fullTextQuery );

// get the results:
List<Object> found = cacheQuery.list();
```

A Lucene Query is often created by parsing a query in text format such as "title:infinispan AND authors.name:sanne", or by using the query builder provided by Hibernate Search.

```
// get the search manager from the cache:
SearchManager searchManager = org.infinispan.query.Search.getSearchManager( cache );

// you could make the queries via Lucene APIs, or use some helpers:
QueryBuilder queryBuilder = searchManager.buildQueryBuilderForClass(Book.class).get();

// the queryBuilder has a nice fluent API which guides you through all options.
// this has some knowledge about your object, for example which Analyzers
// need to be applied, but the output is a failry standard Lucene Query.
org.apache.lucene.search.Query luceneQuery = queryBuilder.phrase()
                  .onField("description")
                  .andField("title")
                  .sentence("a book on highly scalable query engines")
                  .createQuery();

// the query API itself accepts any Lucene Query, and on top of that
// you can restrict the result to selected class types:
CacheQuery query = searchManager.getQuery(luceneQuery, Book.class);

// and there are your results!
List<Book> objectList = query.list();

for (Book book : objectList) {
      System.out.println(book.getTitle());
}
```

A part from *list()* you have the option for streaming results, or use pagination.
This barely scratches the surface of all what is possible to do: see the Hibernate Search reference documentation to learn about sorting, numeric fields, declarative filters, caching filters, complex object graph indexing, custom types and the powerful faceting search API.

6.2.1 Notable differences with Hibernate Search

Using *@DocumentId* to mark a field as identifier does not apply to Infinispan values; in Infinispan Query the identifier for all *@Indexed* objects is the key used to store the value. You can still customize how the key is indexed using a combination of *@Transformable*, *@ProvidedId*, custom types and custom *FieldBridge* implementations.

6.2.2 Requirements on the Key: @Transformable and @ProvidedId

The key for each value needs to be indexed as well, and the key instance must be transformed in a String. Infinispan includes some default transformation routines to encode common primitivies, but to use a custom key you must provide an implementation of *org.infinispan.query.Transformer*.

Registering a Transformer via annotations

You can annotate your key type with *org.infinispan.query.Transformable*:

```
@Transformable(transformer = CustomTransformer.class)
public class CustomKey {
   ...
}

public class CustomTransformer implements Transformer {
   @Override
   public Object fromString(String s) {
      ...
      return new CustomKey(...);
   }

   @Override
   public String toString(Object customType) {
      CustomKey ck = (CustomKey) customType;
      return ...
   }
}
```

Registering a Transformer programmatically

Using this technique, you don't have to annotated your custom key type:

```
org.infinispan.query.SearchManager.registerKeyTransformer(Class<?>, Class<? extends
Transformer>)
```

@ProvidedId

The *org.hibernate.search.annotations.ProvidedId* annotation lets you apply advanced indexing options to the key field: the field name to be used, and/or specify a custom *FieldBridge*.

6.3 Configuration

6.3.1 Configuration via XML

To enable indexing via XML, you need to add the <indexing ... /> element to your cache configuration, and optionally pass additional properties to the embedded Hibernate Search engine:

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:5.2
http://www.infinispan.org/schemas/infinispan-config-5.2.xsd"
      xmlns="urn:infinispan:config:5.2">
   <default>
      <indexing enabled="true" indexLocalOnly="true">
         <properties>
            <property name="hibernate.search.default.directory_provider" value="ram" />
         </properties>
      </indexing>
   </default>
</infinispan>
```

In this example the index is stored in memory, so when this nodes is shutdown the index is lost: good for a quick demo, but in real world cases you'll want to use the default (store on filesystem) or store the index in Infinispan as well. For the complete reference of properties to define, refer to the Hibernate Search documentation.

6.3.2 Lucene Directory

Infinispan Query isn't aware of where you store the indexes, it just passes the configuration of which *Lucene Directory* implementation you want to use to the Hibernate Search engine. There are several *Lucene Directory* implementations bundled, and you can plug your own or add third party implementations: the Directory is the IO API for Lucene to store the indexes.

The most common *Lucene Directory* implementations used with *Infinispan Query* are:

- Ram - stores the index in a local map to the node. This index can't be shared.
- Filesystem - stores the index in a locally mounted filesystem. This could be a network shared FS, but sharing this way is generally not recommended.
- Infinispan - stores the index in a different dedicated Infinispan cache. This cache can be configured as replicated or distributed, to share the index among nodes. See also Infinispan as a Directory for Lucene.

Of course having a shared index vs. an independent index on each node directly affects behaviour of the Query module; some combinations might not make much sense.

6.3.3 Using programmatic configuration and index mapping

In the following example we start Infinispan programmatically, avoiding XML configuration files, and also map an object *Author* which is to be stored in the grid and made searchable on two properties but without annotating the class.

```
SearchMapping mapping = new SearchMapping();
mapping.entity(Author.class).indexed().providedId()
      .property("name", ElementType.METHOD).field()
      .property("surname", ElementType.METHOD).field();

Properties properties = new Properties();
properties.put(org.hibernate.search.Environment.MODEL_MAPPING, mapping);
properties.put("hibernate.search.[other options]", "[...]");

Configuration infinispanConfiguration = new ConfigurationBuilder()
      .indexing()
         .enable()
         .indexLocalOnly(true)
         .withProperties(properties)
      .build();

DefaultCacheManager cacheManager = new DefaultCacheManager(infinispanConfiguration);

Cache<Long, Author> cache = cacheManager.getCache();
SearchManager sm = Search.getSearchManager(cache);

Author author = new Author(1, "Manik", "Surtani");
cache.put(author.getId(), author);

QueryBuilder qb = sm.buildQueryBuilderForClass(Author.class).get();
Query q = qb.keyword().onField("name").matching("Manik").createQuery();
CacheQuery cq = sm.getQuery(q, Author.class);
Assert.assertEquals(cq.getResultSize(), 1);
```

6.4 Cache modes and managing indexes

Index management is currently controlled by the *Configuration.setIndexLocalOnly()* setter, or the <indexing indexLocalOnly="true" /> XML element. If you set this to true, only modifications made locally on each node are considered in indexing. Otherwise, remote changes are considered too.

Regarding actually configuring a Lucene directory, refer to the Hibernate Search documentation on how to pass in the appropriate Lucene configuration via the Properties object passed to QueryHelper.

6.4.1 LOCAL

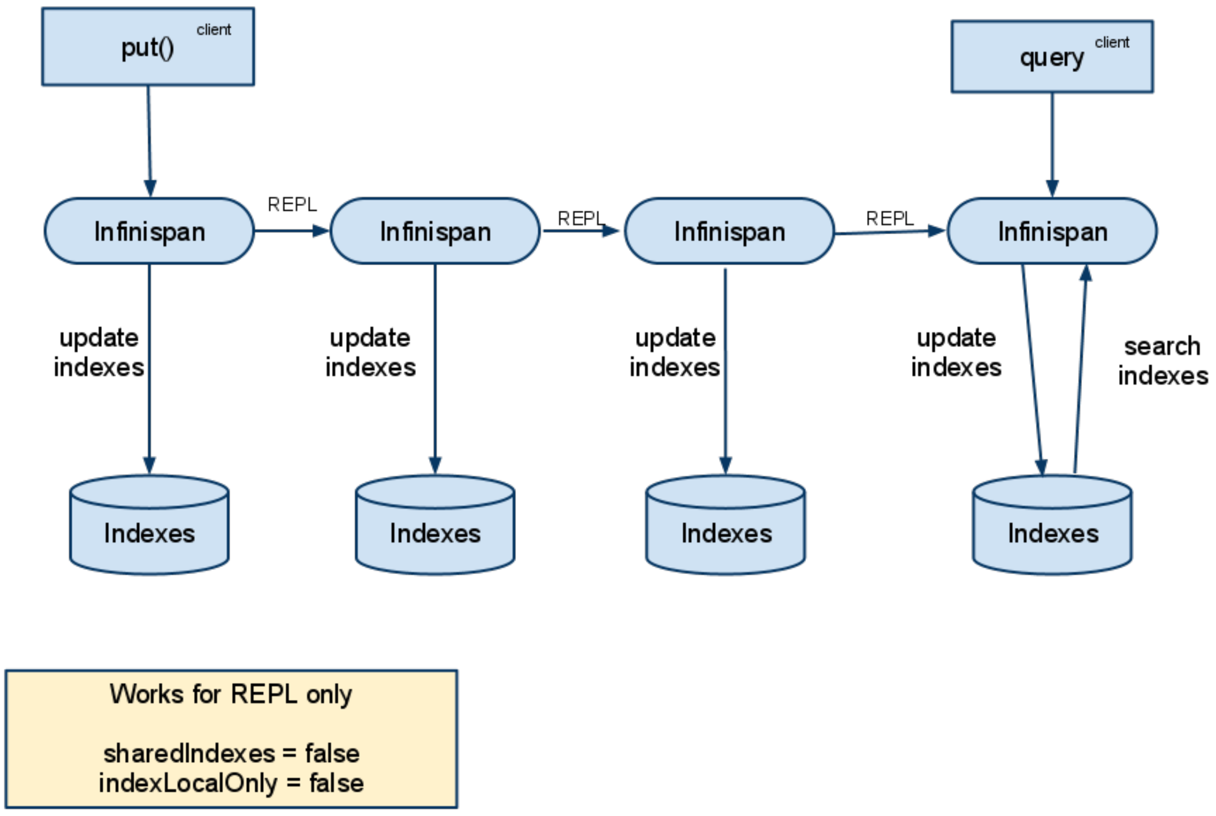
In local mode, you may use any Lucene Directory implementation. Also the option *indexLocalOnly* isn't meaningful.

6.4.2 REPLICATION

In replication mode, each node can have it's own local copy of the index. So indexes can either be stored locally on each node (RAMDirectory, FSDirectory, etc) but you need to set *indexLocalOnly* to *false* , so that each node will apply needed updates it receives from other nodes in addition to the updates started locally. Any Directory implementation can be used, but you have to make sure that when a new node is started it receives an up to date copy of the index; typically rsync is well suited for this task, but being an external operation you might end up with a slightly out-of-sync index, especially if updates are very frequent.

Alternately, if you use some form of shared storage for indexes (see *Sharing the Index*), you then have to set *indexLocalOnly* to *true* so that each node will apply only the changes originated locally; in this case there's no risk in having an out-of-sync index, but to avoid write contention on the index you should make sure that a single node is "in charge" of updating the index. Again, the Hibernate Search reference documentation describes means to use a JMS queue or JGroups to send indexing tasks to a master node.

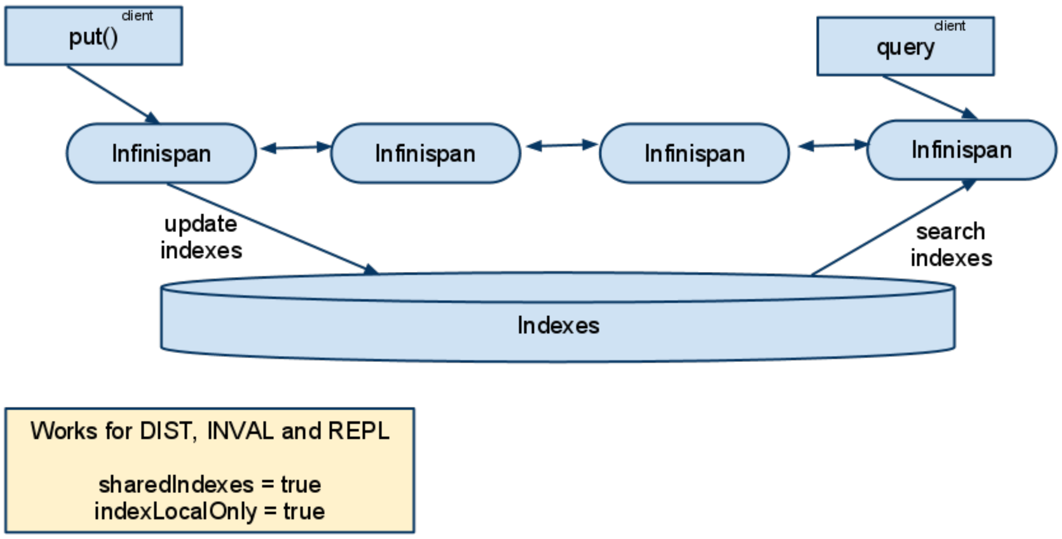
The diagram below shows a replicated deployment, in which each node has a local index.



6.4.3 DISTRIBUTION and INVALIDATION

For these 2 cache modes, you *need* to use a shared index and set *indexLocalOnly* to true.

The diagram below shows a deployment with a shared index. Note that while not mandatory, a shared index can be used for replicated (vs. distributed) caches as well.



6.4.4 Sharing the Index

The most simple way to share an index is to use some form of shared storage for the indexes, like an *FSDirectory* on a shared disk; however this form is problematic as the *FSDirectory* relies on specific locking semantics which are often incompletely implemented on network filesystems, or not reliable enough; if you go for this approach make sure to search for potential problems on the Lucene mailing lists for other experiences and workarounds. Good luck, test well.

There are many alternative Directory implementations you can find, one of the most suited approaches when working with Infinispan is of course to store the index in an Infinispan cache: have a look at the InfinispanDirectoryProvider, as all Infinispan based layers it can be combined with persistent CacheLoaders to keep the index on a shared filesystem without the locking issues, or alternatively in a database, cloud storage, or any other CacheLoader implementation; you could backup the index in the same store used to backup your values.

For full documentation on clustering the Lucene engine, refer to the Hibernate Search documentation to properly configure it clustered.

6.4.5 Clustering the Index in Infinispan

Again the configuration details are in the Hibernate Search reference, in particular in the infinispan-directories section. This backend will by default start a secondary Infinispan CacheManager, and optionally take another Infinispan configuration file: don't reuse the same configuration or you will start grids recursively!
It is currently not possible to share the same CacheManager.

7 Configuring cache programmatically

7.1 Programmatic Configuration

Programmatic Infinispan configuration is centered around CacheManager and ConfigurationBuilder API. Although every single aspect of Infinispan configuration could be set programmatically, the most usual approach is to create a starting point in a form of XML configuration file and then in runtime, if needed, programmatically tune a specific configuration to suit the use case best.

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

Let assume that a new synchronously replicated cache is to be configured programmatically. First, a fresh instance of Configuration object is created using ConfigurationBuilder helper object, and the cache mode is set to synchronous replication. Finally, the configuration is defined/registered with a manager.

```
Configuration c = new
ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC).build();

String newCacheName = "repl";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

The default cache configuration (or any other cache configuration) can be used as a starting point for creation of a new cache. For example, lets say that `infinispan-config-file.xml` specifies a replicated cache as a default and that a distributed cache is desired with a specific L1 lifespan while at the same time retaining all other aspects of a default cache. Therefore, the starting point would be to read an instance of a default Configuration object and use ConfigurationBuilder to construct and modify cache mode and L1 lifespan on a new Configuration object. As a final step the configuration is defined/registered with a manager.

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration dcc = cacheManager.getDefaultCacheConfiguration();
Configuration c = new
ConfigurationBuilder().read(dcc).clustering().cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L)
newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

As long as the based configuration is the default named cache, the previous code works perfectly fine. However, other times the base configuration might be another named cache. So, how can new configurations be defined based on other defined caches? Take the previous example and imagine that instead of taking the default cache as base, a named cache called "replicatedCache" is used as base. The code would look something like this:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration rc = cacheManager.getCacheConfiguration("replicatedCache");
Configuration c = new
ConfigurationBuilder().read(rc).clustering().cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).
newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

Refer to CacheManager, ConfigurationBuilder, Configuration, and GlobalConfiguration javadocs for more details.

7.2 ConfigurationBuilder Programmatic Configuration API

However, users do not have to first read an XML based configuration and then modify it in runtime; they can start from scratch using only programmatic API. This is where powerful ConfigurationBuilder API comes to shine. The aim of this API is to make it easier to chain coding of configuration options in order to speed up the coding itself and make the configuration more readable. This new configuration can be used for both the global and the cache level configuration. GlobalConfiguration objects are constructed using GlobalConfigurationBuilder while Configuration objects are built using ConfigurationBuilder. Let's look at some examples on configuring both global and cache level options with this new API:

One of the most commonly configured global option is the transport layer, where you indicate how an Infinispan node will discover the others:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .clusterName("qa-cluster")
        .addProperty("configurationFile", "jgroups-tcp.xml")
        .machineId("qa-machine").rackId("qa-rack")
      .build();
```

Sometimes you might also want to get global JMX statistics and information about the transport, or the cache manager in general. To enable global JMX statistics simply do:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
  .build();
```

Further options at the global JMX statistics level allows you for example to configure the cache manager name which comes handy when you have multiple cache managers running on the same system, or how to locate the JMX MBean Server:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
    .cacheManagerName("SalesCacheManager")
    .mBeanServerLookupClass(JBossMBeanServerLookup.class)
  .build();
```

Some of the Infinispan features are powered by a group of the thread pool executors which can also be tweaked at this global level. For example:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .replicationQueueScheduledExecutor()
    .factory(DefaultScheduledExecutorFactory.class)
    .addProperty("threadNamePrefix", "RQThread")
  .build();
```

You can not only configure global, cache manager level, options, but you can also configure cache level options such as the cluster mode:

```
Configuration config = new ConfigurationBuilder()
  .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .sync()
    .l1().lifespan(25000L)
    .hash().numOwners(3)
  .build();
```

Or you can configure eviction/expiration settings to:

```
Configuration config = new ConfigurationBuilder()
           .eviction()
             .maxEntries(20000).strategy(EvictionStrategy.LIRS).expiration()
             .wakeUpInterval(5000L)
             .maxIdle(120000L)
           .build();
```

An application might also want to interact with an Infinispan cache within the boundaries of JTA and to do that you need to configure the transaction layer and optionally tweak the locking settings. When interacting with transactional caches, you might want to enable recovery to deal with transactions that finished with an heuristic outcome and if you do that, you will often want to enable JMX management and statistics gathering too:

```
Configuration config = new ConfigurationBuilder()
  .locking()
    .concurrencyLevel(10000).isolationLevel(IsolationLevel.REPEATABLE_READ)
    .lockAcquisitionTimeout(12000L).useLockStriping(false).writeSkewCheck(true)
  .transaction()
    .recovery()
    .transactionManagerLookup(new GenericTransactionManagerLookup())
  .jmxStatistics()
  .build();
```

Configuring Infinispan with one or several chained persistent stores is simple too:

```
Configuration config = new ConfigurationBuilder()
      .loaders()
        .shared(false).passivation(false).preload(false)

.addFileCacheStore().location("/tmp").streamBufferSize(1800).async().enable().threadPoolSize(20).b
```

7.2.1 Advanced programmatic configuration

The fluent configuration can also be used to configure more advanced or exotic options, such as advanced externalizers:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .serialization()
    .addAdvancedExternalizer(PersonExternalizer.class)
    .addAdvancedExternalizer(999, AddressExternalizer.class)
  .build();
```

Or, add custom interceptors:

```
Configuration config = new ConfigurationBuilder()
  .customInterceptors().interceptors()
    .add(new FirstInterceptor()).first()
    .add(new LastInterceptor()).last()
    .add(new FixPositionInterceptor()).atIndex(8)
    .add(new AfterInterceptor()).after(LockingInterceptor.class)
    .add(new BeforeInterceptor()).before(CallInterceptor.class)
  .build();
```

For information on the individual configuration options, please check the configuration guide.

8 Grid File System

Infinispan's GridFileSystem is a new, experimental API that exposes an Infinispan-backed data grid as a file system. This API is available in Infinispan 4.1.0 (from 4.1.0.ALPHA2 onwards).

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

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

Here is a quick code snippet demonstrating usage:

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

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

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

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

Copying stuff to the grid file system:

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

Reading stuff from the grid:

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

8.1 WebDAV demo

Infinispan 4.1.0 also ships with a demo WebDAV application that makes use of the grid file system APIs. This demo app is packaged as a WAR file which can be deployed in a servlet container, such as JBoss AS or Tomcat, and exposes the grid as a file system over WebDAV. This could then be mounted as a remote drive on your operating system.

Here is a short video clip showcasing this demo:

9 Infinispan WebSocket Server

The Infinispan WebSocket Server can be used to expose an Infinispan Cache instance over a WebSocket Interface via a very simple Javascript "Cache" API. The WebSocket Interface was introduced as part of the HTML 5 specification. It defines a full-duplex communication channel to the browser, operating over a single socket (unlike Comet or Ajax) and is exposed to the browser via a Javascript interface.

9.1 Starting The Server

The Infinispan WebSocket server is included in Infinispan distributions from **4.1.0.BETA1** onwards, in both the `-bin.zip` and `-all.zip` archives. To start the server, use the bin/startServer.sh (or bin\startServer.bat) command-line scripts, using the `-r websocket` switch.

```
$ bin/startServer.sh -r websocket
```

For more help on available switches, check out the server command line options article.

9.2 Javascript API

Writing a web page that uses the Infinispan Cache API is trivial. The page simply needs to include a `<script>` declaration for the `infinispan-ws.js` Javascript source file. This script is served up by WebSocket Server.

So, for loading `infinispan-ws.js` from a WebSocket Server instance running on **www.acme.com:8181** (default port):

```
<script type="text/javascript" src="<a href="http://www.acme.com:61999/infinispan-ws.js"
target="_blank">http://www.acme.com:8181/infinispan-ws.js</a>" />
```

9.2.1 Creating a Client-Side Cache Object Instance

The client-side interface to a server-side Infinispan cache is the `Cache` Javascript object. It can be constructed as follows:

```
<script type="text/javascript">
    var cache = new Cache();

    // etc...
</script>
```

By default, the `Cache` instance will interface to the default Infinispan Cache associated with the WebSocket Server from which the `infinispan-ws.js` Javascript source file was loaded. So, in the above case, the `Cache` object instance will connect to the WebSocket Server running on **www.acme.com:8181** (i.e. **ws://www.acme.com:8181**).

The Infinispan Cache name and WebSocket Server address can be specified in the {{Cache} object constructor as follows:

```
var cache = new Cache("omCache", "ws://ws.acmews.com:8181");
// etc...
```

9.2.2 Cache Operations

A number of cache operations can be performed via the `Cache` object instance such as **get**, **put**, **remove**, **notify** and **unnotify**.

The `get` and `notify` operations require a callback function to be registered with the `Cache` object instance. This callback function receives all add/update/remove notifications on any cache entries for which the `notify` function was invoked. It also asynchronously receives the result of a single invocation of the `get` function i.e. `get` can be thought of as "notify once, immediately".

The callback function is registered with the `Cache` object instance via the `registerCallback` function. The function should have 2 parameters - `key` and `value`, relating to the cache key and value.

```
var cache = new Cache();

// Ask to be notified about some cache entries...
cache.notify("orderStatus");
cache.notify("expectedDeliveryTime");

// Register the callback function for receiving notifcations...
cache.registerCallback(cacheCallback);

// Cache callback function...
function cacheCallback(key, value) {
    // Handle notification...
}
```

Getting and updating data in the cache is done by simply calling the `get`, `put` and `remove` functions on the `Cache` object instance. These operations could be triggered by user interaction with a web form e.g.

```
<form onsubmit="return false;">

    <!-- Other form components... -->

    <!-- Buttons for making cache updates... -->
    <input type="button" value="Put"
           onclick="cache.put(this.form.key.value, this.form.val.value)" />
    <input type="button" value="Get"
           onclick="cache.get(this.form.key.value)" />
    <input type="button" value="Remove"
           onclick="cache.remove(this.form.key.value)" />
</form>
```

9.3 Sample code

Infinispan's source tree contains a sample HTML document that makes use of the WebSocket server. Browse through the source of this HTML document here.

9.4 Browser Support

At the time of writing, Google Chrome was the only browser with native WebSocket support. However, the jWebSocket project provides a client side Javascript library that adds WebSocket support to any Flash enabled browser.

9.5 Screencast

See the following demo of the Infinispan WebSocket Server in action.

9.6 Status

Prototype/Alpha.

9.7 Source

Browse Infinispan's Git repository.

10 Using Infinispan Memcached Server

10.1 Introduction

Starting with version 4.1, the Infinispan distribution contains a server module that implements the Memcached text protocol. This allows Memcached clients to talk to one or serveral Infinispan backed Memcached servers. These servers could either be working standalone just like Memcached does where each server acts independently and does not communicate with the rest, or they could be clustered where servers replicate or distribute their contents to other Infinispan backed Memcached servers, thus providing clients with failover capabilities.

10.2 Starting an Infinispan Memcached server

The simplest way to start up an Infinispan memcached server is to simply unzip the all distribution and either run the `startServer.bat` or `startServer.sh` script passing memcached as the protocol to run. For example:

```
$ ./bin/startServer.sh -r memcached
```

When the script is called without any further parameters, the started Infinispan Memcached server binds to **port 11211 on localhost** (127.0.0.1) and uses a local (unclustered) Infinispan cache instance configured with default values underneath.

10.3 Command Line Options

You can optionally pass a set of parameters to the Infinispan Memcached server that allow you to configure different parts of the server. You can find detailed information in the server command line options article.

Please note that, since the Memcached protocol does not allow specifying a cache to use, the mapping between Infinispan Memcached server instances and Infinispan Cache instances is 1 to 1. Therefore, when passing an Infinispan configuration file to the Infinispan Memcache server, either define a named cache with name "memcachedCache" or modify the default cache configuration.

10.4 Enabling Statistics

The memcached module makes use of the JMX statistics in the cache configuration. For example:

```
<default>
   ...
   <jmxStatistics enabled="true"/>
   ...
</default>
```

Infinispan Memcached server has jmx statistics enabled by default.

10.5 Command Clarifications

10.5.1 Flush All

Even in a clustered environment, `flush_all` command leads to the clearing of the Infinispan Memcached server where the call lands. There's no attempt to propagate this flush to other nodes in the cluster. This is done so that `flush_all` with delay use case can be reproduced with the Infinispan Memcached server. The aim of passing a delay to `flush_all` is so that different Memcached servers in a full can be flushed at different times, and hence avoid overloading the database with requests as a result of all Memcached servers being empty. For more info, check the Memcached text protocol section on flush_all.

10.6 Unsupported Features

This section explains those parts of the memcached text protocol that for one reason or the other, are not currently supported by the Infinispan based memcached implementation.

10.6.1 Individual Stats

Due to difference in nature between the original memcached implementation which is C/C++ based and the Infinispan implementation which is Java based, there're some general purpose stats that are not supported. For these unsupported stats, Infinispan memcached server always returns 0.

Here's the list of currently unsupported stats:

- `pid`
- `pointer_size`
- `rusage_user`
- `rusage_system`
- `bytes`
- `curr_connections`
- `total_connections`
- `connection_structures`
- `auth_cmds`
- `auth_errors`
- `limit_maxbytes`
- `threads`
- `conn_yields`
- `reclaimed`

10.6.2 Statistic Settings

The settings statistics section of the text protocol has not been implemented due to its volatility.

10.6.3 Settings with Arguments Parameter

Since the arguments that can be send to the Memcached server are not documented, Infinispan Memcached server does not support passing any arguments to stats command. If any parameters are passed, the Infinispan Memcached server will respond with a `CLIENT_ERROR`.

10.6.4 Delete Hold Time Parameter

Memcached does no longer honor the optional hold time parameter to delete command and so the Infinispan based memcached server does not implement such feature either.

10.6.5 Verbosity Command

Verbosity command is not supported since Infinispan logging cannot be simplified to defining the logging level alone.

11 Asynchronous API

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

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

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

11.1 Why use such an API?

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

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

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

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

11.2 Which processes actually happen asynchronously?

There are 4 things in Infinispan that can be considered to be on the critical path of a typical write operation. These are, in terms of cost, network calls, marshalling, writing to a cache store (optional), and locking. As of Infinispan 4.0, using the async methods will take the network calls and marshalling off the critical path. For various technical reasons, writing to a cache store and acquiring locks, however, still happens in the caller's thread. In future, we plan to take these offline as well. See this developer mail list thread about this topic.

11.3 Notifying futures

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

```
FutureListener futureListener = new FutureListener() {

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

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

11.4 Further reading

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

12 Tree API Module

12.1 Introduction

Infinispan's tree API module offers clients the possibility of storing data using a tree-structure like API. This API is similar to the one provided by JBoss Cache, hence the tree module is perfect for those users wanting to migrate their applications from JBoss Cache to Infinispan, who want to limit changes their codebase as part of the migration. Besides, it's important to understand that Infinispan provides this tree API much more efficiently than JBoss Cache did, so if you're a user of the tree API in JBoss Cache, you should consider migrating to Infinispan.

12.2 What is Tree API about?

The aim of this API is to store information in a hierarchical way. The hierarchy is defined using paths represented as Fqn or fully qualified names, for example: */this/is/a/fqn/path* or */another/path* . In the hierarchy, there's an special path called root which represents the starting point of all paths and it's represented as: /

Each FQN path is represented as a node where users can store data using a key/value pair style API (i.e. a Map). For example, in */persons/john* , you could store information belonging to John, for example: surname=Smith, birthdate=05/02/1980...etc.

Please remember that users should not use root as a place to store data. Instead, users should define their own paths and store data there. The following sections will delve into the practical aspects of this API.

12.3 Using Tree API

12.3.1 Dependencies

For your application to use the tree API, you need to import `infinispan-tree.jar` which can be located in the Infinispan binary distributions, or you can simply add a dependency to this module in your pom.xml:

```
<dependencies>
  ...
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-tree</artifactId>
    <version>$put-infinispan-version-here</version>
  </dependency>
  ...
</dependencies>
```

12.3.2 Creating a Tree Cache

The first step to use the tree API is to actually create a tree cache. To do so, you need to create an Infinispan Cache as you'd normally do, and using the TreeCacheFactory, create an instance of TreeCache. A very important note to remember here is that the Cache instance passed to the factory must be configured with invocation batching. For example:

```
import org.infinispan.config.Configuration;
import org.infinispan.tree.TreeCacheFactory;
import org.infinispan.tree.TreeCache;
...
Configuration config = new Configuration();
config.setInvocationBatchingEnabled(true);
Cache cache = new DefaultCacheManager(config).getCache();
TreeCache treeCache = TreeCacheFactory.createTreeCache(cache);
```

12.3.3 Manipulating data in a Tree Cache

The Tree API effectively provides two ways to interact with the data:

1. Via TreeCache convenience methods: These methods are located within the TreeCache interface and enable users to store, retrieve, move, remove...etc data with a single call that takes the Fqn, in String or Fqn format, and the data involved in the call. For example:

   ```
   treeCache.put("/persons/john", "surname", "Smith");
   ```

   Or:

   ```
   import org.infinispan.tree.Fqn;
   ...
   Fqn johnFqn = Fqn.fromString("persons/john");
   Calendar calendar = Calendar.getInstance();
   calendar.set(1980, 5, 2);
   treeCache.put(johnFqn, "birthdate", calendar.getTime()));
   ```

2. Via Node API: It allows finer control over the individual nodes that form the FQN, allowing manipulation of nodes relative to a particular node. For example:

```
import org.infinispan.tree.Node;
...
TreeCache treeCache = ...
Fqn johnFqn = Fqn.fromElements("persons", "john");
Node<String, Object> john = treeCache.getRoot().addChild(johnFqn);
john.put("surname", "Smith");
```

Or:

```
Node persons = treeCache.getRoot().addChild(Fqn.fromString("persons"));
Node<String, Object> john = persons.addChild(Fqn.fromString("john"));
john.put("surname", "Smith");
```

Or even:

```
Fqn personsFqn = Fqn.fromString("persons");
Fqn johnFqn = Fqn.fromRelative(personsFqn, Fqn.fromString("john"));
Node<String, Object> john = treeCache.getRoot().addChild(johnFqn);
john.put("surname", "Smith");
```

A node also provides the ability to access its parent or children. For example:

```
Node<String, Object> john = ...
Node persons = john.getParent();
```

Or:

```
Set<Node<String, Object>> personsChildren = persons.getChildren();
```

12.3.4 Common Operations

In the previous section, some of the most used operations, such as addition and retrieval, have been shown. However, there are other important operations that are worth mentioning, such as remove:

You can for example remove an entire node, i.e. */persons/john* , using:

```
treeCache.removeNode("/persons/john");
```

Or remove a child node, i.e. persons that a child of root, via:

```
treeCache.getRoot().removeChild(Fqn.fromString("persons"));
```

You can also remove a particular key/value pair in a node:

```
Node john = treeCache.getRoot().getChild(Fqn.fromElements("persons", "john"));
john.remove("surname");
```

Or you can remove all data in a node with:

```
Node john = treeCache.getRoot().getChild(Fqn.fromElements("persons", "john"));
john.clearData();
```

Another important operation supported by Tree API is the ability to move nodes around in the tree. Imagine we have a node called "john" which is located under root node. The following example is going to show how to we can move "john" node to be under "persons" node:

Current tree structure:

```
   /persons
   /john
```

Moving trees from one FQN to another:

```
Node john = treeCache.getRoot().addChild(Fqn.fromString("john"));
Node persons = treeCache.getRoot().getChild(Fqn.fromString("persons"));
treeCache.move(john.getFqn(), persons.getFqn());
```

Final tree structure:

```
/persons/john
```

12.4 Locking In Tree API

Understanding when and how locks are acquired when manipulating the tree structure is important in order to maximise the performance of any client application interacting against the tree, while at the same time maintaining consistency.

Locking on the tree API happens on a per node basis. So, if you're putting or updating a key/value under a particular node, a write lock is acquired for that node. In such case, no write locks are acquired for parent node of the node being modified, and no locks are acquired for children nodes.

If you're adding or removing a node, the parent is not locked for writing. In JBoss Cache, this behaviour was configurable with the default being that parent was not locked for insertion or removal.

Finally, when a node is moved, the node that's been moved and any of its children are locked, but also the target node and the new location of the moved node and its children. To understand this better, let's look at an example:

Imagine you have a hierarchy like this and we want to move c/ to be underneath b/:

```
        /
      --|--
    /      \
    a      c
    |      |
    b      e
    |
    d
```

The end result would be something like this:

```
        /
        |
        a
        |
        b
      --|--
    /      \
    d      c
           |
           e
```

To make this move, locks would have been acquired on:

- */a/b* - because it's the parent underneath which the data will be put
- */c* and */c/e* - because they're the nodes that are being moved
- */a/b/c* and */a/b/c/e* - because that's new target location for the nodes being moved

12.5 Listeners for tree cache events

The current Infinispan listeners have been designed with key/value store notifications in mind, and hence they do not map to tree cache events correctly. Tree cache specific listeners that map directly to tree cache events (i.e. adding a child...etc) are desirable but these are not yet available. If you're interested in this type of listeners, please follow this issue to find out about any progress in this area.

13 Infinispan as a Directory for Lucene

Infinispan is including a highly scalable distributed **Apache Lucene Directory** implementation.

This directory closely mimicks the same semantics of the traditional filesystem and RAM-based directories, being able to work as a drop-in replacement for existing applications using Lucene and providing reliable index sharing and other features of Infinispan like node autodiscovery, automatic failover and rebalancing, optionally transactions, and can be backed by traditional storage solutions as filesystem, databases or cloud store engines.

The implementation extends Lucene's *org.apache.lucene.store.Directory* so it can be used to *store* the index in a cluster-wide shared memory, making it easy to distribute the index. Compared to rsync-based replication this solution is suited for use cases in which your application makes frequent changes to the index and you need them to be quickly distributed to all nodes, having configurable consistency levels, synchronicity and guarantees, total elasticity and autodiscovery; also changes applied to the index can optionally participate in a JTA transaction; since version 5 supporting XA transactions with recovery.

Two different *LockFactory* implementations are provided to guarantee only one *IndexWriter* at a time will make changes to the index, again implementing the same semantics as when opening an index on a local filesystem. As with other Lucene Directories, you can override the *LockFactory* if you prefer to use an alternative implementation.

13.1 Additional Links

Javadoc: http://docs.jboss.org/infinispan/5.2/apidocs/org/infinispan/lucene/InfinispanDirectory.html
Issue tracker: https://jira.jboss.org/browse/ISPN/component/12312732
Source code: http://www.jboss.org/infinispan/sourcecode.html

13.2 Lucene compatibility

Current version was developed and compiled against Lucene 3.6.0, and also tested to work with Lucene versions from 3.0.x to 3.5.0, version 2.9.x, and the older 2.4.1.

13.3 How to use it

To create a Directory instance:

```
import org.apache.lucene.store.Directory;
import org.infinispan.lucene.InfinispanDirectory;
import org.infinispan.Cache;

Cache cache = // create an Infinispan cache, configured as you like
Directory indexDir = new InfinispanDirectory(cache, "indexName");
```

The ***indexName*** is a unique key to identify your index. It takes the same role as the path did on filesystem based indexes: you can create several different indexes giving them different names. When you use the same *indexName* in another instance connected to the same network (or instantiated on the same machine, useful for testing) they will join, form a cluster and share all content.
New nodes can be added or removed dynamically, making the service administration very easy and also suited for cloud environments: it's simple to react to load spikes, as adding more memory and CPU power to the search system is done by just starting more nodes.

13.4 Limitations

As when using an *IndexWriter* on a filesystem based *Directory*, even on the clustered edition only one *IndexWriter* can be opened across the whole cluster.

As an example, Hibernate Search, which includes integration with this Lucene Directory since version 3.3, sends index change requests across a JMS queue, or a *JGroups* channel. Other valid approaches are to proxy the remote *IndexWriter* or just design your application in such a way that only one node attempts to write it.

Reading (searching) is of course possible in parallel, from any number of threads on each node; changes applied to the single IndexWriter are affecting results of all threads on all nodes in a very short time, or guaranteed to be visible after a commit when using transactions.

13.5 Configuration

Infinispan can be configured as LOCAL clustering mode, in which case it will disable clustering features and serve as a cache for the index, or any clustering mode.
A transaction manager is not mandatory, but when enabled the changes to the index can participate in transactions.

Batching was required in previous versions, it's not strictly needed anymore.

As better explained in the javadocs of *org.infinispan.lucene.InfinispanDirectory*, it's possible for it to use more than a single cache, using specific configurations for different purposes. When using readlocks, make sure to not enable transactions on this cache.

Any Infinispan configuration should work fine as long as caches are not configured to remove entries after thresholds.

13.6 Demo

There is a simple command-line demo of it's capabilities distributed with Infinispan under demos/lucene-directory; make sure you grab the *"Binaries, server and demos"* package from download page , which contains all demos.

Start several instances, then try adding text in one instance and searching for it on the other. The configuration is not tuned at all, but should work out-of-the box without any changes. If your network interface has multicast enabled, it will cluster across the local network with other instances of the demo.

13.7 Maven dependencies

All you need is *org.infinispan:infinispan-lucene-directory* :

```
<dependencies>
   <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-lucene-directory</artifactId>
      <version>5.2.0.BETA1</version>
   </dependency>
</dependencies>
```

13.8 Using a CacheLoader

Using a CacheLoader you can have the index content backed up to a permanent storage; you can use a shared store for all nodes or one per node, see Cache Loaders and Stores for more details.

When using a CacheLoader to store a Lucene index, to get best write performance you would need to configure the CacheLoader with *async=true*. However, this should not be done unless ISPN-1174 is resolved.

13.8.1 Storing the index in a database

It might be useful to store the Lucene index in a relational database; this would be very slow but Infinispan can act as a cache between the application and the JDBC interface, making this configuration useful in both clustered and non-clustered configurations.
When storing indexes in a JDBC database, it's suggested to use the JdbcStringBasedCacheStore, which will need this attribute:

```
<property name="key2StringMapperClass" value="org.infinispan.lucene.LuceneKey2StringMapper" />
```

13.8.2 Loading an existing Lucene Index

The *org.infinispan.lucene.cachestore.LuceneCacheLoader* is an Infinispan CacheLoader able to have Infinispan directly load data from an existing Lucene index into the grid. Currently this supports reading only.

Property	Description	Default
location	The path where the indexes are stored. Subdirectories (of first level only) should contain the indexes to be loaded, each directory matching the index name attribute of the InfinispanDirectory constructor.	none (mandatory)
autoChunkSize	A threshold in bytes: if any segment is larger than this, it will be transparently chunked in smaller cache entries up to this size.	32MB

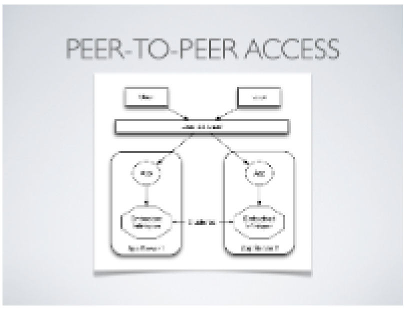
It's worth noting that the IO operations are delegated to Lucene's standard *org.apache.lucene.store.FSDirectory*, which will select an optimal approach for the running platform.

Implementing write-through should not be hard: you're welcome to try implementing it.

14 Infinispan Server Modules

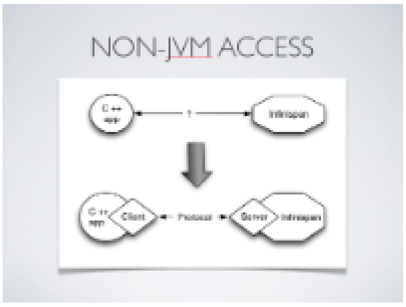
14.1 Introduction

Traditionally, clients have interacted with Infinispan in a peer-to-peer (p2p) fashion where Infinispan and the client code that accessed it lived within the same VM. When Infinispan is queried in this way, it is considered to be accessed in an embedded fashion, as shown in the screenshot below

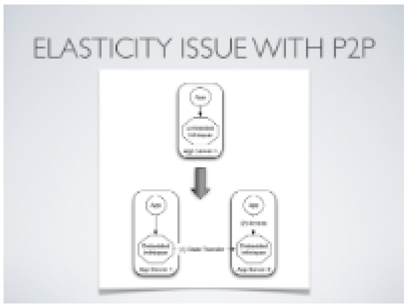


14.1.1 Client-Server over Peer-to-Peer

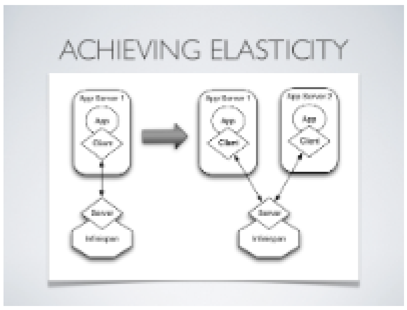
However, there are situations when accessing Infinispan in a client-server mode might make more sense than accessing it via p2p. For example, when trying to **access Infinispan from a non-JVM environment** . Since Infinispan is written in Java, if someone had a C++ application that wanted to access it, it couldn't just do it in a p2p way. On the other hand, client-server would be perfectly suited here assuming that a language neutral protocol was used and the corresponding client and server implementations were available.



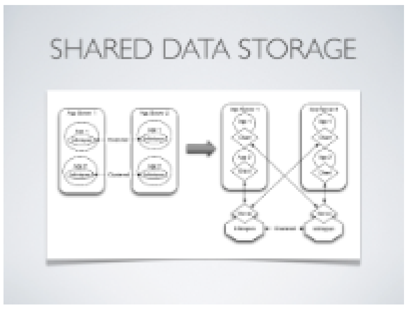
In other situations, Infinispan users want to have an **elastic application tier** where you start/stop business processing servers very regularly. Now, if users deployed Infinispan configured with distribution or state transfer, startup time could be greatly influenced by the shuffling around of data that happens in these situations. So in the following diagram, assuming Infinispan was deployed in p2p mode, the app in the second server could not access Infinispan until state transfer had completed.



This effectively means that bringing up new application-tier servers is impacted by things like state transfer because applications cannot access Infinispan until these processes have finished and if the state being shifted around is large, this could take some time. This is undesirable in an elastic environment where you want quick application-tier server turnaround and predictable startup times. Problems like this can be solved by accessing Infinispan in a client-server mode because starting a new application-tier server is just a matter of starting a lightweight client that can connect to the backing data grid server. No need for rehashing or state transfer to occur and as a result server startup times can be more predictable which is very important for modern cloud-based deployments where elasticity in your application tier is important.



Other times, it's common to find multiple applications needing access to data storage. In this cases, you could in theory deploy an Infinispan instance per each of those applications but this could be wasteful and difficult to maintain. Thing about databases here, you don't deploy a database alongside each of your applications, do you? So, alternatively you could deploy Infinispan in client-server mode keeping a pool of Infinispan data grid nodes acting as a **shared storage tier for your applications** .



Deploying Infinispan in this way also allows you to manage each tier independently, for example, you can upgrade you application or app server without bringing down your Infinispan data grid nodes.

14.1.2 Peer-to-Peer over Client-Server

Before talking about individual Infinispan server modules, it's worth mentioning that in spite of all the benefits, client-server Infinispan still has disadvantages over p2p. Firstly, p2p deployments are simpler than client-server ones because in p2p, all peers are equals to each other and hence this simplifies deployment. So, if this is the first time you're using Infinispan, p2p is likely to be easier for you to get going compared to client-server.

Client-server Infinispan requests are likely to take longer compared to p2p requests, due to the serialization and network cost in remote calls. So, this is an important factor to take in account when designing your application. For example, with replicated Infinispan caches, it might be more performant to have lightweight HTTP clients connecting to a server side application that accesses Infinispan in p2p mode, rather than having more heavyweight client side apps talking to Infinispan in client-server mode, particularly if data size handled is rather large. With distributed caches, the difference might not be so big because even in p2p deployments, you're not guaranteed to have all data available locally.

Environments where application tier elasticity is not so important, or where server side applications access state-transfer-disabled, replicated Infinispan cache instances are amongst scenarios where Infinispan p2p deployments can be more suited than client-server ones.

14.2 Server Modules

So, now that it's clear when it makes sense to deploy Infinispan in client-server mode, what are available solutions? All Infinispan server modules are based on the same pattern where the server backend creates an embedded Infinispan instance and if you start multiple backends, they can form a cluster and share/distribute state if configured to do so. The server types below primarily differ in the type of listener endpoint used to handle incoming connections. Here's a brief look at each of them:

14.2.1 REST Server Module

- This module, which is distributed as a WAR file, can be deployed in any servlet container to allow Infinispan to be accessed via a RESTful HTTP interface.
- To connect to it, you can use any HTTP client out there and there're tons of different client implementations available out there for pretty much any language or system.
- This module is particularly recommended for those environments where HTTP port is the only access method allowed between clients and servers.
- Clients wanting to load balance or failover between different Infinispan REST servers can do so using any standard HTTP load balancer such as mod_cluster. It's worth noting though these load balancers maintain a static view of the servers in the backend and if a new one was to be added, it would require manual update of the load balancer.

14.2.2 Memcached Server Module

- This module is an implementation of the Memcached text protocol backed by Infinispan.
- To connect to it, you can use any of the existing Memcached clients which are pretty diverse.
- As opposed to Memcached servers, Infinispan based Memcached servers can actually be clustered and hence they can replicate or distribute data using consistent hash algorithms around the cluster. So, this module is particularly of interest to those users that want to provide failover capabilities to the data stored in Memcached servers.
- In terms of load balancing and failover, there're a few clients that can load balance or failover given a static list of server addresses (perl's Cache::Memcached for example) but any server addition or removal would require manual intervention.

14.2.3 Hot Rod Server Module

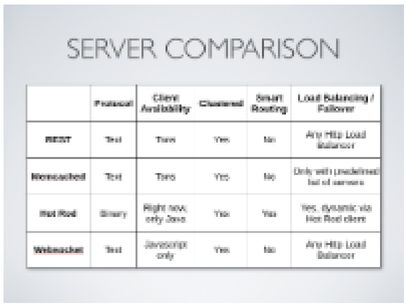
- This module is an implementation of the Hot Rod binary protocol backed by Infinispan which allows clients to do dynamic load balancing and failover and smart routing.
- So far, a single Java client, which is the reference implementation, has been fully developed. A beta version of the Python client is available, and a JRuby and pure Ruby clients are in the making.
- If you're clients are running Java, this should be your defacto server module choice because it allows for dynamic load balancing and failover. This means that Hot Rod clients can dynamically detect changes in the topology of Hot Rod servers as long as these are clustered, so when new nodes join or leave, clients update their Hot Rod server topology view. On top of that, when Hot Rod servers are configured with distribution, clients can detect where a particular key resides and so they can route requests smartly.
- Load balancing and failover is dynamically provided by Hot Rod client implementations using information provided by the server.

14.2.4 WebSocket Server Module

- This module enables Infinispan to be exposed over a Websocket interface via a Javascript API.
- This module is very specifically designed for Javascript clients and so that is the only client implementation available.
- This module is particularly suited for developers wanting to enable access to Infinispan instances from their Javascript codebase.
- Since websockets work on the same HTTP port, any HTTP load balancer would do to load balance and failover.

14.3 Server Comparison Summary

Here's a table comparing and summarizing the capabilities of distinct server modules:



SERVER COMPARISON

	Protocol	Client Availability	Clustered	Smart Routing	Load Balancing / Failover
REST	Text	Tons	Yes	No	Any Http Load Balancer
Memcached	Text	Tons	Yes	No	Only with predefined list of servers
Hot Rod	Binary	Right now, only Java	Yes	Yes	Yes, dynamic via Hot Rod client
Websocket	Text	Javascript only	Yes	No	Any Http Load Balancer

15 Management Tooling

15.1 Introduction

Management of Infinispan instances is all about exposing as much relevant statistical information that allows administrators to get view of the state of each Infinispan instance. Taking in account that a single installation could be made up of several tens or hundreds Infinispan instances, providing clear and concise information in an efficient manner is imperative. The following sections dive into the range of management tooling that Infinispan provides.

15.2 JMX

Over the years, JMX has become the de facto standard for management and administration of middleware and as a result, the Infinispan team has decided to standarize on this technology for the exposure of management or statistical information.

15.2.1 Enabling JMX Statistics

JMX reporting can be enabled at 2 different levels:

1. CacheManager level: The CacheManager is the entity that governs all the cache instances that have been created from it. Details on the information exposed at the CacheManager level can be found below. For the moment, let's just focus on how to enable the CacheManager to report management data via JMX.

- If configuring the CacheManager via XML, make sure you add the following XML under the <global> element:

```
<globalJmxStatistics enabled="true"/>
```

- If configuring the CacheManager programmatically, simply add the following code:

```
GlobalConfiguration globalConfiguration = ...
globalConfiguration.setExposeGlobalJmxStatistics(true);
```

2. Cache level: At this level, you will receive management information generated by individual cache instances. Details on the information exposed at the Cache level are explained below. For the moment, let's just focus on how to enable the Cache to report management data via JMX:

- If configuring the Cache via XML, make sure you add the following XML under the either <default>, if you're configuring the default Cache instance, or under the corresponding <namedCache>:

```
<jmxStatistics enabled="true"/>
```

- If configuring the Cache programmatically, simply add the following code:

```
Configuration configuration = ...
configuration.setExposeJmxStatistics(true);
```

Understanding The MBeans

Once you have enabled JMX reporting at either the CacheManager or Cache level, you should be able to connect to VM(s) where Infinispan is running using a standard JMX GUI such as JConsole or VisualVM, and you should find the following MBeans:

Infinispan 4.1 or earlier

- If you enabled CacheManager level JMX statistics, you should see an MBean called infinispan:cache-name=[global],jmx-resource=CacheManager with properties specified by the CacheManager MBean.
- If you enabled Cache level JMX statistics, you should see several different MBeans depending on which configuration options have been enabled. For example, if you have configured a write behind cache store, you should see an MBean exposing properties belonging to the cache store component. All Cache level MBeans will follow the same format though which is the following: infinispan:cache-name=<name-of-cache>(<cache-mode>),jmx-resource=<component-name> where:
    - <name-of-cache> has been substituted by the actual cache name. If this cache represents the default cache, its name will be "___defaultcache".
    - <cache mode> has been substituted by the cache mode of the cache. The cache mode is represented by the lower case version of the possible enumeration values shown here.
    - <component-name> has been substituted by one of the JMX component names in the JMX reference documentation.

For example, the cache store JMX component MBean for a default cache configured with synchronous distribution would have the following name:infinispan:cache-name=___defaultcache(dist_sync),jmx-resource=CacheStore

Please note that any cache names that contain ':' or '=' characters will be substituted by '_' character. Infinispan does this because ':' and '=' are control characters for JMX object names.

Infinispan 4.2 or later

- If you enabled CacheManager level JMX statistics, without further configuration, you should see an MBean called *org.infinispan:type=CacheManager,name="DefaultCacheManager"* with properties specified by the CacheManager MBean.
- Using the cacheManagerName attribute in globalJmxStatistics XML element, or using the corresponding GlobalConfiguration.setCacheManagerName() call, you can name the cache manager in such way that the name is used as part of the JMX object name. So, if the name had been "Hibernate2LC", the JMX name for the cache manager would have been: *org.infinispan:type=CacheManager,name="Hibernate2LC"* . This offers a nice and clean way to manage environments where multiple cache managers are deployed, which follows JMX best practices.
- If you enabled Cache level JMX statistics, you should see several different MBeans depending on which configuration options have been enabled. For example, if you have configured a write behind cache store, you should see an MBean exposing properties belonging to the cache store component. All Cache level MBeans follow the same format though which is the following: *org.infinispan:type=Cache,name="<name-of-cache>(<cache-mode>)",manager="<name-of-cache-ma* where:
  - <name-of-cache> has been substituted by the actual cache name. If this cache represents the default cache, it's name will be *"_defaultCache"* .
  - <cache-mode> has been substituted by the cache mode of the cache. The cache mode is represented by the lower case version of the possible enumeration values shown here.
  - <name-of-cache-manager> has been substituted by the name of the cache manager to which this cache belongs. The name is derived from the cacheManagerName attribute value in globalJmxStatistics element.
  - <component-name> has been substituted by one of the JMX component names in the JMX reference documentation.

For example, the cache store JMX component MBean for a default cache configured with synchronous distribution would have the following name: *org.infinispan:type=Cache,name="_defaultcache(dist_sync)", manager="DefaultCacheManager",component=CacheStore*

Please note that cache and cache manager names are quoted to protect against illegal characters being used in these user-defined names.

15.2.2 Multiple JMX Domains

There can be situations where several CacheManager instances are created in a single VM, or Cache names belonging to different CacheManagers under the same VM clash.

Infinispan 4.1 or earlier

In order to cope with such situations, Infinispan enabled users to define a particular JMX domain prefix for their MBeans. For example, either of these two options could be used to configure a JMX domain prefix called "myInfinispan" for the CacheManager JMX statistics:

- Via XML:

```
<globalJmxStatistics enabled="true" jmxDomain="myInfinispan"/>
```

- Programmatically:

```
GlobalConfiguration globalConfiguration = ...
globalConfiguration.setExposeGlobalJmxStatistics(true);
globalConfiguration.setJmxDomain("myInfinispan");
```

  Using either of these options should result on the CacheManager MBean name being: *myInfinispan:cache-name=[global],jmx-resource=CacheManager*

As you probably have guessed by now, the default JMX domain prefix is called "*infinispan* ".

Another related configuration option for both CacheManager and Cache JMX statistics allows for duplicate JMX domains to be discovered. Internally, when duplicates are allowed, Infinispan takes the duplicating JMX domain prefix, adds an index that starts at number 2 to the existing prefix and uses that JMX prefix from then onwards. So, for example, if two CacheManagers were started with global JMX statistics enabled, no particular JMX domain was configured, and JMX domain duplicates were allowed, the first CacheManager would be registered under "*infinispan...* ", whereas the second one would be registered under: "*infinispan2...* ". To allow JMX duplicate domains, do the following:

- Via XML:

```
<globalJmxStatistics enabled="true" allowDuplicateDomains="true"/>
```

- Programmatically:

```
GlobalConfiguration globalConfiguration = ...
globalConfiguration.setExposeGlobalJmxStatistics(true);
globalConfiguration.setAllowDuplicateDomains(true)
```

  Remember that by default, duplicate domains are disallowed.

Infinispan 4.2 or later

Using different JMX domains for multi cache manager environments should be last resort. Instead, as mentioned in previous section, it's now possible to name a cache manager in such way that it can easily be identified and used by monitoring tools such as RHQ. For example:

- Via XML:

  ```
  <globalJmxStatistics enabled="true" cacheManagerName="Hibernate2LC"/>
  ```

- Programmatically:

  ```
  GlobalConfiguration globalConfiguration = ...
  globalConfiguration.setExposeGlobalJmxStatistics(true);
  globalConfiguration.setCacheManagerName("Hibernate2LC");
  ```

  Using either of these options should result on the CacheManager MBean name being: org.infinispan:type=CacheManager,name="Hibernate2LC"

Please note as well that since 4.2, the default domain names has changed from "infinispan" to "org.infinispan", as per JMX best practices.

For the time being, you can still set your own jmxDomain if you need to and we also allow duplicate domains, or rather duplicate JMX names, but these should be limited to very special cases where different cache managers within the same JVM are named equally.

15.2.3 Registering MBeans In Non-Default MBean Servers

To finish up with this JMX section, let's quickly discuss where Infinispan registers all these MBeans. By default, Infinispan registers them in the standard JVM MBeanServer plattform. However, users might want to register these MBeans in a different MBeanServer instance. For example, an application server might work with a different MBeanServer instance to the default plattform one. In such cases, users should implement the MBeanServerLookup interface provided by Infinispan so that the getMBeanServer() method returns the MBeanServer under which Infinispan should register the management MBeans. You can find an example in the default PlatformMBeanServerLookup class used by Infinispan. So, once you have your implementation ready, simply configure Infinispan with the fully qualified name of this class. For example:

- Via XML:

```
<globalJmxStatistics enabled="true" mBeanServerLookup="com.acme.MyMBeanServerLookup"/>
```

- Programmatically:

```
GlobalConfiguration globalConfiguration = ...
globalConfiguration.setExposeGlobalJmxStatistics(true);
globalConfiguration.setMBeanServerLookup("com.acme.MyMBeanServerLookup")
```

MBean additions in Infinispan 5.0

There has been a couple of noticeable additions in Infinispan 5.0 in terms of MBean exposed:

1. MBeans related to Infinispan servers are now available that for the moment focus on the transport layer. So, if the Infinispan servers are configured with global JMX statistics, a brand new MBean in *org.infinispan:type=Server,name=<Memcached|Hotrod>,component=Transport* is now available which offers information such as: host name, port, bytes read, byte written, number of worker threads...etc.
2. When global JMX statistics are enabled, JGroups MBeans are also registered automatically, so you can get key information of the group communication transport layer that's used to cluster Infinispan instances. To find out more about the information provided, check the JGroups JMX documentation.

15.3 RHQ

The preferred way to manage multiple Infinispan instances spread accross different servers is to use RHQ, which is JBoss' enterprise management solution. Thanks to RHQ's agent and auto discovery capabilities, monitoring both Cache Manager and Cache instances is a very simple task. With RHQ, administrators have access to graphical views of key runtime parameters or statistics and can also be notified be these exceed or go below certain limits. The Infinispan specific statistics shown by RHQ are a reflection of the JMX information exposed by Infinispan which has been formatted for consumption by RHQ. Please follow these steps to get started with RHQ and have Infinispan instances monitored with it:

1. Firstly, download and install an RHQ server and install and start at least one RHQ agent. The job of the RHQ agent is to send information about the Infinispan instance back to the server which is the one that shows the information via a nice GUI. You can find detailed information on the installation process in RHQ's installation guide and you can find information on how to run an agent in the RHQ agent guide.

   Careful with H2 database installation

   If you're just building a demo or testing RHQ server, you can avoid the need to install a fully fledged database and use an in-memory H2 database instead. However, you might encounter issues after testing database connection as shown here. Simply repeating the installation avoiding testing the connection should work.

   Where do I install the RHQ agent?

   The most common set up is to have the RHQ agent installed in the same machine where Infinispan is running. If you have multiple machines, an agent can be installed in each machine.

2. By now, you should have an RHQ server and agent running. It's time now to download the latest Infinispan binary distribution (*-bin.zip or *-all.zip should do) from the downloads section and locate the RHQ plugin jar file which should be named something like `infinispan-rhq-plugin.jar`. This is located under the `modules/rhq-plugin` directory.

3. The adding and updating plugins section on the RHQ guide contains some more detailed information on how to update both RHQ servers and agents with new plugins, but essentially, this process involves uploading a new plugin to the RHQ server and then pushing the plugin to one, or several, RHQ agents.

   Speeding up plugin installation

   If you're simply demoing or testing and you only have a single agent, once the plugin has been uploaded to the server, simply go to the agent command line interface and type: `plugins update` .This will force the agent to retrieve the latest plugins from the server. Doing this can be considerably faster than some of the other alternatives.

4. At this point, RHQ is ready to start monitoring Infinispan instances, but before firing them up, make sure you start them with the following system properties so that RHQ agents can discover them:

   ```
   -Dcom.sun.management.jmxremote.port=6996 -Dcom.sun.management.jmxremote.ssl=false
   -Dcom.sun.management.jmxremote.authenticate=false
   ```

   Remote JMX port value

   The actual port value used does not really matter here, but what matters is that a port is given, otherwise Infinispan instances cannot be located. So, you can easily start multiple Infinispan instances in a single machine, each with a different remote JMX port, and a locally running agent will be able to discover them all without any problems.

5. Once Infinispan instances have been discovered, you should see a new resource for each of the cache manager running appearing in the Inventory/Discovery Queue of the RHQ server. Simply import it now and you should see each cache manager appearing with as many child cache resources as caches are running in each cache manager. You're now ready to monitor Infinispan!

15.3.1 RHQ monitoring tips

This section focuses on the lessons learned while developing the Infinispan RHQ plugin that are likely to be useful to anyone using RHQ.

- By default, at least in version 2.3.1 of RHQ, the RHQ agent sends an availability report of any managed resources every 5 minutes. The problem with this is that if you're testing whether your Infinispan instance is automatically discovered by the RHQ server, it can take up to 5 minutes to do so! Also, it can take 5 minutes for the RHQ server to figure out that you've shutdown your Infinispan instance. You can change this setting by the following property (default value is 300 seconds) in rhq-agent/conf/agent-configuration.xml. For example, if you wanted the availability to be sent every 1 minute, simply change the value to 60:

```
<entry key="rhq.agent.plugins.availability-scan.period-secs" value="60"/>
```

Careful with agent configuration changes

Please bear in mind the instructions given in the RHQ agent installation and more specifically the paragraph below with regards to changes made to properties in agent-configuration.xml:

> *Once the agent is configured, it persists its configuration in the Java Preferences backing store. Once this happens, agent-configuration.xml is no longer needed or used. Editing agent-configuration.xml will no longer have any effect on the agent, even if you restart the agent. If you want the agent to pick up changes you make to agent-configuration.xml, you must either restart the agent with the "--cleanconfig" command line option or use the "config --import" agent prompt command.*

15.4 Writing plugins for other management tools

As mentioned in the previous section, RHQ consumes the JMX data exposed by Infinispan, and in similar fashion, plugins could be written for other 3rd party management tools that were able to transform these data into the correct representation in these tools, for example graphs,...etc.

16 Asynchronous Options

16.1 Introduction

When Infinispan instances are clustered, regardless of the clustering mode, data can be propagated to other nodes in a synchronous or asynchronous way. When synchronous, the sender waits for replies from the receivers and when asynchronous, the sender sends the data and does not wait for replies from other nodes in the cluster.

With asynchronous modes, speed is more important than consistency and this is particularly advantageous in use cases such as HTTP session replication with sticky sessions enabled. In these scenarios, data, or in this case a particular session, is always accessed on the same cluster node and only in case of failure is data accessed in a different node. This type of architectures allow consistency to be relaxed in favour of increased performance.

In order to choose the asynchronous configuration that best suits your application, it's important to understand the following configuration settings:

16.2 Asynchronous Communications

Whenever you add <async> element within <clustering>, you're telling the underlying JGroups layer in Infinispan to use asynchronous communication. What this means is that JGroups will send any replication/distribution/invalidation request to the wire but will not wait for a reply from the receiver.

16.3 Asynchronous Marshalling

This is a configurable boolean property of <async> element that indicates whether the actual call from Infinispan to the JGroups layer is done on a separate thread or not. When set to true, once Infinispan has figured out that a request needs to be sent to another node, it submits it to the async transport executor so that it can talk to the underlying JGroups layer.

With asynchronous marshalling, Infinispan requests can return back to the client quicker compared to when async marshalling is set to false. The downside though is that client requests can be reordered before they have reached the JGroups layer. In other words, JGroups provides ordering guarantees even for async messages but with async marshalling turned on, requests can reach the JGroups in a different order in which they're called. This can effectively lead to data consistency issues in applications making multiple modifications on the same key/value pair. For example, with async marshalling turned on:

App calls:

```
cache.put("car", "bmw");
cache.remove("car");
```

Other nodes could receive these operations in this order:

```
cache.remove("car");
cache.put("car", "bmw");
```

The end result is clearly different which is often not desirable. So, if your application makes multiple modifications on the same key, you should either: turned off asynchronous marshalling, or set <asyncTransportExecutor> element's maxThreads to 1. The first modification only applies to a particular named cache, whereas the second option affects all named caches in configuration file that are configured with async marshalling. It's worth noting though that having this type of executor configured with a single thread would defeat its purpose adding unnecessary contention point. It'd be better to simply switch off async marshalling.

On the contrary, if your application only ever makes one modification per key/value pair and there's no happens-before relationship between them, then async marshalling is a very valid optimization that can increase performance of your application without data consistency risks.

If you have async marshalling turned on and see exceptions related to `java.util.concurrent.RejectedExecutionException`, as explained in the technical faq page, you should also consider switching off async marshalling.

Back in Infinispan 4.0, when <async> element was used, this property was set to `true` by default. However due to reordering risks mentioned earlier, the default has changed to `false` from Infinispan 4.1 onwards.

16.4 Replication Queue

The aim of the replication queue is to batch the individual cache operations and send them as one, as opposed to sending each cache operation individually. As a result, replication queue enabled configurations perform generally better compared to those that have it switched off because less RPC messages are sent, fewer envelopes are used...etc. The only real trade off to the replication queue is that the queue is flushed periodically (based on time or queue size) and hence it might take longer for the replication/distribution/invalidation to be realised across the cluster. When replication queue is turned off, data is placed directly on the wire and hence it takes less for data to arrive to other nodes.

Until Infinispan 4.1.0.CR2, replication queue always flushed data with async marshalling turned on, which meant that there was a small gap where flush calls could be reordered. Since 4.1.0.CR3, async marshalling configuration is taken into account, and decides whether the flush calls goes directly to the JGroups layer, or whether an intermediate handing over to a different thread occurs. The advantages of using async marshalling with replication queue are less than clear because replication queue itself already makes client requests return faster, so it's generally recommended to have async marshalling turned off, or <asyncTransportExecutor> element's maxThreads set to 1, when replication queue is turned on.

16.5 Asynchronous API

Finally, the asynchronous API can be used to emulate non-blocking APIs, whereby calls are handed over to a different thread and asynchronous API calls return to the client immediately. Similar to async marshalling, using this API can lead to reordering, so you should avoid calling modifying asynchronous methods on the same keys.

16.6 Return Values

Regardless of the asynchronous option used, the return values of cache operations are reliable. If talking about return values of cache operations that return previous value, the correctness of these returns are guaranteed as well regardless of the clustering mode. With replication, the previous value is already available locally, and with distribution, regardless of whether it's asynchronous or synchronous, Infinispan sends a synchronous request to get the previous value if not present locally. If on the other hand the asynchronous API is used, client code needs to get hold of the NotifiyngFuture returned by the async operation in order to be able to query the previous value.

17 Infinispan as Hibernate 2nd-Level Cache in JBoss AS 5.x

A JBoss AS 5.x application can be configured to use Infinispan 4.x as the Hibernate 2nd-level cache, replacing JBoss Cache.

1. Add the attached jar files to the ear lib directory. These include the core 4.1.0.GA Infinispan jar (infinispan-core.jar), the Hibernate/Infinispan integration jar back-ported from Hibernate 3.5 (hibernate-infinispan-3.3.2.GA_CP03.jar), the JGroups jar required by Infinispan 4.1.0 (jgroups-2.10.0.GA.jar), and other required 3rd party jars (river-1.2.3.GA.jar, marshalling-api-1.2.3.GA.jar)
2. Isolate the classloading to be ear-scoped by adding META-INF/jboss-classloading.xml
3.
```
<classloading xmlns="urn:jboss:classloading:1.0" domain="simple-scoped"
parent-first="false" />
```

4. Configure persistence.xml to use Infinispan instead of JBoss Cache:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">
<persistence-unit name="jpa-test">
    <jta-data-source>java:/PostgresDS</jta-data-source>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"
/>

            <property name="hibernate.session_factory_name"
value="SessionFactories/infinispan" />

            <property name="hibernate.cache.use_query_cache" value="true" />
            <property name="hibernate.cache.use_second_level_cache" value="true" />
            <property name="hibernate.generate_statistics" value="true" />
            <property name="hibernate.cache.use_structured_entries" value="true" />

            <property name="hibernate.cache.region_prefix" value="infinispan" />

            <property name="hibernate.show_sql" value="true" />

            <property name="hibernate.hbm2ddl.auto" value="validate" />

            <!-- Infinispan second level cache configuration -->
            <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.infinispan.InfinispanRegionFactory" />
        </properties>
    </persistence-unit>
</persistence>
```

18 Clustered Configuration QuickStart

Infinispan ships with *pre-configured* JGroups stacks that make it easy for you to jump-start a clustered configuration.

18.1 Using an external JGroups file

If you are configuring your cache programmatically, all you need to do is:

```
// ...
GlobalConfiguration gc = new
GlobalConfigurationBuilder().transport().addProperty("configurationFile",
"jgroups.xml").build();
// ...
```

and if you happen to use an XML file to configure Infinispan, just use:

```
<infinispan>
  <global>
    <transport>
      <properties>
        <property name="configurationFile" value="jgroups.xml" />
      </properties>
    </transport>
  </global>

  ...

</infinispan>
```

In both cases above, Infinispan looks for *jgroups.xml* first in your classpath, and then for an absolute path name if not found in the classpath.

18.2 Use one of the pre-configured JGroups files

Infinispan ships with a few different JGroups files (packaged in infinispan-core.jar) which means they will already be on your classpath by default. All you need to do is specify the file name, e.g., instead of jgroups.xml above, specify jgroups-tcp.xml.

The configurations available are:

- jgroups-udp.xml - Uses UDP as a transport, and UDP multicast for discovery. Usually suitable for larger (over 100 nodes) clusters *or* if you are using replication or invalidation. Minimises opening too many sockets.
- jgroups-tcp.xml - Uses TCP as a transport and UDP multicast for discovery. Better for smaller clusters (under 100 nodes) *only if* you are using distribution, as TCP is more efficient as a point-to-point protocol
- jgroups-ec2.xml - Uses TCP as a transport and S3_PING for discovery. Suitable on Amazon EC2 nodes where UDP multicast isn't available.

18.2.1 Fine-tuning JGroups settings

The settings above can be further tuned without editing the XML files themselves. Passing in certain system properties to your JVM at startup can affect the behaviour of some of these settings. The table below shows you which settings can be configured in this way. E.g.,

```
$ java -cp ... -Djgroups.tcp.port=1234 -Djgroups.tcp.address=10.11.12.13
```

jgroups-udp.xml

System Property	Description	Default	Required?
jgroups.udp.mcast_addr	IP address to use for multicast (both for communications and discovery). Must be a valid Class D IP address, suitable for IP multicast.	228.6.7.8	No
jgroups.udp.mcast_port	Port to use for multicast socket	46655	No
jgroups.udp.ip_ttl	Specifies the time-to-live (TTL) for IP multicast packets. The value here refers to the number of network hops a packet is allowed to make before it is dropped	2	No

jgroups-tcp.xml

System Property	Description	Default	Required?
jgroups.tcp.address	IP address to use for the TCP transport.	127.0.0.1	No
jgroups.tcp.port	Port to use for TCP socket	7800	No
jgroups.udp.mcast_addr	IP address to use for multicast (for discovery). Must be a valid Class D IP address, suitable for IP multicast.	228.6.7.8	No
jgroups.udp.mcast_port	Port to use for multicast socket	46655	No
jgroups.udp.ip_ttl	Specifies the time-to-live (TTL) for IP multicast packets. The value here refers to the number of network hops a packet is allowed to make before it is dropped	2	No

jgroups-ec2.xml

System Property	Description	Default	Required?
jgroups.tcp.address	IP address to use for the TCP transport.	127.0.0.1	No
jgroups.tcp.port	Port to use for TCP socket	7800	No
jgroups.s3.access_key	The Amazon S3 access key used to access an S3 bucket		No
jgroups.s3.secret_access_key	The Amazon S3 secret key used to access an S3 bucket		No
jgroups.s3.bucket	Name of the Amazon S3 bucket to use. Must be unique and must already exist		No

18.3 Further reading

JGroups also supports more system property overrides, details of which can be found on this page: SystemProps

In addition, the JGroups configuration files shipped with Infinispan are intended as a jumping off point to getting something up and running, and working. More often than not though, you will want to fine-tune your JGroups stack further to extract every ounce of performance from your network equipment. For this, your next stop should be the JGroups manual which has a detailed section on configuring each of the protocols you see in a JGroups configuration file.

19 Locking and Concurrency

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

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

The rest of this wiki page is broken down into the following sections:

19.1 MVCC implementation details

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

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

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

19.2 Isolation levels

Infinispan offers two isolation levels - READ_COMMITTED (the default) and REPEATABLE_READ, configurable via the <locking /> configuration element. These isolation levels determine when readers see a concurrent write, and are implemented using different subclasses of MVCCEntry, which have different behaviour in how state is committed back to the data container.

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

1. Thread1: tx.begin()
2. Thread1: cache.get(k) returns v
3. Thread2: tx.begin()
4. Thread2: cache.get(k) returns v
5. Thread2: cache.put(k, v2)
6. Thread2: tx.commit()
7. Thread1: cache.get(k) returns v2!

With REPEATABLE_READ, step 7 will still return v. So, if you're gonna retrieve the same key multiple times within a transaction, you should use REPEATABLE_READ.

19.3 The LockManager

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

19.4 Lock striping

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

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

Default lock stripping settings

Since Infinispan 5.0, lock striping is disabled by default, due to potential deadlocks that can happen if locks for different keys end up in the same lock stripe. Previously, in Infinispan 4.x lock striping used to be enabled by default.

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

19.5 Concurrency levels

In addition to determining the size of the striped lock container, this concurreny level is also used to tune any JDK ConcurrentHashMap based collections where related, such as internal to DataContainers. Please refer to the JDK ConcurrentHashMap Javadocs for a detailed discussion of concurrency levels, as this parameter is used in exactly the same way in Infinispan.

19.6 Explicit and implicit distributed eager locking

Infinispan, by default, acquires remote locks lazily. Locks are acquired locally on a node that runs a transaction while other cluster nodes attempt to lock cache keys involved in a transaction during two-phase prepare/commit phase. However, if desired, Infinispan can eagerly lock cache keys either explicitly or implicitly.

Infinispan cache interface exposes lock API that allows cache users to explicitly lock set of cache keys eagerly during a transaction. Lock call attempts to lock specified cache keys across all cluster nodes and it either succeeds or fails. All locks are released during commit or rollback phase.

Consider a transaction running on one of the cache nodes:

```
tx.begin()
cache.lock(K)    // acquire cluster wide lock on K
cache.put(K,V5)  // guaranteed to succeed
tx.commit()      // releases locks
```

Implicit locking goes one step ahead and locks cache keys behind the scene as keys are accessed for modification operations.
Consider a transaction running on one of the cache nodes:

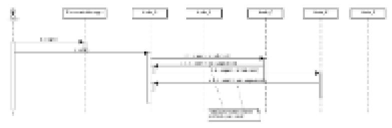
```
tx.begin()
cache.put(K,V)    // acquire cluster wide lock on K
cache.put(K2,V2)  // acquire cluster wide lock on K2
cache.put(K,V5)   // no-op, we already own cluster wide lock for K
tx.commit()       // releases locks
```

Implicit eager locking locks cache keys across cluster nodes only if it is necessary to do so. In a nutshell, if implicit eager locking is turned on then for each modification Infinispan checks if cache key is locked locally. If it is then a global cluster wide lock has already been obtained, otherwise a cluster wide lock request is sent and lock is acquired.
Implicit eager locking is enabled as follows:

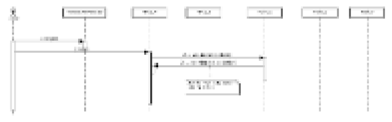
```
<transaction useEagerLocking="true" />
```

19.7 Locking a single remote node

Starting with 4.2, Infinispan allows eagerLockSingleNode configuration option. This only applies for DIST modes. Having this enabled would make the number of remote locks acquired to be always 1, disregarding the configured numOwners. Following diagrams are intended to explain better this option. All diagrams represent an cluster having 5 nodes, with numOwners=2.



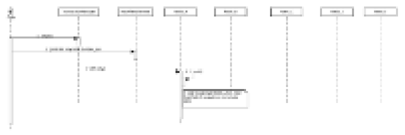
Above diagram shows the situation where eagerLockSingleNode=false (default configuration). On each lock request, numOwners remote calls are performed (in our example 2).



Above diagram shows how lock on the same key are acquired when eagerLockSingleNode=true. The number of remote calls being performed is always 1, disregarding numOwners values (it can actually be 0, as we'll see later on).

In this scenario, if the lock owner fails (Node_C) then the transaction that holds the lock, which originated on Node_A is marked for rollback.

Combining eagerLockSingleNode with the KeyAffinityService can bring some interesting advantages. The next diagram shows this:



By using KeyAffinityService one can generate keys that would always map to the local node. If eagerLockSingleNode=true, then the remote lock acquisition happens locally: this way one can benefit from eager locking semantics and having the same performance as non eager locking. The optimisation is affected by cluster topology changes, so keys might get relocated. But for clusters where topology changes are rather rare this can bring a lot of value.

The following xml snippet shows how can be configured:

```
      <transaction

 transactionManagerLookupClass="org.infinispan.transaction.lookup.GenericTransactionManagerLookup"
 syncRollbackPhase="false"
             syncCommitPhase="false"
             useEagerLocking="true" eagerLockSingleNode="true"/>
```

```
      <transaction

 transactionManagerLookupClass="org.infinispan.transaction.lookup.GenericTransactionManagerLookup"
 syncRollbackPhase="false"
             syncCommitPhase="false"
             useEagerLocking="true" eagerLockSingleNode="true"/>
```

Note that the configuration is ignored if eager locking is disabled or cache mode is not DIST.

19.7.1 Consistency

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

19.8 transactional caches and concurrent updates

This configuration refers to non-transactional distributed and local caches only (doesn't apply to replicated caches) and was added in Infinispan 5.2. Depending on whether one needs to support concurrent updates (e.g. two threads concurrently writing the same key), the following configuration option can be used:

```
<locking supportsConcurrentUpdates="true"/>
```

When enabled (default == true), the *supportConcurrentUpdates* adds internal support for concurrent writes: a locking interceptor that would serialize writes to the same key and a delegation layer, that designates a lock owner and uses it in order to coordinate the writes to a key.

More specific, when a thread running on node A writes on key *k* that mapps according to the consistent hash to nodes {B, C}

(given *numOwners*==2):

- A forwards (RPC) the write to the primary owner. The primary owner is the first node in the list of owners, in our example B
- B acquires a lock on *k.* Once the lock successfully acquired,_ _it forwards (RPC) the request to the rest of owners (in this example C) that apply it locally
- B applies the result locally, releases the lock and then it returns to A

Reasoning about the performance: in order to assure consistency under concurrent update, we do 2 RPCs: from operation originator to main owner and from main owner to the rest of the owners. That's one more than when *supportConcurrentUpdates == false*: in this case the operation originator does a single (multicast) RPC to all the owners. This induces a performance cost and whenever one uses the cache in non-concurrent manner, it is recommended that this configuration to be set to false in order to increase the performance. When using Infinispan in client/server mode with a Hot Rod client, this would use the main data owner in order to write data, so in this scenario there should not be any performance cost when supporting concurrent updates.

20 Configuring Cache declaratively

One of the major goals of Infinispan is to aim for zero configuration. A simple XML configuration file containing nothing more than a single infinispan element is enough to get you started. The configuration file listed below provides sensible defaults and is perfectly valid.

```
<infinispan />
```

However, that would only give you the most basic, local mode, non-clustered cache. Non basic configurations are very likely to use customized global and default cache elements.
Declarative configuration is the most common approach to configuring Infinispan cache instances. In order to read XML configuration files one would typically construct an instance of CacheManager by pointing to an XML file containing Infinispan configuration. Once configuration file is read you can obtain reference to the default cache instance.

```
CacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

or any other named instance specified in "my-config-file.xml".

```
Cache someNamedCache = manager.getCache("someNamedCache");
```

Every time you define <namedCache> element in XML configuration file, in addition to <default> cache element, you are effectively configuring additional caches whose settings are inheriting and/or overriding the default cache.
Refer to Infinispan configuration reference for more details. If you are using XML editing tools for configuration writing you can use provided Infinispan schema to assist you.

21 Configuration Migration Tools

Infinispan has a number of scripts for importing configurations from other cache products. Currently we have scripts to import configurations from:

- JBoss Cache 3.x
- EHCache 1.x
- Coherence 3.x

JBoss Cache 3.x itself supports configuration migration from previous (2.x) versions, so JBoss Cache 2.x configurations can be migrated indirectly.

```
If you wish to help write conversion tools for other caching systems, please contact <a
href="https://lists.jboss.org/mailman/listinfo/infinispan-dev">infinispan-dev</a>.
```

There is a single scripts for importing configurations: ${INFINISPAN_HOME}/bin/importConfig.sh and an equivalent .BAT script for Windows. Just run it and you should get a help message to assist you with the import:

```
C:\infinispan\bin> importConfig.bat
Missing 'source', cannot proceed
Usage:
importConfig [-source <the file to be transformed>]
[-destination <where to store resulting XML>]
[-type <the type of the source, possible values being: [JBossCache3x, Ehcache1x, Coherence35x]
>]

C:\infinispan\bin>
```

Here is how a JBoss Cache 3.x configuration file is imported:

```
C:\infinispan\bin>importConfig.bat -source in\jbosscache_all.xml -destination out.xml -type
JBossCache3x

WARNING! Preload elements cannot be automatically transformed, please do it manually!

WARNING! Please configure cache loader props manually!

WARNING! Singleton store was changed and needs to be configured manually!

IMPORTANT: Please take a look at the generated file
for (possible) TODOs about the elements that couldn't be converted automatically!

---

New configuration file [out.xml] successfully created.

---

C:\infinispan\bin>
```

Please read all warning messages *carefully* and inspect the generated XML for potential TODO statements that indicate the need for manual intervention. In the case of JBoss Cache 3.x this would usually have to do with custom extensions, such as custom CacheLoaders that cannot be automatically migrated.

For EHCache and Coherence these may also contain suggestions and warnings for configuration options that may not have direct equivalents in Infinispan.

22 Consistent Concurrent Updates With Hot Rod Versioned Operations

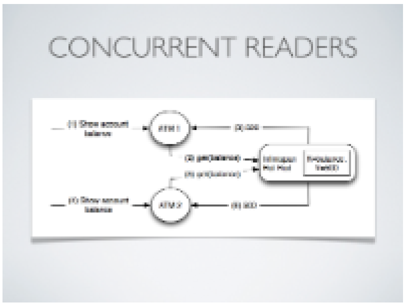
22.1 Introduction

Data structures, such as Infinispan Cache, that are accessed and modified concurrently can suffer from data consistency issues unless there're mechanisms to guarantee data correctness. Infinispan Cache, since it implements ConcurrentMap, provides operations such as conditional replace, putIfAbsent, and conditional remove to its clients in order to guarantee data correctness. It even allows clients to operate against cache instances within JTA transactions, hence providing the necessary data consistency guarantees.

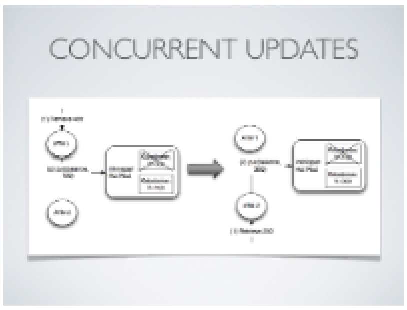
However, when it comes to Hot Rod protocol backed servers, clients do not yet have the ability to start remote transactions but they can call instead versioned operations to mimic the conditional methods provided by the embedded Infinispan cache instance API. Let's look at a real example to understand how it works.

22.2 Data Consistency Problem

Imagine you have two ATMs that connect using Hot Rod to a bank where an account's balance is stored. Two closely followed operations to retrieve the latest balance could return 500 CHF (swiss francs) as shown below:



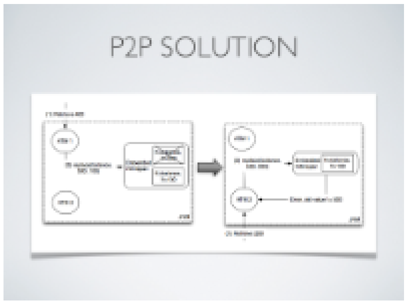
Next a customer connects to the first ATM and requests 400 CHF to be retrieved. Based on the last value read, the ATM could calculate what the new balance is, which is 100 CHF, and request a put with this new value. Let's imagine now that around the same time another customer connects to the ATM and requests 200 CHF to be retrieved. Let's assume that the ATM thinks it has the latest balance and based on its calculations it sets the new balance to 300 CHF:



Obviously, this would be wrong. Two concurrent updates have resulted in an incorrect account balance. The second update should not have been allowed since the balance the second ATM had was incorrect. Even if the ATM would have retrieved the balance before calculating the new balance, someone could have updated between the new balance being retrieved and the update. Before finding out how to solve this issue in a client-server scenario with Hot Rod, let's look at how this is solved when Infinispan clients run in peer-to-peer mode where clients and Infinispan live within the same JVM.

22.2.1 Peer-to-Peer Solution

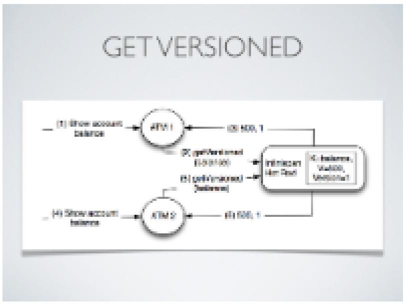
If the ATM and the Infinispan instance storing the bank account lived in the same JVM, the ATM could use the conditional replace API referred at the beginning of this article. So, it could send the previous known value to verify whether it has changed since it was last read. By doing so, the first operation could double check that the balance is still 500 CHF when it was to update to 100 CHF. Now, when the second operation comes, the current balance would not be 500 CHF any more and hence the conditional replace call would fail, hence avoiding data consistency issues:



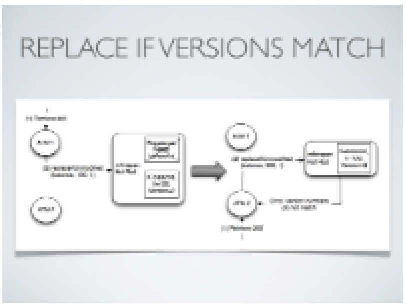
22.2.2 Client-Server Solution

In theory, Hot Rod could use the same p2p solution but sending the previous value would be not practical. In this example, the previous value is just an integer but the value could be a lot bigger and hence forcing clients to send it to the server would be rather wasteful. Instead, Hot Rod offers versioned operations to deal with this situation.

Basically, together with each key/value pair, Hot Rod stores a version number which uniquely identifies each modification. So, using an operation called getVersioned or getWithVersion, clients can retrieve not only the value associated with a key, but also the current version. So, if we look at the previous example once again, the ATMs could call getVersioned and get the balance's version:



When the ATMs wanted to modify the balance, instead of just calling put, they could call replaceIfUnmodified operation passing the latest version number of which the clients are aware of. The operation will only succeed if the version passed matches the version in the server. So, the first modification by the ATM would be allowed since the client passes 1 as version and the server side version for the balance is also 1. On the other hand, the second ATM would not be able to make the modification because after the first ATMs modification the version would have been incremented to 2, and now the passed version (1) and the server side version (2) would not match:



Finally, to find out how to call these operations from a Java environment, checkout the Java Hot Rod Client article.

23 Cache Loaders and Stores

23.1 Introduction

Cache loader is Infinispan's connection to a (persistent) data store. Cache loader fetches data from a store when that data is not in the cache, and when modifications are made to data in the cache the CacheLoader is called to store those modifications back to the store. Cache loaders are associated with individual caches, i.e. different caches from the same cache manager might have different cache store configurations.

23.2 Configuration

Cache loaders can be configured in a chain. Cache read operations 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.

```
<loaders passivation="false" shared="false" preload="true">
   <!-- We can have multiple cache loaders, which get chained -->
   <fileStore
           fetchPersistentState="true"
           purgerThreads="3"
           purgeSynchronously="true"
           ignoreModifications="false"
           purgeOnStartup="false"
           location="${java.io.tmpdir}" />
      <async enabled="true" flushLockTimeout="15000" threadPoolSize="5"/>
      <singletonStore enabled="true" pushStateWhenCoordinator="true" pushStateTimeout="20000"/>
   </fileStore>
</loaders>
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.loaders()
    .passivation(false).shared(false).preload(true)
    .addFileCacheStore()
        .fetchPersistentState(true)
        .purgerThreads(3)
        .purgeSynchronously(true)
        .ignoreModifications(false)
        .purgeOnStartup(false)
        .location(System.getProperty("java.io.tmpdir"))
        .async()
           .enabled(true)
           .flushLockTimeout(15000)
           .threadPoolSize(5)
        .singletonStore()
          .enabled(true)
          .pushStateWhenCoordinator(true)
          .pushStateTimeout(20000);
```

- passivation (false by default) has a significant impact on how Infinispan interacts with the loaders, and is discussed in the next paragraph.
- shared (false by default) indicates that the cache loader is shared among different cache instances, for example where all instances in a cluster use the same JDBC settings to 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.
- preload (false by default) if true, when the cache starts, data stored in the cache loader will be pre-loaded into memory. This is particularly useful when data in the cache loader is needed immediately after startup and you want to avoid cache operations being delayed as a result of loading this data lazily. Can be used to provide a 'warm-cache' on startup, however there is a performance penalty as startup time is affected by this process. Note that preloading is done in a local fashion, so any data loaded is only stored locally in the node. No replication or distribution of the preloaded data happens. Also, Infinispan only preloads up to the maximum configured number of entries in eviction.
- class attribute (mandatory) defines the class of the cache loader implementation.
- fetchPersistentState (false by default) determines whether or not to fetch the persistent state of a cache when joining a cluster. The aim here is to take the persistent state of a cache and apply it to the local cache store of the joining node. Hence, if cache store is configured to be shared, since caches access the same cache store, fetch persistent state is ignored. 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.
- purgeSynchronously will control whether the expiration takes place in the eviction thread, i.e. if purgeSynchronously (false by default) is set to true, the eviction thread will block until the purging is finished, otherwise would return immediately. If the cache loader supports multi-threaded purge then purgeThreads (1 by default) are used for purging expired entries. There are cache loaders that support multi-threaded purge (e.g. FileCacheStore) and caches that don't (e.g. JDBCCacheStore); check the actual cache loader configuration in order to see that.

- ignoreModifications(false by default) 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 JDBC based cache loader used by all servers in the network. This feature allows you to write to the 'local' file cache loader but not the shared JDBCCacheLoader.
- purgeOnStatup empties the specified cache loader (if ignoreModifications is false) when the cache loader starts up.
- additional attributes configure aspects specific to each cache loader, e.g. the location attribute in the previous example refers to where the FileCacheStore will keep the files that contain data. Other loaders, with more complex configuration, also introduce additional sub-elements to the basic configuration. See for example the JDBC cache store configuration examples below
- singletonStore (default for enabled is false) 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(or distributed) 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 attribute to true in all nodes, but again only the coordinator of the cluster will the modifications in the underlying cache loader as defined in loader element. You cannot define a shared and with singletonStore enabled at the same time.
- pushStateWhenCoordinator (true by default) If true, when a node becomes the coordinator, it will transfer in-memory state to the underlying cache loader. This can be very useful in situations where the coordinator crashes and the new coordinator is elected.
- async element has to do with cache store persisting data (a)synchronously to the actual store. It is discussed in detail here.

23.3 Cache Passivation

A cache loader can be used to enforce entry 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 an entry from the cache, if passivation is enabled, a notification that the entry is being passivated will be emitted to the cache listeners and the entry will be stored. When a user attempts to retrieve a entry that was evicted earlier, the entry is (lazily) loaded from the cache loader into memory. When the entry and its children have been loaded, they're removed from the cache loader and a notification is emitted to the cache listeners that the entry has been activated. In order to enable passivation just set passivation to true (false by default). When passivation is used, only the first cache loader configured is used and all others are ignored.

23.3.1 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 entries 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 keyOne
2. Insert keyTwo
3. Eviction thread runs, evicts keyOne
4. Read keyOne
5. Eviction thread runs, evicts keyTwo
6. Remove keyTwo

When passivation is **disabled** :

1. Memory: keyOne Disk: keyOne
2. Memory: keyOne, keyTwo Disk: keyOne, keyTwo
3. Memory: keyTwo Disk: keyOne, keyTwo
4. Memory: keyOne, keyTwo Disk: keyOne, keyTwo
5. Memory: keyOne Disk: keyOne, keyTwo
6. Memory: keyOne Disk: keyOne

When passivation is **enabled** :

1. Memory: keyOne Disk:
2. Memory: keyOne, keyTwo Disk:
3. Memory: keyTwo Disk: keyOne
4. Memory: keyOne, keyTwo Disk:
5. Memory: keyOne Disk: keyTwo
6. Memory: keyOne Disk:

23.4 File system based cache loaders

Infinispan ships with several cache loaders that utilize the file system as a data store. They all require a location attribute, which maps to a directory to be used as a persistent store. (e.g., location="/tmp/myDataStore").

- FileCacheStore, which is a simple filesystem-based implementation. 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. 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. Please visit the file cache store configuration documentation for more information on the configurable parameters of this store.
- BdbjeCacheStore, which is a cache loader implementation based on the Oracle/Sleepycat's BerkeleyDB Java Edition.
- JdbmCacheStore, which is a cache loader implementation based on the JDBM engine, 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 (seehttp://www.oracle.com/database/berkeley-db/index.html for details).

For detailed description of all the parameters supported by the stores, please consult the javadoc.

23.5 JDBC based cache loaders

Based on the type of keys to be persisted, there are three JDBC cache loaders:

- JdbcBinaryCacheStore - can store any type of keys. It stores all the keys that have the same hash value (hashCode method on key) in the same table row/blob, having as primary key the hash value. While this offers great flexibility (can store any key type), it impacts concurrency/throughput. E.g. If storing k1,k2 and k3 keys, and keys had same hash code, then they'd persisted in the same table row. Now, if 3 threads try to concurrently update k1, k2 and k3 respectively, they would need to do it sequentially since these threads would be updating the same row.
- JdbcStringBasedCacheStore - stores each key in its own row, increasing throughput under concurrent load. In order to store each key in its own column, it relies on a (pluggable) bijection that maps the each key to a String object. The bijection is defined by the Key2StringMapper interface. Infinispans ships a default implementation (smartly named DefaultKey2StringMapper) that knows how to handle primitive types.
- JdbcMixedCacheStore - it is a hybrid implementation that, based on the key type, delegates to either JdbcBinaryCacheStore or JdbcStringBasedCacheStore.

23.5.1 Which JDBC cache loader should I use?

It is generally preferable to use JdbcStringBasedCacheStore when you are in control of the key types, as it offers better throughput under heavy load. One scenario in which it is not possible to use it though, is when you can't write an Key2StringMapper to map the keys to to string objects (e.g. when you don't have control over the types of the keys, for whatever reason). Then you should use either JdbcBinaryCacheStore or JdbcMixedCacheStore. The later is preferred to the former when the majority of the keys are handled by JdbcStringBasedCacheStore, but you still have some keys you cannot convert through Key2StringMapper.

23.5.2 Connection management (pooling)

In order to obtain a connection to the database all the JDBC cache loaders rely on an ConnectionFactory implementation. The connection factory can be specified through the *connectionFactoryClass* configuration attribute. Infinispan ships with three ConnectionFactoy implementations:

- PooledConnectionFactory is a factory based on C3P0. Refer to javadoc for details on configuring it.
- ManagedConnectionFactory is a connection factory that can be used within managed environments, such as application servers. It knows how to look into the JNDI tree at a certain location (configurable) and delegate connection management to the DataSource. Refer to javadoc javadoc for details on how this can be configured.
- SimpleConnectionFactory is a factory implementation that will create database connection on a per invocation basis. Not recommended in production.

The PooledConnectionFactory is generally recommended for stand-alone deployments (i.e. not running within AS or Servlet container). ManagedConnectionFactory can be used when running in a managed environment where a DataSource is present, so that connection pooling is performed within the DataSource.

23.5.3 Sample configurations

Bellow is an sample configuration for the JdbcBinaryCacheStore. For detailed description of all the parameters used refer to the javadoc. Please note the use of multiple XML schemas, since each cachestore has its own schema.

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

<infinispan
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:infinispan:config:5.2
http://www.infinispan.org/schemas/infinispan-config-5.2.xsd
                       urn:infinispan:config:jdbc:5.2
http://www.infinispan.org/schemas/infinispan-cachestore-jdbc-config-5.2.xsd"
   xmlns="urn:infinispan:config:5.2"
   xmlns:jdbc="urn:infinispan:config:jdbc:5.2" >

 :

<loaders>
   <binaryKeyedJdbcStore xmlns="urn:infinispan:config:jdbc:5.2"
fetchPersistentState="false"ignoreModifications="false" purgeOnStartup="false">
       <connectionPool connectionUrl="jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1"
username="sa" driverClass="org.h2.Driver"/>
       <binaryKeyedTable dropOnExit="true" createOnStart="true" prefix="ISPN_BUCKET_TABLE">
         <idColumn name="ID_COLUMN" type="VARCHAR(255)" />
         <dataColumn name="DATA_COLUMN" type="BINARY" />
         <timestampColumn name="TIMESTAMP_COLUMN" type="BIGINT" />
       </binaryKeyedTable>
   </binaryKeyedJdbcStore>
</loaders>

 :

</infinispan>
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
  builder.loaders().addLoader(JdbcBinaryCacheStoreConfigurationBuilder.class)
     .fetchPersistentState(false).ignoreModifications(false).purgeOnStartup(false)
     .table()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_BUCKET_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
        .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
     .connectionPool()
        .connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
        .username("sa")
        .driverClass("org.h2.Driver");
```

Bellow is an sample configuration for the JdbcStringBasedCacheStore. For detailed description of all the parameters used refer to the javadoc.

```
<loaders>
   <stringKeyedJdbcStore xmlns="urn:infinispan:config:jdbc:5.2" fetchPersistentState="false"
ignoreModifications="false" purgeOnStartup="false">
       <connectionPool connectionUrl="jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1"
username="sa" driverClass="org.h2.Driver"/>
       <stringKeyedTable dropOnExit="true" createOnStart="true" prefix="ISPN_STRING_TABLE">
         <idColumn name="ID_COLUMN" type="VARCHAR(255)" />
         <dataColumn name="DATA_COLUMN" type="BINARY" />
         <timestampColumn name="TIMESTAMP_COLUMN" type="BIGINT" />
       </stringKeyedTable>
   </stringKeyedJdbcStore>
</loaders>
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
  builder.loaders().addLoader(JdbcStringBasedCacheStoreConfigurationBuilder.class)
     .fetchPersistentState(false).ignoreModifications(false).purgeOnStartup(false)
     .table()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_STRING_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
        .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
     .connectionPool()
        .connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
        .username("sa")
        .driverClass("org.h2.Driver");
```

Bellow is an sample configuration for the JdbcMixedCacheStore. For detailed description of all the parameters used refer to the javadoc.

```
<loaders>
   <mixedKeyedJdbcStore xmlns="urn:infinispan:config:jdbc:5.2" fetchPersistentState="false"
ignoreModifications="false" purgeOnStartup="false">
      <connectionPool connectionUrl="jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1"
username="sa" driverClass="org.h2.Driver" />
      <stringKeyedTable dropOnExit="true" createOnStart="true" prefix="ISPN_MIXED_STR_TABLE">
         <idColumn name="ID_COLUMN" type="VARCHAR(255)" />
         <dataColumn name="DATA_COLUMN" type="BINARY" />
         <timestampColumn name="TIMESTAMP_COLUMN" type="BIGINT" />
      </stringKeyedTable>
      <binaryKeyedTable dropOnExit="true" createOnStart="true" prefix="ISPN_MIXED_BINARY_TABLE">
         <idColumn name="ID_COLUMN" type="VARCHAR(255)" />
         <dataColumn name="DATA_COLUMN" type="BINARY" />
         <timestampColumn name="TIMESTAMP_COLUMN" type="BIGINT" />
      </binaryKeyedTable>
   </loader>
</loaders>
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
  builder.loaders().addLoader(JdbcMixedCacheStoreConfigurationBuilder.class)
     .fetchPersistentState(false).ignoreModifications(false).purgeOnStartup(false)
     .stringTable()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_MIXED_STR_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
        .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
     .binaryTable()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_MIXED_BINARY_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
        .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
     .connectionPool()
        .connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
        .username("sa")
        .driverClass("org.h2.Driver");
```

Finally, below is an example of a JDBC cache store with a managed connection factory, which is chosen implicitly by specifying a datasource JNDI location:

```
<stringKeyedJdbcStore xmlns="urn:infinispan:config:jdbc:5.2" fetchPersistentState="false"
ignoreModifications="false" purgeOnStartup="false">
   <dataSource jndiUrl="java:/StringStoreWithManagedConnectionTest/DS" />
   <stringKeyedTable dropOnExit="true" createOnStart="true" prefix="ISPN_STRING_TABLE">
       <idColumn name="ID_COLUMN" type="VARCHAR(255)" />
       <dataColumn name="DATA_COLUMN" type="BINARY" />
       <timestampColumn name="TIMESTAMP_COLUMN" type="BIGINT" />
   </stringKeyedTable>
</stringKeyedJdbcStore>
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
    builder.loaders().addLoader(JdbcStringBasedCacheStoreConfigurationBuilder.class)
     .fetchPersistentState(false).ignoreModifications(false).purgeOnStartup(false)
     .table()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_STRING_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
        .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
     .dataSource()
        .jndiUrl("java:/StringStoreWithManagedConnectionTest/DS");
```

Apache Derby users

If you're connecting to an Apache Derby database, make sure you set `dataColumnType` to `BLOB`:

```
<dataColumn name="DATA_COLUMN" type="BLOB"/>
```

23.6 Cloud cache loader

The CloudCacheStore implementation utilizes JClouds to communicate with cloud storage providers such as Amazon's S3, Rackspace's Cloudfiles or any other such provider supported by JClouds. If you're planning to use Amazon S3 for storage, consider using it with Infinispan. Infinispan itself provides in-memory caching for your data to minimize the amount of remote access calls, thus reducing the latency and cost of fetching your Amazon S3 data. With cache replication, you are also able to load data from your local cluster without having to remotely access it every time. Note that Amazon S3 does not support transactions. If transactions are used in your application then there is some possibility of state inconsistency when using this cache loader. However, writes are atomic, in that if a write fails nothing is considered written and data is never corrupted. For a list of configuration refer to the javadoc.

23.7 Remote cache loader

The RemoteCacheStore is a cache loader implementation that stores data in a remote infinispan cluster. In order to communicate with the remote cluster, the RemoteCacheStore uses the HotRod client/server architecture. HotRod bering the load balancing and fault tolerance of calls and the possibility to fine-tune the connection between the RemoteCacheStore and the actual cluster. Please refer to HotRod for more information on the protocol, client and server configuration. For a list of RemoteCacheStore configuration refer to the javadoc. Example:

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

<infinispan
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:infinispan:config:5.2
http://www.infinispan.org/schemas/infinispan-config-5.2.xsd
                       urn:infinispan:config:remote:5.2
http://www.infinispan.org/schemas/infinispan-cachestore-remote-config-5.2.xsd"
   xmlns="urn:infinispan:config:5.2"
   xmlns:remote="urn:infinispan:config:remote:5.2" >

 :
<loaders>
   <remoteStore xmlns="urn:infinispan:config:remote:5.2" fetchPersistentState="false"
             ignoreModifications="false" purgeOnStartup="false" remoteCache="mycache">
      <servers>
         <server host="one" port="12111"/>
         <server host="two" />
      </servers>
      <connectionPool maxActive="10" exhaustedAction="CREATE_NEW" />
      <async enabled="true" />
   </remoteStore>
</loaders>

 :

</infinispan>
```

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.loaders().addStore(RemoteCacheStoreConfigurationBuilder.class)
     .fetchPersistentState(false)
     .ignoreModifications(false)
     .purgeOnStartup(false)
     .remoteCacheName("mycache")
     .addServer()
        .host("one").port(12111)
     .addServer()
        .host("two")
     .connectionPool()
        .maxActive(10)
        .exhaustedAction(ExhaustedAction.CREATE_NEW)
     .async().enable();
```

In this sample configuration, the remote cache store is configured to use the remote cache named "mycache" on servers "one" and "two". It also configures connection pooling and provides a custom transport executor. Additionally the cache store is asynchronous.

23.8 Cassandra cache loader

The CassandraCacheStore was introduced in Infinispan 4.2. Read the specific page for details on implementation and configuration.

23.9 Cluster cache loader

The ClusterCacheLoader is a cache loader implementation that retrieves data from other cluster members.

It is a cache loader only as it doesn't persist anything (it is not a Store), therefore features like *fetchPersistentState* (and like) are not applicable.

A cluster cache loader can be used as a non-blocking (partial) alternative to *stateTransfer* : keys not already available in the local node are fetched on-demand from other nodes in the cluster. This is a kind of lazy-loading of the cache content.

```
<loaders>
   <clusterLoader remoteCallTimeout="500" />
</loaders>
```

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.loaders().addClusterCacheLoader()
    .remoteCallTimeout(500);
```

For a list of ClusterCacheLoader configuration refer to the javadoc.

Note: The ClusterCacheLoader does not support preloading(preload=true). It also won't provide state if fetchPersistentSate=true.

23.10 Cache Loaders and transactional caches

When a cache is transactional and a cache loader is present, the cache loader won't be enlisted in the transaction in which the cache is part. That means that it is possible to have inconsistencies at cache loader level: the transaction to succeed applying the in-memory state but (partially) fail applying the changes to the store. Manual recovery would not work with caches stores.

24 Portable Serialization For Hot Rod With Apache Avro

24.1 Introduction

Starting with Infinispan 5.0, Hot Rod clients can be configured with a marshaller that produces plattform independent payloads using Apache Avro. This means that payloads generated by a Java, Avro-based, marshaller could be read by a Python, Avro-based, marshaller. When Hot Rod clients in other languages such as Python or Ruby become available, this will mean that for example, data stored via Java Hot Rod client will be readable by a Python Hot Rod client.

24.2 Languages Supported

Avro currently supports providing portable serialization payloads for the following languages:

- C
- C++
- Java
- Python
- Ruby

So interoperability of payloads is limited to these languages. The choice of Avro over other existing portable serialization libraries (i.e. Google Protocol Buffers, Apache Thrift, MessagePack...etc) was done based languages supported, ease of use, and payload size.

24.3 Object Types Supported

Avro based marshaller currently supports basic type and collection marshalling:

- Basic types include:
    - UTF-8 string
    - Integer
    - Long
    - Float
    - Double
    - Boolean
    - Byte Array
- Collections composed of the basic types :
    - Arrays
    - Lists
    - Maps
    - Sets

Marshalling of complex objects is not supported because it's not fully solvable as some language don't support some concepts other languages offer. Instead, it is recommended that for any complex object marshalling requirements, users serialize these complex objects into byte arrays using portable serialization libraries such as Google Protocol Buffers, Apache Thrift, MessagePack, or Apache Avro. Once the byte array has been created, simply pass it to the Hot Rod client API which will handle it accordingly.

Therefore, it's clear that users are free to use any portable serialization library to transform their complex objects into byte arrays, regardless of the fact that Infinispan uses Apache Avro for basic type and collection marshalling. The only limitation here is that the target language used to serialize complex objects needs to be amongst the languages that Apache Avro supports.

Short and Byte Java primitive types are not supported per se. Instead, clients should pass integers which will be encoded efficiently using variable-length zig zag coding. Primitive arrays not supported except byte arrays. Instead, use their object counter partners, i.e. Integer...etc.

24.4 Java Configuration

To configure a Java Hot Rod client with Avro marshaller, simply pass a `infinispan.client.hotrod.marshaller` property to the `RemoteCacheManager` constructor containing the value of `org.infinispan.client.hotrod.marshall.ApacheAvroMarshaller`. For example:

```
Properties props = new Properties();
props.put("infinispan.client.hotrod.marshaller",
   "org.infinispan.client.hotrod.marshall.ApacheAvroMarshaller");
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(props);
```

Note however that if all you're gonna be using are Java Hot Rod clients, there's no need to configure Avro based marshaller. The default marshaller is capable of transforming any Serializable or Externalizable java object into a byte array in a quick and efficient manner.

Please remember as well that when remote java clients are configured with the apache avro marshaller, the server's marshaller does not need changing. This is because the server does not make any attempt at unmarshalling the data that's been passed to it. Internally, it keeps the data as byte arrays.

25 The Grouping API

In some cases you may wish to co-locate a group of entries onto a particular node. In this, the group API will be useful for you.

25.1 How does it work?

Infinispan allocates each node a portion of the total hash space. Normally, when you store an entry, Infinispan will take a hash of the key, and store the entry on the node which owns that portion of the hash space. Infinispan always uses an algorithm to locate a key in the hash space, never allowing the node on which the entry is stored to be specified manually. This scheme allows any node to know which nodes owns a key, without having to distribute such ownership information. This reduces the overhead of Infinispan, but more importantly improves redundancy as there is no need to replicate the ownership information in case of node failure.

If you use the grouping API , then Infinispan will ignore the hash of the key when deciding which **node** to store the entry on, and instead use a hash of the group. Infinispan still uses the hash of the key to store the entry on a node. When the group API is in use, it is important that every node can still compute, using an algorithm, the owner of every key. For this reason, the group cannot be specified manually. The group can either be intrinsic to the entry (generated by the key class) or extrinsic (generated by an external function).

25.2 How do I use the grouping API?

First, you must enable groups. If you are configuring Infinispan programmatically, then call:

```
Configuration c = new ConfigurationBuilder().clustering().hash().groups().enabled().build();
```

Or, if you are using XML:

```
<clustering>
  <hash>
     <groups enabled="true" />
  </hash>
</clustering>
```

If you have control of the key class (you can alter the class definition, it's not part of an unmodifiable library), and the determination of the group is not an orthogonal concern to the key class, then we recommend you use an intrinsic group. The intrinsic group can be specified using the @Group annotation placed on the method. Let's take a look at an example:

```
class User {

   ...
   String office;
   ...

   int hashCode() {
      // Defines the hash for the key, normally used to determine location
      ...
   }

   // Override the location by specifying a group, all keys in the same
   // group end up with the same owner
   @Group
   String getOffice() {
      return office;
   }

}
```

⚠ The group must be a `String`.

If you don't have control over the key class, or the determination of the group is an orthogonal concern to the key class, we recommend you use an extrinsic group. In extrinsic group is specified by implementing the Grouper interface, which has a single method computeGroup, which should return the group. Grouper acts as an interceptor, passing the previously computed value in. The group passed to the first Grouper will be that determined by @Group (if @Groupis defined). This allows you even greater control over the group when using an intrinsic group. For a grouper to be used when determining the group for a key, it's keyType must be assignable from the key being grouped.

Let's take a look at an example of a Grouper:

```
public class KXGrouper implements Grouper<String> {

   // A pattern that can extract from a "kX" (e.g. k1, k2) style key
   // The pattern requires a String key, of length 2, where the first character is
   // "k" and the second character is a digit. We take that digit, and perform
   // modular arithmetic on it to assign it to group "1" or group "2".
   private static Pattern kPattern = Pattern.compile("(^k)(<a>\\d</a>)$");

    public String computeGroup(String key, String group) {
        Matcher matcher = kPattern.matcher(key);
        if (matcher.matches()) {
            String g = Integer.parseInt(matcher.group(2)) % 2 + "";
            return g;
        } else
            return null;
    }

    public Class<String> getKeyType() {
        return String.class;
    }

}
```

Here we determine a simple grouper that can take the key class and extract from the group from the key using a pattern. We ignore any group information specified on the key class.

You must register every grouper you wish to have used. If you are configuring Infinispan programmatically:

```
Configuration c = new ConfigurationBuilder().clustering().hash().groups().addGrouper(new
KXGrouper()).build();
```

Or, if you are using XML:

```
<clustering>
  <hash>
     <groups enabled="true">
        <grouper class="com.acme.KXGrouper" />
     </groups>
  </hash>
</clustering>
```

26 Infinispan transactions

26.1 JTA Support

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

On every cache operation Infinispan does the following:

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

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

Infinispan ships with several transaction manager lookup classes:

- DummyTransactionManagerLookup : This provides with a dummy transaction manager which should only be used for testing. Being a dummy, this is not recommended for production use a it has some severe limitations to do with concurrent transactions and recovery.
- JBossStandaloneJTAManagerLookup : If you're running Infinispan in a standalone environment, this should be your default choice for transaction manager. It's a fully fledged transaction manager based on JBoss Transactions which overcomes all the deficiencies of the dummy transaction manager.
- GenericTransactionManagerLookup : This is a lookup class that locate transaction managers in the most popular Java EE application servers. If no transaction manager can be found, it defaults on the dummy transaction manager.
- JBossTransactionManagerLookup : This lookup class locates the transaction manager running within a JBoss Application Server instance.

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

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

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

26.2 Configuring transactions

Transactions are being configured at cache level; bellow is a sample configuration:

```
<transaction

transactionManagerLookupClass="org.infinispan.transaction.lookup.GenericTransactionManagerLookup"
transactionMode="TRANSACTIONAL"
      lockingMode="OPTIMISTIC"/>
```

- transactionManagerLookupClass fully qualified class name of a class that looks up a reference to a javax.transaction.TransactionManager
- transactionMode - configures whether the cache is transactional or not
- lockingMode - configures whether the cache uses optimistic or pessimistic locking.

For more details on how two phase commit (2PC) is implemented in Infinispan and how locks are being acquired see the section below. All possible transactional settings are available in Configuration reference

26.3 Transactional modes

Starting with Infinispan 5.1 release a cache can accessed either transactionally or non-transactionally. The mixed access mode is no longer supported. There are several reasons for going this path, but one of them most important is a cleaner semantic on how concurrency is managed between multiple requestors for the same cache entry.

By default, all Infinispan caches are non-transactional. A cache can be made transactional by changing the transactionMode attribute:

```
<namedCache name="transactional">
  <transaction transactionMode="TRANSACTIONAL"/>
</namedCache>
```

transactionMode can only take two values: TRANSACTIONAL and NON_TRANSACTIONAL; one can configure a transactional cache programatically as well:

```
Configuration c = new
ConfigurationBuilder().transaction().transactionMode(TransactionMode.TRANSACTIONAL).build();
assert c.transaction().transactionalCache();
```

⚠ Do not forget to configure a TransactionManagerLookup for transactional caches.

Supported transaction models are optimistic and pessimistic. Optimistic model is an improvement over the old transaction model as it completely defers lock acquisition to transaction prepare time. New approach reduces lock acquisition duration and increases throughput which in turn avoids deadlocks significantly. In pessimistic model, cluster wide-locks are acquired on each write operation only being released after the transaction completed.

26.3.1 Optimistic Transactions

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

Optimistic transactions can be enabled in the configuration file:

```
<namedCache name="transactional">
  <transaction transactionMode="TRANSACTIONAL" lockingMode="OPTIMISTIC"/>
</namedCache>
```

or programatically:

```
Configuration c = new
ConfigurationBuilder().transaction().lockingMode(LockingMode.OPTIMISTIC).build();
assert c.transaction().lockingMode() == LockingMode.OPTIMISTIC;
```

⚠ By default, a transactional cache is optimistic.

26.3.2 Pessimistic Transactions

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

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

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

Pessimistic transactions can be enabled in the configuration file:

```
<namedCache name="transactional"/>
  <transaction transactionMode="TRANSACTIONAL" lockingMode="PESSIMISTIC"/>
</namedCache>
```

or programatically:

```
Configuration c = new
ConfigurationBuilder().transaction().lockingMode(LockingMode.PESSIMISTIC).build();
assert c.transaction().lockingMode() == LockingMode.PESSIMISTIC;
```

26.3.3 Backward compatibility

The autoCommit attribute was added in order to assure backward compatibility. If a cache is transactional and autoCommit is enabled (defaults to true) then any call that is performed outside of a transaction's scope is transparently wrapped within a transaction. In other words Infinispan adds the logic for starting a transaction before the call and committing it after the call.

Therefore if your code accesses a cache both transactionally and non-transactionally all you have to do when migrating to Infinispan 5.1 is mark the cache as transactional and enable autoCommit (that's actually enabled by default)

The autoCommit feature can be managed through configuration:

```
<namedCache name="transactional">;
  <transaction transactionMode="TRANSACTIONAL" autoCommit="true"/>
</namedCache>
```

26.3.4 What do I need - pessimistic or optimistic transactions?

From a use case perspective, optimistic transactions should be used when there is **not** a lot of contention between multiple transactions running at the same time. That is because the optimistic transactions rollback if data has changed between the time it was read and the time it was committed (writeSkewCheck).

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

26.4 Deadlock detection

Deadlocks can significantly (up to one order of magnitude, see benchmarks) reduce the throughput of a system, especially when multiple transactions are operating agains the same key set. Deadlock detection is disabled by default, but can be enabled/configured per cache (i.e. under namedCache config element) by adding the following:

```
<deadlockDetection enabled="true" spinDuration="1000"/>
```

Some clues on when to enable deadlock detection. A high number of transaction rolling back due to TimeoutException is an indicator that this functionality might help. TimeoutException might be caused by other causes as well, but deadlocks will always result in this exception being thrown. Generally, when you have a high contention on a set of keys, deadlock detection may help. But the best way is not to guess the performance improvement but to benchmark and monitor it: you can have access to statistics (e.g. number of deadlocks detected) through JMX, as it is exposed via the DeadlockDetectingLockManager MBean. For more details on how deadlock detection works, benchmarks and design details refer to this article.
Note: deadlock detection only runs on an a per cache basis: deadlocks that spread over two or more caches won't be detected.

26.5 Transaction and exceptions

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

26.6 Transaction recovery on node failures

Transaction recovery is discussed in this document.

26.7 Enlisting Synchronization

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

Starting with 5.0 release, Infinispan allows transaction enlistment through Synchronisation. This can be enabled through the *useSynchronization* attribute on the *transaction* element:

```
<transaction useSynchronization="true"/>
```

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

27 Load Testing Infinispan Server Modules

Infinispan comes with different server implementations that allow Infinispan to be accessed remotely. You can find an overview about these servers here.

This wiki explains how to load test and benchmark these server implementations using Grinder, which is a Java load testing framework that makes it easy to run a distributed test using many load injector machines.

1. First of all, download Grinder version 3.4 or higher and unzip it somewhere locally.
2. Next, download and unzip the Infinispan version that you want to be testing. Make sure you download the `-all.zip` distribution to run these tests. Note that although Infinispan REST server was included in 4.0, Memcached and Hot Rod servers are only included from 4.1 onwards.
3. Download Infinispan Grinder scripts and other material from this source repository.
4. Next up, modify `bin/setGrinderEnv.sh` file to set correct environment paths. Primarily, you should be looking at setting the following properties correctly (The rest of properties not mentioned below can be left as they are) :
    1. `INFINISPAN_HOME`: It should point to the location where Infinispan was unzipped.
    2. `GRINDER_HOME`: It should point to where Grinder was unzipped.
    3. `GRINDER_PROPERTIES`: It should point to the grinder.properties provided with the Infinispan Grinder scripts
    4. `SPYMEMCACHED_HOME`: It should only be modified to point to Spymemcached Memcached client library if you're planning to load test Infinispan Memcached server.
5. It is time now to start the Infinispan server that you want to be testing. For detailed information on starting up a Hot Rod server, check this wiki. If you want to start the Hot Rod server quickly, the following command should do the job:
    1. `$INFINISPAN_HOME/bin/startServer.sh -r hotrod -l 0.0.0.0`
6. Finally, let's start the load test. Currently, a single load test exists which after a warmup phase, it does a certain amount of put/get calls on a pool of keys with a configurable distribution. By default, it executes 20% put operations and 80% get operations. So, if you're going to start the Hot Rod test, you'd simply do:
    1. `cd infinispan-server-grinder/`
    2. `./bin/startHotRodAgent.sh`

7. Once the test has finished, the output can be located in `infinispan-server-grinder/log/out-<host>-0.log` and will show something like this at the bottom:

```
         Tests     Errors    Mean Test    Test Time    TPS
                             Time (ms)    Standard
                                          Deviation
                                          (ms)

Test 1   25        0         13742.56     457.22       0.29       "Warmup (1000 ops)"
Test 2   400136    0         1.92         2.55         4687.08    "Read (20000 ops per thread)"
Test 3   99864     0         2.41         4.97         1169.78    "Put (20000 ops per thread)"

Totals   500025    0         2.70         97.26        1952.38
```

The output is saying that it first run 25 threads, each sending 1000 operations in order to warm up the cache. Afterwards, each of these 25 threads send 20.000 operations making a total of 500.000 operations, out of which 400136 were get calls and 99864 were put calls, and the test here indicates what was the mean test time for each of these operations, including standard deviation and how many operations per second (TPS) were executed. In this case, 4686 get operations per second and 1169 put operations per second.

28 Using Infinispan as JPA-Hibernate Second Level Cache Provider

28.1 Overview

Following some of the principles set out by Brian Stansberry in Using JBoss Cache 3 as a Hibernate 3.5 Second Level Cache and taking in account improvements introduced by Infinispan, an Infinispan JPA/Hibernate second level cache provider has been developed. This wiki explains how to configure JPA/Hibernate to use the Infinispan and for those keen on lower level details, the key design decisions made and differences with previous JBoss Cache based cache providers.

If you're unfamiliar with JPA/Hibernate Second Level Caching, I'd suggest you to read Chapter 2.1 in this guide which explains the different types of data that can be cached.

On Caching

Query result caching, or entity caching, **may or may not improve** application performance. Be sure to benchmark your application with and without caching.

28.2 Configuration

1. First of all, to enable JPA/Hibernate second level cache with query result caching enabled, add either of the following:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.cache.use_second_level_cache" value="true" />
<property name="hibernate.cache.use_query_cache" value="true" />
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
```

2. Now, configure the Infinispan cache region factory using one of the two options below:

• If the Infinispan CacheManager instance happens to be bound to JNDI select JndiInfinispanRegionFactory as the cacheregion factory and add the cache manager's JNDI name:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.infinispan.JndiInfinispanRegionFactory" />
<property name="hibernate.cache.infinispan.cachemanager" value="java:CacheManager" />
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property
name="hibernate.cache.region.factory_class">org.hibernate.cache.infinispan.JndiInfinispanRegionFac
name="hibernate.cache.infinispan.cachemanager">java:CacheManager/entity</property>
```

JBoss Application Server

JBoss Application Server 6 and 7 deploy a shared Infinispan cache manager that can be used by all services, so when trying to configure applications with Infinispan second level cache, you should use the JNDI name for the cache manager responsible for the second level cache. By default, this is "java:CacheManager/entity". In any other application server, you can deploy your own cache manager and bind the CacheManager to JNDI, but in this cases it's generally preferred that the following method is used.

• If running JPA/Hibernate and Infinispan standalone or within third party Application Server, select the *InfinispanRegionFactory* as the cache region factory:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.infinispan.InfinispanRegionFactory"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property
name="hibernate.cache.region.factory_class">org.hibernate.cache.infinispan.InfinispanRegionFactory
```

This is all the configuration you need to have JPA/Hibernate use Infinispan as cache provider with the default settings. You will still need to define which entities and queries need to be cached as defined in the Hibernate reference documentation, but that configuration aspect is not peculiar to Infinispan.
This default configuration should suit the majority of use cases but sometimes, further configuration is required and to help with such situations, please check the following section where more advanced settings are discussed.

28.3 Default Configuration Explained

The aim of this section is to explain the default settings for each of the different global data type (entity, collection, query and timestamps) caches, why these were chosen and what are the available alternatives.

28.3.1 Defaults for Entity/Collection Caching

- By default, for all entities and collections, whenever an new **entity or collection is read from database** and needs to be cached, **it's only cached locally** in order to reduce intra-cluster traffic. This option cannot be changed.
- By default, all **entities and collections are configured to use a synchronous invalidation** as clustering mode. This means that when an entity is updated, the updated cache will send a message to the other members of the cluster telling them that the entity has been modified. Upon receipt of this message, the other nodes will remove this data from their local cache, if it was stored there. This option can be changed to use replication by configuring entities or collections to use "replicated-entity" cache but it's generally not a recommended choice.
- By default, all **entities and collections have initial state transfer disabled** since there's no need for it. It's not recommended that this is enabled.
- By default, **entities and collections are configured to use READ_COMMITTED** as cache isolation level. It would only make sure to configure REPEATABLE_READ if the application evicts/clears entities from the Hibernate Session and then expects to repeatably re-read them in the same transaction. Otherwise, the Session's internal cache provides repeatable-read semantics. If you really need to use REPEATABLE_READ, you can simply configure entities or collections to use "entity-repeatable" cache.
- By default, entities and collections are configured with the following eviction settings:
    - Eviction wake up interval is 5 seconds.
    - Max number of entries are 10.000
    - Max idle time before expiration is 100 seconds

You can change these settings on a per entity or collection basis or per individual entity or collection type.

More information in the "Advanced Configuration" section below.

- By default **entites and collections are configured with lazy deserialization** which helps deserialization when entities or collections are stored in isolated deployments. If you're sure you'll never deploy your entities or collections in classloader isolated deployment archives, you can disable this setting.

28.3.2 Defaults for Query Caching

- By default, query cache is configured so that **queries are only cached locally**. Alternatively, you can configure query caching to use replication by selecting the "replicated-query" as query cache name. However, replication for query cache only makes sense if, and only if, all of this conditions are true:
  - Performing the query is quite expensive.
  - The same query is very likely to be repeatedly executed on different cluster nodes.
  - The query is unlikely to be invalidated out of the cache (Note: Hibernate must aggressively invalidate query results from the cache any time any instance of one of the entity types targeted by the query. All such query results are invalidated, even if the change made to the specific entity instance would not have affected the query result)
- By default, **query cache** uses the **same cache isolation levels and eviction/expiration settings as for entities/collections**.
- By default, **query cache has initial state transfer disabled**. It is not recommended that this is enabled.

28.3.3 Defaults for Timestamps Cache

- By default, the ***timestamps* cache is configured with asynchronous replication** as clustering mode. Local or invalidated cluster modes are not allowed, since all cluster nodes must store all timestamps. As a result, **no eviction/expiration is allowed for timestamp caches either**.
- By default, the ***timestamps* cache is configured with a cluster cache loader (in Hibernate 3.6.0 or earlier it had state transfer enabled)** so that joining nodes can retrieve all timestamps. You shouldn't attempt to disable the cluster cache loader for the timestamps cache.

28.4 JTA Transactions Configuration

It is highly recommended that Hibernate is configured with JTA transactions so that both Hibernate and Infinispan cooperate within the same transaction and the interaction works as expected.

Otherwise, if Hibernate is configured for example with JDBC transactions, Hibernate will create a Transaction instance via java.sql.Connection and Infinispan will create a transaction via whatever TransactionManager returned by hibernate.transaction.manager_lookup_class. If hibernate.transaction.manager_lookup_class has not been populated, it will default on the dummy transaction manager. So, any work on the 2nd level cache will be done under a different transaction to the one used to commit the stuff to the database via Hibernate. In other words, your operations on the database and the 2LC are not treated as a single unit. Risks here include failures to update the 2LC leaving it with stale data while the database committed data correctly. It has also been observed that under some circumstances where JTA was not used, commit/rollbacks are not propagated to Infinispan.

To sum up, if you configure Hibernate with Infinispan, apply the following changes to your configuration file:

1. Unless your application uses JPA, you need to select the correct Hibernate transaction factory via the property *hibernate.transaction.factory_class*:

- If you're running within an application server, it's recommended that you use:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.transaction.factory_class"
value="org.hibernate.transaction.CMTTransactionFactory"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property
name="hibernate.transaction.factory_class">org.hibernate.transaction.CMTTransactionFactory</
```

- If you're running in a standalone environment and you wanna enable JTA transaction factory, use:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.transaction.factory_class"
value="org.hibernate.transaction.JTATransactionFactory"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property
name="hibernate.transaction.factory_class">org.hibernate.transaction.JTATransactionFactory</
```

The reason why JPA does not require a transaction factory class to be set up is because the entity manager already sets it to a variant of CMTTransactionFactory.

2. Select the correct Hibernate transaction manager lookup:

- If you're running within an application server, select the appropriate lookup class according to "JTA Transaction Managers" table.

For Example if you were running with the JBoss Application Server you would set:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.transaction.manager_lookup_class"
   value="org.hibernate.transaction.JBossTransactionManagerLookup"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property name="hibernate.transaction.manager_lookup_class">
   org.hibernate.transaction.JBossTransactionManagerLookup
</property>
```

- If you are running standalone and you want to add a JTA transaction manager lookup, things get a bit more complicated. Due to a current limitation, Hibernate does not support injecting a JTA TransactionManager or JTA UserTransaction that are not bound to JNDI. In other words, if you want to use JTA, Hibernate expects your TransactionManager to be bound to JNDI and it also expects that UserTransaction instances are retrieved from JNDI. This means that in an standalone environment, you need to add some code that binds your TransactionManager and UserTransaction to JNDI. With this in mind and with the help of one of our community contributors, we've created an example that does just that: JBoss Standalone JTA Example. Once you have the code in place, it's just a matter of selecting the correct Hibernate transaction manager lookup class, based on the JNDI names given. If you take *JBossStandaloneJtaExample* as an example, you simply have to add:

  ```
  <!-- If using JPA, add to your persistence.xml -->
  <property name="hibernate.transaction.manager_lookup_class"
     value="org.hibernate.transaction.JBossTransactionManagerLookup"/>
  ```

  ```
  <!-- If using Hibernate, add to your hibernate.cfg.xml -->
  <property name="hibernate.transaction.manager_lookup_class">
     org.hibernate.transaction.JBossTransactionManagerLookup
  </property>
  ```

  As you probably have noted through this section, there wasn't a single mention of the need to configure Infinispan's transaction manager lookup and there's a good reason for that. Basically, the code within Infinispan cache provider takes the transaction manager that has been configured at the Hibernate level and uses that. Otherwise, if no Hibernate transaction manager lookup class has been defined, Infinispan uses a default dummy transaction manager.

Since Hibernate 4.0, the way Infinispan hooks into the transaction manager can be configured. By default, since 4.0, Infinispan interacts with the transaction manager as an JTA synchronization, resulting in a faster interaction with the 2LC thanks to some key optimisations that the transaction manager can apply. However if desired, users can configure Infinispan to act as an XA resource (just like it did in 3.6 and earlier) by disabling the use of the synchronization. For example:

```
<!-- If using JPA, add to your persistence.xml: -->
<property name="hibernate.cache.infinispan.use_synchronization"  value="false"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml: -->
<property name="hibernate.cache.infinispan.use_synchronization">
   false
</property>
```

28.4.1 Standalone JTA for JPA/Hibernate using Infinispan as 2LC

The JBoss standalone JTA example referred to in the previous section is inspired by the great work of Guenther Demetz, one of the members of the Infinispan community, who wrote an impressive wiki explaining how to set up standalone JTA with different transaction managers running outside of an EE server, and how to get this to work with an Infinispan backed JPA/Hibernate application.

28.5 Advanced Configuration

Infinispan has the capability of exposing statistics via JMX and since Hibernate 3.5.0.Beta4, you can enable such statistics from the Hibernate/JPA configuration file. By default, Infinispan statistics are turned off but when these are disabled via the following method, statistics for the Infinispan Cache Manager and all the managed caches (entity, collection,...etc) are enabled:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.cache.infinispan.statistics" value="true"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml: -->
<property name="hibernate.cache.infinispan.statistics">true</property>
```

The Infinispan cache provider jar file contains an Infinispan configuration file, which is the one used by default when configuring the Infinispan standalone cache region factory. This file contains default cache configurations for all Hibernate data types that should suit the majority of use cases. However, if you want to use a different configuration file, you can configure it via the following property:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.cache.infinispan.cfg"
   value="/home/infinispan/cacheprovider-configs.xml"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml: -->
<property name="hibernate.cache.infinispan.cfg">
   /home/infinispan/cacheprovider-configs.xml
</property>
```

For each Hibernate cache data types, Infinispan cache region factory has defined a default cache name to look up in either the default, or the user defined, Infinispan cache configuration file. These default values can be found in the Infinispan cache provider javadoc. You can change these cache names using the following properties:

```
<!-- If using JPA, add to your persistence.xml: -->
<property name="hibernate.cache.infinispan.entity.cfg"
   value="custom-entity"/>
<property name="hibernate.cache.infinispan.collection.cfg"
   value="custom-collection"/>
<property name="hibernate.cache.infinispan.query.cfg"
   value="custom-collection"/>
<property name="hibernate.cache.infinispan.timestamp.cfg"
   value="custom-timestamp"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property name="hibernate.cache.infinispan.entity.cfg">
   custom-entity
</property>
<property name="hibernate.cache.infinispan.collection.cfg">
   custom-collection
</property>
<property name="hibernate.cache.infinispan.query.cfg">
   custom-collection
</property>
<property name="hibernate.cache.infinispan.timestamp.cfg">
   custom-timestamp
</property>
```

One of the key improvements brought in by Infinispan is the fact that cache instances are more lightweight than they used to be in JBoss Cache. This has enabled a radical change in the way entity/collection type cache management works. With the Infinispan cache provider, each entity/collection type gets each own cache instance, whereas in old JBoss Cache based cache providers, all entity/collection types would be sharing the same cache instance. As a result of this, locking issues related to updating different entity/collection types concurrently are avoided completely.

This also has an important bearing on the meaning of hibernate.cache.infinispan.entity.cfg and hibernate.cache.infinispan.collection.cfg properties. These properties define the template cache name that should be used for all entity/collection data types. So, with the above hibernate.cache.infinispan.entity.cfg configuration, when a region needs to be created for entity com.acme.Person, the cache instance to be assigned to this entity will be based on a "custom-entity" named cache.

On top of that, this finer grained cache definition enables users to define cache settings on a per entity/collection basis. For example:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.cache.infinispan.com.acme.Person.cfg"
   value="person-entity"/>
<property name="hibernate.cache.infinispan.com.acme.Person.addresses.cfg"
   value="addresses-collection"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property name="hibernate.cache.infinispan.com.acme.Person.cfg">
   person-entity
</property>
<property name="hibernate.cache.infinispan.com.acme.Person.addresses.cfg">
   addresses-collection
</property>
```

Here, we're configuring the Infinispan cache provider so that for com.acme.Person entity type, the cache instance assigned will be based on a "person-entity" named cache, and for com.acme.Person.addresses collection type, the cache instance assigned will be based on a "addresses-collection" named cache. If either of these two named caches did not exist in the Infinispan cache configuration file, the cache provider would create a cache instance for com.acme.Person entity and com.acme.Person.addresses collection based on the default cache in the configuration file.

Furthermore, thanks to the excellent feedback from the Infinispan community and in particular, Brian Stansberry, we've decided to allow users to define the most commonly tweaked Infinispan cache parameters via hibernate.cfg.xml or persistence.xml, for example eviction/expiration settings. So, with the Infinispan cache provider, you can configure eviction/expiration this way:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.cache.infinispan.entity.eviction.strategy"
   value= "LRU"/>
<property name="hibernate.cache.infinispan.entity.eviction.wake_up_interval"
   value= "2000"/>
<property name="hibernate.cache.infinispan.entity.eviction.max_entries"
   value= "5000"/>
<property name="hibernate.cache.infinispan.entity.expiration.lifespan"
   value= "60000"/>
<property name="hibernate.cache.infinispan.entity.expiration.max_idle"
   value= "30000"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property name="hibernate.cache.infinispan.entity.eviction.strategy">
   LRU
</property>
<property name="hibernate.cache.infinispan.entity.eviction.wake_up_interval">
   2000
</property>
<property name="hibernate.cache.infinispan.entity.eviction.max_entries">
   5000
</property>
<property name="hibernate.cache.infinispan.entity.expiration.lifespan">
   60000
</property>
<property name="hibernate.cache.infinispan.entity.expiration.max_idle">
   30000
</property>
```

With the above configuration, you're overriding whatever eviction/expiration settings were defined for the default entity cache name in the Infinispan cache configuration used, regardless of whether it's the default one or user defined. More specifically, we're defining the following:

- All entities to use LRU eviction strategy
- The eviction thread to wake up every 2000 milliseconds
- The maximum number of entities for each entity type to be 5000 entries
- The lifespan of each entity instance to be 600000 milliseconds
- The maximum idle time for each entity instance to be 30000

You can also override eviction/expiration settings on a per entity/collection type basis in such way that the overriden settings only afftect that particular entity (i.e. com.acme.Person) or collection type (i.e. com.acme.Person.addresses). For example:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.cache.infinispan.com.acme.Person.eviction.strategy"
   value= "FIFO"/>
<property name="hibernate.cache.infinispan.com.acme.Person.eviction.wake_up_interval"
   value= "2500"/>
<property name="hibernate.cache.infinispan.com.acme.Person.eviction.max_entries"
   value= "5500"/>
<property name="hibernate.cache.infinispan.com.acme.Person.expiration.lifespan"
   value= "65000"/>
<property name="hibernate.cache.infinispan.com.acme.Person.expiration.max_idle"
   value= "35000"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property name="hibernate.cache.infinispan.com.acme.Person.eviction.strategy">
   FIFO
</property>
<property name="hibernate.cache.infinispan.com.acme.Person.eviction.wake_up_interval">
   2500
</property>
<property name="hibernate.cache.infinispan.com.acme.Person.eviction.max_entries">
   5500
</property>
<property name="hibernate.cache.infinispan.com.acme.Person.expiration.lifespan">
   65000
</property>
<property name="hibernate.cache.infinispan.com.acme.Person.expiration.max_idle">
   35000
</property>
```

The aim of these configuration capabilities is to reduce the number of files needed to modify in order to define the most commonly tweaked parameters. So, by enabling eviction/expiration configuration on a per generic Hibernate data type or particular entity/collection type via hibernate.cfg.xml or persistence.xml, users don't have to touch to Infinispan's cache configuration file any more. We believe users will like this approach and so, if you there are any other Infinispan parameters that you often tweak and these cannot be configured via hibernate.cfg.xml or persistence.xml, please let the Infinispan team know by sending an email to infinispan-dev@lists.jboss.org.

Please note that query/timestamp caches work the same way they did with JBoss Cache based cache providers. In other words, there's a query cache instance and timestamp cache instance shared by all. It's worth noting that eviction/expiration settings are allowed for query cache but not for timestamp cache. So configuring an eviction strategy other than NONE for timestamp cache would result in a failure to start up.

Finally, from Hibernate 3.5.4 and 3.6 onwards, queries with specific cache region names are stored under matching cache instances. This means that you can now set query cache region specific settings. For example, assuming you had a query like this:

```
Query query = session.createQuery(
  "select account.branch from Account as account where account.holder = ?");
query.setCacheable(true);
query.setCacheRegion("AccountRegion");
```

The query would be stored under "AccountRegion" cache instance and users could control settings in similar fashion to what was done with entities and collections. So, for example, you could define specific eviction settings for this particular query region doing something like this:

```
<!-- If using JPA, add to your persistence.xml -->
<property name="hibernate.cache.infinispan.AccountRegion.eviction.strategy"
   value= "FIFO"/>
<property name="hibernate.cache.infinispan.AccountRegion.eviction.wake_up_interval"
   value= "10000"/>
```

```
<!-- If using Hibernate, add to your hibernate.cfg.xml -->
<property name="hibernate.cache.infinispan.AccountRegion.eviction.strategy">
   FIFO
</property>
<property name="hibernate.cache.infinispan.AccountRegion.eviction.wake_up_interval">
   10000
</property>
```

28.6 Integration with JBoss Application Server

In JBoss Application Server 7, Infinispan is the default second level cache provider and you can find details about its configuration the AS7 JPA reference guide.

Infinispan based Hibernate 2LC was developed as part of Hibernate 3.5 release and so it currently only works within AS 6 or higher. Hibernate 3.5 is not designed to work with AS/EAP 5.x or lower. To be able to run Infinispan based Hibernate 2LC in a lower AS version such as 5.1, the Infinispan 2LC module would require porting to Hibernate 3.3.

Recently, William Decoste has helped migrate the Infinispan 2LC module to Hibernate 3.3, and in this wiki, he explains how to integrate Infinispan Hibernate 2LC with JBoss AS/EAP 5.x.

28.7 Using Infinispan as remote Second Level Cache?

Lately, several questions (here & here) have appeared in the Infinispan user forums asking whether it'd be possible to have an Infinispan second level cache that instead of living in the same JVM as the Hibernate code, it resides in a remote server, i.e. an Infinispan Hot Rod server. It's important to understand that trying to set up second level cache in this way is generally not a good idea for the following reasons:

- The purpose of a JPA/Hibernate second level cache is to store entities/collections recently retrieved from database or to maintain results of recent queries. So, part of the aim of the second level cache is to have data accessible locally rather than having to go to the database to retrieve it everytime this is needed. Hence, if you decide to set the second level cache to be remote as well, you're losing one of the key advantages of the second level cache: the fact that the cache is local to the code that requires it.
- Setting a remote second level cache can have a negative impact in the overall performance of your application because it means that cache misses require accessing a remote location to verify whether a particular entity/collection/query is cached. With a local second level cache however, these misses are resolved locally and so they are much faster to execute than with a remote second level cache.

There are however some edge cases where it might make sense to have a remote second level cache, for example:

- You are having memory issues in the JVM where JPA/Hibernate code and the second level cache is running. Off loading the second level cache to remote Hot Rod servers could be an interesting way to separate systems and allow you find the culprit of the memory issues more easily.
- Your application layer cannot be clustered but you still want to run multiple application layer nodes. In this case, you can't have multiple local second level cache instances running because they won't be able to invalidate each other for example when data in the second level cache is updated. In this case, having a remote second level cache could be a way out to make sure your second level cache is always in a consistent state, will all nodes in the application layer pointing to it.
- Rather than having the second level cache in a remote server, you want to simply keep the cache in a separate VM still within the same machine. In this case you would still have the additional overhead of talking across to another JVM, but it wouldn't have the latency of across a network. The benefit of doing this is that:
  - Size the cache separate from the application, since the cache and the application server have very different memory profiles. One has lots of short lived objects, and the other could have lots of long lived objects.
  - To pin the cache and the application server onto different CPU cores (using *numactl*), and even pin them to different physically memory based on the NUMA nodes.

28.8 Frequently Asked Questions

To find out more please go to the Hibernate 2nd level cache section in the Technical FAQ wiki.

29 Default Values For Property Based Attributes

The aim of this article is to complement the configuration reference documentation with information on default values that could not be automatically generated. Please find below the name of the XML elements and their corresponding property default values:

29.1 asyncListenerExecutor

- maxThreads = 1
- threadNamePrefix = "notification-thread"

29.2 asyncTransportExecutor

- maxThreads = 25
- threadNamePrefix = "transport-thread"

29.3 evictionScheduledExecutor

- maxThreads = 1 (cannot be changed)
- threadNamePrefix = "eviction-thread"

29.4 replicationQueueScheduledExecutor

- maxThreads = 1 (cannot be changed)
- threadNamePrefix = "replicationQueue-thread"

30 Running Infinispan on Amazon Web Services

Infinispan can be used on the Amazon Web Service (AWS) platform and similar cloud based environment in several ways. As Infinispan uses JGroups as the underlying communication technology, the majority of the configuration work is done JGroups. The default auto discovery won't work on EC2 as multicast is not allowed, but JGroups provides several other discovery protocols so we only have to choose one.

30.1 TCPPing, GossipRouter, S3_Ping

The TCPPing approach contains a static list of the IP address of each member of the cluster in the JGroups configuration file. While this works it doesn't really help when cluster nodes are dynamically added to the cluster. See http://community.jboss.org/wiki/JGroupsTCPPING for more information about TCPPing.
Sample TCPPing configuration

```
<config xmlns="urn:org:jgroups"
     xmlns:xsi="[http://www.w3.org/2001/XMLSchema-instance]"
     xsi:schemaLocation="urn:org:jgroups      file:schema/JGroups-2.8.xsd">
      <TCP bind_port="7800" />
      <TCPPING timeout="3000"

initial_hosts="$\{jgroups.tcpping.initial_hosts:localhost\[7800\],localhost\[7801\]\}"
           port_range="1"
           num_initial_members="3"/>
      <MERGE2 max_interval="30000"  min_interval="10000"/>
      <FD_SOCK/>
      <FD timeout="10000" max_tries="5" />
      <VERIFY_SUSPECT timeout="1500"  />
      <pbcast.NAKACK
           use_mcast_xmit="false" gc_lag="0"
           retransmit_timeout="300,600,1200,2400,4800"
           discard_delivered_msgs="true"/>
      <UNICAST timeout="300,600,1200" />
      <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"  max_bytes="400000"/>
      <pbcast.GMS print_local_addr="true" join_timeout="3000"   view_bundling="true"/>
      <FC max_credits="2000000"  min_threshold="0.10"/>
      <FRAG2 frag_size="60000"  />
      <pbcast.STREAMING_STATE_TRANSFER/>
</config>
```

30.2 GossipRouter

Another approach is to have a central server (Gossip, which each node will be configured to contact. This central server will tell each node in the cluster about each other node. More on Gossip Router @ http://www.jboss.org/community/wiki/JGroupsGossipRouter

The address (ip:port) that the Gossip router is listening on can be injected into the JGroups configuration used by Infinispan. To do this pass the gossip routers address as a system property to the JVM e.g. -DGossipRouterAddress="10.10.2.4[12001]" and reference this property in the JGroups configuration that Infinispan is using e.g.
Sample JGroups configuration for Gossip Router

```
<config>
   <TCP bind_port="7800" />
   <TCPGOSSIP timeout="3000" initial_hosts="${GossipRouterAddress}" num_initial_members="3" />
.
.
</config>
```

30.3 S3_Ping

Finally you can configure your JGroups instances to use a shared storage to exchange the details of the cluster nodes. S3_ping was added to JGroups in 2.6.12 and 2.8, and allows the Amazon S3 to be used as the shared storage. It is experimental at the moment but offers another method of clustering without a central server. Be sure that you have signed up for Amazon S3 as well as EC2 to use this method.
Sample S3_Ping configuration

```
<?xml version="1.0" encoding="UTF-8"?><config>     <TCP bind_port="7800" />     <S3_PING
secret_access_key="replace this with you secret access key" access_key="replace this with
your           access key" location="replace this with your S3 bucket location" />     <MERGE2
max_interval="30000" min_interval="10000" />     <FD_SOCK />     <FD timeout="10000"
max_tries="5" />     <VERIFY_SUSPECT timeout="1500" />     <pbcast.NAKACK use_mcast_xmit="false"
gc_lag="0" retransmit_timeout="300,600,1200,2400,4800"          discard_delivered_msgs="true"
/>     <UNICAST timeout="300,600,1200,2400,3600" />     <pbcast.STABLE stability_delay="1000"
desired_avg_gossip="50000" max_bytes="400000" />     <VIEW_SYNC avg_send_interval="60000" />
<pbcast.GMS print_local_addr="true" join_timeout="60000" view_bundling="true" />     <FC
max_credits="20000000" min_threshold="0.10" />     <FRAG2 frag_size="60000" />
<pbcast.STATE_TRANSFER />     <pbcast.FLUSH timeout="0" /></config>
```

30.4 JDBC_PING

A similar approach to S3_PING, but using a JDBC connection to a shared database. On EC2 that is quite easy using Amazon RDS. See the JDBC_PING Wiki page for details.

30.5 Creating a cluster node with distributed cache

Creating a cluster

```
GlobalConfiguration gc = GlobalConfiguration.getClusteredDefault();
gc.setClusterName("infinispan-test-cluster");
gc.setTransportClass(JGroupsTransport.class.getName());
//Load the jgroups properties
Properties p = newProperties();
p.setProperty("configurationFile","jgroups-config.xml");
gc.setTransportProperties(p);
Configuration c = new Configuration();
//Distributed cache mode
c.setCacheMode(Configuration.CacheMode.DIST_SYNC);
c.setExposeJmxStatistics(true);
// turn functionality which returns the previous value when setting
c.setUnsafeUnreliableReturnValues(true);
//data will be distributed over 3 nodes
c.setNumOwners(3);
c.setL1CacheEnabled(true);
//Allow batching
c.setInvocationBatchingEnabled(true);
c.setL1Lifespan(6000000);
cache_manager = new DefaultCacheManager(gc, c, false);
```

31 Eviction Examples

31.1 Introduction

1. **Expiration** is a top-level construct, represented in the configuration as well as in the cache API. More on this later.
2. While eviction is **local to each cache instance** , expiration is **cluster-wide** . Expiration lifespans and maxIdle values are replicated along with the cache entry.
3. Expiration `lifespan` and `maxIdle` are also persisted in CacheStores, so this information survives eviction/passivation.
4. Four eviction strategies are shipped, EvictionStrategy.NONE, EvictionStrategy.LRU, EvictionStrategy.UNORDERED, and EvictionStrategy.LIRS.

31.2 Configuration

Eviction may be configured using the Configuration bean or the XML file. Eviction configuration is on a per-cache basis. Valid eviction-related configuration elements are:

```
<eviction strategy="FIFO" wakeupInterval="1000" maxEntries="2000"/>
<expiration lifespan="1000" maxIdle="500" />
```

XML changes in Infinispan 5.0

From Infinispan 5.0 onwards, `wakeupInterval` attribute has been moved to `expiration` XML element. This is because since 4.1, eviction happens in the user thread, and so the old eviction thread now simply purges expired entries from memory and any attached cache store. So, effectively, `wakeUpInterval` controls how often this purging occurs:

```
<eviction strategy="FIFO" maxEntries="2000"/>
<expiration lifespan="1000" maxIdle="500" wakeupInterval="1000"/>
```

Programmatically, the same would be defined using:

```
Configuration c = new ConfigurationBuilder().eviction().strategy(EvictionStrategy.LRU)

.maxEntries(2000).expiration().wakeUpInterval(5000l).lifespan(1000l).maxIdle(1000l)
               .build();
```

31.2.1 Default values

Eviction is disabled by default. If enabled (using an empty <eviction /> element), certain default values are used:

- strategy: EvictionStrategy.NONE is assumed, if a strategy is not specified..
- wakeupInterval: 5000 is used if not specified.
    - If you wish to disable the eviction thread, set wakeupInterval to -1.
- maxEntries: -1 is used if not specified, which means unlimited entries.
    - 0 means no entries, and the eviction thread will strive to keep the cache empty.

Expiration lifespan and maxIdle both default to -1.

31.2.2 Using expiration

Expiration allows you to set either a lifespan or a maximum idle time on each key/value pair stored in the cache. This can either be set cache-wide using the configuration, as described above, or it can be defined per-key/value pair using the Cache interface. Any values defined per key/value pair overrides the cache-wide default for the specific entry in question.

For example, assume the following configuration:

```
<expiration lifespan="1000" />
```

```
// this entry will expire in 1000 millis
cache.put("pinot noir", pinotNoirPrice);

// this entry will expire in 2000 millis
cache.put("chardonnay", chardonnayPrice, 2, TimeUnit.SECONDS);

// this entry will expire 1000 millis after it is last accessed
cache.put("pinot grigio", pinotGrigioPrice, -1,
          TimeUnit.SECONDS, 1, TimeUnit.SECONDS);

// this entry will expire 1000 millis after it is last accessed, or
// in 5000 millis, which ever triggers first
cache.put("riesling", rieslingPrice, 5,
          TimeUnit.SECONDS, 1, TimeUnit.SECONDS);
```

31.3 Eviction designs

Central to eviction is an EvictionManager - which is only available if eviction or expiration is configured.

The purpose of the EvictionManager is to drive the eviction/expiration thread which periodically purges items from the DataContainer. If the eviction thread is disabled (wakeupInterval set to -1) eviction can be kicked off manually using EvictionManager.processEviction(), for example from another maintenance thread that may run periodically in your application.

The eviction manager processes evictions in the following manner:

1. Causes the data container to purge expired entries
2. Causes cache stores (if any) to purge expired entries
3. Prunes the data container to a specific size, determined by maxElements

32 Clustering modes

32.1 Introduction

Infinispan can be configured to be either local (standalone) or clustered. If in a cluster, the cache can be configured to replicate changes to all nodes, to invalidate changes across nodes and finally to be used in distributed mode - state changes are replicated to a small subset of nodes enough to be fault tolerant but not to many nodes to prevent scalability.

32.2 Local Mode

While Infinispan is particularly interesting in clustered mode, it also offers a very capable local mode, where it acts as a simple, in-memory data cache similar to JBoss Cache and EHCache. But why would one use a local cache rather than a map? Caches offer a lot of features over and above a simple map, including write-through and write-behind caching to persist data, eviction of entries to prevent running out of memory, and support for expirable entries. Infinispan, specifically, is built around a high-performance, read-biased data container which uses modern techniques like MVCC locking - which buys you non-blocking, thread-safe reads even when concurrent writes are taking place. Infinispan also makes heavy use of compare-and-swap and other lock-free algorithms, making it ideal for high-throughput, multi-CPU/multi-core environments. Further, Infinispan's Cache API extends the JDK's ConcurrentMap - making migration from a map to Infinispan trivial. For more details refer to Non-clustered, LOCAL mode section.

32.3 Replicated Mode

Replication is a simple clustered mode where cache instances automatically discover neighboring instances on other JVMs on the same local network, and form a cluster. Entries added to any of these cache instances will be replicated to all other cache instances in the cluster, and can be retrieved locally from any instance. This clustered mode provides a quick and easy way to share state across a cluster, however replication practically only performs well in small clusters (under 10 servers), due to the number of replication messages that need to happen - as the cluster size increases. Infinispan can be configured to use UDP multicast which mitigates this problem to some degree.

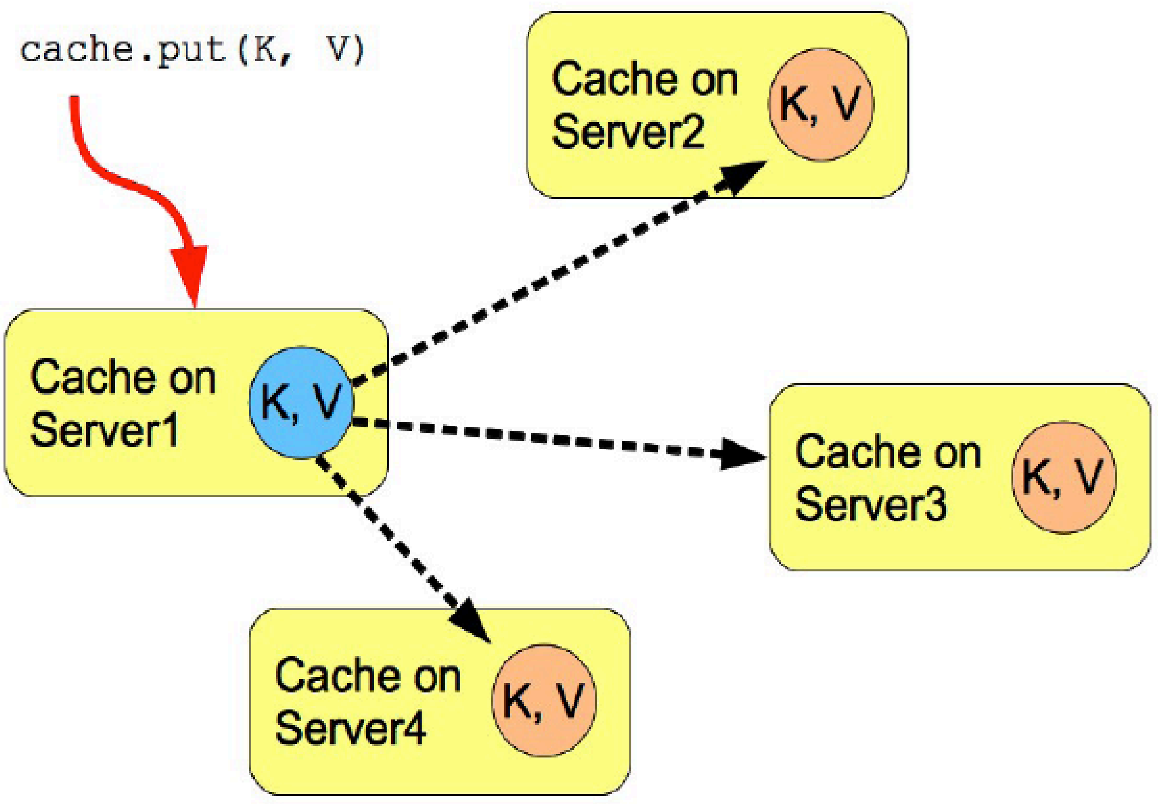


Figure 1. Replication mode

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). Infinispan 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. A replication queue can therefore offer much higher performance as the actual replication is performed by a background thread.

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.

32.4 Invalidation Mode

Invalidation is a clustered mode that does not actually share any data at all, but simply aims to remove data that may be stale from remote caches. This cache mode only makes sense if you have another, permanent store for your data such as a database and are only using Infinispan as an optimization in a read-heavy system, to prevent hitting the database every time you need some state. 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.

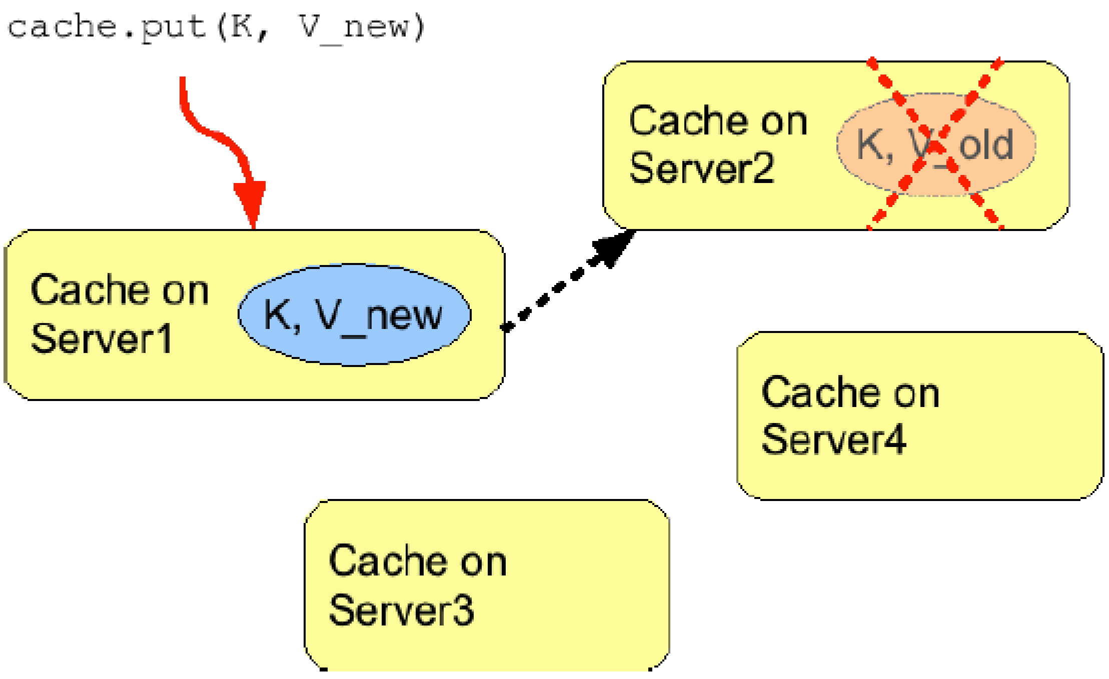


Figure 2. Invalidation mode

Invalidation, when used with a shared cache loader would cause remote caches to refer to the shared cache loader to retrieve modified data. The benefit of this is twofold: network traffic is minimized 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 batches), or at the end of a transaction or batch, upon successful commit. This is usually more efficient as invalidation messages can be optimized 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.

32.5 Distribution Mode

Distribution is a powerful clustering mode which allows Infinispan to scale linearly as more servers are added to the cluster. Distribution makes use of a consistent hash algorithm to determine where in a cluster entries should be stored. Hashing algorithm is configured with the number of copies each cache entry should be maintained cluster-wide. Number of copies represents the tradeoff between performance and durability of data. The more copies you maintain, the lower performance will be, but also the lower the risk of losing data due to server outages. Regardless of how many copies are maintained, distribution still scales linearly and this is key to Infinispan scalability. Another feature of the consistent hash algorithm is that it is deterministic in locating entries without resorting to multicasting requests or maintaining expensive metadata. Doing a PUT would result in at most num_copies remote calls, and doing a GET anywhere in the cluster would result in at most 1 remote call. In reality, num_copies remote calls are made even for a GET, but these are done in parallel and as soon as any one of these returns, the entry is passed back to the caller.

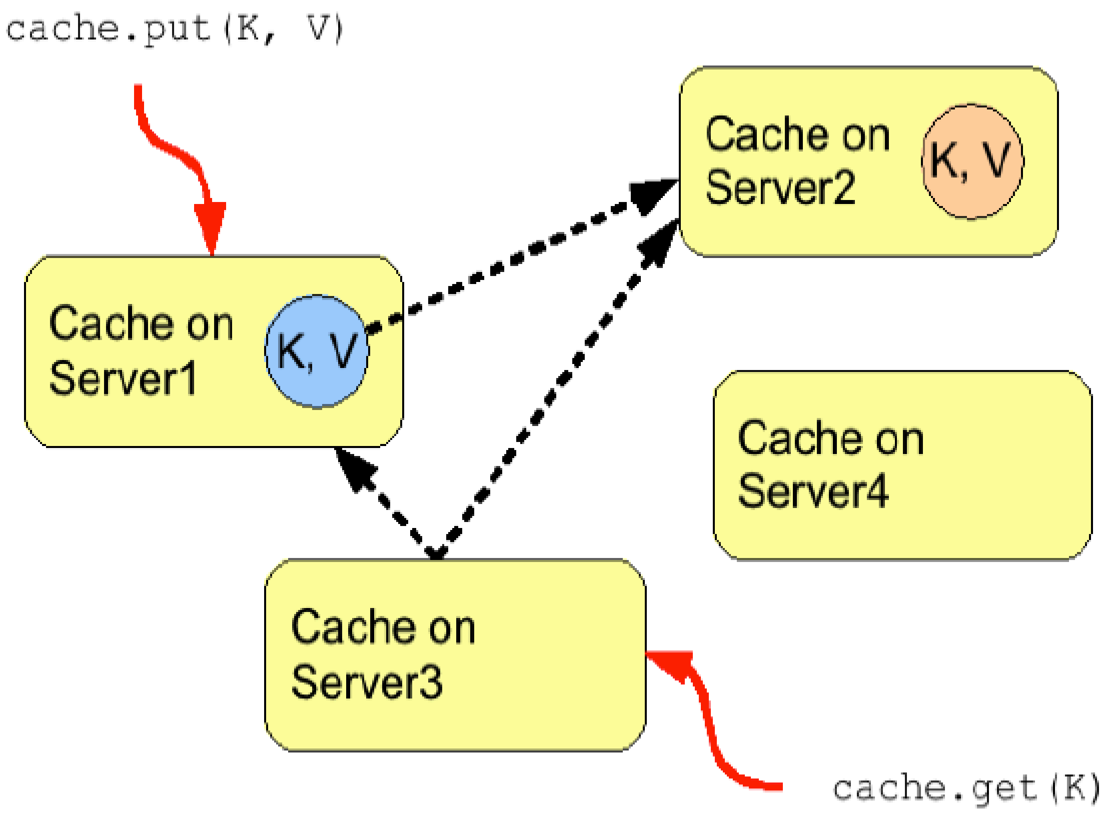


Figure 3. Distribution mode

32.5.1 Virtual Nodes - Improving the distribution of data

Infinispan does not attempt to evenly split the hash space between nodes – by not trying to split it evenly, it means that if a node joins or leaves the grid, there is no need to adjust the ownership of every node, just the neighbours of the joiner/leaver. This has a positive impact on network traffic. However this can mean that some nodes take on substantially larger portions of the hash space than others. This, combined with potential irregularities in the hash functions of keys, can mean the distribution of entries across the grid becomes poor. In order to address the irregularities in the hash of keys, Infinispan uses an advanced hashing function (Murmur Hash 3) by default, as well as using a bit spreader. In order to address the irregularities in the node distribution, Infinispan uses virtual nodes.

First, let's consider how virtual nodes help conceptually by taking a couple of distribution examples, and armed with that knowledge, look at how Infinispan uses them.

Consider a hash space of 1000 (there are 1000 buckets into which data can be placed). If there were two nodes, it is possible you can have 1 node being used for 1 bucket, and 1 node for 999 buckets (this is the most pessimistic distribution!). If there were 200 nodes, the worst distribution of node would end up being 199 nodes responsible for one bucket each (199 buckets in total), and 1 node being responsible for 801 buckets. If there were 1000 nodes, then each node must be responsible for 1 bucket each. From this we can deduce that as the number of nodes tends to the size of the hash space, that the distribution of buckets to nodes improves.

A guiding principle of Infinispan is that it always uses an algorithm to locate a key in the hash space, never allowing the node on which the entry is stored to be specified manually. This scheme allows any node to know which nodes owns a key, without having to distribute such ownership information. This reduces the overhead of Infinispan, but more importantly improves redundancy as there is no need to replicate the ownership information in case of node failure.

With this in mind, we can see that virtual nodes are an ideal solution to the distribution of nodes problem, as it allows the location of an entry to be determined algorithmically.

Infinispan implements virtual nodes by altering the algorithm for splitting the hash space whenever a node joins or leaves the grid. Rather than allocating a block of the hashspace to the node, it allocates a number of blocks from throughout the hash space.

Often it's easier to understand the topology changes that virtual nodes introduce through diagrams.

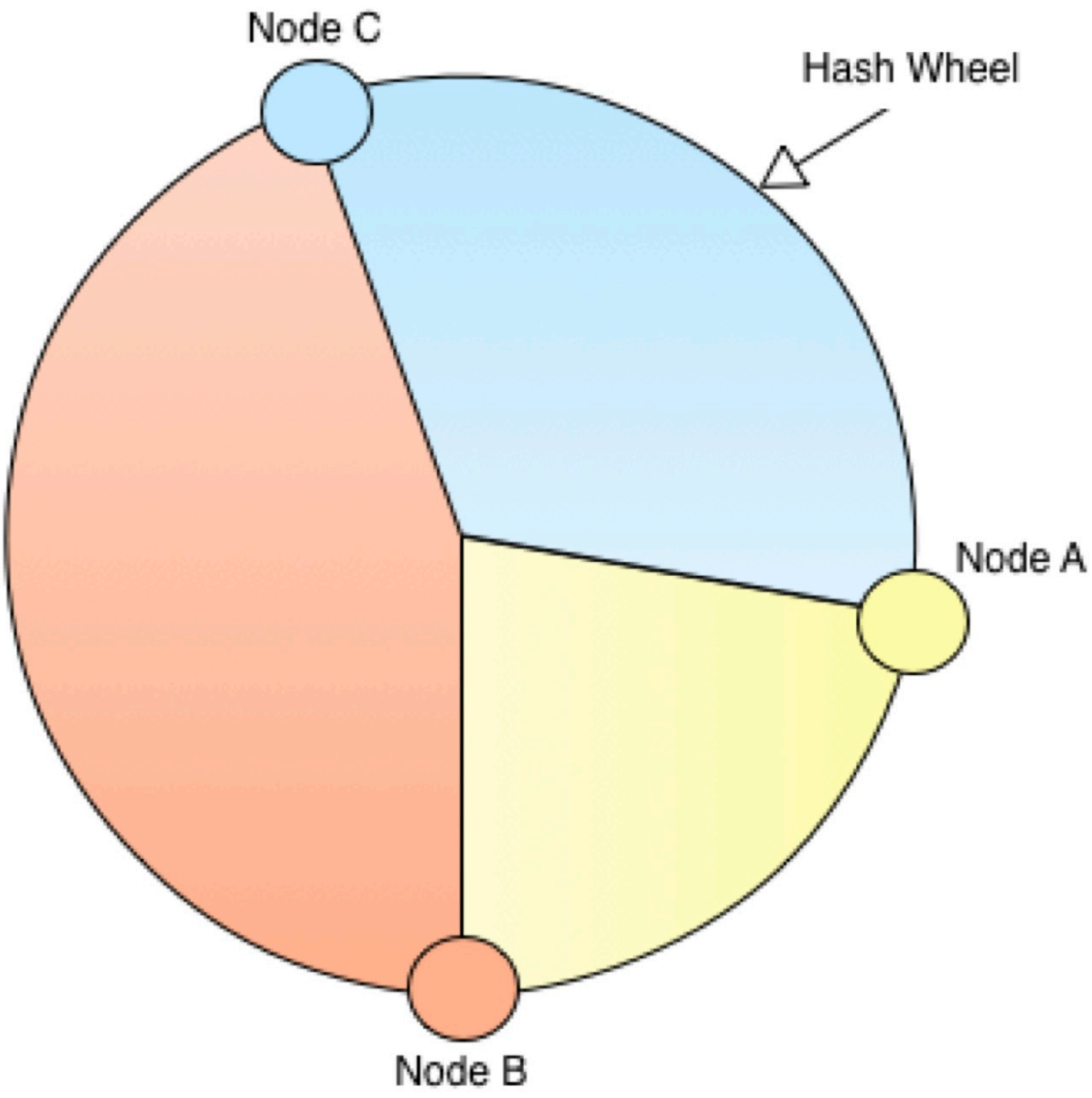


Figure: Topology Without Virtual Nodes

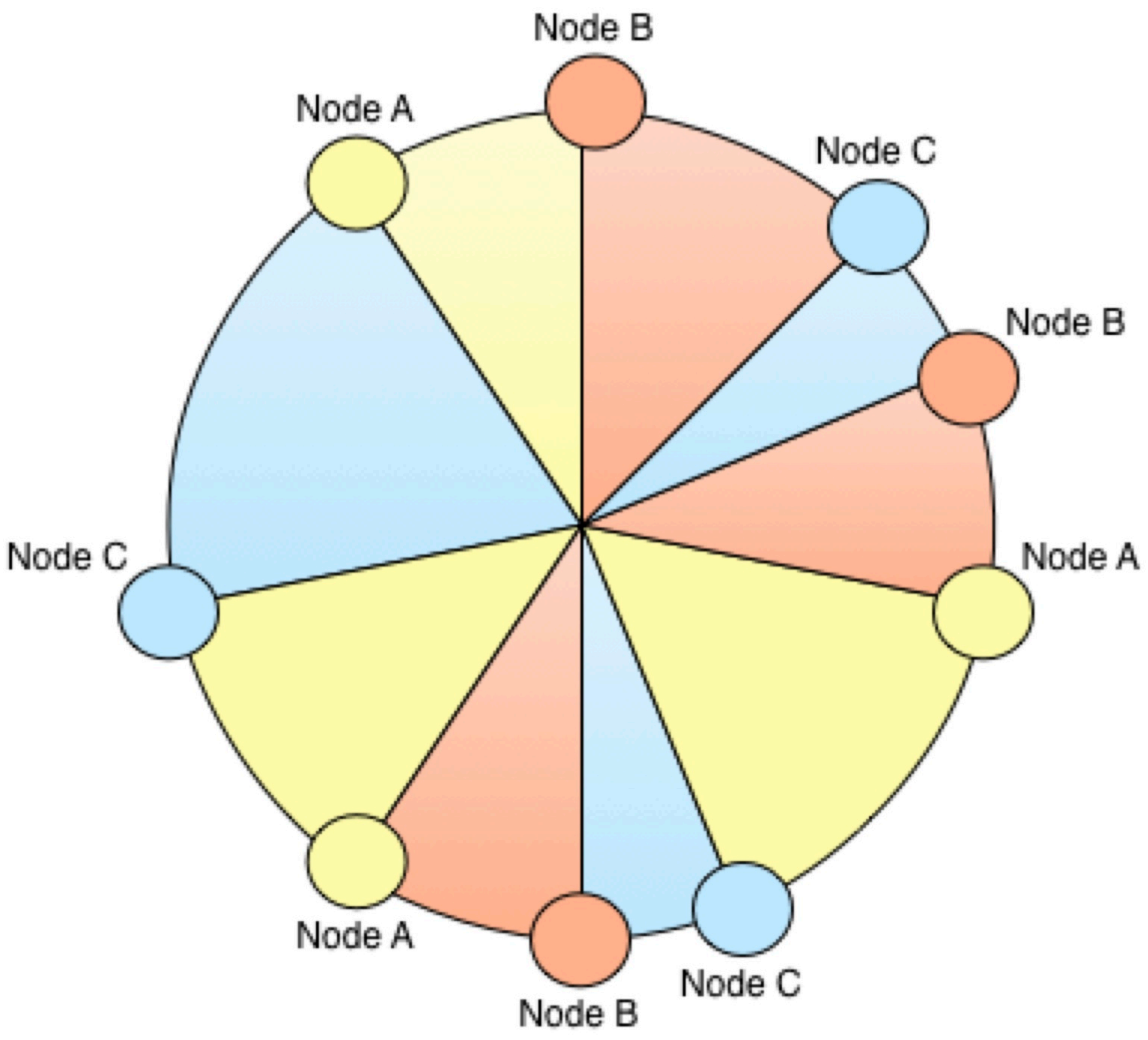


Figure: Topology With Virtual Nodes

To use virtual nodes, simply set the number of virtual nodes higher than one. For example

```
<namedCache name="cacheWithVirtualNodes">
      <clustering>
         <hash numVirtualNodes="10" />
      </clustering>
   </namedCache>
```

Alternatively, you can enable virtual nodes programmatically

```
new ConfigurationBuilder()
   .clustering()
      .hash()
         .numVirtualNodes(10)
   .build();
```

TODO Add notes on how to select number of virtual nodes.

32.6 L1 Caching

To prevent repeated remote calls when doing multiple GETs, L1 caching can be enabled. L1 caching places remotely received values in a near cache for a short period of time (configurable) so repeated lookups would not result in remote calls. In the above diagram, if L1 was enabled, a subsequent GET for the same key on Server3 would not result in any remote calls.

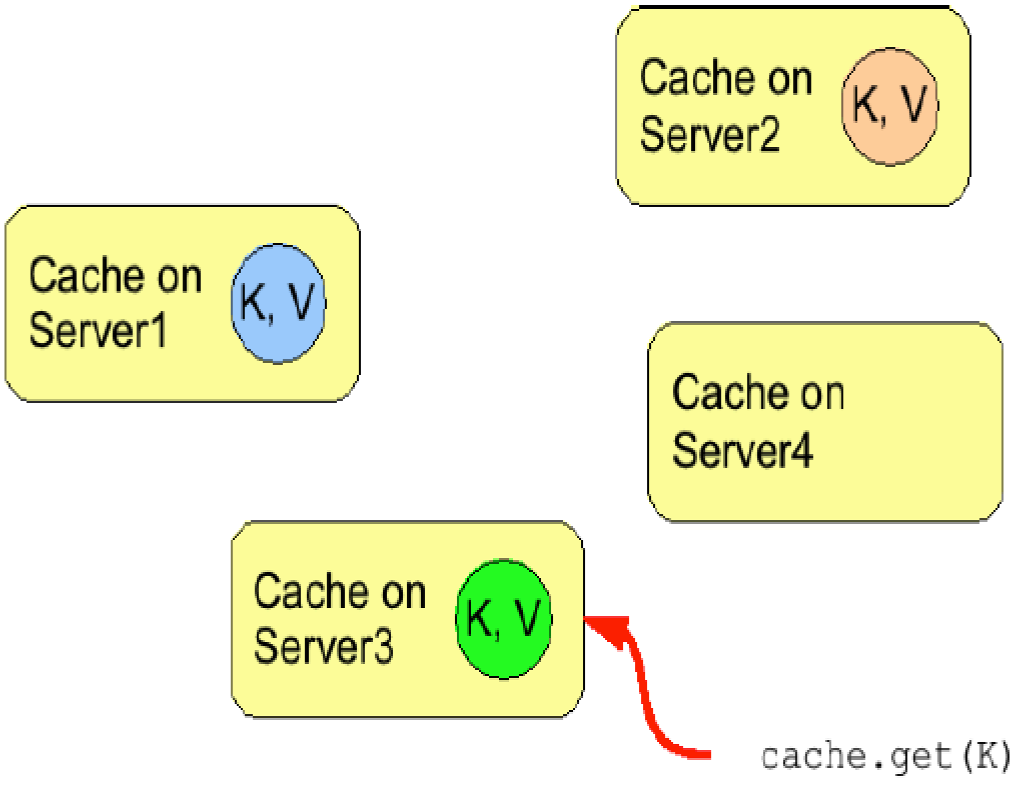


Figure 4. L1 caching

L1 caching is not free though. Enabling it comes at a cost, and this cost is that every time a key is updated, an invalidation message needs to be multicast to ensure nodes with the entry in L1 invalidates the entry. L1 caching causes the requesting node to cache the retrieved entry locally and listen for changes to the key on the wire. L1-cached entries are given an internal expiry to control memory usage. Enabling L1 will improve performance for repeated reads of non-local keys, but will increase memory consumption to some degree. It offers a nice tradeoff between the "read-mostly" performance of an invalidated data grid with the scalability of a distributed one. Is L1 caching right for you? The correct approach is to benchmark your application with and without L1 enabled and see what works best for your access pattern.

```
Looking for Buddy Replication?  Buddy Replication - from JBoss Cache - does not exist in
Infinispan.  See this blog article which discusses the reasons why Buddy Replication was not
implemented in Infinispan, and how the same effects can be achieved using Infinispan -
<a
href="http://infinispan.blogspot.com/2009/08/distribution-instead-of-buddy.html">http://infinispan
```

33 Using Infinispan as a Spring Cache provider

```
h1. Introduction

Starting with 3.1 Spring offers a [cache abstraction|<a
href="http://blog.springsource.com/2011/02/23/spring-3-1-m1-caching/">http://blog.springsource.com
enabling users to declaratively add caching support to applications via two simple annotations,
{font:monospace}@Cacheable{font} and {font:monospace}@CacheEvict{font}. While out of the box
Spring 3.1's caching support is backed by [Ehcache|<a
href="http://ehcache.org/">http://ehcache.org/</a>] it has been designed to easily support
different cache providers. To that end Spring 3.1 defines a simple and straightforward SPI other
caching solutions may implement. Infinispan's very own spring module does - amongst other things
- exactly this and therefore users invested in Spring's programming model may easily have all
their caching needs fulfilled through Infinispan. Here's how.

(Note that the following is based on a small but fully functional example that is part of Spring
Infinispan's test suite. For further details you are encouraged to look at
{font:monospace}org.infinispan.spring.provider.sample.CachingBookDaoContextTest{font} and its
ilk.)

h1. Activating Spring Cache support

You activate Spring's cache support via putting
{code:xml}
<beans xmlns="<a
href="http://www.springframework.org/schema/beans">http://www.springframework.org/schema/beans</a>

            xmlns:xsi="<a
href="http://www.w3.org/2001/XMLSchema-instance">http://www.w3.org/2001/XMLSchema-instance</a>"
            xmlns:cache="<a
href="http://www.springframework.org/schema/cache">http://www.springframework.org/schema/cache</a>
xsi:schemaLocation="
                <a
href="http://www.springframework.org/schema/beans">http://www.springframework.org/schema/beans</a>

                <a
href="http://www.springframework.org/schema/beans/spring-beans.xsd">http://www.springframework.org
<a
href="http://www.springframework.org/schema/cache">http://www.springframework.org/schema/cache</a>

                <a
href="http://www.springframework.org/schema/cache/spring-cache.xsd">http://www.springframework.org
<cache:annotation-driven />

</beans>
```

somewhere in your application context. This will tell Spring to be on the lookout for {font:monospace}@Cacheable{font} and {font:monospace}@CacheEvict{font} within your application code.Now, assuming you've already {font:monospace}infinispan.jar{font} and its dependencies on your classpath, all that's left to do is installing {font:monospace}infinispan-spring{font} and {font:monospace}spring{font}. For maven users this translates into

```
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-context</artifactId>
          <version>3.1.0.M1</version>
          <scope>compile</scope>
       </dependency>
       <dependency>
          <groupId>org.infinispan</groupId>
          <artifactId>infinispan-spring</artifactId>
          <version>5.0.0</version>
          <scope>compile</scope>
       </dependency>
```

{font:monospace}Gradle{font} users will most likely know how to adapt this to their needs. Those leaning towards a more ... pedestrian way of managing their dependencies will need to download {font:monospace}Spring 3.1.0 M1{font} and install it manually alongside {font:monospace}infinispan-spring.jar{font} from the Infinispan distribution.h1. Telling Spring to use Infinispan as its caching providerSpring 3.1's cache provider SPI comprises two interfaces, {font:monospace}org.springframework.cache.CacheManager{font} and {font:monospace}org.springframework.cache.Cache{font} where a {font:monospace}CacheManager{font} serves as a factory for named {font:monospace}Cache{font} instances. By default Spring will look at runtime for a {font:monospace}CacheManager{font} implementation having the bean name "cacheManager" in an application's application context. So by putting

```
<!-- Infinispan cache manager -->
<bean id="cacheManager"
          class="org.infinispan.spring.provider.SpringEmbeddedCacheManagerFactoryBean"

p:configurationFileLocation="classpath:/org/infinispan/spring/provider/sample/books-infinispan-con
/>
```

somewhere in your application context you tell Spring to henceforth use Infinispan as its caching provider.h1. Adding caching to your application codeAs outlined above enabling caching in your application code is as simple as adding {font:monospace}@Cacheable{font} and {font:monospace}@CacheEvict{font} to select methods. Suppose you've got a DAO for, say, books and you want book instances to be cached once they've been loaded from the underlying database using {font:monospace}BookDao#findBook(Integer bookId){font}. To that end you annotate {font:monospace}findBook(Integer bookId){font} with {font:monospace}@Cacheable{font}, as in

```
@Transactional
@Cacheable(value = "books", key = "#bookId")
Book findBook(Integer bookId) {...}
```

This will tell Spring to cache Book instances returned from calls to {font:monospace}findBook(Integer bookId){font} in a named cache "books", using the parameter's "bookId" value as a cache key. Here, "#bookId" is an expression in the [Spring Expression Language| http://static.springsource.org/spring/docs/current/spring-framework-reference/html/expressions.html] that evaluates to the {font:monospace}bookId{font} argument. If you don't specify the {font:monospace}key{font} attribute Spring will generate a hash from the supplied method arguments - in this case only {font:monospace}bookId{font} - and use that as a cache key. Essentially, you relinquish control over what cache key to use to Spring. Which may or may not be fine depending on your application's needs.Though the notion of actually deleting a book will undoubtedly seem alien and outright abhorrent to any sane reader there might come the time when your application needs to do just that. For whatever reason. In this case you will want for such a book to be removed not only from the underlying database but from the cache, too. So you annotate {font:monospace}deleteBook(Integer bookId){font} with {font:monospace}@CacheEvict{font} as in

```
@Transactional
@CacheEvict(value = "books", key = "#bookId")
void deleteBook(Integer bookId) {...}
```

and you may rest assured that no stray books be left in your application once you decide to remove them.h1. OutroHopefully you enjoyed our quick tour of Infinispan's support for Spring's cache abstraction and saw how easy it is for all your caching woes to be taken care of by Infinispan. More information may be found in Spring's as usual rather excellent [reference documentation| http://static.springsource.org/spring/docs/3.1.0.M1/spring-framework-reference/html/cache.html] and [javadocs| http://static.springsource.org/spring/docs/3.1.0.M1/javadoc-api/index.html?org/springframework/cache/packag]. Also see [this|http://blog.springsource.com/2011/02/23/spring-3-1-m1-caching/] very nice posting on the official Spring blog for a somewhat more comprehensive introduction to Spring's cache abstraction.{code}

34 Note: This page is obsolete. Please refer to Infinispan Distributed Execution Framework .

35 Invocation Flags

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

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

35.1 Examples

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

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

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

35.2 DecoratedCache

Another approach would be to use the DecoratedCache wrapper. This allows you to reuse flags. For example:

```
AdvancedCache cache = ...
DecoratedCache strictlyLocal = new DecoratedCache(cache, Flag.CACHE_MODE_LOCAL,
Flag.SKIP_CACHE_STORE);
strictlyLocal.put("local_1", "only");
strictlyLocal.put("local_2", "only");
strictlyLocal.put("local_3", "only");
```

This approach makes your code more readable.

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

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

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

For more information, please check the Flag enumeration javadoc.

36 Key affinity service

36.1 Introduction

The key affinity service solves the following problem: for a distributed Infinispan cluster one wants to make sure that a value is placed in a certain node. Based on a supplied cluster address identifying the node, the service returns a key that will be hashed to that particular node.

36.2 API

Following code snippet depicts how a reference to this service can be obtained and used.

```
//1. obtain a reference to a cache manager
EmbeddedCacheManager cacheManager = getCacheManager();//obtain a reference to a cache manager
Cache cache = cacheManager.getCache();

//2. create the affinity service
KeyAffinityService keyAffinityService =
KeyAffinityServiceFactory.newLocalKeyAffinityService(cache, new RndKeyGenerator(),
                                 Executors.newSingleThreadExecutor(), 100);

//3. obtain a key to be mapped to a certain address
Object localKey = keyAffinityService.getKeyForAddress(cacheManager.getAddress());

//4. this put makes sure that the key resigns on the local node (as obtained
cacheManager.getAddress())
cache.put(localKey, "yourValue");
```

The service is started at step 2: after this point it uses the supplied Excutor to generate and queue keys. At step 3, we obtain a key for this service, and use it at step 4, with that guarantee that it is distributed in node identified by cacheManager.getAddress().

36.3 Lifecycle

KeyAffinityService extends Lifecycle, which allows stopping and (re)starting it:

```
public interface Lifecycle {
   void start();
   void stop();
}
```

The service is instantiated through KeyAffinityServiceFactory. All the factory method have an Executors parameter, that is used for asynchronous key generation (so that it won't happen in the caller's thread). It is user's responsibility to handle the shutdown of this Executor.
The KeyAffinityService, once started, needs to be explicitly stopped. This stops the async key generation and releases other held resources.

The only situation in which KeyAffinityService stops by itself is when the cache manager with wich it was registered is shutdown.

36.4 Topology changes

When a topology change takes place the key ownership from the KeyAffinityService might change. The key affinity service keep tracks of these topology changes and updates and doesn't return stale keys, i.e. keys that would currently map to a different node than the one specified. However, this does not guarantee that at the time the key is used its node affinity hasn't changed, e.g.:

- thread T1 reads a key k1 that maps to node A

- a topology change happens which makes k1 map to node B

- T1 uses k1 to add something to the cache. At this point k1 maps to B, different node than the one requested at the time of read.

Whilst this is not ideal, it should be a supported behaviour for the application as all the already in-use keys might me moved over during cluster change. The KeyAffinityService provides an access proximity optimisation for stable clusters which doesn't apply during topology changes.

37 Transaction recovery

37.1

37.2 When to use recovery

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

37.3 How does it work

Recovery is coordinated by the TransactionManager (TM). The TM works with Infinispan to determine the list of in-doubt transactions that require manual intervention and informs the system administrator (SA) (email, logs). This process is TM specific, but generally requires some configuration on TM's side.

Knowing the in-doubt transaction ids, the SA can now connect to the Infinispan cluster and replay the commit of transactions or force the rollback. Infinispan provides JMX tooling for this - this is explained extensively in the Reconciliate state section.

37.4 Configuring recovery

Recovery is **not** enabled by default in Infinispan. If disabled the TransactionManager won't be able to work with Infinispan to determine the in-doubt transactions. In order to enable recovery through xml configuration:

```
<transaction useEagerLocking="true" eagerLockSingleNode="true">
    <recovery enabled="true" recoveryInfoCacheName="noRecovery"/>
</transaction>
```

Note: the *recoveryInfoCacheName* attribute is not mandatory. More information about it can be found in the **Recovery Cache** section below.

Alternatively you can enable it through the fluent configuration API as follows:

```
//simply calling .recovery() enables it
configuration.fluent().transaction().recovery();

//then you can disable it
configuration.fluent().transaction().recovery().disable();

//or just check its status
boolean isRecoveryEnabled = configuration.isTransactionRecoveryEnabled();
```

Recovery can be enabled/disabled o a per cache level: e..g it is possible to have a transaction spanning a cache that is has it enabled and another one that doesn't.

37.4.1 Enable JMX support

Important: In order to be able to use JMX for managing recovery JMX support must be explicitly enabled. More about enabling JMX here.

37.4.2 Recovery cache

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

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

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

37.5 Integration with the TM

Even though this is TM specific, generally the TM would need a reference to a XAResource imlementation in order to run XAResource.recover on it. In order to obtain a reference to a Infinispan XAResource following API can be used:

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

Note: It is a common practice to run the recovery in a different process than the one running the transaction. At the moment it is not possible to do this with infinispan: the recovery must be run from the same process where the infinispan instance exists. This limitation will be dropped once ransactions over HotRod are available.

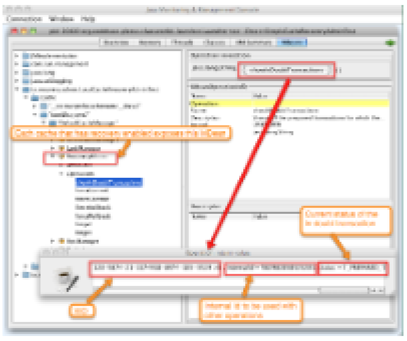
37.6 conciliate state

The TM informs the SA on in-doubt transaction in a proprietary way. At this stage it is assumed that the SA knows transaction's XID (byte array).

A normal recovery flow is:

1. SA connects to an Infinispan server through JMX, and lists the in doubt transactions

The image below is taken with JCosole connecting to an Infinispan node that has an in doubt transaction.

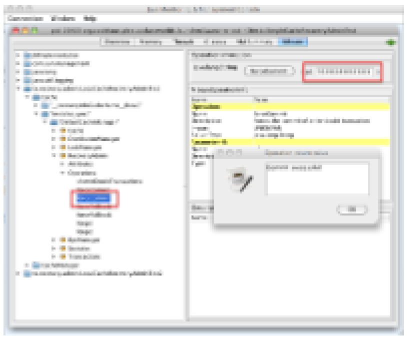


The status of each in-doubt transaction is displayed(in this example "*PREPARED*"). There might be multiple elements in the status field, e.g. "PREPARED" and "COMMITTED" in the case the transaction committed on certain nodes but not on all of them.

2. SA visually maps the XID received from the TM to an Infinispna internal id, represented as a number. This step is needed because the XID, a byte array, cannot conveniently be passed to the JMX tool (e.g. JConsole) and then re-assembled on infinispan's side.

3. SA forces the transaction's commit/rollback through the corresponding jmx operation, based on the internal id.

The image below is obtained by forcing the commit of the transaction based on its internal id.



Note: All JMX operations described above can be executed on any node, disregarding where the transaction originated.

37.6.1 Force commit/rollback based on XID

XID-based JMX operations for forcing in-doubt transactions' commit/rollback are available as well: these methods receive byte[] arrays describing the XID instead of the number associated with the transactions (as previously described at step 2). These can be useful e.g. if one wants to set up an automatic completion job for certain in-doubt transactions. This process is plugged into TM's recovery and has access to the TM's XID objects.

37.7 Want to know more?

The [recovery design document] describes in more detail the insides of transaction recovery implementation.

38 Implementing standalone JPA JTA Hibernate application outside J2EE server using Infinispan 2nd level cache

38.1 Introduction

IMPORTANT NOTE: From Hibernate 4.0.1 onwards, Infinispan now interacts as a synchronization rather than as an XA resource with the transaction manager when used as second-level cache, so there's no longer need to apply any of the changes suggested below!

Infinispans predecessor JBossCache requires integration with JTA when used as 2L-cache for a Hibernate application. At the moment of writing this article (Hibernate 3.5.0.Beta3) also Infinspan requires integration with JTA. Hibernate integrated with JTA is already largely used in EJB applications servers, but most users using Hibernate with Java SE outside any EJB container, still use the plain JDBC approach instead to use JTA.

According Hiberante documentation it should also possible to integrate JTA in a standalone application outside any EJB container, but I did hardly find any documentation how to do that in detail. (probably the reason is, that probably 95% of people is using hibernate within a EJB app. server or using SPRING). This article should give you some example how to realize a standalone Hibernate app. outside of a EJB container with JTA integration (and using Infinispan 2nd level cache).

As first thing you have to choose which implementation of TransactionManager to take. This article comes with examples for following OpenSource TransactionManagers:

1. JBoss
2. JOTM
3. Bitronix
4. Atomikos

Datasource/Transaction interaction

A very important aspect is not forgetting to couple the datasource with your transaction manager. In other words, the corresponding XAResource must be onto the transaction manager, otherwise only DML-statements but no commits/rollbacks are propagated to your database.

38.2 JBoss Transactions

The example with JBoss Transactions Transaction Manager was the most complex to implement, as JBoss's TransactionManager and UserTransaction objects are not declared serializable whilst it's JNDI-server isn't able to bind non serializable objects out of the box. Special use of NonSerializableFactory is needed, requiring some additional custom code:

```
import hello.A;  // a persistent class
import java.io.Serializable;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.Name;
import javax.naming.NameNotFoundException;
import javax.naming.Reference;
import javax.naming.StringRefAddr;
import javax.persistence.EntityManager;
import javax.persistence.Persistence;
import javax.transaction.TransactionManager;
import javax.transaction.UserTransaction;

import org.enhydra.jdbc.standard.StandardXADataSource;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.ejb.HibernateEntityManagerFactory;
import org.hibernate.transaction.JBossTransactionManagerLookup;
import org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup;
import org.jboss.util.naming.NonSerializableFactory;
import org.jnp.interfaces.NamingContext;
import org.jnp.server.Main;
import org.jnp.server.NamingServer;

public class JTAStandaloneExampleJBossTM  {

    static JBossStandaloneJTAManagerLookup _ml =  new JBossStandaloneJTAManagerLookup();


    public static void main(String[] args) {
        try {
            // Create an in-memory jndi
            NamingServer namingServer = new NamingServer();
            NamingContext.setLocal(namingServer);
            Main namingMain = new Main();
            namingMain.setInstallGlobalService(true);
            namingMain.setPort(-1);
            namingMain.start();

            Properties props = new Properties();
            props.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
            props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
            InitialContext ictx = new InitialContext( props );


            // as JBossTransactionManagerLookup extends JNDITransactionManagerLookup we must
also register the TransactionManager
            bind("java:/TransactionManager", _ml.getTransactionManager(),
_ml.getTransactionManager().getClass(), ictx);

            // also the UserTransaction must be registered on jndi:
org.hibernate.transaction.JTATransactionFactory#getUserTransaction() requires this
            bind(new
JBossTransactionManagerLookup().getUserTransactionName(),_ml.getUserTransaction(),_ml.getUserTrans
ictx);

            ExtendedXADataSource xads = new ExtendedXADataSource();
            xads.setDriverName("org.hsqldb.jdbcDriver");
            xads.setDriverName("com.p6spy.engine.spy.P6SpyDriver"); // comment this line if you
don't want p6spy-logging
            xads.setUrl("jdbc:hsqldb:hsql://localhost");
            //xads.setTransactionManager(_ml.getTransactionManager()); useless here as this
information is not serialized

            ictx.bind("java:/MyDatasource", xads);

            final HibernateEntityManagerFactory emf =  (HibernateEntityManagerFactory)
Persistence.createEntityManagerFactory("helloworld");

            UserTransaction userTransaction = _ml.getUserTransaction();
            userTransaction.setTransactionTimeout(300000);
            //SessionFactory sf = (SessionFactory)
ictx.lookup("java:/hibernate/MySessionFactory"); // if hibernate.session_factory_name set
            final SessionFactory sf = emf.getSessionFactory();

            userTransaction.begin();
            EntityManager em = emf.createEntityManager();

            // do here your persistent work
            A a = new A();
            a.name= "firstvalue";
            em.persist(a);
            em.flush();     // do manually flush here as apparently FLUSH_BEFORE_COMPLETION
seems not work, bug ?

            System.out.println("\nCreated and flushed instance a with id : " + a.oid + "  a.name
set to:" + a.name);

            System.out.println("Calling userTransaction.commit() (Please check if the commit is
effectively executed!)");
            userTransaction.commit();


            ictx.close();
            namingMain.stop();
            emf.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
        System.exit(0);
    }

   public static class ExtendedXADataSource extends StandardXADataSource { // XAPOOL

        @Override
        public Connection getConnection() throws SQLException {

            if (getTransactionManager() == null) { // although already set before, it results
null again after retrieving the datasource by jndi
                TransactionManager tm;  // this is because the TransactionManager information is
not serialized.
                try {
                    tm = _ml.getTransactionManager();
                } catch (Exception e) {
                    throw new SQLException(e);
                }
                setTransactionManager(tm);  //  resets the TransactionManager on the datasource
retrieved by jndi,
                                            //  this makes the datasource JTA-aware
            }

            // According to Enhydra documentation, here we must return the connection of our
XAConnection
            // see
http://cvs.forge.objectweb.org/cgi-bin/viewcvs.cgi/xapool/xapool/examples/xapooldatasource/Databas
return super.getXAConnection().getConnection();
        }
    }

    /**
     * Helper method that binds the a non serializable object to the JNDI tree.
     *
     * @param jndiName Name under which the object must be bound
     * @param who Object to bind in JNDI
     * @param classType Class type under which should appear the bound object
     * @param ctx Naming context under which we bind the object
     * @throws Exception Thrown if a naming exception occurs during binding
     */
    private static void bind(String jndiName, Object who, Class classType, Context ctx) throws
Exception {
       // Ah ! This service isn't serializable, so we use a helper class
       NonSerializableFactory.bind(jndiName, who);
       Name n = ctx.getNameParser("").parse(jndiName);
       while (n.size() > 1) {
          String ctxName = n.get(0);
          try {
             ctx = (Context) ctx.lookup(ctxName);
          } catch (NameNotFoundException e) {
             System.out.println("Creating subcontext:" + ctxName);
             ctx = ctx.createSubcontext(ctxName);
          }
          n = n.getSuffix(1);
       }

       // The helper class NonSerializableFactory uses address type nns, we go on to
       // use the helper class to bind the service object in JNDI
       StringRefAddr addr = new StringRefAddr("nns", jndiName);
       Reference ref = new Reference(classType.getName(), addr,
NonSerializableFactory.class.getName(), null);
       ctx.rebind(n.get(0), ref);
    }

    private static void unbind(String jndiName, Context ctx) throws Exception {
       NonSerializableFactory.unbind(jndiName);
       ctx.unbind(jndiName);
    }

}
```

The content of the corresponding complete persistence.xml:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"  version="1.0">
   <persistence-unit name="helloworld" transaction-type="JTA">
      <jta-data-source>java:/MyDatasource</jta-data-source>
      <properties>
       <property name="hibernate.hbm2ddl.auto" value = "create"/>
       <property name="hibernate.archive.autodetection" value="class, hbm"/>
           <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>

           <property name="hibernate.jndi.class"
value="org.jnp.interfaces.NamingContextFactory"/>
           <property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.JBossTransactionManagerLookup"/>

        <property name="current_session_context_class" value="jta"/>
           <!-- <property name="hibernate.session_factory_name"
value="java:/hibernate/MySessionFactory"/> optional -->
           <property name="hibernate.transaction.factory_class"
value="org.hibernate.transaction.JTATransactionFactory"/>
           <property name="hibernate.connection.release_mode" value="auto"/>
           <!-- setting above is important using XA-DataSource on SQLServer,
                otherwise SQLServerException: The function START: has failed. No transaction
cookie was returned.-->

         <property name="hibernate.cache.use_second_level_cache" value="true"/>
            <property name="hibernate.cache.use_query_cache" value="true"/>

         <property name="hibernate.cache.region.factory_class"
value="org.hibernate.cache.infinispan.InfinispanRegionFactory"/>

      </properties>
   </persistence-unit>
</persistence>
```

38.3 JOTM

The example with JOTM is more simple, but apparently it's JNDI implementation is not useable without wasting any rmi port. So it is not completely 'standalone' as the JNDI service is exposed outside your virtual machine.

```
import hello.A; // a persistent class

import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.transaction.TransactionManager;
import javax.transaction.UserTransaction;

import org.enhydra.jdbc.standard.StandardXADataSource;
import org.hibernate.transaction.JOTMTransactionManagerLookup;
import org.objectweb.jotm.Jotm;
import org.objectweb.transaction.jta.TMService;


public class JTAExampleJOTM {

 static JOTMTransactionManagerLookup _ml =  new JOTMTransactionManagerLookup();

 public static class ExtendedXADataSource extends StandardXADataSource { // XAPOOL
        @Override
        public Connection getConnection() throws SQLException {
            if (getTransactionManager() == null) { // although already set before, it results
null again after retrieving the datasource by jndi
                TransactionManager tm;  // this is because the TransactionManager information is
not serialized.
                try {
                    tm = _ml.getTransactionManager(null);
                } catch (Exception e) {
                    throw new SQLException(e);
                }
                setTransactionManager(tm);  //  resets the TransactionManager on the datasource
retrieved by jndi,
                                            //  this makes the datasource JTA-aware
            }

            // According to Enhydra documantation, here we must return the connection of our
XAConnection
            // see
http://cvs.forge.objectweb.org/cgi-bin/viewcvs.cgi/xapool/xapool/examples/xapooldatasource/Databas
return super.getXAConnection().getConnection();
```

```
        }
    }


    public static void main( String[] args )
    {
        try
        {
            java.rmi.registry.LocateRegistry.createRegistry(1099); // also possible to lauch by
command line rmiregistry
            System.out.println("RMI registry ready.");


           // following properties can be left out if specifying thes values in a file
jndi.properties located into classpath
            Properties props = new Properties();
            props.put(Context.INITIAL_CONTEXT_FACTORY,
"org.ow2.carol.jndi.spi.MultiOrbInitialContextFactory");
           InitialContext jndiCtx = new InitialContext(props);


        // XAPOOL
           ExtendedXADataSource xads = new ExtendedXADataSource();
           xads.setDriverName("org.hsqldb.jdbcDriver");
           xads.setDriverName("com.p6spy.engine.spy.P6SpyDriver");
           xads.setUrl("jdbc:hsqldb:hsql://localhost");

           jndiCtx.bind("java:/MyDatasource", xads);



           /* startup JOTM */
           TMService jotm = new Jotm(true, false);
           jotm.getUserTransaction().setTransactionTimeout(36000); // secs, important JOTM
default is only 60 secs !


           /* and get a UserTransaction */
           UserTransaction userTransaction = jotm.getUserTransaction();


           jndiCtx.bind("java:comp/UserTransaction", jotm.getUserTransaction()); // this is
needed by hibernates JTATransactionFactory

           /* get the Hibernate SessionFactory */
           EntityManagerFactory emf =    Persistence.createEntityManagerFactory("helloworld");
           //SessionFactory sf = (SessionFactory)
jndiCtx.lookup("java:/hibernate/MySessionFactory");

           // begin a new Transaction
           userTransaction.begin();
           EntityManager em = emf.createEntityManager();

           A a = new A();
           a.name= "firstvalue";
           em.persist(a);
           em.flush();     // do manually flush here as apparently FLUSH_BEFORE_COMPLETION seems
not work, bug ?
```

```
            System.out.println("Calling userTransaction.commit() (Please check if the commit is
effectively executed!)");
            userTransaction.commit();


            // stop the transaction manager
            jotm.stop();
            jndiCtx.close();
            emf.close();


        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
        System.exit(0);
    }

}
```

Adjust following 2 properties in your persistence.xml:

```
<property name="hibernate.jndi.class"
value="org.ow2.carol.jndi.spi.MultiOrbInitialContextFactory"/>
<property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.JOTMTransactionManagerLookup"/>
```

For using the JTA Hibernate application as servlet in tomcat please read http://jotm.objectweb.org/current/jotm/doc/howto-tomcat-jotm.html and also https://forum.hibernate.org/viewtopic.php?f=1&t=1003866

38.4 Bitronix

The Transaction Manager comes bundled with a fake in memory jndi-implementation which is ideal for standalone purpose. To integrate with Infinispan I did need a ad-hoc pre-alpha improvement (see attached btm-ispn.jar by courtesy of Mr. Ludivic Orban). BitronixTM offers the so-called Last Resource Commit optimization (aka Last Resource Gambit or Last Agent optimization) and it allows a single non-XA database to participate in a XA transaction by cleverly ordering the resources. "Last Resource Commit" is not part of the XA spec as it doesn't cover the transaction-recovery aspect, so if your database does not support XA (or if you don't wish to have the Xa-driver performance overhead against the plain jdbc) then the "Last Resource Commit" feature should be ideal for the combination 1 single database plus infinispan.

```
import hello.A; // a persistent class

import java.util.Properties;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.persistence.EntityManager;
import javax.persistence.Persistence;
import javax.transaction.UserTransaction;

import org.hibernate.cache.infinispan.InfinispanRegionFactory;
import org.hibernate.ejb.HibernateEntityManagerFactory;
import org.hibernate.impl.SessionFactoryImpl;
import org.infinispan.manager.CacheManager;

import bitronix.tm.resource.ResourceRegistrar;
import bitronix.tm.resource.infinispan.InfinispanCacheManager;
import bitronix.tm.resource.jdbc.PoolingDataSource;



public class JTAExampleBTM  {
    public static void main(String[] args) {
        try {
             Properties props = new Properties();
             props.put(Context.INITIAL_CONTEXT_FACTORY,
"bitronix.tm.jndi.BitronixInitialContextFactory");
             // Attention: BitronixInitialContextFactory is'nt a real jndi implementation: you
can't do explicit bindings
             // It is ideal for hiberante standalone usage, as it automatically 'binds' the
needed things: datasource + usertransaction

             System.out.println("create initial context");
             InitialContext ictx = new InitialContext(props);

             PoolingDataSource myDataSource = new PoolingDataSource();
             myDataSource.setClassName("bitronix.tm.resource.jdbc.lrc.LrcXADataSource");

             myDataSource.setMaxPoolSize(5);
             myDataSource.setAllowLocalTransactions(true);

             myDataSource.getDriverProperties().setProperty("driverClassName",
"com.p6spy.engine.spy.P6SpyDriver");
             myDataSource.getDriverProperties().setProperty("url",
"jdbc:hsqldb:hsql://localhost");
             myDataSource.getDriverProperties().setProperty("user", "sa");
             myDataSource.getDriverProperties().setProperty("password", "");
             myDataSource.setUniqueName("java:/MyDatasource");
             myDataSource.setAutomaticEnlistingEnabled(true); // important to keep it to true
(default), otherwise commits/rollbacks are not propagated
             myDataSource.init(); // does also register the datasource on the Fake-JNDI with
Unique Name

             org.hibernate.transaction.BTMTransactionManagerLookup lokhiberante = new
org.hibernate.transaction.BTMTransactionManagerLookup();

             HibernateEntityManagerFactory emf =  (HibernateEntityManagerFactory)
Persistence.createEntityManagerFactory("helloworld");
             SessionFactoryImpl sfi = (SessionFactoryImpl) emf.getSessionFactory();
             InfinispanRegionFactory infinispanregionfactory = (InfinispanRegionFactory)
sfi.getSettings().getRegionFactory();
             CacheManager manager = infinispanregionfactory.getCacheManager();

             // register Inifinispan as a BTM resource
```

```
              InfinispanCacheManager icm = new InfinispanCacheManager();
              icm.setUniqueName("infinispan");
              ResourceRegistrar.register(icm);
              icm.setManager(manager);

             final UserTransaction userTransaction = (UserTransaction)
ictx.lookup(lokhiberante.getUserTransactionName());

             // begin a new Transaction
             userTransaction.begin();
             EntityManager em = emf.createEntityManager();

             A a = new A();
             a.name= "firstvalue";
             em.persist(a);
             em.flush();     // do manually flush here as apparently FLUSH_BEFORE_COMPLETION
seems not work, bug ?

             System.out.println("Calling userTransaction.commit() (Please check if the commit is
effectively executed!)");
             userTransaction.commit();

             emf.close();

         } catch (Exception e) {
             e.printStackTrace();
             System.exit(1);
         }
          System.exit(0);

     }
}
```

Adjust following 2 properties in your corresponding persistence.xml:

```
<property name="hibernate.jndi.class" value="bitronix.tm.jndi.BitronixInitialContextFactory"/>
<property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.BTMTransactionManagerLookup"/>
```

38.5 Atominkos

Last but not least, the Atomikos Transaction manager. It is currently the unique Transaction manager I've found with a online-documentation on how to integrate with Hiberantewithout Spring, outside any J2EE container.. It seems to be the unique supporting XaDataSource together with Pooling, so it doesn't matter that It does not come with it's own JNDI implementation (we will use the one of JBoss in following example).

```
import hello.A; // a persistent class

import java.io.Serializable;
import java.sql.Connection;
import java.sql.SQLException;
import java.util.Properties;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.Name;
import javax.naming.NameNotFoundException;
import javax.naming.Reference;
import javax.naming.StringRefAddr;
import javax.persistence.EntityManager;
import javax.persistence.Persistence;
import javax.transaction.TransactionManager;
import javax.transaction.UserTransaction;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.ejb.HibernateEntityManagerFactory;
import org.hibernate.impl.SessionFactoryImpl;

import org.jboss.util.naming.NonSerializableFactory;
import org.jnp.interfaces.NamingContext;
import org.jnp.server.Main;
import org.jnp.server.NamingServer;

import com.atomikos.icatch.jta.hibernate3.TransactionManagerLookup;
import com.atomikos.jdbc.AtomikosDataSourceBean;
import com.atomikos.jdbc.SimpleDataSourceBean;

public class JTAStandaloneExampleAtomikos  {

    public static void main(String[] args) {
        try {
            // Create an in-memory jndi
            NamingServer namingServer = new NamingServer();
            NamingContext.setLocal(namingServer);
            Main namingMain = new Main();
            namingMain.setInstallGlobalService(true);
            namingMain.setPort(-1);
            namingMain.start();

            Properties props = new Properties();
            props.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
            props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");

            InitialContext ictx = new InitialContext( props );

            AtomikosDataSourceBean ds = new AtomikosDataSourceBean();
            ds.setUniqueResourceName("sqlserver_ds");
            ds.setXaDataSourceClassName("com.microsoft.sqlserver.jdbc.SQLServerXADataSource");
            Properties p = new Properties();
            p.setProperty ( "user" , "sa" );
            p.setProperty ( "password" , "" );
            p.setProperty ( "serverName" , "myserver" );
            ds.setXaProperties ( p );
            ds.setPoolSize(5);
            bind("java:/MyDatasource", ds, ds.getClass(), ictx);
            TransactionManagerLookup _ml = new TransactionManagerLookup();
            UserTransaction userTransaction = new com.atomikos.icatch.jta.UserTransactionImp();

            bind("java:/TransactionManager", _ml.getTransactionManager(null),
_ml.getTransactionManager(null).getClass(), ictx);
            bind("java:comp/UserTransaction", userTransaction, userTransaction.getClass(),
ictx);

            HibernateEntityManagerFactory emf =  (HibernateEntityManagerFactory)
Persistence.createEntityManagerFactory("helloworld");

            // begin a new Transaction
            userTransaction.begin();
            EntityManager em = emf.createEntityManager();

            A a = new A();
            a.name= "firstvalue";
            em.persist(a);
            em.flush();     // do manually flush here as apparently FLUSH_BEFORE_COMPLETION
seems not work, bug ?

            System.out.println("Calling userTransaction.commit() (Please check if the commit is
effectively executed!)");
            userTransaction.commit();

            emf.close();

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
         System.exit(0);
    }

    /**
     * Helper method that binds the a non serializable object to the JNDI tree.
     *
     * @param jndiName Name under which the object must be bound
     * @param who Object to bind in JNDI
     * @param classType Class type under which should appear the bound object
     * @param ctx Naming context under which we bind the object
     * @throws Exception Thrown if a naming exception occurs during binding
     */
    private static void bind(String jndiName, Object who, Class<?> classType, Context ctx)
throws Exception {
       // Ah ! This service isn't serializable, so we use a helper class
       NonSerializableFactory.bind(jndiName, who);
       Name n = ctx.getNameParser("").parse(jndiName);
       while (n.size() > 1) {
          String ctxName = n.get(0);
          try {
             ctx = (Context) ctx.lookup(ctxName);
          } catch (NameNotFoundException e) {
             System.out.println("Creating subcontext:" + ctxName);
             ctx = ctx.createSubcontext(ctxName);
          }
          n = n.getSuffix(1);
       }
       // The helper class NonSerializableFactory uses address type nns, we go on to
       // use the helper class to bind the service object in JNDI
       StringRefAddr addr = new StringRefAddr("nns", jndiName);
       Reference ref = new Reference(classType.getName(), addr,
NonSerializableFactory.class.getName(), null);
       ctx.rebind(n.get(0), ref);
    }

    private static void unbind(String jndiName, Context ctx) throws Exception {
       NonSerializableFactory.unbind(jndiName);
       ctx.unbind(jndiName);
    }

}
```

Adjust follwing 2 properties in your corresponding persistence.xml:

```
<property name="hibernate.jndi.class" value="org.jnp.interfaces.NamingContextFactory"/>
<property name="hibernate.transaction.manager_lookup_class"
value="com.atomikos.icatch.jta.hibernate3.TransactionManagerLookup"/>
```

And create a file named jta.properties in your classpath with following content:

```
com.atomikos.icatch.service=com.atomikos.icatch.standalone.UserTransactionServiceFactory
com.atomikos.icatch.automatic_resource_registration=false
com.atomikos.icatch.console_log_level=WARN
com.atomikos.icatch.force_shutdown_on_vm_exit=true
com.atomikos.icatch.enable_logging=false
```

39 Infinispan Maven Archetypes

Infinispan currently has 2 separate Maven archetypes you can use to create a skeleton project and get started using Infinispan. This is an easy way to get started using Infinispan as the archetype generates sample code, a sample Maven pom.xml with necessary depedencies, etc.

```
NOTE:  You don't need to have any experience with or knowledge of Maven's Archetypes to use
this!  Just follow the simple steps below.
```

```
WARNING: These archetypes have only been tested with <a
href="http://maven.apache.org/docs/3.0.1/release-notes.html">Maven 3</a>.  Please report back if
you have any success with using Maven 2.
```

39.1 Starting a new project

Use the newproject-archetype project. The simple command below will get you started, and

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.infinispan.archetypes \
    -DarchetypeArtifactId=newproject-archetype \
    -DarchetypeVersion=1.0.10 \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

You will be prompted for a few things, including the *artifactId* , *groupId* and *version* of your new project. And that's it - you're ready to go!

39.1.1 Playing with your new project

The skeleton project ships with a sample application class, interacting with Infinispan. You should open this new project in your IDE - most good IDEs such as IntelliJ and Eclipse allow you to import Maven projects, see this guide and this guide. Once you open your project in your IDE, you should examine the generated classes and read through the comments.

39.1.2 On the command line...

Try running

```
$ mvn install -Prun
```

in your newly generated project! This runs the main() method in the generated application class.

39.2 Writing a test case for Infinispan

This archetype is useful if you wish to contribute a test to the Infinispan project and helps you get set up to use Infinispan's testing harness and related tools.

Use

```
$ mvn archetype:generate \
    -DarchetypeGroupId=org.infinispan.archetypes \
    -DarchetypeArtifactId=testcase-archetype \
    -DarchetypeVersion=1.0.10 \
    -DarchetypeRepository=http://repository.jboss.org/nexus/content/groups/public
```

As above, this will prompt you for project details and again as above, you should open this project in your IDE. Once you have done so, you will see some sample tests written for Infinispan making use of Infinispan's test harness and testing tools along with extensive comments and links for further reading.

39.2.1 On the command line...

Try running

```
$ mvn test
```

in your newly generated project to run your tests.

The generated project has a few different profiles you can use as well, using Maven's -P flag. E.g.,

```
$ mvn test -Pudp
```

39.2.2 Available profiles

The profiles available in the generated sample project are:

- udp: use UDP for network communications rather than TCP
- tcp: use TCP for network communications rather than UDP
- jbosstm: Use the embedded JBoss Transaction Manager rather than Infinispan's dummy test transaction manager

39.2.3 Contributing tests back to Infinispan

If you have written a functional, unit or stress test for Infinispan and want to contribute this back to Infinispan, your best bet is to fork the Infinispan sources on GitHub. The test you would have prototyped and tested in an isolated project created using this archetype can be simply dropped in to Infinispan's test suite. Make your changes, add your test, prove that it fails even on Infinispan's upstream source tree and issue a pull request.

```
TIP: New to working with Infinispan and GitHub?  Want to know how best to work with the
repositories and contribute code?  Read <a __default_attr="16089" __jive_macro_name="document"
_modifiedtitle="Infinispan and GitHub" class="jive_macro jive_macro_document"
href="javascript:;" modifiedtitle="Infinispan and GitHub" title="Infinispan and GitHub"></a>
```

39.3 Versions

The archetypes generate poms with dependencies to specific versions of Infinispan. You should edit these generated poms by hand to point to other versions of Infinispan that you are interested in.

39.4 Source Code

The source code used to generate these archetypes are on GitHub. If you wish to enhance and contribute back to the project, fork away!

40 Accessing data in Infinispan via RESTful interface

Standards Red Hat is working towards standardization of the REST API as a part of the REST-* effort. To participate in this standard, please visit this Google Group

40.1 Putting data in

HTTP PUT and POST methods are used to place data in the cache - the data being the body of the request (the data can be anything you like). It is important that a `Content-Type` header is set.

40.1.1 PUT /{cacheName}/{cacheKey}

A PUT request of the above URL form will place the payload (body) in the given cache, with the given key (the named cache must exist on the server). For example http://someserver/hr/payRoll/3 (in which case "hr" is the cache name, and "payRoll/3" is the key). Any existing data will be replaced, and `Time-To-Live` and `Last-Modified` values etc will updated (if applicable).

40.1.2 POST /{cacheName}/{cacheKey}

Exactly the same as PUT, only if a value in a cache/key already exists, it will return a Http `CONFLICT` status (and the content will not be updated).

40.1.3 Headers:

- `Content-Type`: MANDATORY (use media/mime-types for example: "application/json").

> **Special case**
>
> If you set the `Content-Type` to `application/x-java-serialized-object`, then it will be stored as a Java object

- `performAsync`: OPTIONAL true/false (if true, this will return immediately, and then replicate data to the cluster on its own. Can help with bulk data inserts/large clusters.)
- `timeToLiveSeconds`: OPTIONAL number (the number of seconds before this entry will automatically be deleted)
  - If no parameter is sent, Infinispan assumes -1 as default value, which means that the entry will not expire as a result of ttl. Passing any negative value will have the same effect.
- `maxIdleTimeSeconds`: OPTIONAL number (the number of seconds after last usage of this entry when it will automatically be deleted)
  - If no parameter is sent, Infinispan assumes -1 as default value, which means that the entry will not expire as a result of idle time. Passing any negative value will have the same effect.

> **Passing 0 as parameter for timeToLiveSeconds and/or maxIdleTimeSeconds**
>
> - If both `timeToLiveSeconds` and `maxIdleTimeSeconds` are 0, the cache will use the default `lifespan` and `maxIdle` values configured in XML/programmatically
> - If **only** `maxIdleTimeSeconds` is 0, it uses the timeToLiveSeconds value passed as parameter (or -1 if not present), and default `maxIdle` configured in XML/programmatically
> - If **only** `timeToLiveSeconds` is 0, it uses 0 as timeToLiveSeconds meaning that it will expire immediately, and `maxIdle` is set whatever came as parameter (or -1 if not present)

40.2 Getting data back out

HTTP `GET` and `HEAD` are used to retrieve data from entries.

40.2.1 GET /{cacheName}/{cacheKey}

This will return the data found in the given cacheName, under the given key - as the body of the response. A Content-Type header will be supplied which matches what the data was inserted as (other then if it is a Java object, see below). Browsers can use the cache directly of course (eg as a CDN).
An ETag will be returned unique for each entry, as will the Last-Modified header field indicating the state of the data at the given URL. ETags allow browsers (and other clients) to ask for data only in the case where it has changed (to save on bandwidth) - this is standard HTTP and is honoured by Infinispan.

40.2.2 HEAD /{cacheName}/{cacheKey}

The same as GET, only no content is returned (only the header fields). You will receive the same content that you stored. E.g., if you stored a String, this is what you get back. If you stored some XML or JSON, this is what you will receive. If you stored a binary (base 64 encoded) blob, perhaps a serialized; Java; object - you will need to; deserialize this yourself.

Prior to Infinispan 4.2

The behaviour was as follows:
If the data in the grid is a Java object - there are a few options in how it can be returned, which use the HTTP Accept header:

- `application/xml` - the object will be serialized via XStream to XML representation
- `application/json` - the object will be sent via a JSON format (via Jackson)
- `application/x-java-serialized-object` - if the object is Serializable, you can use this and get a stream of the Java Serialization form of the object (obviously only makes sense for java clients with a suitable class loaded). If its not Serializable, you have to use one of the other options.

40.3 Removing data

Data can be removed at the cache key/element level, or via a whole cache name using the HTTP delete method.

40.3.1 DELETE /{cacheName}/{cacheKey}

Removes the given key name from the cache.

40.3.2 DELETE /{cacheName}

Removes ALL the entries in the given cache name (ie everything from that path down). If the operation is successful, it returns 200 code.

Make it quicker!

Set the header performAsync to true to return immediately and let the removal happen in the background.

41

41.1 Introduction

MapReduce is a programming model and a framework for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate k. MapReduce framework enables users to transparently parallelize their tasks and execute them on a large cluster of machines.

41.2 Scope and objectives

Infinispan's distributed execution framework focuses on developing a simple distributed execution framework for tasks that are already defined as Callables as well as an adapted version of Google's MapReduce framework for large computation tasks. The framework is defined as "adapted" because the input data for map reduce tasks is taken from Infinispan nodes themselves rather than using input files as it was defined by the original proposal.

41.3 Cache data as task input data

Infinispan's distributed task execution and MapReduce frameworks use data from Infinispan nodes as input for execution tasks. Most other distributed frameworks do not have that leverage and users have to specify input for distributed tasks from some well known location. Furthermore, users of Infinispan distributed execution framework do not have to configure store for intermediate and final results thus removing another layer of complexity and maintenance.

41.4 Load balanced execution - built in by default

Our distributed execution framework capitalizes on the fact input data in Infinispan data grid is already load balanced (DIST Infinispan mode). Since input data is already balanced execution tasks will be automatically balanced as well; users do not have to explicitly assign work tasks to specific Infinispan nodes. However, our framework accommodates users to specify arbitrary subset of cache data as input for distributed execution tasks.

41.5 Distributed tasks input

After an execution task is assigned for execution to a specific Infinispan node our framework provides API to access local data on Infinispan node used as an input to a distributed execution task. Access to Infinispan cache runtime environment is provided by setEnvironment callback of DistributedCallable. Users can access cache whose data is used as an input for distributed task or any other cache using a familiar CacheManager API.

41.6 Distributed task failover and migration

Distributed execution framework will support task failover. There are two orthogonal issues related to task failover:

a) Failover due to node failure where task is executing

b) Failover due to task failure (e.g. Callable task throws Exception).

Distributed tasks, taking input data from Infinispan nodes themselves, can rely on Infinispan's consistent hashing (CH) for failover of uncompleted tasks. CH based failover will migrate failed task T to cluster node(s) having a backup of input data that used to belong to failed node F. Task T executing on node F has to be re-spawned/migrated to node F' which is next to node F based on a hash wheel position. Task migration is continued until a running node - hash wheel neighbour of F is found. In another words, task migration continues until all hash wheel neghbours of F are exhausted or task completes successfully. Utilizing CH each newly spawned task migrated to node F' will be able to locate data that used to belong to failed node F. Implementation of a default node failover is planed for first release - time permitting. For distributed tasks that do not rely on pulling data from Infinispan nodes we can provide other policies: fail-fast, fail-slow etc etc.

Both node failover and task failover policy will be pluggable. Initial implementation will define interfaces to implement various node failover policies but we will provide only a simple policy that throws an exception if a node fails. In terms of task failure the default initial implementation will simply re-spawn the failed task until it reaches some failure threshold. Future implementations might migrate such a failing task to another node etc.

41.7 Distributed execution model

The main interfaces for distributed task execution are DistributedCallable and DistributedExecutorService. DistributedCallable is a subtype of the existing Callable from java.util.concurrent package; DistributedCallable can be executed in a remote JVM and receive input from Infinispan cache. Task's main algorithm could essentially remain unchanged, only the input source is changed. Exisiting Callable implementations most likely get its input in a form of some Java object/primitive while DistributedCallable gets its input from Infinispan cache. Therefore, users who have already implemented Callable interface to describe their task units would simply extend DistributedCallable and use keys from Infinispan execution environment as input for the task. Implentation of DistributedCallable can in fact continue to support implementation of an already existing Callable while simultaneously be ready for distribited execution by extending DistributedCallable.

```
public interface DistributedCallable<K, V, T> extends Callable<T> {

   /**
    * Invoked by execution environment after DistributedCallable
    * has been migrated for execution to a specific Infinispan node.
    *
    * @param cache
    *           cache whose keys are used as input data for this
    *           DistributedCallable task
    * @param inputKeys
    *           keys used as input for this DistributedCallable task
    */
   public void setEnvironment(Cache<K, V> cache, Set<K> inputKeys);

}
```

DistributedExecutorService is a simple extension of a familiar ExecutorService from java.util.concurrent package. However, advantages of DistributedExecutorService are not to be overlooked. Existing Callable tasks, instead of being executed in JDK's ExecutorService, are also eligible for execution on Infinispan cluster. Infinispan execution environment would migrate a task to execution node(s), run the task and return the result(s) to the calling node. Of course, not all Callable tasks would benefit from parallel distributed execution. Excellent candidates are long running and computationally intensive tasks that can run concurrently and/or tasks using input data that can be processed concurrently. For more details about good candidates for parallel execution and parallel algorithms in general refer to Introduction to Parallel Computing.

The second advantage of the DistributedExecutorService is that it allows a quick and simple implementation of tasks that take input from Infinispan cache nodes, execute certain computation and return results to the caller. Users would specify which keys to use as input for specified DistributedCallable and submit that callable for execution on Infinispan cluster. Infinispan runtime would locate the appriate keys, migrate DistributedCallable to target execution node(s) and finally return a list of results for each executed Callable. Of course, users can omit specifying input keys in which case Infinispan would execute DistributedCallable on all keys for a specified cache.

41.8 MapReduce model

Infinispan's own MapReduce model is an adaptation of Google's original MapReduce. There are four main components in each map reduce task: Mapper, Reducer, Collator and MapReduceTask.

Implementation of a Mapper class is a component of a MapReduceTask invoked once for each input entry K,V. Every Mapper instance migrated to an Infinispan node, given a cache entry K,V input pair transforms that input pair into intermediate key/value pair emitted into a provided Collector. Intermediate results are further reduced using a Reducer.

```
public interface Mapper<KIn, VIn, KOut, VOut> extends Serializable {

   /**
    * Invoked once for each input cache entry KIn,VOut .
    */
   void map(KIn key, VIn value, Collector<KOut, VOut> collector);
}
```

The Reducer, as its name implies, reduces a list of intermediate results from map phase of MapReduceTask. Infinispan distributed execution environment creates one instance of Reducer per execution node.

```
public interface Reducer<KOut, VOut> extends Serializable {
   /**
    * Combines/reduces all intermediate values for a particular
    * intermediate key to a single value.
    * <p>
    *
    */
   VOut reduce(KOut reducedKey, Iterator<VOut> iter);

}
```

Collator coordinates results from Reducers executed on Infinispan cluster and assembles a final result returned to an invoker of MapReduceTask. Collator is applied to final Map<KOut,VOut> result of MapReduceTask.

```
public interface Collator<KOut, VOut, R> {
   /**
    * Collates all reduced results and returns R to invoker
    * of distributed task.
    *
    * @return final result of distributed task computation
    */
   R collate(Map<KOut, VOut> reducedResults);
}
```

Finally, MapReduceTask is a distributed task uniting Mapper, Reducer and Collator into a cohesive large scale computation to be transparently parallelized across Infinispan cluster nodes. Users of MapReduceTask need to provide a cache whose data is used as input for this task. Infinispan execution environment will instantiate and migrate instances of provided mappers and reducers seamlessly across Infinispan nodes. Unless otherwise specified using onKeys method input keys filter all available key value pairs of a specified cache will be used as input data for this task.

41.9 Examples

Pi approximation can greatly benefit from parallel distributed execution in DistributedExecutorService. Recall that area of the square is Sa = 4r2 and area of the circle is Ca=pi*r2. Substituting r2 from the second equation into the first one it turns out that pi = 4 * Ca/Sa. Now, image that we can shoot very large number of darts into a square; if we take ratio of darts that land inside a circle over a total number of darts shot we will approximate Ca/Sa value. Since we know that pi = 4 * Ca/Sa we can easily derive approximate value of pi. The more darts we shoot the better approximation we get. In the example below we shoot 10 million darts but instead of "shooting" them serially we parallelize work of dart shooting across entire Infinispan cluster.

```
  public class PiAppx {

   public static void main (String [] arg){
      List<Cache> caches = ...;
      Cache cache = ...;

      int numPoints = 10000000;
      int numServers = caches.size();
      int numberPerWorker = numPoints / numServers;

      DistributedExecutorService des = new DefaultExecutorService(cache);
      long start = System.currentTimeMillis();
      CircleTest ct = new CircleTest(numberPerWorker);
      List<Future<Integer>> results = des.submitEverywhere(ct);
      int countCircle = 0;
      for (Future<Integer> f : results) {
         countCircle += f.get();
      }
      double appxPi = 4.0 * countCircle / numPoints;

      System.out.println("Distributed PI appx is " + appxPi +
      " completed in " + (System.currentTimeMillis() - start) + " ms");
   }

   private static class CircleTest implements Callable<Integer>, Serializable {

      /** The serialVersionUID */
      private static final long serialVersionUID = 3496135215525904755L;

      private final int loopCount;

      public CircleTest(int loopCount) {
         this.loopCount = loopCount;
      }

      @Override
      public Integer call() throws Exception {
         int insideCircleCount = 0;
         for (int i = 0; i < loopCount; i++) {
            double x = Math.random();
            double y = Math.random();
            if (insideCircle(x, y))
               insideCircleCount++;
         }
         return insideCircleCount;
      }

      private boolean insideCircle(double x, double y) {
         return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))
         <= Math.pow(0.5, 2);
      }
   }
}
```

Word count is a classic, if not overused, example of map/reduce paradigm. Assume we have a mapping of key-->sentence stored on Infinispan nodes. Key is a String, each sentence is also a String, and we have to count occurrence of all words in all sentences available. The implementation of such a distributed task could be defined as follows:

```
public class WordCountExample {

   /**
    * In this example replace c1 and c2 with
    * real Cache references
    *
    * @param args
    */
   public static void main(String[] args) {
      Cache c1 = null;
      Cache c2 = null;

      c1.put("1", "Hello world here I am");
      c2.put("2", "Infinispan rules the world");
      c1.put("3", "JUDCon is in Boston");
      c2.put("4", "JBoss World is in Boston as well");
      c1.put("12","JBoss Application Server");
      c2.put("15", "Hello world");
      c1.put("14", "Infinispan community");
      c2.put("15", "Hello world");

      c1.put("111", "Infinispan open source");
      c2.put("112", "Boston is close to Toronto");
      c1.put("113", "Toronto is a capital of Ontario");
      c2.put("114", "JUDCon is cool");
      c1.put("211", "JBoss World is awesome");
      c2.put("212", "JBoss rules");
      c1.put("213", "JBoss division of RedHat ");
      c2.put("214", "RedHat community");

      MapReduceTask<String, String, String, Integer> t =
         new MapReduceTask<String, String, String, Integer>(c1);
      t.mappedWith(new WordCountMapper())
         .reducedWith(new WordCountReducer());
      Map<String, Integer> wordCountMap = t.execute();
   }

   static class WordCountMapper implements Mapper<String,String,String,Integer> {
      /** The serialVersionUID */
      private static final long serialVersionUID = -5943370243108735560L;

      @Override
      public void map(String key, String value, Collector<String, Integer> c) {
         StringTokenizer tokens = new StringTokenizer(value);
         while (tokens.hasMoreElements()) {
            String s = (String) tokens.nextElement();
            c.emit(s, 1);
         }
      }
   }
```

```
    static class WordCountReducer implements Reducer<String, Integer> {
       /** The serialVersionUID */
       private static final long serialVersionUID = 1901016598354633256L;

       @Override
       public Integer reduce(String key, Iterator<Integer> iter) {
          int sum = 0;
          while (iter.hasNext()) {
             Integer i = (Integer) iter.next();
             sum += i;
          }
          return sum;
       }
    }
}
```

As we have seen it is relatively easy to specify map reduce task counting number of occurrences for each word in all sentences. Best of all result is returned to task invoker in the form of Map<KOut,VOut> rather than being written to a stream.

What if we need to find the most frequent word in our word count example? All we have to do is to define a Collator that will transform the result of MapReduceTask Map<KOut,VOut> into a String which in turn is returned to a task invoker. We can think of Collator as transformation function applied to a final result of MapReduceTask.

```
MapReduceTask<String, String, String, Integer> t =
      new MapReduceTask<String, String, String, Integer>(cache);
t.mappedWith(new WordCountMapper()).reducedWith(new WordCountReducer());
String mostFrequentWord = t.execute(
      new Collator<String,Integer,String>() {

         @Override
         public String collate(Map<String, Integer> reducedResults) {
            String mostFrequent = "";
            int maxCount = 0;
            for (Entry<String, Integer> e : reducedResults.entrySet()) {
               Integer count = e.getValue();
               if(count > maxCount) {
                  maxCount = count;
                  mostFrequent = e.getKey();
               }
            }
         return mostFrequent;
         }

      });
System.out.println("The most frequent word is " + mostFrequentWord);
```

42 Listeners and Notifications

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

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

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

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

```
@Listener
public class PrintWhenAdded {

  @CacheEntryCreated
  public void print(CacheEntryCreatedEvent event) {
    System.out.println("New entry " + event.getKey() + " created in the cache");
  }

}
```

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

42.1 Cache-level notifications

Cache-level events occur on a per-cache basis, and is global and cluster-wide. Examples of cache-level events are entries being added, removed, modified, etc. These events trigger notifications to listeners registered to a specific cache.

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

```
NOTE: In Infinispan 5.0, additional events were added.  Please refer to the <a
href="http://docs.jboss.org/infinispan/5.0/apidocs/org/infinispan/notifications/cachelistener/anno
on the org.infinispan.notifications.cachelistener.annotation package</a> for Infinispan 5.0 for
the list of cache-level notifications available in Infinispan 5.0.
```

42.2 Cache manager-level notifications

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

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

42.3 Synchronicity

By default, all notifications are dispatched in the same thread that generates the event. This means that you ***must*** write your listener such that it does not block or do anything that takes too long, as it would prevent the thread from progressing. Alternatively, you could annotate your listener as *asynchronous* , in which case a separate thread pool will be used to dispatch the notification and prevent blocking the event originating thread. To do this, simply annotate your listener such:

```
@Listener (sync = false)
public class MyAsyncListener { .... }
```

42.3.1 Asynchronous thread pool

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

42.4 Listeners on RemoteCacheManager

At the moment there is no support for registering listeners for the RemoteCacheManager. Whenever calling one of the add/remove/getListener methods on the RemoteCacheManager
an UnsupportedOperationException will be thrown. There are plans to support remote listeners in future versions; in order to be notified when this feature will be added you can watch/vote for ISPN-374.

43 Eviction

Infinispan supports eviction of entries, such that you do not run out of memory. Eviction is typically used in conjunction with a cache store, so that entries are not permanently lost when evicted, since eviction only removes entries from memory and not from cache stores or the rest of the cluster.

Passivation is also a popular option when using eviction, so that only a single copy of an entry is maintained - either in memory or in a cache store, but not both. The main benefit of using passivation over a regular cache store is that updates to entries which exist in memory are cheaper since the update doesn't need to be made to the cache store as well.

Note that eviction occurs on a *local* basis, and is not cluster-wide. Each node runs an eviction thread to analyse the contents of its in-memory container and decide what to evict. Eviction does not take into account the amount of free memory in the JVM as threshold to starts evicting entries. You have to set maxEntries attribute of the eviction element to be greater than zero in order for eviction to be turned on. If maxEntries is too large you can run out of memory. maxEntries attribute will probably take some tuning in each use case.

43.1 Eviction in 5.2

Eviction is configured by adding the <eviction /> element to your <default /> or <namedCache /> configuration sections or using EvictionConfigurationBuilder API programmatic approach. LIRS is default eviction algorithm in Infinispan 5.2 release.

All cache entry are evicted by piggybacking on user threads that are hitting the cache. Periodic pruning of expired cache entries from cache is done on a dedicated thread which is turned on by enabling reaper in expiration configuration element/API.

43.1.1 NONE

This eviction strategy effectively disables the eviction thread.

43.1.2 UNORDERED

UNORDERED eviction strategy is a legacy eviction strategy that has been deprecated. If UNORDERED strategy is specified LRU eviction algorithm will be used.

43.1.3 LRU

If LRU eviction is used cache entries are selected for eviction using a well known least-recently-used pattern.

43.1.4 LIRS

LRU eviction algorithm, although simple and easy to understand, under performs in cases of weak access locality (one time access entries are not timely replaced, entries to be accessed soonest are unfortunately replaced, and so on). Recently, a new eviction algorithm - LIRS has gathered a lot of attention because it addresses weak access locality shortcomings of LRU yet it retains LRU's simplicity. Eviction in LIRS algorithm relies on history information of cache entries accesses using so called Inter-Reference Recency (a.k.a IRR) and the Recency. The IRR of a cache entry A refers to number of other distinct entries accessed between the last two consecutive accesses to cache entry A, while recency refers to the number of other entries accessed from last reference to A up to current time point. If we relied only on cache recency we would essentially have LRU functionality. However, in addition to recency LIRS tracks elements that are in low IRR and high IRR, aptly named LIR and HIR cache entry blocks respectively. LIRS eviction algorithm essentially keeps entries with a low IRR in the cache as much as possible while evicting high IRR entries if eviction is required. If recency of a LIR cache entry increases to a certain point and entry in HIR gets accessed at a smaller recency than that of the LIR entry, the LIR/HIR statuses of the two blocks are switched. Entries in HIR may be evicted regardless of its recency, even if element was recently accessed.

43.2 Configuration and defaults in 5.2.x

By default when no <eviction> element is specified, no eviction takes place.

In case there is an eviction element, this table describes behaviour of eviction based on information provided in the xml configuration ("-" in Supplied maxEntries or Supplied strategy column means that the attribute wasn't supplied)

Supplied maxEntries	**Supplied strategy**	**Example**	**Eviction behaviour**
-	-	<eviction />	no eviction
> 0	-	<eviction maxEntries="100" />	the strategy defaults to LIRS and eviction takes place
> 0	NONE	<eviction maxEntries="100" strategy="NONE" />	the strategy defaults to LIRS and eviction takes place
> 0	!= NONE	<eviction maxEntries="100" strategy="LRU" />	eviction takes place with defined strategy
0	-	<eviction maxEntries="0" />	no eviction
0	NONE	<eviction maxEntries="0" strategy="NONE" />	no eviction
0	!= NONE	<eviction maxEntries="0" strategy="LRU" />	ConfigurationException
< 0	-	<eviction maxEntries="-1" />	no eviction
< 0	NONE	<eviction maxEntries="-1" strategy="NONE" />	no eviction
< 0	!= NONE	<eviction maxEntries="-1" strategy="LRU" />	ConfigurationException

43.3 Advanced Eviction Internals

Implementing eviction in a scalable, low lock contention approach while at the same time doing meaningful selection of entries for eviction is not an easy feat. Data container needs to be locked until appropriate eviction entries are selected. Having such a lock protected data container in turn causes high lock contention offsetting any eviction precision gained by sophisticated eviction algorithms. In order to get superior throughput while retaining high eviction precision both low lock contention data container and high precision eviction algorithm implementation are needed. Infinispan evicts entries from cache on a segment level (segments similar to ConcurrentHashMap), once segment is full entries are evicted according to eviction algorithm. However, there are two drawbacks with this approach. Entries might get evicted from cache even though maxEntries has not been reached yet and maxEntries is a theoretical limit for cache size but in practical terms it will be slightly less than maxEntries. For more details refer to Infinispan eviction design.

43.4 Expiration

Similar to, but unlike eviction, is expiration. Expiration allows you to attach lifespan and/or maximum idle times to entries. Entries that exceed these times are treated as invalid and are removed. When removed expired entries are not passivated like evicted entries (if passivation is turned on).

✓ Unlike eviction, expired entries are removed globally - from memory, cache stores, and cluster-wide.

By default entries created are immortal and do not have a lifespan or maximum idle time. Using the cache API, mortal entries can be created with lfiespans and/or maximum idle times. Further, default lifespans and/or maximum idle times can be configured by adding the <expiration /> element to your <default /> or <namedCache /> configuration sections.

43.5 Difference between Eviction and Expiration

Both Eviction and Expiration are means of cleaning the cache of unused entries and thus guarding the heap against OutOfMemory exceptions, so now a brief explanation of the difference.

With **eviction** you set **maximal number of entries** you want to keep in the cache and if this limit is exceeded, some candidates are found to be removed according to a choosen **eviction strategy** (LRU, LIRS, etc...). Eviction can be setup to work with passivation (evicting to a cache store).

With **expiration** you set **time criteria** for entries, **how long you want to keep them** in cache. Either you set maximum **lifespan** of the entry - time it is allowed to stay in the cache or **maximum idle time**, time it's allowed to be untouched (no operation performed with given key).

44 ServerHinting

44.1 What is server hinting?

The motivations behind this feature is to ensure when using distribution, backups are not picked to reside on the same physical server, rack or data centre. For obvious reasons it doesn't work with total replication.

44.2 Version

Server hinting was added in Infinspan 4.2.0 "Ursus". You'll need this release or later in order to use it.

44.3 Configuration

The hints are configured at transport level:

```
<transport     clusterName = "MyCluster"
     machineId = "LinuxServer01"
     rackId = "Rack01"
     siteId = "US-WestCoast" />
```

Following topology hints can be specified:

- machineId - this is probably the most useful, to disambiguate between multiple JVM instances on the same node, or even multiple virtual hosts on the same physical host.
- rackId - in larger clusters with nodes occupying more than a single rack, this setting would help prevent backups being stored on the same rack.
- siteId - to differentiate between nodes in different data centres replicating to each other.   All of the above are optional, and if not provided, the distribution algorithms provide no guarantees that backups will not be stored in instances on the same host/rack/site.

44.4 Algorithm

This is an advanced topic, useful e.g. if you need to change distribution behaviour.

The consistent hash beyond this implementation is wheel based. Conceptually this works as follows: each node is placed on a wheel ordered by the hash code of its address. When an entry is added its owners are chosen using this algorithm:

- key's hash code is calculated
- the first node on the wheel with a value grater than key's hash code is the first owner
- for subsequent nodes, walk clockwise and pick nodes that have a different site id
- if not enough nodes found repeat walk again and pick nodes that have different site id and rack id
- if not enough nodes found repeat walk again and pick nodes that have different site id, rack id and machine id
- Ultimately cycle back to the first node selected, don't discard any nodes, regardless of machine id/rack

45 Java Hot Rod client

45.1 Introduction

Hot Rod is a binary, language neutral protocol. This article explains how a Java client can interact with a server via the Hot Rod protocol. A reference implementation of the protocol written in Java can be found in all Infinispan distributions since 4.1, and this article focuses on the capabilities of this java client.

45.2 Basic API

Bellow is a sample code snippet on how the client API can be used to store or retrieve information from a Hot Rod server using the Java Hot Rod client. It assumes that a Hot Rod server has been started bound to the default location (localhost:11222)

```
//API entry point, by default it connects to localhost:11222
CacheContainer cacheContainer = new RemoteCacheManager();

//obtain a handle to the remote default cache
Cache<String, String> cache = cacheContainer.getCache();

//now add something to the cache and make sure it is there
cache.put("car", "ferrari");
assert cache.get("car").equals("ferrari");

//remove the data
cache.remove("car");
assert !cache.containsKey("car") : "Value must have been removed!";
```

The client API maps the local API: RemoteCacheManager corresponds to DefaultCacheManager (both implement CacheContainer). This common API facilitates an easy migration from local calls to remote calls through Hot Rod: all one needs to do is switch between DefaultCacheManager and RemoteCacheManager- which is further simplified by the common CacheContainer interface that both inherit.
Starting from Infinispan 5.2, all keys can be retrieved from the remote cache (whether it's local, replicated, or distributed) by using keySet() method. If the remote cache is a distributed cache, the server will start a map/reduce job to retrieve all keys from clustered nodes, and return all keys to the client. Please use this method with care if there are large number of keys.

```
Set keys = remoteCache.keySet();
```

45.3 Versioned API

A RemoteCacheManager provides instances of RemoteCache interface that represents a handle to the named or default cache on the remote cluster. API wise, it extends the Cache interface to which it also adds some new methods, including the so called versioned API. Please find below some examples of this API but to understand the motivation behind it, make sure you read this article.

The code snippet bellow depicts the usage of these versioned methods:

```
// To use the versioned API, remote classes are specifically needed
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
RemoteCache<String, String> cache = remoteCacheManager.getCache();

remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");

// removal only takes place only if the version has not been changed
// in between. (a new version is associated with 'car' key on each change)
assert remoteCache.remove("car", valueBinary.getVersion());
assert !cache.containsKey("car");
```

In a similar way, for replace:

```
remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");
assert remoteCache.replace("car", "lamborghini", valueBinary.getVersion());
```

For more details on versioned operations refer to RemoteCache's javadoc.

45.4 Async API

This cool feature is "borrowed" from the Infinispan core and it is largely discussed here.

45.5 Unsupported methods

Some of the Cache methods are not being supported by the RemoteCache. Calling one of these methods results in an UnsupportedOperationException being thrown. Most of these methods do not make sense on the remote cache (e.g. listener management operations), or correspond to methods that are not supported by local cache as well (e.g. containsValue). Another set of unsupported operations are some of the atomic operations inherited from ConcurrentMap:

```
boolean remove(Object key, Object value);
boolean replace(Object key, Object value);
boolean replace(Object key, Object oldValue, Object value);
```

RemoteCache offers alternative versioned methods for these atomic operations, that are also network friendly, by not sending the whole value object over the network, but a version identifier. See the section on versioned API.
Each one of these unsupported operation is documented in the RemoteCache javadoc.

45.6 Return values

There is a set of methods that alter an cached entry and return the previous existing value, e.g.:

```
V remove(Object key);
V put(K key, V value);
```

By default on RemoteCache, these operations return null even if such a previous value exists. This approach reduces the amount of data sent over the network. However, if these return values are needed they can be enforced on an per invocation basis using flags:

```
cache.put("aKey", "initialValue");
assert null == cache.put("aKey", "aValue");
assert "aValue".equals(cache.withFlags(Flag.FORCE_RETURN_VALUE).put("aKey",
   "newValue"));
```

This default behaviour can can be changed through force-return-value=true configuration parameter (see configuration section bellow).

45.7 Intelligence

HotRod defines three level of intelligence for the clients:

- basic client, interested in neither cluster nor hash information
- topology-aware client, interested in cluster information
- hash-distribution-aware client, that is interested in both cluster and hash information

The java client supports all 3 levels of intelligence. It is transparently notified whenever a new server is added/removed from the HotRod cluster. At startup it only needs to know the address of one HotRod server (ip:host). On connection to the server the cluster topology is piggybacked to the client, and all further requests are being dispatched to all available servers. Any further topology change is also piggybacked.

Hash-distribution-aware client is discussed in the next section.

45.8 Hash-distribution-aware client

Another aspect of the 3rd level of intelligence is the fact that it is hash-distribution-aware. This means that, for each operation, the client chooses the most appropriate remote server to go to: the data owner. As an example, for a put(k,v) operation, the client calculates k's hash value and knows exactly on which server the data resides on. Then it picks up a tcp connection to that particular server and dispatches the operation to it. This means less burden on the server side which would otherwise need to lookup the value based on the key's hash. It also results in a quicker response from the server, as an additional network roundtrip is skipped. This hash-distribution-aware aspect is only relevant to the distributed HotRod clusters and makes no difference for replicated server deployments.

45.9 Request Balancing

Request balancing is only relevant when the server side is configured with replicated infinispan cluster (on distributed clusters the hash-distribution-aware client logic is used, as discussed in the previos paragraph). Because the client is topology-aware, it knows the list of available servers at all the time. Request balancing has to do with how the client dispatches requests to the available servers.

The default strategy is round-robin: requests are being dispatched to all existing servers in a circular manner. E.g. given a cluster of servers {s1, s2, s3} here is how request will be dispatched:

```
CacheContainer cacheContainer = new RemoteCacheManager();
Cache<String, String> cache = cacheContainer.getCache();

cache.put("key1", "aValue"); //this goes to s1
cache.put("key2", "aValue"); //this goes to s2
String value = cache.get("key1"); //this goes to s3

cache.remove("key2"); //this is dispatched to s1 again, and so on...
```

Custom types of balancing policies can defined by implementing the RequestBalancingStrategy and by specifying it through the infinispan.client.hotrod.request-balancing-strategy configuration property. Please refer to configuration section for more details on this.

45.10 Persistent connections

In order to avoid creating a TCP connection on each request (which is a costly operation), the client keeps a pool of persistent connections to all the available servers and it reuses these connections whenever it is possible. The validity of the connections is checked using an async thread that iterates over the connections in the pool and sends a HotRod ping command to the server. By using this connection validation process the client is being proactive: there's a hight chance for broken connections to be found while being idle in the pool and no on actual request from the application.

The number of connections per server, total number of connections, how long should a connection be kept idle in the pool before being closed - all these (and more) can be configured. Please refer to the javadoc of RemoteCacheManager for a list of all possible configuration elements.

45.11 Marshalling data

The Hot Rod client allows one to plug in a custom marshaller for transforming user objects into byte arrays and the other way around. This transformation is needed because of Hot Rod's binary nature - it doesn't know about objects.

The marshaller can be plugged through the "marshaller" configuration element (see Configuration section): the value should be the fully qualified name of a class implementing the Marshaller interface. This is a optional parameter, if not specified it defaults to the GenericJBossMarshaller - a highly optimized implementation based on the JBoss Marshalling library.

Since version 5.0, there's a new marshaller available to Java Hot Rod clients based on Apache Avro which generates portable payloads. You can find more information about it here

45.12 Statistics

Various server usage statistics can be obtained through the RemoteCache.stats() method. This returns an ServerStatistics object - please refer to javadoc for details on the available statistics.

45.13 Configuration

All the configurations are passed to the RemoteCacheManager's constructor as key-value pairs, through an instance of java.util.Properties or reference to a .properties file. Please refer to the javadoc of RemoteCacheManager for a exhaustive list of the possible configuration elements.

45.14 Multi-Get Operations

The Java Hot Rod client does not provide multi-get functionality out of the box but clients can build it themselves with the given APIs. More information on this topic can be found in the Hot Rod protocol article.

45.15 More info

It is highly recommended to read the following Javadocs (this is pretty much all the public API of the client):

- RemoteCacheManager
- RemoteCache

46 Configuring cache

Infinispan offers both Configuring Cache declaratively and Configuring cache programmatically configuration approaches.

Declarative configuration comes in a form of XML document that adheres to a provided Infinispan configuration XML schema. Every aspect of Infinispan that can be configured declaratively can also be configured programmatically. In fact, declarative configuration, behind the scenes, invokes programmatic configuration API as the XML configuration file is being processed. One can even use combination of these approaches. For example, you can read static XML configuration files and at runtime programmatically tune that same configuration. Or you can use a certain static configuration defined in XML as a starting point or template for defining additional configurations in runtime.

There are two main configuration abstractions in Infinispan: global and default configuration.

Global cache configuration defines global settings shared among all cache instances created by a single CacheManager. Shared resources like thread pools, serialization/marshalling settings, transport and network settings, JMX domains are all part of global configuration.

Default cache configuration is more specific to actual caching domain itself. It specifies eviction, locking, transaction, clustering, cache store settings etc. The default cache can be retrieved via the CacheManager.getCache() API. However, the real power of default cache mechanism comes to light when used in conjuction with *named caches* . Named caches have the same XML schema as the default cache. Whenever they are specified, named caches inherit settings from the default cache while additional behavior can be specified or overridden. Named caches are retrieved via CacheManager.getCache(String name) API. Therefore, note that the *name* attribute of named cache is both mandatory and unique for every named cache specified.

Do not forget to refer to Infinispan configuration reference for more details.

For more details, refer to the following documents:

- Configuring Cache declaratively
- Configuring cache programmatically
- Configuration Migration Tools

47 Write-Through And Write-Behind Caching

47.1 Introduction

Infinispan can optionally be configured with one or several cache stores allowing it to store data in a persistent location such as shared JDBC database, a local filesystem...etc. Infinispan can handle updates to the cache store in two different ways:

- Write-Through (Synchronous)
- Write-Behind (Asynchronous)

47.2 Write-Through (Synchronous)

In this mode, which is supported in version 4.0, when clients update a cache entry, i.e. via a Cache.put() invocation, the call will not return until Infinispan has gone to the underlying cache store and has updated it. Normally, this means that updates to the cache store are done within the boundaries of the client thread.

The main advantage of this mode is that the cache store is updated at the same time as the cache, hence the cache store is consistent with the cache contents. On the other hand, using this mode reduces performance because the latency of having to access and update the cache store directly impacts the duration of the cache operation.

Configuring a write-through or synchronous cache store does not require any particular configuration option. By default, unless marked explicitly as write-behind or asynchronous, all cache stores are write-through or synchronous. Please find below a sample configuration file of a write-through unshared local file cache store:

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:5.0">
  <global />
  <default />
  <namedCache name="persistentCache">
    <loaders shared="false">
      <loader
          class="org.infinispan.loaders.file.FileCacheStore"
          fetchPersistentState="true" ignoreModifications="false"
          purgeOnStartup="false">
        <properties>
          <property name="location" value="${java.io.tmpdir}" />
        </properties>
      </loader>
    </loaders>
  </namedCache>
</infinispan>
```

47.3 Write-Behind (Asynchronous)

In this mode, updates to the cache are asynchronously written to the cache store. Normally, this means that updates to the cache store are done by a separate thread to the client thread interacting with the cache.

One of the major advantages of this mode is that the performance of a cache operation does not get affected by the update of the underlying store. On the other hand, since the update happens asynchronously, there's a time window during the which the cache store can contain stale data compared to the cache. Even within write-behind, there are different strategies that can be used to store data:

47.4 Unscheduled Write-Behind Strategy

In this mode, which is supported in version 4.0, Infinispan tries to store changes as quickly as possible by taking the pending changes and applying them in paralel. Normally, this means that there are several threads waiting for modifications to occur and once they're available, they apply them to underlying cache store.

This strategy is suited for cache stores with low latency and cheap operation cost. One such example would a local unshared file based cache store, where the cache store is local to the cache itself. With this strategy, the window of inconsistency between the contents of the cache and the cache store are reduced to the lowest possible time. Please find below a sample configuration file of this strategy:

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:5.0">
  <global />
  <default />
  <namedCache name="persistentCache">
    <loaders shared="false">
      <loader
          class="org.infinispan.loaders.file.FileCacheStore"
          fetchPersistentState="true" ignoreModifications="false"
          purgeOnStartup="false">
        <properties>
          <property name="location" value="${java.io.tmpdir}" />
        </properties>
        <!-- write-behind configuration starts here -->
        <async enabled="true" threadPoolSize="10" />
        <!-- write-behind configuration ends here -->
      </loader>
    </loaders>
  </namedCache>
</infinispan>
```

47.5 Scheduled Write-Behind Strategy

First of all, please note that this strategy is not included in version 4.0 but it will be implemented at a later stage. ISPN-328 has been created to track this feature request. If you want it implemented, please vote for it and don't forget to watch it to be notified of any changes. The following explanation refers to how we envision it to work.

In this mode, Infinispan would periodically store changes to the underlying cache store. The periodicity could be defined in seconds, minutes, days...etc.

Since this strategy is oriented at cache stores with high latency or expensive operation cost, it makes sense to coalesce changes, so that if there are multiple operations queued on the same key, only the latest value is applied to cache store. With this strategy, the window of inconsistency between the contents of the cache and the cache store depends on the delay or periodicity configured. The higher the periodicity, the higher the chance of inconsistency.

48 Using Hot Rod Server

48.1 Introduction

Starting with version 4.1, Infinispan distribution contains a server module that implements Infinispan's custom binary protocol called Hot Rod. The protocol was designed to enable faster client/server interactions compared to other existing text based protocols and to allow clients to make more intelligent decisions with regards to load balancing, failover and even data location operations.

48.2 Starting an Infinispan Hot Rod server

The simplest way to start up Hot Rod server is to simply unzip the all distribution and either run the startServer.bat or startServer.sh script passing 'hotrod' as the protocol to run. For example:

```
[g@eq]~/infinispan-4.1.0-SNAPSHOT% ./bin/startServer.sh -r hotrod
```

When the script is called without any further parameters, the started Hot Rod server bounds to port 11222 (Infinispan 4.2.0.BETA1 or earlier listened on port 11311 by default) on localhost (127.0.0.1). If no further parameters is given at startup, this means that any cache instance queried will be based on the default cache instance which will be a local (unclustered) Infinispan cache instance configured with default values. Please visit the Configuration Reference Guide for more information on these values. If you want to use non-default values, please check the next section on command line options.

48.2.1 Command Line Options

You can optionally pass a set of parameters to the Hot Rod server that allow you to configure different parts of the server. You can find detailed information in the server command line options article.

So, following the discussion earlier on configuring a Hot Rod server with a custom Infinispan configuration, you'd use -c parameter to pass the corresponding file. Hot Rod clients could then send requests to specific cache instances whose name match one of the names in the custom configuration file.

48.2.2 Predefining Hot Rod Caches

In order to provide a more consistent experience when interacting with Hot Rod servers and avoid issues related to lazily started cache instances, on startup, Hot Rod server starts all defined caches in the Infinispan configuration file including the default cache. If no configuration file was provided, only the default cache will be started. Any request to an undefined cache will be rejected by the server.

So, as user, this means that any caches you interact with must be defined in the Infinispan configuration file by providing a namedCache XML element entry for each of them. Besides, as explained the 'Cache Name' section of the Hot Rod protocol, you can also interact with the default cache by passing an empty cache name.

49 Cassandra CacheStore

Infinispan's CassandraCacheStore leverages Apache Cassandra's distributed database architecture to provide a virtually unlimited, horizontally scalable persistent store for Infinispan's caches.

49.1 Version

The Cassandra CacheStore was added in Infinspan 4.2.0 "Ursus". You'll need this release or later in order to use it.

49.2

Configuration

In order to use this CacheStore you need to create an appropriate keyspace on your Cassandra database. The following storage-conf.xml excerpt shows the declaration of the keyspace:

```
<Keyspace Name="Infinispan">
      <ColumnFamily CompareWith="BytesType" Name="InfinispanEntries" KeysCached="10%" />
      <ColumnFamily CompareWith="LongType" Name="InfinispanExpiration" KeysCached="10%"
ColumnType="Super" CompareSubcolumnsWith="BytesType"/>
      <ColumnFamily CompareWith="BytesType" Name="InfinispanEntries" KeysCached="0" />
      <ColumnFamily CompareWith="LongType" Name="InfinispanExpiration" KeysCached="0"
ColumnType="Super" CompareSubcolumnsWith="BytesType"/>

<ReplicaPlacementStrategy>org.apache.cassandra.locator.RackUnawareStrategy</ReplicaPlacementStrate
<ReplicationFactor>2</ReplicationFactor>
      <EndPointSnitch>org.apache.cassandra.locator.EndPointSnitch</EndPointSnitch>
</Keyspace>
```

The important bits are the CompareWith, ColumnType and CompareSubColumnsWith declarations. Everything else can be changed at will. You can also have more than one Keyspace to accomodate for multiple caches.
You then need to add an appropriate cache declaration to your infinispan.xml (or whichever file you use to configure Infinispan):

```
<namedCache name="cassandraCache">
        <loaders passivation="false" shared="true" preload="false">
                <loader
                        class="org.infinispan.loaders.cassandra.CassandraCacheStore"
                        fetchPersistentState="true" ignoreModifications="false"
                        purgeOnStartup="false">
                        <properties>
                                <property name="host" value="localhost" />
                                <property name="keySpace" value="Infinispan" />
                                <property name="entryColumnFamily" value="InfinispanEntries" />
                                <property name="expirationColumnFamily"
value="InfinispanExpiration" />
                                <property name="sharedKeyspace" value="false" />
                                <property name="readConsistencyLevel" value="ONE" />
                                <property name="writeConsistencyLevel" value="ONE" />
                                <property name="configurationPropertiesFile"
value="cassandrapool.properties" />
                                <property name="keyMapper"
value="org.infinispan.loaders.keymappers.DefaultTwoWayKey2StringMapper" />
                        </properties>
                </loader>
        </loaders>
</namedCache>
```

It is important the the shared property on the loader element is set to true because all the Infinispan nodes will share the same Cassandra cluster.
Since the Cassandra client library doesn't provide connection pooling, a separate project has been created at http://github.com/tristantarrant/cassandra-connection-pool.
Configuration of the connection pool can be done by creating an appropriate properties file and specifying its name in the configuration (*configurationPropertiesFile*).
The following is an example file:

```
socketTimeout = 5000
initialSize = 10
maxActive = 100
maxIdle = 20
minIdle = 10
maxWait = 30000testOnBorrow = false
testOnReturn = false
testWhileIdle = false
timeBetweenEvictionRunsMillis = 5000removeAbandoned = false
removeAbandonedTimeout = 60
logAbandoned = false
```

Here is a description of the various properties which can be configured:

host	a hostname (or a comma-separated list of hostnames)
port	the thrift port (usually 9160)
keySpace	the keyspace to use (defaults to Infinispan)
entryColumnFamily	the column family where entry values will be stored. Make sure you configure your keyspace accordingly (defaults to InfinispanEntries)
expirationColumnFamily	the column family where element expiration information is stored. Make sure your (defaults to InfinispanExpiration)
sharedKeyspace	whether multiple caches are to be stored in a single keyspace. In this case the cache name is prefixed to the key entries (defaults to false).
readConsistencyLevel	The consistency level to use when reading from Cassandra. Possible values are ONE, QUORUM, DCQUORUM, ALL. For an explanation of these refer to the Cassandra API documentation. (defaults to ONE)
writeConsistencyLevel	The consistency level to use when writing to Cassandra. Possible values are ZERO, ANY, ONE, QUORUM, DCQUORUM, ALL. For an explanation of these refer to the Cassandra API documentation. (defaults to ONE)
keyMapper	A class which implements Infinispan's TwoWayKey2StringMapper interface for mapping cache keys to Cassandra row keys (defaults to org.infinispan.loaders.keymappers.DefaultTwoWayKey2StringMapper)

50 Infinispan Custom Interceptors

50.1 Introduction

It is possible to add custom interceptors to Infinispan, both declaratively and programatically. Custom interceptors are an way of extending Infinispan by being able to influence or respond to any modifications to cache. Example of such modifications are: elements are added/removed/updated or transactions are committed. For a detailed list refer to CommandInterceptor API.

50.2 Adding custom interceptors declaratively

Custom interceptors can be added on a per named cache basis. This is because each named cache have it's own interceptor stack. Following xml snippet depicts the ways in which an custom interceptor can be added.

```
   <namedCache name="cacheWithCustomInterceptors">
      <!--
      Define custom interceptors.  All custom interceptors need to extend
org.jboss.cache.interceptors.base.CommandInterceptor
      -->
      <customInterceptors>
         <interceptor position="FIRST" class="com.mycompany.CustomInterceptor1">
            <properties>
               <property name="attributeOne" value="value1" />
               <property name="attributeTwo" value="value2" />
            </properties>
         </interceptor>
         <interceptor position="LAST" class="com.mycompany.CustomInterceptor2"/>
         <interceptor index="3" class="com.mycompany.CustomInterceptor1"/>
         <interceptor before="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor2"/>
         <interceptor after="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor1"/>
      </customInterceptors>
   </namedCache>
```

Adding custom interceptors programatically
In order to do that one needs to obtain a reference to the AdvancedCache. This can be done ass follows:

```
CacheManager cm = getCacheManager();//magic
Cache aCache = cm.getCache("aName");
AdvancedCache advCache = aCache.getAdvancedCache();
```

Then one of the *addInterceptor()* methods should be used to add the actual interceptor. For further documentation refer to AdvancedCache javadoc.

50.3 Custom interceptor design

- Custom interceptors must extend BaseCustomInterceptor
- Custom interceptors must declare a public, empty constructor to enable construction.
- Custom interceptors will have setters for any property defined through property tag.

51 Talking To Infinispan Memcached Servers From Non-Java Clients

This wiki shows how to talk to Infinispan memcached server via non-java client, such as a python script.

51.1 Multi Clustered Server Tutorial

The second example showcases the distribution capabilities of Infinispan memcached severs that are not available in the original memcached implementation.

- Start first Infinispan memcached server giving to it a port number and an Infinispan configuration supporting distribution. This configuration is the same one used for the GUI demo:

```
./bin/startServer.sh -r memcached -c etc/config-samples/gui-demo-cache-config.xml -p 12211
```

- Start a second Infinispan memcached server passing to it a different port number:

```
./bin/startServer.sh -r memcached -c etc/config-samples/gui-demo-cache-config.xml -p 13211
```

- Execute test_memcached_write.py script which basically executes several write operations againts the Infinispan memcached server bound to port 12211. If the script is executed successfully, you should see an output similar to this:

```
<li>Connecting to 127.0.0.1:12211</li><li>Testing set ['Simple_Key': Simple value] ...
OK</li><li>Testing set ['Expiring_Key' : 999 : 3] ... OK</li><li>Testing increment 3 times
['Incr_Key' : starting at 1 ]</li><li>Initialise at 1 ... OK</li><li>Increment by one ...
OK</li><li>Increment again ... OK</li><li>Increment yet again ... OK</li><li>Testing
decrement 1 time ['Decr_Key' : starting at 4 ]</li><li>Initialise at 4 ...
OK</li><li>Decrement by one ... OK</li><li>Testing decrement 2 times in one call
['Multi_Decr_Key' : 3 ]</li><li>Initialise at 3 ... OK</li><li>Decrement by 2 ... OK</li>
```

- Execute test_memcached_read.py script which connects to server bound to 127.0.0.1:13211 and verifies that it can read the data that was written by the writer script to the first server. If the script is executed successfully, you should see an output similar to this:

```
<li>Connecting to 127.0.0.1:13211</li><li>Testing get ['Simple_Key'] should return Simple
value ... OK</li><li>Testing get ['Expiring_Key'] should return nothing...
OK</li><li>Testing get ['Incr_Key'] should return 4 ... OK</li><li>Testing get
['Decr_Key'] should return 3 ... OK</li><li>Testing get ['Multi_Decr_Key'] should return 1
... OK</li>
```

52 Plugging Infinispan With User Defined Externalizers

52.1 Introduction

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

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

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

52.2 Benefits of Externalizers

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

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

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

52.2.1 User Friendly Externalizers

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

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

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

   final String name;
   final int age;

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

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

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

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

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

- Due to several constraints of the model, such as support different versions of the same class or the need to marshall the Externalizer class, the payload sizes generated via this method are not the most efficient.
- This model requires for the marshalled class to be annotated with {@link SerializeWith} but a user might need to provide an Externalizer for a class for which source code is not available, or for any other constraints, it cannot be modified.
- The use of annotations by this model might be limiting for framework developers or service providers that try to abstract lower level details, such as the marshalling layer, away from the user.

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

52.2.2 Advanced Externalizers

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

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

public class Person {

   final String name;
   final int age;

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

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

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

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

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

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

Linking Externalizers with Marshaller Classes

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

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

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

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

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

Externalizer Identifier

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

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

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

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

Registering Advanced Externalizers

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

XML:

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

Programmatically:

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

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

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer id="123"
                              externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

Programmatically:

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

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

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

Preassigned Externalizer Id Ranges

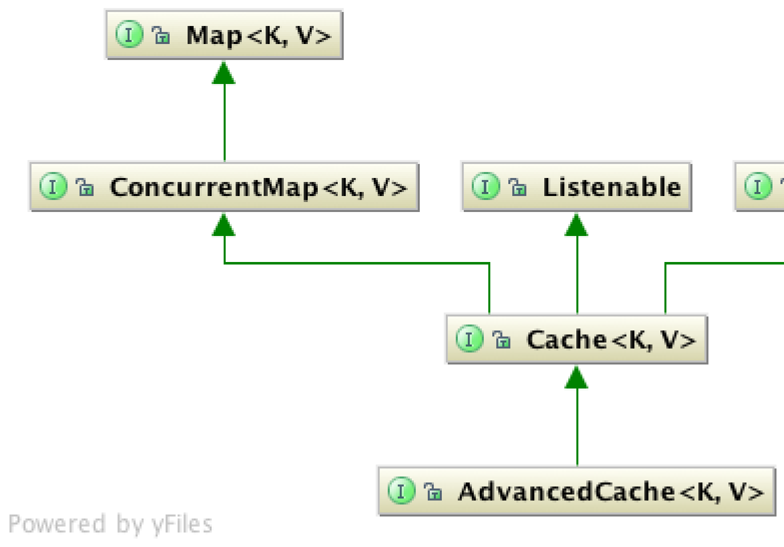
This is the list of Externalizer identifiers that are used by Infinispan based modules or frameworks. Infinispan users should avoid using ids within these ranges.

Infinispan Tree Module:	1000 - 1099
Infinispan Server Modules:	1100 - 1199
Hibernate Infinispan Second Level Cache:	1200 - 1299
Infinispan Lucene Directory:	1300 - 1399
Hibernate OGM:	1400 - 1499
Hibernate Search:	1500 - 1599

53 Using the Cache API

53.1 The Cache interface

Infinispan exposes a simple, JSR-107 compliant Cache interface.



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

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

53.1.1 Limitations of Certain Map Methods

Certain methods exposed in Map have certain limitations when used with Infinispan, such as size(), values(), keySet() and entrySet(). Specifically, these methods are *unreliable* and only provide a best-effort guess. They do not acquire locks, either local or global, and concurrent modifications, additions and removals will not be considered in the result of any of these calls. Further, they only operate on the local data container, and as such, do not give you a global view of state.

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

53.1.2 Mortal and Immortal Data

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

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

53.1.3 Example of Using Expiry and Mortal Data

See this page for an example of using mortal data with Infinispan

53.1.4 putForExternalRead operation

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

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

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

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

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

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

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

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

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

53.2 The AdvancedCache interface

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

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

53.2.1 Flags

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

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

53.2.2 Custom Interceptors

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

For more information on writing custom interceptors, see /javascript:;

54 Local mode cache

Introduction

Even though Infinispan's biggest potential is as distributed, in-memory data grid platform, one aspect of it often gets overlooked - it can be used as a standalone cache node. But why would anyone use Infinispan over, say, a ConcurrentHashMap? Here are some reasons:

—

- **Eviction.** Built-in eviction ensures you don't run out of memory.
- **Write-through and write-behind caching.** Going beyond memory and onto disk (or any other pluggable CacheStore) means that your state survives restarts, and preloaded hot caches can be configured.
- **JTA support and XA compliance.** Participate in ongoing transactions with any JTA-compliant transaction manager.
- **MVCC-based concurrency.** Highly optimized for fast, non-blocking readers.
- **Manageability.** Simple JMX or rich GUI management console via JOPR, you have a choice.
- **Not just for the JVM.** RESTful API, and upcoming client/server modules speaking Memcached and HotRodprotocols help non-JVM platforms use Infinispan.
- **Cluster-ready.** Should the need arise.

  —

  So how do you get started with Infinispan in local mode? The simplest configuration file containing just

```
<infinispan/>
```

is enough to get you started, or you can create DefaultCacheManager with *no -argument* constructor. Either approach creates local default cache.

All the features above are exposed via an easy-to-use Cache interface, which extends ConcurrentMap and is compatible with many other cache systems. Infinispan even ships with migration tools to help you move off other cache solutions onto Infinispan, whether you need a cache to store data retrieved remotely or simply as a 2nd level cache for Hibernate.

Performance

In the process of testing and tuning Infinispan on very large clusters, we have started to put together a benchmarking framework. As a part of this framework, we have the ability to measure cache performance in standalone, local mode. We compared Infinispan 4.0 in local mode against the latest JBoss Cache release (3.2.2.GA) and EHCache (1.7.2). Some background on the tests:

- Used a latest snapshot of the CacheBenchFwk
- Run on a RHEL 5 server with 4 Intel Xeon cores, 4GB of RAM
- Sun JDK 1.6.0_18, with -Xms1g -Xmx1g

- Test run on a single node, with 25 concurrent threads, using randomly generated Strings as keys and values and a 1kb payload for each entry, with a 80/20 read/write ratio.
- Performance measured in transactions per second (higher = better).

In summary, what we have here is that when run in local mode, Infinispan is a high-performance standalone caching engine which offers a rich set of features while still being trivially simple to configure and use.

55 Infinispan Command-line Console

Documentation for ispncon version 0.8.1

55.1 What is ispncon ?

Ispncon (Infinispan Command-Line Console) is a simple command-line interface to infinispan cache written in python. It accesses the cache in client/server fashion, whereby infinispan server modules (Hot Rod, Memcached, REST) have to be properly configured. Once you have a running instance of infinispan with one of the server modules, you can issue simple cache requests via command line, like:

```
$ ispncon put keyA "Sample value under keyA"
$ ispncon get keyA
$ ispncon delete keyA
```

Furthermore it creates an abstraction above the specifics of the client/server protocol you use to access the cache. The basic commands look the same whether you use Hot Rod, memcached or REST. The client/protocol specifics are handled via protocol-specific configuration options or command arguments. Your shell scripts should look basically the same for all the client modes, if you don't use any protocol specific features.

Behind the scenes the console tool uses existing python clients to handle the communication with particular server modules. The implementations are listed in the following table:

Client	Client implementation	Web
Hot Rod	python-client for hotrod	https://github.com/infinispan/python-client
Memcached	python-memcached	https://launchpad.net/python-memcached
REST	httplib (standard module for python)	http://docs.python.org/library/httplib.html

The source code and issue reporting for ispncon can be found here: https://github.com/infinispan/ispncon

55.2 Installation

Ispncon requires

- python 2.6
- python modules: python-devel, setuptools, infinispan, python-memcached

Installation steps:

```
$ git clone <a class="jive-link-external-small" href="https://github.com/infinispan/ispncon.git"
target="_blank">https://github.com/infinispan/ispncon.git</a>
$ cd ispncon/src
$ sudo python setup.py install
```

Make this part of your .bashrc or whatever so you have ispncon command on path

```
export ISPNCON_HOME=/path/to/ispncon
export PATH=$ISPNCON_HOME/bin:$PATH
```

55.3 Basic usage

Format

```
ispncon [options] <operation> [operation_options] <op_arguments>
```

Options

Option	Meaning
-c --client	Client to use, possible values: memcached, rest, hotrod (default) Configuration key: ispncon.client_type
-h --host <host>	Host/ip address to connect to. Default: localhost. Configuration key: ispncon.host
-p --port <port>	Port to connect to. Default: 11222 (hotrod default port) Configuration key: ispncon.port
-C --cache-name <name>	Named cache to use. Default: use default cache (no value) Configuration key: ispncon.cache
-v --version	Print ispncon version and exit
-e --exit-on-error	If true and operation fails, then fail with an exit code. If false just print ERROR message and continue. This mostly makes sense for batch files.
-P --config "<key> <value>"	override configuration option <key> with value <value>. NOTE: the quotes are necessary, because getopt parser needs to get these as one string.

The possible Operations are *put, get, delete, clear, exists, help, config, include* and each one is described in the sections *Cache operations* and *Other commands* .

55.4 Configuration

On start-up ispncon reads the file ~/.ispncon to configure some of the default values to use, so that the user doesn't have to enter the options like host, port, client type and cache name everytime he types a cache command.

The format of the config file is like this:

```
[ispncon]
host = localhost
port = 8080
client_type = rest
cache =
exit_on_error = False
default_codec = None
[rest]
server_url = /infinispan-server-rest/rest
content_type = text/plain
[hotrod]
use_river_string_keys = True
```

The config parameters in the section **ispncon** are described in the_Basic usage_ section. The rest is described here:

Config key	Meaning
rest.server_url	location of the REST server service, in the above example the HTTP requests would be sent tohttp://localhost:8080/infinispan-server-rest/rest
rest.content_type	default MIME content type for entries stored via REST interface
default_codec	Default codec to use to encode/decode values. Possible values: None, RiverString, RiverByteArray see section Interoperability with java clients for further info
hotrod.use_river_string_keys	If set to True, hotrod client will encode keys with RiverString codec - this is necessary to be able to access the same data as via Java HotRod Client using the same string keys. see section Interoperability with java clients for further info

55.5 Cache operations

55.5.1 put

Put data under a specified key.

Format

```
put [options] <key> <value>
```

Options

Option	Meaning
-i --input-filename <filename>	Don't specify the value, instead put the contents of the specified file.
-v --version <version>	Put only if version equals version given. Version format differs between protocols: HotRod: 64-bit integer version number Memcached: 64-bit integer unique version id REST: ETag string Not yet implemented for REST client in infinispan, watch ISPN-1084 for more info.
-l --lifespan <seconds>	Specifies lifespan of the entry. Integer, number of seconds.
-I --max-idle <seconds>	Specifies max idle time for the entry. Integer, number of seconds.
-a --put-if-absent	Return CONFLICT if value already exists and don't put anything in that case
-e --encode <codec>	Encode value using the specified codec

Return values

Exit code	Output	Result description
0	STORED	Entry was stored sucessfully
1	ERROR <msg>	General error occurred
2	NOT_FOUND	-v option was used and entry doesn't exist
3	CONFLICT	-a option was used and the entry already exists, or -v was used and versions don't match

NOTE: memcached client won't distinguish between states NOT_FOUND, CONFLICT and ERROR and always will return ERROR if operation wasn't successfull. this is a limitation of python-memcached client.

see issues:

https://bugs.launchpad.net/python-memcached/+bug/684689

https://bugs.launchpad.net/python-memcached/+bug/684690 for discussion.

In later ispncon versions python-memcached client might get replaced by a customized version.

55.5.2 get

Get the data stored under the specified key.

Format

```
get [options] <key>
```

Options

Option	Meaning
-o --output-filename <filename>	Stores the output of the get operation into the file specified.
-v --version	Get version along with the data. Version format differs between protocols: HotRod: 64-bit integer version number Memcached: 64-bit integer unique version id REST: ETag string
-d --decode <codec>	Decode the value using the specified codec.

Return values

Exit code	Output	Result description
0	In case no filename was specified: <data, possibly multi-line> (NOTE: the data might contain binary content, that is not suitable for reading in terminal) In case a filename was specified, nothing is printed on standard output. In case -v was specified, the output is prepended with one line: VERSION <version>	Entry was found and is returned.
1	ERROR <msg>	General error occurred
2	NOT_FOUND	Requested entry wasn't found in the cache

55.5.3 delete

Delete the entry with the specified key.

Format

```
delete [options] <key>
```

Options

Option	Meaning
-v --version <version>	Deletes only if the specified version matches the version in the cache NOTE: versioned delete is not supported with memcached client. attempt to delete with -v flag will end in ERROR message. with REST client the situation is different, the protocol allows this, but it's not yet implemented in infinispan, watch ISPN-1084 for more info

Return values

Exit code	Output	Result description
0	DELETED	Entry was successfully deleted
1	ERROR <msg>	General error occurred
2	NOT_FOUND	Entry wasn't found in the cache.
3	CONFLICT	Option -v was used and versions don't match

55.5.4 clear

Clear the cache

Format

```
clear
```

Return values

Exit code	Output	Result description
0	DELETED	Cache was sucessfully cleared
1	ERROR <msg>	General error occurred

55.5.5 exists

Verify if the entry exists in the cache

Format

```
exists <key>
```

Return values

Exit code	Output	Result description
0	EXISTS	Entry with the given key exists
1	ERROR <msg>	General error occurred
2	NOT_FOUND	Entry with the given key wasn't found in the cache

NOTE: memcached protocol doesn't support querying for existence of an entry in the cache so exists operation is implemented (inefficiently) by get opeartion, that gets the whole entry with all the data from the server.

55.5.6 version

Get version of the entry. Version format differs between protocols:

HotRod: 64-bit integer version number

Memcached: 64-bit integer unique version id

REST: ETag string

NOTE: The purpose of this command is to facilitate the parsing of the version string. HotRod and Memcached client don't support efficient implementation of this operation. They transfer the whole entry from the server to determine the version, so if applicable you are encouraged to use "get -v" command to obtain version together with the data.

REST client implements this operation efficiently by executing HEAD method.

Format

```
version <key>
```

Return values

Exit code	Output	Result description
0	<version>	If the entry exists.
1	ERROR <msg>	General error occurred
2	NOT_FOUND	Requested entry wasn't found in the cache

55.6 Other commands

55.6.1 help

Print help about an operation

Format

```
help <operation>
```

NOTE: if no operation is supplied, prints list of supported operations

55.6.2 config

Change internal state/config of the client. This opeartion has only client-side effect.

Format

```
config               - to print current config
config save          - to save config to ~/.ispncon
config <key> <value>  - to change config for currently running session
```

Configuration values

see section Configuration for the meaning of different configuration options. Currently supported keys are:

- cache
- host
- port
- client_type
- exit_on_error
- rest.server_url
- rest.content_type

These values directly correspond to the keys in the ~/.ispncon config file. The format of the key is

```
<section>.<config_key>
```

If no section is given, "ispncon" is implied.
Return values

Exit code	Output	Result description
0	STORED	If configuration/client state was updated successfully.
0	multi-line output with config values	If config command with no parameters was entered.
1	ERROR <msg>	General error occurred

55.6.3 include

Process cache commands from the specified batch file. The commands will be processed line by line.

Format

```
include <filename>
```

Return values

Exit code	Output	Result description
exit code of the last command in the file.	The output depends on the commands present in the input file	depends on the commands in the batch file

NOTE: The name of this command and it's behaviour is going to change in the next version.

55.7 Interoperability with java clients

55.7.1 HTTP clients

when exchanging data via REST interface, the values are interpreted by any client as sequence of bytes. The meaning is given to this byte-sequence by using MIME type specified via "Content-Type" HTTP header. No special interoperability measures are needed here.

55.7.2 Hot Rod Java Client

If we want to read in ispncon the entries that were put with Hot Rod Java client, we need to use a special option **hotrod.use_river_string_keys = True**. This will cause the string keys to be encoded the same way the Java client does it.

Using **hotrod.use_river_string_keys = True** we're able to access the data that has been writen by the java client, but we still see the raw binary values. To be able to see a value that has been put by Hot Rod java client in a readable form and vice versa - to be able to see in Hot Rod Java client what we've put via ispncon we need to use a **codec**. Currently there are two types of codecs: **RiverString** and **RiverByteArray**

RiverString - will decode a value that has been put as java.lang.String and vice versa - a value encoded with this codec will be returned as java.lang.String on the java side

RiverByteArray - analogous to RiverString but works with byte[] (java byte array)

Codecs can be used either by specifying a **default_codec** option in the ~/.ispncon config file (in section ispncon) or by specifying a codec on each put resp get using **-e (--encode)** resp **-d (--decode) options**.

55.7.3 SpyMemcached Java Client

Tested with spymemcached 2.7.

Value that is put by ispncon is interpreted as an UTF-8 string. meaning if we supply some bytes, on the java side it will be recreated as new java.lang.String(bytes, "UTF-8")

This also works reversely: values put by java side as java.lang.String will be returned as UTF-8 bytes in ispncon

56 Server Command Line Options

Infinispan ships several server modules, some of which can be started via calling startServer.sh or startServer.bat scripts from command line. These currently include Hot Rod, Memcached and Web Socket servers. Please find below the set of common command line parameters that can be passed to these servers:

Note that starting with Infinispan 4.2.0.CR1, default Hot Rod port has changed from 11311 to 11222.

```
[g@eq]~/infinispan-4.1.0-SNAPSHOT% ./bin/startServer.sh -h
usage: startServer [options]

options:
  -h, --help
        Show this help message

  -V, --version
        Show version information

  --
        Stop processing options

  -p, --port=<num>
        TCP port number to listen on
        (default: 11211 for Memcached, 11222 for Hot Rod
        and 8181 for WebSocket server)

  -l, --host=<host or ip>
        Interface to listen on
        (default: 127.0.0.1, localhost)

  -m, --master_threads=<num>
        Number of threads accepting incoming connections
        (default: unlimited while resources are available)

  -t, --work_threads=<num>
        Number of threads processing incoming requests and sending responses
        (default: unlimited while resources are available)

  -c, --cache_config=<filename>
        Cache configuration file
        (default: creates cache with default values)

  -r, --protocol=[memcached|hotrod|websocket]
        Protocol to understand by the server.
        This is a mandatory option and you should choose one of these options

  -i, --idle_timeout=<num>
        Idle read timeout, in seconds, used to detect stale connections
        (default: -1)
        If no new messages have been read within this time,
        the server disconnects the channel.
        Passing -1 disables idle timeout.
```

```
-n, --tcp_no_delay=[true|false]
      TCP no delay flag switch (default: true)

-s, --send_buf_size=<num>
      Send buffer size (default: as defined by the OS).

-e, --recv_buf_size=<num>
      Receive buffer size (default: as defined by the OS).

-o, --proxy_host=<host or ip>
      Host address to expose in topology information sent to clients.
      If not present, it defaults to configured host.
      Servers that do not transmit topology information ignore this setting.

-x, --proxy_port=<num>
      Port to expose in topology information sent to clients.
      If not present, it defaults to configured port.
      Servers that do not transmit topology information ignore this setting.

-k, --topo_lock_timeout=<num>
      Controls lock timeout (in milliseconds) for those servers that maintain
      the topology information in an internal cache.

-u, --topo_repl_timeout=<num>
      Sets the maximum replication time (in milliseconds) for transfer of
      topology information between servers.
      If state transfer is enabled, this setting also controls the topology
      cache state transfer timeout.
      If state transfer is disabled, it controls the amount of time to wait
      for this topology data to be lazily loaded from a different node when
      not present locally.

-a, --topo_state_trasfer=[true|false]
      Enabling topology information state transfer means that when a server
      starts it retrieves this information from a different node.
      Otherwise, if set to false, the topology information is lazily loaded
      if not available locally.

-D<name>[=<value>]
      Set a system property
```

57 Interacting With Hot Rod Server From Within Same JVM

57.1 Introduction

Normally, a Hot Rod server is accessed via a Hot Rod protocol client such as the Java Hot Rod client. However, there might be situations where not only do you want to access the Hot Rod server remotely, you might also want to access it locally from within the same JVM that the Hot Rod server is running. For example, you might have an Infinispan cache pushing changes via the RemoteCacheStore to a Hot Rod server, and if the cache goes down, you might want to access the data directly from the Hot Rod server itself.

In this situations, we have to remember that the Hot Rod protocol specifies that keys and values are stored as byte arrays. This means that if the client code, using an existing Hot Rod client, stored Strings or Integers, or any other complex serializable or externalizable object, you won't be able to retrieve these objects straight from the cache that the Hot Rod server uses.

To actually get the fully constructed objects that you're after, you're gonna need to take the byte arrays stored within the Hot Rod server and unmarshall them into something that you can use. In the future, this is something that might be done for you, as suggested in ISPN-706, but for the time being, clients wanting to access Hot Rod server data will have to do it themselves.

Two different use cases need to be differentiated at this stage and to explain how to transform the Hot Rod server data into something usable, we'll assume that the clients are java clients:

57.2 Data Stored Directly Via A Hot Rod Client

The most common case is for a client to use a Hot Rod client library directly to store data in the Hot Rod server. In this case, assuming that the client used the existing Java Hot Rod client, the default marshaller used to marshall objects into byte arrays is the GenericJBossMarshaller. So, if a user wants to read data from the Hot Rod server directly, it would need to execute something along the lines of:

```
import org.infinispan.marshall.jboss.GenericJBossMarshaller;
import org.infinispan.util.ByteArrayKey;
import org.infinispan.server.core.CacheValue;
...

// Create a new instance of the marshaller:
GenericJBossMarshaller marshaller = new GenericJBossMarshaller();
Object key = ...

// Take the cache key and convert into a byte array,
// and wrap it with an instance of ByteArrayKey
ByteArrayKey bytesKey = new ByteArrayKey(marshaller.objectToByteBuffer(key));

// Internally, Hot Rod stores values wrapped in a CacheValue, so retrieve it
CacheValue cacheValue = (CacheValue) cache.get(bytesKey);

// Take the data part which is byte array and unmarshall it to retrieve the value
Object value = marshaller.objectFromByteBuffer(cacheValue.data());
```

If you want to store data directly in the HotRod server, you'd have to execute something like this:

```
import org.infinispan.marshall.jboss.GenericJBossMarshaller;
import org.infinispan.util.ByteArrayKey;
import org.infinispan.server.core.CacheValue;
...

// Create a new instance of the marshaller:
GenericJBossMarshaller marshaller = new GenericJBossMarshaller();
Object key = ...
Object value = ...

// Take the cache key and convert into a byte array,
// and wrap it with an instance of ByteArrayKey
ByteArrayKey bytesKey = new ByteArrayKey(marshaller.objectToByteBuffer(key));

// Internally, Hot Rod stores values wrapped in a CacheValue, so create instance
// Remember that you need to give it a version number, so either:
// 1. Increment previous value's version
// 2. Or generate a new version number that minimises potential clash
//    with a concurrent update to the same key in the cluster
CacheValue cacheValue = new CacheValue(marshaller.objectToByteBuffer(value), 1)

// Finally, store it in the cache
cache.put(bytesKey, cacheValue);
```

57.3 Data Stored Via Remote Cache Store

Other times, Hot Rod server might be storing data coming from a RemoteCacheStore, rather than user code. In this case, there're a couple of differences to the code above. First of all, the marshaller is slightly different. Instead, the RemoteCacheStore uses the VersionAwareMarshaller which all it does is add Infinispan version information to the byte array generated. The second difference is that RemoteCacheStore stores internal cache entry classes, which apart from the value part, they contain other extra information. So, any code trying to read these directly from the Hot Rod server would need to take in account. For example, to read data from such Hot Rod server:

```
import org.infinispan.marshall.VersionAwareMarshaller;
import org.infinispan.util.ByteArrayKey;
import org.infinispan.server.core.CacheValue;
import org.infinispan.container.entries.CacheEntry;
...

// Create a new instance of the marshaller
VersionAwareMarshaller marshaller = new VersionAwareMarshaller();
Object key = ...

// Take the cache key and convert into a byte array,
// and wrap it with an instance of ByteArrayKey
ByteArrayKey bytesKey = new ByteArrayKey(marshaller.objectToByteBuffer(key));

// Internally, Hot Rod stores values wrapped in a CacheValue, so retrieve it
CacheValue cacheValue = (CacheValue) cache.get(bytesKey);

// However, in this case the data part of CacheValue does not contain directly
// the value Instead, it contains an instance of CacheEntry, so we need to
// unmarshall that and then get the actual value
CacheEntry cacheEntry = (CacheEntry)
   marshaller.objectFromByteBuffer(cacheValue.data());
Object value = cacheEntry.getValue();
```

And to actually write data back into the Hot Rod server directly:

```
import org.infinispan.marshall.VersionAwareMarshaller;
import org.infinispan.util.ByteArrayKey;
import org.infinispan.server.core.CacheValue;
import org.infinispan.container.entries.CacheEntry;
import org.infinispan.container.entries.InternalEntryFactory;
...

// Create a new instance of the marshaller:
VersionAwareMarshaller marshaller = new VersionAwareMarshaller();
Object key = ...
Object value = ...

// Take the cache key and convert into a byte array
ByteArrayKey bytesKey = new ByteArrayKey(marshaller.objectToByteBuffer(key));

// With the value to store, a new CacheEntry instance needs to be created:
CacheEntry cacheEntry = InternalEntryFactory.create(bytesKey, value, ...)

// Internally, Hot Rod stores values wrapped in a CacheValue, so create instance
// Remember that you need to give it a version number, so either:
// 1. Increment previous value's version
// 2. Or generate a new version number that minimises potential clash
//    with a concurrent update to the same key in the cluster
CacheValue cacheValue = new CacheValue(
   marshaller.objectToByteBuffer(cacheEntry), 1)

// Finally, store it in the cache
cache.put(bytesKey, cacheValue);
```

58 Multiple Tiers of Caches

58.1 Introduction

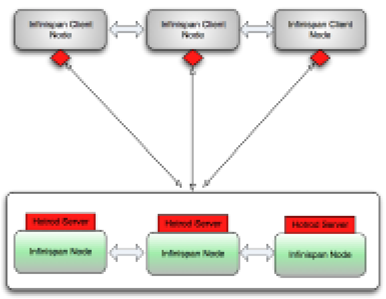
The introduction of HotRod protocol and RemoteCacheLoader opened the way for a set of new architectures in Infinispan, where layers of caches can exists and interact. This article takes a look at such an layered architecture.

58.2 Building blocks

HotRod is a binary protocol defined for exposing an Infinispan cluster as an caching server to multiple platforms. It has support for load balancing and smart routing.

RemoteCacheLoader is a cache loader that knows how to read/store data in a remote infinispan cluster. For that it makes use of the java hotrod client.

58.3 Sample architecture/near caching



The diagram above shows an Infinispan server cluster running 3 hotrod servers. This cluster is accessed remotely, through HotRod, by another infinispan cluster: client cluster (upper part of the image). All the nodes in the server cluster are configured to run HotRod servers, so requests from remote loader are being balanced between them. The client cluster is configured with invalidation as cluster mode and a RemoteCacheLoader to acess data stored in the server cluster. Application data is held on the server cluster which runs in DIST mode for scalability.

In this deployment the client code, running in same address space with the client cluster, holds all its data in the server cluster. Client cluster acts as an **near-cache** for frequently accessed entries.

58.4 Want to know more?

On the documentation main page:

- cache loaders are described in the "Cache loaders" section
- Hotrod's specification, server and client in the "Hot Rod" section

59 Infinispan REST Server

59.1 Introduction

This server provides easy to use RESTful HTTP access to the Infinispan data grid, build on RESTEasy. This application is delivered (currently) as a war, which you can deploy to a servlet container (as many instances as you need).

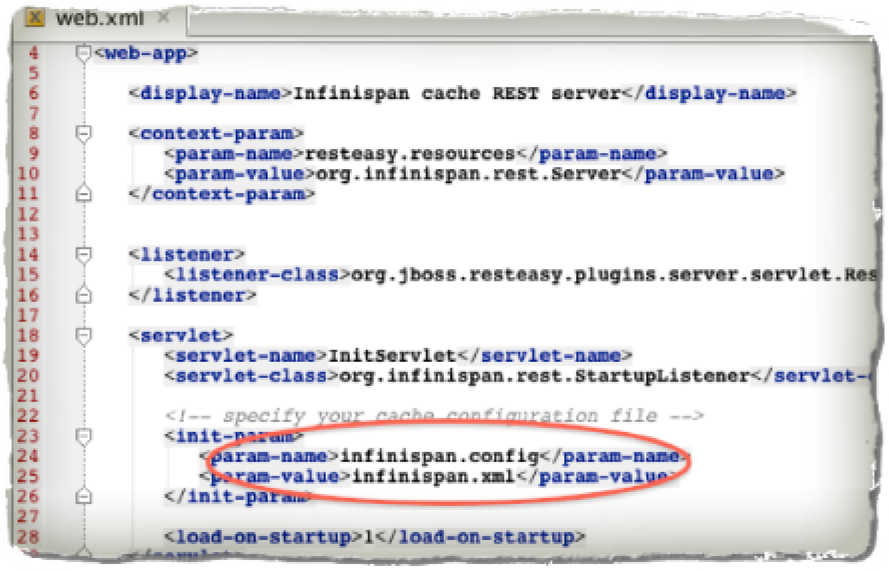
59.2 Configuration

Out of the box, Infinispan will create and use a new LOCAL mode cache. To set a custom configuration:

1. Unzip the REST WAR file (or use an exploded deployment)
2. Create an Infinispan configuration XML file and name this infinispan.xml
3. Place this file in `infinispan-server-rest.war/WEB-INF/classes`

Alternatively, you could:

1. Unzip the REST WAR file (or use an exploded deployment)
2. Create an Infinispan configuration XML file, call it whatever you want and place it wherever you want
3. Edit `infinispan-server-rest.war/WEB-INF/web.xml` and look for the `infinispan.config` init-param. Change the value of this init-param to the full path to your Infinispan configuration.



```
<web-app>

    <display-name>Infinispan cache REST server</display-name>

    <context-param>
        <param-name>resteasy.resources</param-name>
        <param-value>org.infinispan.rest.Server</param-value>
    </context-param>


    <listener>
        <listener-class>org.jboss.resteasy.plugins.server.servlet.Res
    </listener>

    <servlet>
        <servlet-name>InitServlet</servlet-name>
        <servlet-class>org.infinispan.rest.StartupListener</servlet-

        <!-- specify your cache configuration file -->
        <init-param>
            <param-name>infinispan.config</param-name>
            <param-value>infinispan.xml</param-value>
        </init-param>

        <load-on-startup>1</load-on-startup>
```

Please note that the REST server only allows interaction with either the default cache (named `___defaultcache`) or one of the named caches in the configuration file. This is because the REST server starts the default and pre-defined caches on startup in order to provide consistent behaivor.

Warning

Creation of new named caches on the fly is not supported.

As a result, if you don't use a custom configuration file, you'll only be able to interact with the default cache. To interact with more caches, use a configuration file with the desired named caches.

59.3 Accessing Data - via URLs

HTTP PUT and POST methods are used to place data in the cache, with URLs to address the cache name and key(s) - the data being the body of the request (the data can be anything you like). It is important that a Content-Type header is set. GET/HEAD are used to retrieve data Please see here for the details. Other headers are used to control the cache settings and behaviour (detailed in that link).

59.4 Client side code

Part of the point of a RESTful service is that you don't need to have tightly coupled client libraries/bindings. All you need is a HTTP client library. For Java, Apache HTTP Commons Client works just fine (and is used in the integration tests), or you can use java.net API.

59.4.1 Ruby client code:

```
# Shows how to interact with Infinispan REST api from ruby.
# No special libraries, just standard net/http
#
# Author: Michael Neale
#
require 'net/http'

http = Net::HTTP.new('localhost', 8080)

#Create new entry
http.post('/infinispan/rest/MyData/MyKey', 'DATA HERE', {"Content-Type" => "text/plain"})

#get it back
puts http.get('/infinispan/rest/MyData/MyKey').body

#use PUT to overwrite
http.put('/infinispan/rest/MyData/MyKey', 'MORE DATA', {"Content-Type" => "text/plain"})

#and remove...
http.delete('/infinispan/rest/MyData/MyKey')

#Create binary data like this... just the same...
http.put('/infinispan/rest/MyImages/Image.png', File.read('/Users/michaelneale/logo.png'),
{"Content-Type" => "image/png"})


#and if you want to do json...
require 'rubygems'
require 'json'

#now for fun, lets do some JSON !
data = {:name => "michael", :age => 42 }
http.put('/infinispan/rest/Users/data/0', data.to_json, {"Content-Type" => "application/json"})
```

59.4.2 Python client code:

```
# Sample python code using the standard http lib only
#

import httplib


#putting data in
conn = httplib.HTTPConnection("localhost:8080")
data = "SOME DATA HERE \!" #could be string, or a file...
conn.request("POST", "/infinispan/rest/Bucket/0", data, {"Content-Type": "text/plain"})
response = conn.getresponse()
print response.status

#getting data out
import httplib
conn = httplib.HTTPConnection("localhost:8080")
conn.request("GET", "/infinispan/rest/Bucket/0")
response = conn.getresponse()
print response.status
print response.read()
```

59.4.3 Java client code:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.URL;

/**
 * Rest example accessing Infinispan Cache.
 * @author Samuel Tauil (samuel@redhat.com)
 *
 */
public class RestExample {

   /**
    * Method that puts a String value in cache.
    * @param urlServerAddress
    * @param value
    * @throws IOException
    */
   public void putMethod(String urlServerAddress, String value) throws IOException {
      System.out.println("----------------------------------------");
      System.out.println("Executing PUT");
      System.out.println("----------------------------------------");
      URL address = new URL(urlServerAddress);
      System.out.println("executing request " + urlServerAddress);
      HttpURLConnection connection = (HttpURLConnection) address.openConnection();
      System.out.println("Executing put method of value: " + value);
      connection.setRequestMethod("PUT");
      connection.setRequestProperty("Content-Type", "text/plain");
      connection.setDoOutput(true);

      OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(connection.getOutputStream());
      outputStreamWriter.write(value);

      connection.connect();
      outputStreamWriter.flush();

      System.out.println("----------------------------------------");
      System.out.println(connection.getResponseCode() + " " + connection.getResponseMessage());
      System.out.println("----------------------------------------");

      connection.disconnect();
   }

   /**
    * Method that gets an value by a key in url as param value.
    * @param urlServerAddress
    * @return String value
    * @throws IOException
    */
   public String getMethod(String urlServerAddress) throws IOException {
      String line = new String();
      StringBuilder stringBuilder = new StringBuilder();

      System.out.println("----------------------------------------");
      System.out.println("Executing GET");
      System.out.println("----------------------------------------");

      URL address = new URL(urlServerAddress);
      System.out.println("executing request " + urlServerAddress);

      HttpURLConnection connection = (HttpURLConnection) address.openConnection();
      connection.setRequestMethod("GET");
      connection.setRequestProperty("Content-Type", "text/plain");
      connection.setDoOutput(true);

      BufferedReader&nbsp; bufferedReader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));

      connection.connect();

      while ((line = bufferedReader.readLine()) \!= null) {
         stringBuilder.append(line + '\n');
      }

      System.out.println("Executing get method of value: " + stringBuilder.toString());

      System.out.println("----------------------------------------");
      System.out.println(connection.getResponseCode() + " " + connection.getResponseMessage());
      System.out.println("----------------------------------------");
      connection.disconnect();

      return stringBuilder.toString();
   }

   /**
    * Main method example.
    * @param args
    * @throws IOException
    */
   public static void main(String\[\] args) throws IOException {
      //Attention to the cache name "cacheX" it was configured in xml file with tag <namedCache
name="cacheX">
      RestExample restExample = new RestExample();
      restExample.putMethod("http://localhost:8080/infinispan/rest/cacheX/1", "Infinispan REST
Test");
      restExample.getMethod("http://localhost:8080/infinispan/rest/cacheX/1");
   }
}
```

59.5 Future:

- Sample persistence options to make this a long term data grid
- Query and indexing (of known MIME types, and JSON, XML etc)
- Returning both lists of buckets + entries as <link> relations (where it makes sense)
- Monitoring of stats via Web interface
- (optional: WADL?)

60 CDI Support

60.1 Introduction

Infinispan includes integration with CDI in the `infinispan-cdi` module. Configuration and injection of the Inifispan's Cache API is provided, and it is planned to bridge Cache listeners to the CDI event system. The module also provide partial support of the JCache (JSR-107) caching annotations - for further details see Chapter 8 of the JCACHE specification.

60.2 Maven Dependencies

All you need is `org.infinispan:infinispan-cdi`

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

Which version of Infinispan should I use?

We recommend using the latest final version of the `infinispan-cdi` module. This module is available since Infinispan version `5.0.0.CR8`.

60.3 Embedded cache integration

60.3.1 Inject an embedded cache

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

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

public class GreetingService {
    @Inject
    private Cache<String, String> cache;

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

If you want to use a specific cache you just have to provide your own cache configuration and cache qualifier. For example, if you want to use a custom cache for the `GreetingService` you should write your own qualifier (here `GreetingCache`) and define its configuration:

The new configuration system is used since 5.1.0.CR2

As you probably know Infinispan 5.1.0 comes with a new way to configure your cache programmatically and a bit more (more information are available here). Now to configure a cache you must use this new configuration system.

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

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

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

public class Config {
    @ConfigureCache("greeting-cache") // This is the cache name.
    @GreetingCache // This is the cache qualifier.
    @Produces
    public Configuration greetingCacheConfiguration() {
        return new ConfigurationBuilder()
                    .eviction()
                        .strategy(EvictionStrategy.LRU)
                        .maxEntries(1000)
                    .build();
    }

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

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

60.3.2 Override the default embedded cache manager and configuration

Since 5.1.0.CR1 @OverrideDefault is deprecated

To migrate your code remove this annotation from your producers as detailed in the following section.

You can override the default cache configuration used by the default embedded cache manager. For that, you just have to create one `Configuration` producer with the `@Default` qualifier as illustrated in the following snippet:

```
public class Config {
    // By default CDI adds the @Default qualifier if no other qualifier is provided.
    @Produces
    public Configuration defaultEmbeddedCacheConfiguration() {
        return new ConfigurationBuilder()
                    .eviction()
                        .strategy(EvictionStrategy.LRU)
                        .maxEntries(100)
                    .build();
    }
}
```

It's also possible to override the default embedded cache manager used. The new default cache manager produced must have the `@Default` qualifier and the scope `@ApplicationScoped`.

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

public class Config {
    @Produces
    @ApplicationScoped
    public EmbeddedCacheManager defaultEmbeddedCacheManager() {
      return new DefaultCacheManager(new ConfigurationBuilder()
                                          .eviction()
                                              .strategy(EvictionStrategy.LRU)
                                              .maxEntries(100)
                                          .build());
   }
}
```

60.3.3 Configure the transport for clustered use

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

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

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

60.4 Remote cache integration

60.4.1 Inject a remote cache

With the CDI integration it's also possible to use a remote cache. For example you can inject the default `RemoteCache` as illustrated in the following snippet:

```
public class GreetingService {
    @Inject
    private RemoteCache<String, String> cache;

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

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

```
public class GreetingService {
    @Inject @Remote("greeting-cache")
    private RemoteCache<String, String> cache;

    ...
}
```

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

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

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

60.4.2 Override the default remote cache manager

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

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

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

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

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

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

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

Producer method scope

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

60.6 Use a JBoss AS 7 configured cache

With JBoss AS 7, you can setup an Infinispan cache manager in the server configuration file. This allows you to externalize your Infinispan configuration and also to lookup the cache manager from JNDI, normally with the `@Resource` annotation.

As we mentioned earlier, you can override the default cache manager used by the Infinispan CDI extension. To use a JBoss AS 7 configured cache, you need to use the cache manager defined in JBoss AS 7. You only need to annotate the default cache manager producer with @Resource. The following example shows how use an embedded cache manager configured in JBoss AS 7.

```
...
import javax.annotation.Resource;

public class Config {
    @Produces
    @ApplicationScoped
    @Resource(lookup="java:jboss/infinispan/my-container-name")
    private EmbeddedCacheManager defaultCacheManager;
}
```

60.7 Use JCache caching annotations

The `infinispan-cdi` module provides a partial support of JCache caching annotations. These annotations provide a simple way to handle common use cases. The following caching annotations are defined in this specification:

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

Annotations target type

These annotations must only be used on methods.

To use these annotations the following interceptors must be declared in your application `beans.xml`.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
    <interceptors>
        <class>org.infinispan.cdi.interceptor.CacheResultInterceptor</class>
        <class>org.infinispan.cdi.interceptor.CachePutInterceptor</class>
        <class>org.infinispan.cdi.interceptor.CacheRemoveEntryInterceptor</class>
        <class>org.infinispan.cdi.interceptor.CacheRemoveAllInterceptor</class>
    </interceptors>
</beans>
```

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

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

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

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

Can I use a different cache?

To use another cache specify its name with the `cacheName` attribute of the cache annotation. For example:

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

61 Marshalling

61.1 Introduction

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

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

61.2 The Role Of JBoss Marshalling

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

61.3 Support For Non-Serializable Objects

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

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

If either of these options is true, key/value pairs in the cache will need to be marshalled and currently they require to either to extend java.io.Serializable or java.io.Externalizable. However, since Infinispan 5.0, marshalling non-Serializable key/value objects is supported as long as users can to provide meaningful Externalizer implementations for these non-Seralizable objects, see this article to find out more.

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

61.3.1 Lazy Deserialization (storeAsBinary)

Lazy deserialization is the mechanism by which Infinispan by which serialization and deserialization of objects is deferred till the point in time in which they are used and needed. This typically means that any deserialization happens using the thread context class loader of the invocation that requires deserialization, and is an effective mechanism to provide classloader isolation. By default lazy deserialization is disabled but if you want to enable it, you can do it like this:

- Via XML at the Cache level, either under <namedCache> or <default> elements:

```
<lazyDeserialization enabled="true"/>
```

- Programmatically:

```
Configuration configuration = ...
configuration.setUseLazyDeserialization(true);
```

Note: Since 5.0 onwards, lazyDeserialization has been renamed to storeAsBinary to better represent its functionality:

```
<storeAsBinary enabled="true"/>
```

The programmatic configuration has changed as well together with a move towards more fluent configuration:

```
Configuration configuration = ...
configuration.fluent().storeAsBinary();
```

Equality Considerations

When using lazy deserialization/storing as binary, keys and values are wrapped as MarshalledValue s. It is this wrapper class that transparently takes care of serialization and deserialization on demand, and internally may have a reference to the object itself being wrapped, or the serialized, byte array representation of this object
This has a particular effect on the behavior of equality. The equals() method of this class will either compare binary representations (byte arrays) or delegate to the wrapped object instance's equals() method, depending on whether both instances being compared are in serialized or deserialized form at the time of comparison. If one of the instances being compared is in one form and the other in another form, then one instance is either serialized or deserialized. The preference will be to compare object representations, unless the cache is compacted, in which case byte array comparison is favored.

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

61.4 Advanced Configuration

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

By default, Infinispan uses the VersionAwareMarshaller which, as the name suggests, 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. 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. Optionally, Infinispan users to optionally provide their own marshaller, for example:

- Via XML at the CacheManager level, under <global> element:

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

- Programatically:

```
GlobalConfiguration globalConfiguration = ...
globalConfiguration.setMarshallerClass("com.acme.MyMarshaller")
```

61.4.1 Troubleshooting

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

```
<code class="jive-code jive-java">java.io.NotSerializableException: java.lang.Object
at org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.java:857)
at org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at
org.infinispan.marshall.exts.ReplicableCommandExternalizer.writeObject(ReplicableCommandExternaliz
org.infinispan.marshall.jboss.ConstantObjectTable$ExternalizerAdapter.writeObject(ConstantObjectTa
org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.java:143)
at org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at org.infinispan.marshall.jboss.JBossMarshaller.objectToObjectStream(JBossMarshaller.java:167)
at org.infinispan.marshall.VersionAwareMarshaller.objectToBuffer(VersionAwareMarshaller.java:92)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToByteBuffer(VersionAwareMarshaller.java:170)
org.infinispan.marshall.VersionAwareMarshallerTest.testNestedNonSerializable(VersionAwareMarshalle
by: an exception which occurred:
in object java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
... Removed 22 stack frames
</code>
```

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

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

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

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

```
<code class="jive-code jive-java">java.io.IOException: Injected failue!
at
org.infinispan.marshall.VersionAwareMarshallerTest$1.readExternal(VersionAwareMarshallerTest.java:
org.jboss.marshalling.river.RiverUnmarshaller.doReadNewObject(RiverUnmarshaller.java:1172)
at org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:273)
at org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:210)
at org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller.java:85)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectFromObjectStream(JBossMarshaller.java:210)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarshaller.java:10
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarshaller.java:17
org.infinispan.marshall.VersionAwareMarshallerTest.testErrorUnmarshalling(VersionAwareMarshallerTe
by: an exception which occurred:
in object of type org.infinispan.marshall.VersionAwareMarshallerTest$1
</code>
```

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

```
<code class="jive-code jive-java">java.io.IOException: Injected failue!
...
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.VersionAwareMarshallerTest$1
-> classloader hierarchy:
-> type classloader = sun.misc.Launcher$AppClassLoader@198dfaf
->...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/eclipse-testng.jar
->...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/lib/testng-jdk15.jar
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/test-classes/
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/classes/
->...file:/home/galder/.m2/repository/org/testng/testng/5.9/testng-5.9-jdk15.jar
->...file:/home/galder/.m2/repository/net/jcip/jcip-annotations/1.0/jcip-annotations-1.0.jar
->...file:/home/galder/.m2/repository/org/easymock/easymockclassextension/2.4/easymockclassextensi
parent classloader = sun.misc.Launcher$ExtClassLoader@1858610
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/localedata.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunpkcs11.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunjce_provider.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/dnsns.jar
... Removed 22 stack frames
</code>
```

Finding the root cause of marshalling/unmarshalling exceptions can sometimes be really daunting but we hope that the above improvements would help get to the bottom of those in a more quicker and efficient manner.

62 Batching

62.1 Introduction

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

62.2 Configuring batching

To use batching, you need to enable invocation batching in your cache configuration, either on the Configuration object:

```
Configuration.setInvocationBatchingEnabled(true);
```

or in your XML file:

```
<invocationBatching enabled="true" />
```

By default, invocation batching is disabled.

Note that you do not have to have a transaction manager defined to use batching.

62.3 Batching API

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

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

// using a batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k2", "value");
cache.endBatch(true); // This will now replicate the modifications since the batch was started.
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false); // This will "discard" changes made in the batch
```

62.4 Advanced: batching and JTA

Behinds the scene, the batching functionality starts a JTA transactions, and all the invocations in that scope are associated with it. For this it uses a very simple (e.g. no recovery) TransactionManager implementation behind the scene. From this you get all sorts of nice behaviour, including:

1. Locks you acquire during an invocation are held until the transaction commits or rolls back.

2. Changes are all replicated around the cluster in a batch as part of the transaction commit process. Reduces replication chatter if multiple changes occur during the transaction.

3. If synchronous replication or invalidation are used, a failure in replication/invalidation will cause the transaction to roll back.

4. If a CacheLoader is used, and that cache loader works with a JTA resource (e.g. a JTA DataSource), the JTA resource can also participate in the transaction.

5. All the transaction related configurations apply for batching as well:

```
<transaction
    syncRollbackPhase="false" syncCommitPhase="false"
    useEagerLocking="true" eagerLockSingleNode="true" />
```

63 Hot Rod Hash Functions

Infinispan makes use of a consistent hash function to place nodes on a hash wheel, and to place keys of entries on the same wheel to determine where entries live.

In Infinispan 4.2 and earlier, the hash space was hardcoded to 10240, but since 5.0, the hash space is Integer.MAX_INT. Please note that since Hot Rod clients should not assume a particular hash space by default, everytime a hash-topology change is detected, this value is sent back to the client via the Hot Rod protocol.

When interacting with Infinispan via the Hot Rod protocol, it is mandated that keys (and values) are byte arrays, to ensure platform neutral behavior. As such, smart-clients which are aware of hash distribution on the backend would need to be able to calculate the hash codes of such byte array keys, again in a platform-neutral manner. To this end, the hash functions used by Infinispan are versioned and documented, so that it can be re-implemented by non-Java clients if needed.

The version of the hash function in use is provided in the Hot Rod protocol, as the hash function version parameter.

1. Version 1 (single byte, 0x01)
   The initial version of the hash function in use is Austin Appleby's MurmurHash 2.0 algorithm, a fast, non-cryptographic hash that exhibits excellent distribution, collision resistance and avalanche behavior. The specific version of the algorithm used is the slightly slower, endian-neutral version that allows consistent behavior across both big- and little-endian CPU architectures. Infinispan's version also hard-codes the hash seed as -1. For details of the algorithm, please visit Austin Appleby's MurmurHash 2.0 page. Other implementations are detailed on Wikipedia. This hash function was the default one used by the Hot Rod server until Infinispan 4.2.1.

1. Version 2 (single byte, 0x02)
   Since Infinispan 5.0, a new hash function is used by default which is Austin Appleby's MurmurHash 3.0 algorithm. Detailed information about the hash function can be found in this wiki. Compared to 2.0, it provides better performance and spread.

64 Hot Rod Protocol

64.1 Hot Rod Protocol - Version 1.0

64.1.1 Introduction

This article provides detailed information about the first version of the custom TCP client/server Hot Rod protocol.

Infinispan versions

This version of the protocol is implemented since Infinispan 4.1.0.Final

All key and values are sent and stored as byte arrays. Hot Rod makes no assumptions about their types. Some clarifications about the other types:

- `vInt`: Refers to unsigned variable length integer values as specified in here. They're between 1 and 5 bytes long.
- `vLong`: Refers to unsigned variable length long values similar to vInt but applied to longer values. They're between 1 and 9 bytes long.
- `String`: Strings are always represented using UTF-8 encoding.

64.1.2 Request Header

The header for a request is composed of:

Magic [1b]	Message Id [vLong]	Version [1b]	Opcode [1b]	Cache Name Length [vInt]	Cache Name [string]	Flags [vInt]	Client Intelligence [1b]	Topology Id [vInt]	Transaction Type [1b]

- **Magic** : Possible values are:
  - `0xA0` - Infinispan Cache Request Marker
  - `0xA1` - Infinispan Cache Response Marker
- **Message Id** : Id of the message that will be copied back in the response. This allows for hot rod clients to implement the protocol in an asynchronous way.
- **Version** : Infinispan hot rod server version. In this particular case, this is `10`

- **Opcode** : Possible values are only the ones on the request column:

Request operation codes	Response operation codes
`0x01` - put request	`0x02` - put response
`0x03` - get request	`0x04` - get response
`0x05` - putIfAbsent request	`0x06` - putIfAbsent response
`0x07` - replace request	`0x08` - replace response
`0x09` - replaceIfUnmodified request	`0x0A` - replaceIfUnmodified response
`0x0B` - remove request	`0x0C` - remove response
`0x0D` - removeIfUnmodified request	`0x0E` - removeIfUnmodified response
`0x0F` - containsKey request	`0x10` - containsKey response
`0x11` - getWithVersion request	`0x12` - getWithVersion response
`0x13` - clear request	`0x14` - clear response
`0x15` - stats request	`0x16` - stats response
`0x17` - ping request	`0x18` - ping response
`0x19` - bulkGet request	`0x1A` - bulkGet response
-	`0x50` - error response

- **Cache Name Length** : Length of cache name. If the passed length is 0 (followed by no cache name), the operation will interact with the default cache.
- **Cache Name** : Name of cache on which to operate. This name must match the name of predefined cache in the Infinispan configuration file.
- **Flags** : A variable length number representing flags passed to the system. Each flags is represented by a bit. Note that since this field is sent as variable length, the most significant bit in a byte is used to determine whether more bytes need to be read, hence this bit does not represent any flag. Using this model allows for flags to be combined in a short space. Here are the current values for each flag:

`0x0001`	ForceReturnPreviousValue

- **Client Intelligence** : This byte hints the server on the client capabilities:
  - `0x01` - basic client, interested in neither cluster nor hash information
  - `0x02` - topology-aware client, interested in cluster information
  - `0x03` - hash-distribution-aware client, that is interested in both cluster and hash information
- **Topology Id** : This field represents the last known view in the client. Basic clients will only send 0 in this field. When topology-aware or hash-distribution-aware clients will send 0 until they have received a reply from the server with the current view id. Afterwards, they should send that view id until they receive a new view id in a response

- **Transaction Type** : This is a 1 byte field, containing one of the following well-known supported transaction types (For this version of the protocol, the only supported transaction type is 0):
  - `0` - Non-transactional call, or client does not support transactions. The subsequent TX_ID field will be omitted.
  - `1` - X/Open XA transaction ID (XID). This is a well-known, fixed-size format.
- **Transaction Id** : The byte array uniquely identifying the transaction associated to this call. It's length is determined by the transaction type. If transaction type is 0, no transaction id will be present.

64.1.3 Response Header

Magic [1b]	Message Id [vLong]	Op code [1b]	Status [1b]	Topology Change Marker [1b]

- **Opcode** : Op code representing a response to a particular operation, or error condition.
- **Status** : Status of the response, possible values:

`0x00` - No error	`0x01` - Not put/removed/replaced	`0x02` - Key does not exist
`0x81` - Invalid magic or message id	`0x82` - Unknown command	`0x83` - Unknown version
`0x84` - Request parsing error	`0x85` - Server Error	`0x86` - Command timed out

  Exceptional error status responses, those that start with `0x8`..., are followed by the length of the error message (as a `vInt`) and error message itself as String.
- **Topology Change Marker** : This is a marker byte that indicates whether the response is prepended with topology change information. When no topology change follows, the content of this byte is 0. If a topology change follows, its contents are 1.

Topology Change Headers

The following section discusses how the response headers look for topology-aware or hash-distribution-aware clients when there's been a cluster or view formation change. Note that it's the server that makes the decision on whether it sends back the new topology based on the current topology id and the one the client sent. If they're different, it will send back the new topology.

Topology-Aware Client Topology Change Header

This is what topology-aware clients receive as response header when a topology change is sent back:

Response header with topology change marker	**Topology Id [`vInt`]**	**Num servers in topology [`vInt`]**
m1: Host/IP length [`vInt`]	m1: Host/IP address [`string`]	m1: Port [`2b - Unsigned Short`]
m2: Host/IP length [`vInt`]	m2: Host/IP address [`string`]	m2: Port [`2b - Unsigned Short`]
...etc		

- **Num servers in topology** : Number of Infinispan Hot Rod servers running within the cluster. This could be a subset of the entire cluster if only a fraction of those nodes are running Hot Rod servers.
- **Host/IP address length** : Length of hostname or IP address of individual cluster member that Hot Rod client can use to access it. Using variable length here allows for covering for hostnames, IPv4 and IPv6 addresses.
- **Host/IP address** : String containing hostname or IP address of individual cluster member that Hot Rod client can use to access it.
- **Port** : Port that Hot Rod clients can use to communicat with this cluster member.

Hash-Distribution-Aware Client Topology Change Header

This is what hash-distribution-aware clients receive as response header when a topology change is sent back:

Response header with topology change marker	**Topology Id [vInt]**	**Num Key Owners [2b - Unsigned Short]**	**Hash Function Version [1b]**	**Hash space size [vInt]**	**Num servers in topology [vInt]**
m1: Host/IP length [vInt]	m1: Host/IP address [string]	m1: Port [2b - unsigned short]	m1: Hashcode [4b]		
m2: Host/IP length [vInt]	m2: Host/IP address [string]	m2: Port [2b - unsigned short]	m2: Hashcode [4b]		
...etc					

It's important to note that since hash headers rely on the consistent hash algorithm used by the server and this is a factor of the cache interacted with, hash-distribution-aware headers can only be returned to operations that target a particular cache. Currently ping command does not target any cache (this is to change as per ISPN-424, hence calls to ping command with hash-topology-aware client settings will return a hash-distribution-aware header with "Num Key Owners", "Hash Function Version", "Hash space size" and each individual host's hash code all set to 0. This type of header will also be returned as response to operations with hash-topology-aware client settings that are targeting caches that are not configured with distribution.

- **Number key owners** : Globally configured number of copies for each Infinispan distributed key
- **Hash function version** : Hash function version, pointing to a specific hash function in use. See Hot Rod hash functions for details.
- **Hash space size** : Modulus used by Infinispan for for all module arithmetic related to hash code generation. Clients will likely require this information in order to apply the correct hash calculation to the keys.
- **Num servers in topology** : If virtual nodes are disabled, this number represents the number of Hot Rod servers in the cluster. If virtual nodes are enabled, this number represents all the virtual nodes in the cluster which are calculated as (num configured virtual nodes) * (num cluster members). Regardless of whether virtual nodes are configured or not, the number represented by this field indicates the number of 'host:port:hashId' tuples to be read in the response.
- **Hashcode** : 32 bit integer representing the hashcode of a cluster member that a Hot Rod client can use indentify in which cluster member a key is located having applied the CSA to it.

64.1.4 Operations

Get/Remove/ContainsKey/GetWithVersion

- Common request format:

Header	Key Length [vInt]	Key [byte-array]

  - **Key Length** : Length of key. Note that the size of a vint can be up to 5 bytes which in theory can produce bigger numbers than `Integer.MAX_VALUE`. However, Java cannot create a single array that's bigger than Integer.MAX_VALUE, hence the protocol is limiting vint array lengths to Integer.MAX_VALUE.
  - **Key** : Byte array containing the key whose value is being requested.

- Response status:
  - `0x00` - success, if key present/retrieved/removed
  - `0x02` - if key does not exist

- Get response:

Header	Value Length [vInt]	Value [byte-array]

  - **Value Length** : Length of value
  - **Value** : The requested value. If key does not exist, status returned in 0x02. See encoding section for more info.

- Remove response:
  If `ForceReturnPreviousValue` has been passed, remove response will contain previous value (including value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no `ForceReturnPreviousValue` was sent, the response would be empty.

- ContainsKey response:
  Empty

- GetWithVersion response:

Header	Entry Version [8b]	Value Length [vInt]	Value [byte-array]

  - **Entry Version** : Unique value of an existing entry's modification. The protocol does not mandate that entry_version values are sequential. They just need to be unique per update at the key level.

BulkGet

- Request format:

Header	Entry Count [`vInt`]

  - **Entry Count** : Maximum number of Infinispan entries to be returned by the server (entry == key + associated value). Needed to support CacheLoader.load(int). If 0 then all entries are returned (needed for CacheLoader.loadAll()).
- Response:

Header	More [1b]	Key Size 1	Key 1	Value Size 1	Value 1	More [1b]	Key Size 2	Key 2	Value Size 2	Value 2	More [1b] ...

  - **More** : One byte representing whether more entries need to be read from the stream. So, when it's set to 1, it means that an entry followes, whereas when it's set to 0, it's the end of stream and no more entries are left to read.
    For more information on BulkGet look here

Put/PutIfAbsent/Replace

- Common request format:

Header	Key Length [`vInt`]	Key [`byte-array`]	Lifespan [`vInt`]	Max Idle [`vInt`]	Value Length [`vInt`]	Value [`byte-array`]

  - **Lifespan** : Number of seconds that a entry during which the entry is allowed to life. If number of seconds is bigger than 30 days, this number of seconds is treated as UNIX time and so, represents the number of seconds since 1/1/1970. If set to 0, lifespan is unlimited.
  - **Max Idle** : Number of seconds that a entry can be idle before it's evicted from the cache. If 0, no max idle time.
- Put response status:
  - `0x00` if stored
- Replace response status:
  - `0x00` if stored
  - `0x01` if store did not happen because key does not exist
- PutIfAbsent response status:
  - `0x00` if stored
  - `0x01` if store did not happen because key was present
- Put/PutIfAbsent/Replace response:
  If `ForceReturnPreviousValue` has been passed, these responses will contain previous value (and corresponding value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no `ForceReturnPreviousValue` was sent, the response would be empty.

ReplaceIfUnmodified

- Request format:

Header	Key Length [vInt]	Key [byte-array]	Lifespan [vInt]	Max Idle [vInt]	Entry Version [8b]	Value Length [vInt]	Value [byte-array]

  - **Entry Version** : Use the value returned by GetWithVersion operation.
- Response status
  - `0x00` status if replaced/removed
  - `0x01` status if replace/remove did not happen because key had been modified
  - `0x02` status if key does not exist
- Response:
  If `ForceReturnPreviousValue` has been passed, this responses will contain previous value (and corresponding value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no ForceReturnPreviousValue was sent, the response would be empty.

RemoveIfUnmodified

- Request format:

Header	Key Length [vInt]	Key [byte-array]	Entry Version [8b]

- Response status
  - `0x00` status if replaced/removed
  - `0x01` status if replace/remove did not happen because key had been modified
  - `0x02` status if key does not exist
- Response:
  If `ForceReturnPreviousValue` has been passed, this responses will contain previous value (and corresponding value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no ForceReturnPreviousValue was sent, the response would be empty.

Clear

- Request format:

Header

- Response status:
- `0x00` status if infinispan was cleared

Stats

Returns a summary of all available statistics. For each statistic returned, a name and a value is returned both in String UTF-8 format. The supported stats are the following:

Name	Explanation
`timeSinceStart`	Number of seconds since Hot Rod started.
`currentNumberOfEntries`	Number of entries currently in the Hot Rod server.
`totalNumberOfEntries`	Number of entries stored in Hot Rod server.
`stores`	Number of put operations.
`retrievals`	Number of get operations.
`hits`	Number of get hits.
`misses`	Number of get misses.
`removeHits`	Number of removal hits.
`removeMisses`	Number of removal misses.

- Response

Header	Number of stats [`vInt`]	Name1 length [`vInt`]	Name1 [`string`]	Value1 length [`vInt`]	Value1 [`String`]	Name2 length	Name2	Value2 length	Value2	...

  - **Number of stats** : Number of individual stats returned
  - **Name length** : Length of named statistic
  - **Name** : String containing statistic name
  - **Value length** : Length of value field
  - **Value** : String containing statistic value.

Ping

Application level request to see if the server is available.

- Response status:
  - `0x00` - if no errors

64.1.5 Error Handling

Response header	Error Message Length `vInt`	Error Message `string`

Response header contains error op code response and corresponding error status number as well as the following two:

- **Error Message Length** : Length of error message
- **Error message** : Error message. In the case of `0x84`, this error field contains the latest version supported by the hot rod server. Length is defined by total body length.

64.1.6 Multi-Get Operations

A multi-get operation is a form of get operation that instead of requesting a single key, requests a set of keys. The Hot Rod protocol does not include such operation but remote Hot Rod clients could easily implement this type of operations by either parallelizing/pipelining individual get requests. Another possibility would be for remote clients to use async or non-blocking get requests. For example, if a client wants N keys, it could send send N async get requests and then wait for all the replies. Finally, multi-get is not to be confused with bulk-get operations. In bulk-gets, either all or a number of keys are retrieved, but the client does not know which keys to retrieve, whereas in multi-get, the client defines which keys to retrieve.

64.1.7 Example - Put request

- Coded request

Byte	0	1	2	3	4	5	6	7
8	`0xA0`	`0x09`	`0x41`	`0x01`	`0x07`	`0x4D ('M')`	`0x79 ('y')`	`0x43 ('C')`
16	`0x61 ('a')`	`0x63 ('c')`	`0x68 ('h')`	`0x65 ('e')`	`0x00`	`0x03`	`0x00`	`0x00`
24	`0x00`	`0x05`	`0x48 ('H')`	`0x65 ('e')`	`0x6C ('l')`	`0x6C ('l')`	`0x6F ('o')`	`0x00`
32	`0x00`	`0x05`	`0x57 ('W')`	`0x6F ('o')`	`0x72 ('r')`	`0x6C ('l')`	`0x64 ('d')`	

- Field explanation

Field Name	Value	Field Name	Value
Magic (0)	`0xA0`	Message Id (1)	`0x09`
Version (2)	`0x41`	Opcode (3)	`0x01`
Cache name length (4)	`0x07`	Cache name(5-11)	`'MyCache'`
Flag (12)	`0x00`	Client Intelligence (13)	`0x03`
Topology Id (14)	`0x00`	Transaction Type (15)	`0x00`
Transaction Id (16)	`0x00`	Key field length (17)	`0x05`
Key (18 - 22)	`'Hello'`	Lifespan (23)	`0x00`
Max idle (24)	`0x00`	Value field length (25)	`0x05`
Value (26-30)	`'World'`		

- Coded response

Byte	0	1	2	3	4	5	6	7
8	`0xA1`	`0x09`	`0x01`	`0x00`	`0x00`			

- Field Explanation

Field Name	Value	Field Name	Value
Magic (0)	`0xA1`	Message Id (1)	`0x09`
Opcode (2)	`0x01`	Status (3)	`0x00`
Topology change marker (4)	`0x00`		

64.2 Hot Rod Protocol - Version 1.1

64.2.1 Introduction

This article provides detailed information about the first version of the custom TCP client/server Hot Rod protocol.

Infinispan versions

This version of the protocol is implemented since Infinispan 5.1.0.FINAL

All key and values are sent and stored as byte arrays. Hot Rod makes no assumptions about their types. Some clarifications about the other types:

- `vInt`: Refers to unsigned variable length integer values as specified in here. They're between 1 and 5 bytes long.
- `vLong`: Refers to unsigned variable length long values similar to vInt but applied to longer values. They're between 1 and 9 bytes long.
- `String`: Strings are always represented using UTF-8 encoding.

64.2.2 Request Header

The header for a request is composed of:

Magic [1b]	Message Id [vLong]	Version [1b]	Opcode [1b]	Cache Name Length [vInt]	Cache Name [string]	Flags [vInt]	Client Intelligence [1b]	Topology Id [vInt]	Transaction Type [1b]

- **Magic** : Possible values are:
  - `0xA0` - Infinispan Cache Request Marker
  - `0xA1` - Infinispan Cache Response Marker
- **Message Id** : Id of the message that will be copied back in the response. This allows for hot rod clients to implement the protocol in an asynchronous way.
- **Version** : Infinispan hot rod server version.

Updated for 1.1

The value of this field in version 1.1 is `11`

- **Opcode** : Possible values are only the ones on the request column:

Request operation codes	Response operation codes
`0x01` - put request	`0x02` - put response
`0x03` - get request	`0x04` - get response
`0x05` - putIfAbsent request	`0x06` - putIfAbsent response
`0x07` - replace request	`0x08` - replace response
`0x09` - replaceIfUnmodified request	`0x0A` - replaceIfUnmodified response
`0x0B` - remove request	`0x0C` - remove response
`0x0D` - removeIfUnmodified request	`0x0E` - removeIfUnmodified response
`0x0F` - containsKey request	`0x10` - containsKey response
`0x11` - getWithVersion request	`0x12` - getWithVersion response
`0x13` - clear request	`0x14` - clear response
`0x15` - stats request	`0x16` - stats response
`0x17` - ping request	`0x18` - ping response
`0x19` - bulkGet request	`0x1A` - bulkGet response
-	`0x50` - error response

- **Cache Name Length** : Length of cache name. If the passed length is 0 (followed by no cache name), the operation will interact with the default cache.
- **Cache Name** : Name of cache on which to operate. This name must match the name of predefined cache in the Infinispan configuration file.
- **Flags** : A variable length number representing flags passed to the system. Each flags is represented by a bit. Note that since this field is sent as variable length, the most significant bit in a byte is used to determine whether more bytes need to be read, hence this bit does not represent any flag. Using this model allows for flags to be combined in a short space. Here are the current values for each flag:

`0x0001`	ForceReturnPreviousValue

- **Client Intelligence** : This byte hints the server on the client capabilities:
  - `0x01` - basic client, interested in neither cluster nor hash information
  - `0x02` - topology-aware client, interested in cluster information
  - `0x03` - hash-distribution-aware client, that is interested in both cluster and hash information
- **Topology Id** : This field represents the last known view in the client. Basic clients will only send 0 in this field. When topology-aware or hash-distribution-aware clients will send 0 until they have received a reply from the server with the current view id. Afterwards, they should send that view id until they receive a new view id in a response

- **Transaction Type** : This is a 1 byte field, containing one of the following well-known supported transaction types (For this version of the protocol, the only supported transaction type is 0):
    - `0` - Non-transactional call, or client does not support transactions. The subsequent TX_ID field will be omitted.
    - `1` - X/Open XA transaction ID (XID). This is a well-known, fixed-size format.
- **Transaction Id** : The byte array uniquely identifying the transaction associated to this call. It's length is determined by the transaction type. If transaction type is 0, no transaction id will be present.

64.2.3 Response Header

Magic [1b]	Message Id [vLong]	Op code [1b]	Status [1b]	Topology Change Marker [1b]

- **Opcode** : Op code representing a response to a particular operation, or error condition.
- **Status** : Status of the response, possible values:

`0x00` - No error	`0x01` - Not put/removed/replaced	`0x02` - Key does not exist
`0x81` - Invalid magic or message id	`0x82` - Unknown command	`0x83` - Unknown version
`0x84` - Request parsing error	`0x85` - Server Error	`0x86` - Command timed out

  Exceptional error status responses, those that start with `0x8`..., are followed by the length of the error message (as a `vInt`) and error message itself as String.
- **Topology Change Marker** : This is a marker byte that indicates whether the response is prepended with topology change information. When no topology change follows, the content of this byte is 0. If a topology change follows, its contents are 1.

Topology Change Headers

The following section discusses how the response headers look for topology-aware or hash-distribution-aware clients when there's been a cluster or view formation change. Note that it's the server that makes the decision on whether it sends back the new topology based on the current topology id and the one the client sent. If they're different, it will send back the new topology.

Topology-Aware Client Topology Change Header

This is what topology-aware clients receive as response header when a topology change is sent back:

Response header with topology change marker	**Topology Id [`vInt`]**	**Num servers in topology [`vInt`]**
m1: Host/IP length [`vInt`]	m1: Host/IP address [`string`]	m1: Port [`2b - Unsigned Short`]
m2: Host/IP length [`vInt`]	m2: Host/IP address [`string`]	m2: Port [`2b - Unsigned Short`]
...etc		

- **Num servers in topology** : Number of Infinispan Hot Rod servers running within the cluster. This could be a subset of the entire cluster if only a fraction of those nodes are running Hot Rod servers.
- **Host/IP address length** : Length of hostname or IP address of individual cluster member that Hot Rod client can use to access it. Using variable length here allows for covering for hostnames, IPv4 and IPv6 addresses.
- **Host/IP address** : String containing hostname or IP address of individual cluster member that Hot Rod client can use to access it.
- **Port** : Port that Hot Rod clients can use to communicat with this cluster member.

Hash-Distribution-Aware Client Topology Change Header

Updated for 1.1

This section has been modified to be more efficient when talking to distributed caches with virtual nodes enabled.

This is what hash-distribution-aware clients receive as response header when a topology change is sent back:

Response header with topology change marker	**Topology Id [vInt]**	**Num Key Owners [2b - Unsigned Short]**	**Hash Function Version [1b]**	**Hash space size [vInt]**	**Num servers in topology [vInt]**	**Num Virtual Nodes Owners [vInt]**
m1: Host/IP length [vInt]	m1: Host/IP address [string]	m1: Port [2b - unsigned short]	m1: Hashcode [4b]			
m2: Host/IP length [vInt]	m2: Host/IP address [string]	m2: Port [2b - unsigned short]	m1: Hashcode [4b]			
...etc						

- **Number key owners** : Globally configured number of copies for each Infinispan distributed key. If the cache is not configured with distribution, this field will return `0`.
- **Hash function version** : Hash function version, pointing to a specific hash function in use. See Hot Rod hash functions for details. If cache is not configured with distribution, this field will contain `0`.
- **Hash space size** : Modulus used by Infinispan for for all module arithmetic related to hash code generation. Clients will likely require this information in order to apply the correct hash calculation to the keys. If cache is not configured with distribution, this field will contain `0`.
- **Num servers in topology** : Represents the number of servers in the Hot Rod cluster which represents the number of host:port pairings to be read in the header.
- **Number virtual nodes** : Field added in version 1.1 of the protocol that represents the number of configured virtual nodes. If no virtual nodes are configured or the cache is not configured with distribution, this field will contain `0`.

Server node hash code calculation

Adding support for virtual nodes has made version 1.0 of the Hot Rod protocol impractical due to bandwidth it would have taken to return hash codes for all virtual nodes in the clusters (this number could easily be in the millions). So, as of version 1.1 of the Hot Rod protocol, clients are given the base hash id or hash code of each server, and then they have to calculate the real hash position of each server both with and without virtual nodes configured. Here are the rules clients should follow when trying to calculate a node's hash code:

1. With **virtual nodes disabled**:
   Once clients have received the base hash code of the server, they need to normalize it in order to find the exact position of the hash wheel. The process of normalization involves passing the base hash code to the hash function, and then do a small calculation to avoid negative values. The resulting number is the node's position in the hash wheel:

   ```
   public static int getNormalizedHash(int nodeBaseHashCode, Hash hashFct) {
      return hashFct.hash(nodeBaseHashCode) & Integer.MAX_VALUE; // make sure no negative
   numbers are involved.
   }
   ```

2. With **virtual nodes enabled**:
   In this case, each node represents N different virtual nodes, and to calculate each virtual node's hash code, we need to take the the range of numbers between 0 and N-1 and apply the following logic:
   - For virtual node with 0 as id, use the technique used to retrieve a node's hash code, as shown in the previous section.
   - For virtual nodes from 1 to N-1 ids, execute the following logic:

     ```
     public static int virtualNodeHashCode(int nodeBaseHashCode, int id, Hash hashFct) {
        int virtualNodeBaseHashCode = id;
        virtualNodeBaseHashCode = 31 * virtualNodeBaseHashCode + nodeBaseHashCode;
        return getNormalizedHash(virtualNodeBaseHashCode, hashFct);
     }
     ```

64.2.4 Operations

Get/Remove/ContainsKey/GetWithVersion

- Common request format:

Header	Key Length [`vInt`]	Key [`byte-array`]

  - **Key Length** : Length of key. Note that the size of a vint can be up to 5 bytes which in theory can produce bigger numbers than `Integer.MAX_VALUE`. However, Java cannot create a single array that's bigger than Integer.MAX_VALUE, hence the protocol is limiting vint array lengths to Integer.MAX_VALUE.
  - **Key** : Byte array containing the key whose value is being requested.

- Response status:
  - `0x00` - success, if key present/retrieved/removed
  - `0x02` - if key does not exist

- Get response:

Header	Value Length [`vInt`]	Value [`byte-array`]

  - **Value Length** : Length of value
  - **Value** : The requested value. If key does not exist, status returned in 0x02. See encoding section for more info.

- Remove response:
  If `ForceReturnPreviousValue` has been passed, remove response will contain previous value (including value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no `ForceReturnPreviousValue` was sent, the response would be empty.

- ContainsKey response:
  Empty

- GetWithVersion response:

Header	Entry Version [8b]	Value Length [`vInt`]	Value [`byte-array`]

  - **Entry Version** : Unique value of an existing entry's modification. The protocol does not mandate that entry_version values are sequential. They just need to be unique per update at the key level.

BulkGet

- Request format:

Header	Entry Count [`vInt`]

  - **Entry Count** : Maximum number of Infinispan entries to be returned by the server (entry == key + associated value). Needed to support CacheLoader.load(int). If 0 then all entries are returned (needed for CacheLoader.loadAll()).
- Response:

Header	More [`1b`]	Key Size 1	Key 1	Value Size 1	Value 1	More [`1b`]	Key Size 2	Key 2	Value Size 2	Value 2	More [`1b`] ...

  - **More** : One byte representing whether more entries need to be read from the stream. So, when it's set to 1, it means that an entry followes, whereas when it's set to 0, it's the end of stream and no more entries are left to read.
    For more information on BulkGet look here

Put/PutIfAbsent/Replace

- Common request format:

Header	Key Length [`vInt`]	Key [`byte-array`]	Lifespan [`vInt`]	Max Idle [`vInt`]	Value Length [`vInt`]	Value [`byte-array`]

  - **Lifespan** : Number of seconds that a entry during which the entry is allowed to life. If number of seconds is bigger than 30 days, this number of seconds is treated as UNIX time and so, represents the number of seconds since 1/1/1970. If set to 0, lifespan is unlimited.
  - **Max Idle** : Number of seconds that a entry can be idle before it's evicted from the cache. If 0, no max idle time.
- Put response status:
  - `0x00` if stored
- Replace response status:
  - `0x00` if stored
  - `0x01` if store did not happen because key does not exist
- PutIfAbsent response status:
  - `0x00` if stored
  - `0x01` if store did not happen because key was present
- Put/PutIfAbsent/Replace response:
  If `ForceReturnPreviousValue` has been passed, these responses will contain previous value (and corresponding value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no `ForceReturnPreviousValue` was sent, the response would be empty.

ReplaceIfUnmodified

- Request format:

Header	Key Length [vInt]	Key [byte-array]	Lifespan [vInt]	Max Idle [vInt]	Entry Version [8b]	Value Length [vInt]	Value [byte-array]

  - **Entry Version** : Use the value returned by GetWithVersion operation.
- Response status
  - `0x00` status if replaced/removed
  - `0x01` status if replace/remove did not happen because key had been modified
  - `0x02` status if key does not exist
- Response:
  If `ForceReturnPreviousValue` has been passed, this responses will contain previous value (and corresponding value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no ForceReturnPreviousValue was sent, the response would be empty.

RemoveIfUnmodified

- Request format:

Header	Key Length [vInt]	Key [byte-array]	Entry Version [8b]

- Response status
  - `0x00` status if replaced/removed
  - `0x01` status if replace/remove did not happen because key had been modified
  - `0x02` status if key does not exist
- Response:
  If `ForceReturnPreviousValue` has been passed, this responses will contain previous value (and corresponding value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no ForceReturnPreviousValue was sent, the response would be empty.

Clear

- Request format:

Header

- Response status:
- `0x00` status if infinispan was cleared

Stats

Returns a summary of all available statistics. For each statistic returned, a name and a value is returned both in String UTF-8 format. The supported stats are the following:

Name	Explanation
`timeSinceStart`	Number of seconds since Hot Rod started.
`currentNumberOfEntries`	Number of entries currently in the Hot Rod server.
`totalNumberOfEntries`	Number of entries stored in Hot Rod server.
`stores`	Number of put operations.
`retrievals`	Number of get operations.
`hits`	Number of get hits.
`misses`	Number of get misses.
`removeHits`	Number of removal hits.
`removeMisses`	Number of removal misses.

- Response

Header	Number of stats [`vInt]`	Name1 length [`vInt]`	Name1 [`string`]	Value1 length [`vInt]`	Value1 [`String`]	Name2 length	Name2	Value2 length	Value2	...

  - **Number of stats** : Number of individual stats returned
  - **Name length** : Length of named statistic
  - **Name** : String containing statistic name
  - **Value length** : Length of value field
  - **Value** : String containing statistic value.

Ping

Application level request to see if the server is available.

- Response status:
  - `0x00` - if no errors

64.2.5 Error Handling

Response header	Error Message Length `vInt`	Error Message `string`

Response header contains error op code response and corresponding error status number as well as the following two:

- **Error Message Length** : Length of error message
- **Error message** : Error message. In the case of `0x84`, this error field contains the latest version supported by the hot rod server. Length is defined by total body length.

64.2.6 Multi-Get Operations

A multi-get operation is a form of get operation that instead of requesting a single key, requests a set of keys. The Hot Rod protocol does not include such operation but remote Hot Rod clients could easily implement this type of operations by either parallelizing/pipelining individual get requests. Another possibility would be for remote clients to use async or non-blocking get requests. For example, if a client wants N keys, it could send send N async get requests and then wait for all the replies. Finally, multi-get is not to be confused with bulk-get operations. In bulk-gets, either all or a number of keys are retrieved, but the client does not know which keys to retrieve, whereas in multi-get, the client defines which keys to retrieve.

64.2.7 Example - Put request

- Coded request

Byte	0	1	2	3	4	5	6	7
8	`0xA0`	`0x09`	`0x41`	`0x01`	`0x07`	`0x4D ('M')`	`0x79 ('y')`	`0x43 ('C')`
16	`0x61 ('a')`	`0x63 ('c')`	`0x68 ('h')`	`0x65 ('e')`	`0x00`	`0x03`	`0x00`	`0x00`
24	`0x00`	`0x05`	`0x48 ('H')`	`0x65 ('e')`	`0x6C ('l')`	`0x6C ('l')`	`0x6F ('o')`	`0x00`
32	`0x00`	`0x05`	`0x57 ('W')`	`0x6F ('o')`	`0x72 ('r')`	`0x6C ('l')`	`0x64 ('d')`	

- Field explanation

Field Name	Value	Field Name	Value
Magic (0)	`0xA0`	Message Id (1)	`0x09`
Version (2)	`0x41`	Opcode (3)	`0x01`
Cache name length (4)	`0x07`	Cache name(5-11)	`'MyCache'`
Flag (12)	`0x00`	Client Intelligence (13)	`0x03`
Topology Id (14)	`0x00`	Transaction Type (15)	`0x00`
Transaction Id (16)	`0x00`	Key field length (17)	`0x05`
Key (18 - 22)	`'Hello'`	Lifespan (23)	`0x00`
Max idle (24)	`0x00`	Value field length (25)	`0x05`
Value (26-30)	`'World'`		

- Coded response

Byte	0	1	2	3	4	5	6	7
8	`0xA1`	`0x09`	`0x01`	`0x00`	`0x00`			

- Field Explanation

Field Name	Value	Field Name	Value
Magic (0)	`0xA1`	Message Id (1)	`0x09`
Opcode (2)	`0x01`	Status (3)	`0x00`
Topology change marker (4)	`0x00`		

64.3 Hot Rod Protocol - Version 1.2

64.3.1 Introduction

This article provides detailed information about the first version of the custom TCP client/server Hot Rod protocol.

Infinispan versions

This version of the protocol is implemented since Infinispan 5.2.0.FINAL

All key and values are sent and stored as byte arrays. Hot Rod makes no assumptions about their types. Some clarifications about the other types:

- `vInt`: Refers to unsigned variable length integer values as specified in here. They're between 1 and 5 bytes long.
- `vLong`: Refers to unsigned variable length long values similar to vInt but applied to longer values. They're between 1 and 9 bytes long.
- `String`: Strings are always represented using UTF-8 encoding.

64.3.2 Request Header

The header for a request is composed of:

Magic [1b]	Message Id [vLong]	Version [1b]	Opcode [1b]	Cache Name Length [vInt]	Cache Name [string]	Flags [vInt]	Client Intelligence [1b]	Topology Id [vInt]	Transaction Type [1b]

- **Magic** : Possible values are:
  - `0xA0` - Infinispan Cache Request Marker
  - `0xA1` - Infinispan Cache Response Marker
- **Message Id** : Id of the message that will be copied back in the response. This allows for hot rod clients to implement the protocol in an asynchronous way.
- **Version** : Infinispan hot rod server version.

Updated for 1.2

The value of this field in version 1.2 is `12`

- **Opcode** : Possible values are only the ones on the request column:

Request operation codes	Response operation codes
`0x01` - put request	`0x02` - put response
`0x03` - get request	`0x04` - get response
`0x05` - putIfAbsent request	`0x06` - putIfAbsent response
`0x07` - replace request	`0x08` - replace response
`0x09` - replaceIfUnmodified request	`0x0A` - replaceIfUnmodified response
`0x0B` - remove request	`0x0C` - remove response
`0x0D` - removeIfUnmodified request	`0x0E` - removeIfUnmodified response
`0x0F` - containsKey request	`0x10` - containsKey response
`0x11` - getWithVersion request	`0x12` - getWithVersion response
`0x13` - clear request	`0x14` - clear response
`0x15` - stats request	`0x16` - stats response
`0x17` - ping request	`0x18` - ping response
`0x19` - bulkGet request	`0x1A` - bulkGet response
`0x1B` - getWithMetadata request	`0x1C` - getWithMetadata response
`0x1D` - bulkKeysGet request	`0x1E` - bulkKeysGet response
-	`0x50` - error response

- **Cache Name Length** : Length of cache name. If the passed length is 0 (followed by no cache name), the operation will interact with the default cache.
- **Cache Name** : Name of cache on which to operate. This name must match the name of predefined cache in the Infinispan configuration file.
- **Flags** : A variable length number representing flags passed to the system. Each flags is represented by a bit. Note that since this field is sent as variable length, the most significant bit in a byte is used to determine whether more bytes need to be read, hence this bit does not represent any flag. Using this model allows for flags to be combined in a short space. Here are the current values for each flag:

`0x0001`	ForceReturnPreviousValue
`0x0002`	DefaultLifespan
`0x0004`	DefaultMaxIdle

- **Client Intelligence** : This byte hints the server on the client capabilities:
  - `0x01` - basic client, interested in neither cluster nor hash information
  - `0x02` - topology-aware client, interested in cluster information
  - `0x03` - hash-distribution-aware client, that is interested in both cluster and hash information
- **Topology Id** : This field represents the last known view in the client. Basic clients will only send 0 in this field. When topology-aware or hash-distribution-aware clients will send 0 until they have received a reply from the server with the current view id. Afterwards, they should send that view id until they receive a new view id in a response
- **Transaction Type** : This is a 1 byte field, containing one of the following well-known supported transaction types (For this version of the protocol, the only supported transaction type is 0):
  - `0` - Non-transactional call, or client does not support transactions. The subsequent TX_ID field will be omitted.
  - `1` - X/Open XA transaction ID (XID). This is a well-known, fixed-size format.
- **Transaction Id** : The byte array uniquely identifying the transaction associated to this call. It's length is determined by the transaction type. If transaction type is 0, no transaction id will be present.

64.3.3 Response Header

Magic [1b]	Message Id [vLong]	Op code [1b]	Status [1b]	Topology Change Marker [1b]

- **Opcode** : Op code representing a response to a particular operation, or error condition.
- **Status** : Status of the response, possible values:

`0x00` - No error	`0x01` - Not put/removed/replaced	`0x02` - Key does not exist
`0x81` - Invalid magic or message id	`0x82` - Unknown command	`0x83` - Unknown version
`0x84` - Request parsing error	`0x85` - Server Error	`0x86` - Command timed out

  Exceptional error status responses, those that start with `0x8`..., are followed by the length of the error message (as a `vInt`) and error message itself as String.
- **Topology Change Marker** : This is a marker byte that indicates whether the response is prepended with topology change information. When no topology change follows, the content of this byte is 0. If a topology change follows, its contents are 1.

Topology Change Headers

The following section discusses how the response headers look for topology-aware or hash-distribution-aware clients when there's been a cluster or view formation change. Note that it's the server that makes the decision on whether it sends back the new topology based on the current topology id and the one the client sent. If they're different, it will send back the new topology.

Topology-Aware Client Topology Change Header

This is what topology-aware clients receive as response header when a topology change is sent back:

Response header with topology change marker	**Topology Id [`vInt`]**	**Num servers in topology [`vInt`]**
m1: Host/IP length [`vInt`]	m1: Host/IP address [`string`]	m1: Port [`2b - Unsigned Short`]
m2: Host/IP length [`vInt`]	m2: Host/IP address [`string`]	m2: Port [`2b - Unsigned Short`]
...etc		

- **Num servers in topology** : Number of Infinispan Hot Rod servers running within the cluster. This could be a subset of the entire cluster if only a fraction of those nodes are running Hot Rod servers.
- **Host/IP address length** : Length of hostname or IP address of individual cluster member that Hot Rod client can use to access it. Using variable length here allows for covering for hostnames, IPv4 and IPv6 addresses.
- **Host/IP address** : String containing hostname or IP address of individual cluster member that Hot Rod client can use to access it.
- **Port** : Port that Hot Rod clients can use to communicat with this cluster member.

Hash-Distribution-Aware Client Topology Change Header

This is what hash-distribution-aware clients receive as response header when a topology change is sent back:

Response header with topology change marker	**Topology Id [`vInt`]**	**Num Key Owners [`2b - Unsigned Short`]**	**Hash Function Version [`1b`]**	**Hash space size [`vInt`]**	**Num servers in topology [vInt]**	**Num Virtual Nodes Owners [`vInt`]**
m1: Host/IP length [`vInt`]	m1: Host/IP address [`string`]	m1: Port [`2b - unsigned short`]	m1: Hashcode [4b]			
m2: Host/IP length [`vInt`]	m2: Host/IP address [`string`]	m2: Port [`2b - unsigned short`]	m1: Hashcode [4b]			
...etc						

- **Number key owners** : Globally configured number of copies for each Infinispan distributed key. If the cache is not configured with distribution, this field will return `0`.
- **Hash function version** : Hash function version, pointing to a specific hash function in use. See Hot Rod hash functions for details. If cache is not configured with distribution, this field will contain `0`.
- **Hash space size** : Modulus used by Infinispan for for all module arithmetic related to hash code generation. Clients will likely require this information in order to apply the correct hash calculation to the keys. If cache is not configured with distribution, this field will contain `0`.
- **Num servers in topology** : Represents the number of servers in the Hot Rod cluster which represents the number of host:port pairings to be read in the header.
- **Number virtual nodes** : Field added in version 1.1 of the protocol that represents the number of configured virtual nodes. If no virtual nodes are configured or the cache is not configured with distribution, this field will contain `0`.

Server node hash code calculation

Adding support for virtual nodes has made version 1.0 of the Hot Rod protocol impractical due to bandwidth it would have taken to return hash codes for all virtual nodes in the clusters (this number could easily be in the millions). So, as of version 1.1 of the Hot Rod protocol, clients are given the base hash id or hash code of each server, and then they have to calculate the real hash position of each server both with and without virtual nodes configured. Here are the rules clients should follow when trying to calculate a node's hash code:

1. With **virtual nodes disabled**:
   Once clients have received the base hash code of the server, they need to normalize it in order to find the exact position of the hash wheel. The process of normalization involves passing the base hash code to the hash function, and then do a small calculation to avoid negative values. The resulting number is the node's position in the hash wheel:

```
public static int getNormalizedHash(int nodeBaseHashCode, Hash hashFct) {
   return hashFct.hash(nodeBaseHashCode) & Integer.MAX_VALUE; // make sure no negative
numbers are involved.
}
```

2. With **virtual nodes enabled**:
   In this case, each node represents N different virtual nodes, and to calculate each virtual node's hash code, we need to take the the range of numbers between 0 and N-1 and apply the following logic:
   - For virtual node with 0 as id, use the technique used to retrieve a node's hash code, as shown in the previous section.
   - For virtual nodes from 1 to N-1 ids, execute the following logic:

```
public static int virtualNodeHashCode(int nodeBaseHashCode, int id, Hash hashFct) {
   int virtualNodeBaseHashCode = id;
   virtualNodeBaseHashCode = 31 * virtualNodeBaseHashCode + nodeBaseHashCode;
   return getNormalizedHash(virtualNodeBaseHashCode, hashFct);
}
```

64.3.4 Operations

Get/Remove/ContainsKey/GetWithVersion/GetWithMetadata

- Common request format:

Header	Key Length [vInt]	Key [byte-array]

  - **Key Length** : Length of key. Note that the size of a vint can be up to 5 bytes which in theory can produce bigger numbers than `Integer.MAX_VALUE`. However, Java cannot create a single array that's bigger than Integer.MAX_VALUE, hence the protocol is limiting vint array lengths to Integer.MAX_VALUE.
  - **Key** : Byte array containing the key whose value is being requested.

- Response status:
  - `0x00` - success, if key present/retrieved/removed
  - `0x02` - if key does not exist

- Get response:

Header	Value Length [vInt]	Value [byte-array]

  - **Value Length** : Length of value
  - **Value** : The requested value. If key does not exist, status returned in 0x02. See encoding section for more info.

- Remove response:
  If `ForceReturnPreviousValue` has been passed, remove response will contain previous value (including value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no `ForceReturnPreviousValue` was sent, the response would be empty.

- ContainsKey response:
  Empty

- GetWithVersion response:

Header	Entry Version [8b]	Value Length [vInt]	Value [byte-array]

  - **Entry Version** : Unique value of an existing entry's modification. The protocol does not mandate that entry_version values are sequential. They just need to be unique per update at the key level.

- GetWithMetadata response:

Header	Flag (byte)	Created [Long] (optional)	Lifespan [vInt] (optional)	LastUsed [Long] (optional)	MaxIdle [vInt] (optional)	Entry Version [8b]	Value Length [vInt]	Value [byte-array]

  - **Flag** : a flag indicating whether the response contains expiration information. The value of the flag is obtained as a bitwise OR operation between INFINITE_LIFESPAN (0x01) and INFINITE_MAXIDLE (0x02)
  - **Created** : a Long representing the timestamp when the entry was created on the server. This value is returned only if the flag's INFINITE_LIFESPAN bit is not set
  - **Lifespan** : a vInt representing the lifespan of the entry in seconds. This value is returned only if the flag's INFINITE_LIFESPAN bit is not set
  - **LastUsed** : a Long representing the timestamp when the entry was last accessed on the server. This value is returned only if the flag's INFINITE_MAXIDLE bit is not set
  - **MaxIdle :** a vInt representing the maxIdle of the entry in seconds. This value is returned only if the flag's INFINITE_MAXIDLE bit is not set
  - **Entry Version** : Unique value of an existing entry's modification. The protocol does not mandate that entry_version values are sequential. They just need to be unique per update at the key level.

BulkGet

- Request format:

Header	Entry Count [vInt]

  - **Entry Count** : Maximum number of Infinispan entries to be returned by the server (entry == key + associated value). Needed to support CacheLoader.load(int). If 0 then all entries are returned (needed for CacheLoader.loadAll()).
- Response:

Header	More [1b]	Key Size 1	Key 1	Value Size 1	Value 1	More [1b]	Key Size 2	Key 2	Value Size 2	Value 2	More [1b] ...

  - **More** : One byte representing whether more entries need to be read from the stream. So, when it's set to 1, it means that an entry followes, whereas when it's set to 0, it's the end of stream and no more entries are left to read.
    For more information on BulkGet look here

BulkKeysGet

- Request format:

Header	Scope [`vInt`]

  - **Scope** :
    0 - Default Scope - This scope is used by RemoteCache.keySet() method. If the remote cache is a distributed cache, the server launch a map/reduce operation to retrieve all keys from all of the nodes. (Remember, a topology-aware Hot Rod Client could be load balancing the request to any one node in the cluster). Otherwise, it'll get keys from the cache instance local to the server receiving the request (that is because the keys should be the same across all nodes in a replicated cache).
    1 - Global Scope - This scope behaves the same to Default Scope.
    2 - Local Scope - In case when remote cache is a distributed cache, the server will not launch a map/reduce operation to retrieve keys from all nodes. Instead, it'll only get keys local from the cache instance local to the server receiving the request.
- Response:

Header	More [1b]	Key Size 1	Key 1 More [1b]	Key Size 2	Key 2	More [1b] ...

  - **More** : One byte representing whether more keys need to be read from the stream. So, when it's set to 1, it means that a key followes, whereas when it's set to 0, it's the end of stream and no more entries are left to read.

Put/PutIfAbsent/Replace

- Common request format:

Header	Key Length [`vInt]`	Key [`byte-array ]`	Lifespan [`vInt]`	Max Idle [`vInt]`	Value Length `[vInt]`	Value [`byte-array]`

  - **Lifespan** : Number of seconds that a entry during which the entry is allowed to life. If number of seconds is bigger than 30 days, this number of seconds is treated as UNIX time and so, represents the number of seconds since 1/1/1970. If set to 0, lifespan is unlimited.
  - **Max Idle** : Number of seconds that a entry can be idle before it's evicted from the cache. If 0, no max idle time.
- Put response status:
  - `0x00` if stored
- Replace response status:
  - `0x00` if stored
  - `0x01` if store did not happen because key does not exist
- PutIfAbsent response status:
  - `0x00` if stored
  - `0x01` if store did not happen because key was present
- Put/PutIfAbsent/Replace response:
  If `ForceReturnPreviousValue` has been passed, these responses will contain previous value (and corresponding value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no `ForceReturnPreviousValue` was sent, the response would be empty.

ReplaceIfUnmodified

- Request format:

Header	Key Length [`vInt]`	Key [`byte-array ]`	Lifespan `[vInt]`	Max Idle [`vInt]`	Entry Version [`8b]`	Value Length [`vInt]`	Value [`byte-array ]`

  - **Entry Version** : Use the value returned by GetWithVersion operation.
- Response status
  - `0x00` status if replaced/removed
  - `0x01` status if replace/remove did not happen because key had been modified
  - `0x02` status if key does not exist
- Response:
  If `ForceReturnPreviousValue` has been passed, this responses will contain previous value (and corresponding value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no ForceReturnPreviousValue was sent, the response would be empty.

RemoveIfUnmodified

- Request format:

Header	Key Length [vInt]	Key [byte-array]	Entry Version [8b]

- Response status
    - `0x00` status if replaced/removed
    - `0x01` status if replace/remove did not happen because key had been modified
    - `0x02` status if key does not exist
- Response:
  If `ForceReturnPreviousValue` has been passed, this responses will contain previous value (and corresponding value length) for that key. If the key does not exist or previous was null, value length would be 0. Otherwise, if no ForceReturnPreviousValue was sent, the response would be empty.

Clear

- Request format:

Header

- Response status:
- `0x00` status if infinispan was cleared

Stats

Returns a summary of all available statistics. For each statistic returned, a name and a value is returned both in String UTF-8 format. The supported stats are the following:

Name	Explanation
`timeSinceStart`	Number of seconds since Hot Rod started.
`currentNumberOfEntries`	Number of entries currently in the Hot Rod server.
`totalNumberOfEntries`	Number of entries stored in Hot Rod server.
`stores`	Number of put operations.
`retrievals`	Number of get operations.
`hits`	Number of get hits.
`misses`	Number of get misses.
`removeHits`	Number of removal hits.
`removeMisses`	Number of removal misses.

- Response

Header	Number of stats [`vInt`]	Name1 length [`vInt`]	Name1 [`string`]	Value1 length [`vInt`]	Value1 [`String`]	Name2 length	Name2	Value2 length	Value2	...

  - **Number of stats** : Number of individual stats returned
  - **Name length** : Length of named statistic
  - **Name** : String containing statistic name
  - **Value length** : Length of value field
  - **Value** : String containing statistic value.

Ping

Application level request to see if the server is available.

- Response status:
  - `0x00` - if no errors

64.3.5 Error Handling

Response header	Error Message Length `vInt`	Error Message `string`

Response header contains error op code response and corresponding error status number as well as the following two:

- **Error Message Length** : Length of error message
- **Error message** : Error message. In the case of `0x84`, this error field contains the latest version supported by the hot rod server. Length is defined by total body length.

64.3.6 Multi-Get Operations

A multi-get operation is a form of get operation that instead of requesting a single key, requests a set of keys. The Hot Rod protocol does not include such operation but remote Hot Rod clients could easily implement this type of operations by either parallelizing/pipelining individual get requests. Another possibility would be for remote clients to use async or non-blocking get requests. For example, if a client wants N keys, it could send send N async get requests and then wait for all the replies. Finally, multi-get is not to be confused with bulk-get operations. In bulk-gets, either all or a number of keys are retrieved, but the client does not know which keys to retrieve, whereas in multi-get, the client defines which keys to retrieve.

64.3.7 Example - Put request

- Coded request

Byte	0	1	2	3	4	5	6	7
8	`0xA0`	`0x09`	`0x41`	`0x01`	`0x07`	`0x4D ('M')`	`0x79 ('y')`	`0x43 ('C')`
16	`0x61 ('a')`	`0x63 ('c')`	`0x68 ('h')`	`0x65 ('e')`	`0x00`	`0x03`	`0x00`	`0x00`
24	`0x00`	`0x05`	`0x48 ('H')`	`0x65 ('e')`	`0x6C ('l')`	`0x6C ('l')`	`0x6F ('o')`	`0x00`
32	`0x00`	`0x05`	`0x57 ('W')`	`0x6F ('o')`	`0x72 ('r')`	`0x6C ('l')`	`0x64 ('d')`	

- Field explanation

Field Name	Value	Field Name	Value
Magic (0)	`0xA0`	Message Id (1)	`0x09`
Version (2)	`0x41`	Opcode (3)	`0x01`
Cache name length (4)	`0x07`	Cache name(5-11)	`'MyCache'`
Flag (12)	`0x00`	Client Intelligence (13)	`0x03`
Topology Id (14)	`0x00`	Transaction Type (15)	`0x00`
Transaction Id (16)	`0x00`	Key field length (17)	`0x05`
Key (18 - 22)	`'Hello'`	Lifespan (23)	`0x00`
Max idle (24)	`0x00`	Value field length (25)	`0x05`
Value (26-30)	`'World'`		

- Coded response

Byte	0	1	2	3	4	5	6	7
8	`0xA1`	`0x09`	`0x01`	`0x00`	`0x00`			

- Field Explanation

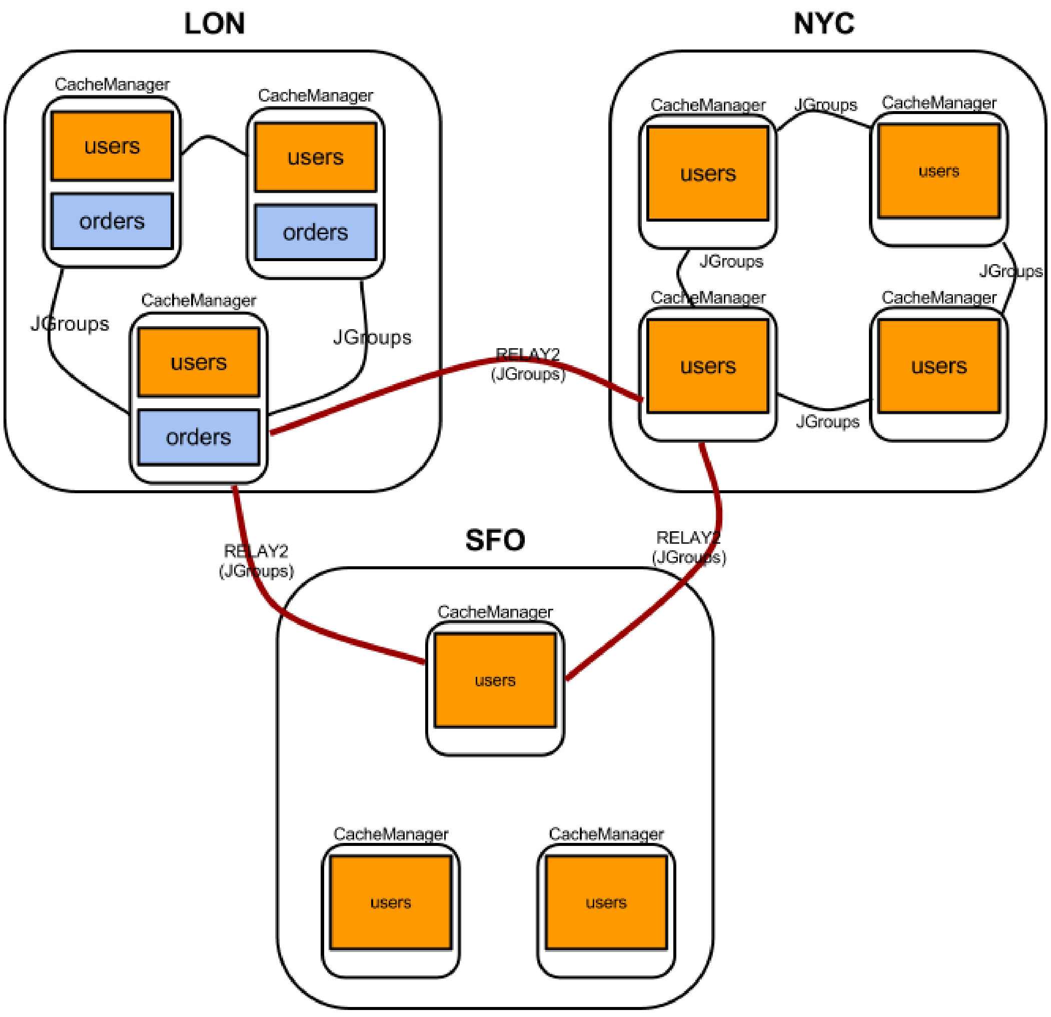
Field Name	Value	Field Name	Value
Magic (0)	`0xA1`	Message Id (1)	`0x09`
Opcode (2)	`0x01`	Status (3)	`0x00`
Topology change marker (4)	`0x00`		

65 Cross site replication

Cross site (x-site) replication allows backing up the data from one cluster to other clusters, potentially situated in different geographical location. The cross-site replication is built on top of JGroups' RELAY2 protocol. This document describes the technical design of cross site replication in more detail.

65.1 Sample deployment

The diagram below depicts a possible setup of replicated sites, followed by a description of individual elements present in the deployment. Options are then explained at large in future paragraphs.
Comments on the diagram above:



- there are 3 sites: LON, NYC and SFO.
- in each site there is a running Infinispan cluster with a (potentially) different number of physical nodes: 3 nodes in LON, 4 nodes in NYC and 3 nodes in SFO
- the "users" cache is active in LON, NYC and SFO. Updates on the "users" cache in any of these sites gets replicated to the other sites as well
- it is possible to use different replication mechanisms between sites. E.g. One can configure SFO to backup data synchronously to NYC and asynchronously to LON
- the "users" cache can have a different configuration from one site to the other. E.g. it might be configured as distributed with numOwners=2 in the LON site, REPL in the NYC site and distributed with numOwners=1 in the SFO site
- JGroups is used for both inter-site and intra-site communication. RELAY2 is used for inter-site communication
- "orders" is a site local to LON, i.e. updates to the data in "orders" don't get replicated to the remote sites

The following sections discuss specific aspects of cross site replication into more detail. The foundation of the cross-site replication functionality is RELAY2 so it highly recommended to read JGroups' RELAY2 documentation before moving on into cross-site.

65.2 Configuration

The cross-site replication configuration spreads over the following files:

1. the backup policy for each individual cache is defined in infinispan's .xml configuration file (infinispan.xml in attachment)
2. cluster's jgroups xml configuration file: RELAY2 protocol needs to be added to the JGroups protocol stack (jgroups.xml)
3. RELAY2 configuration file: RELAY2 has an own configuration file (relay2.xml)
4. the jgroups channel that is used by RELAY2 has its own configuration file (jgroups-relay2.xml)

65.2.1 Infinispan XML configuration file

The local site is defined in the the global configuration section. The local is the site where the node using this configuration file resides (in the example above local site is "LON").

```
<global>
  ...
  <sites local="LON"/>
  ...
</global>
```

The same setup can be achieved programatically:

```
GlobalConfigurationBuilder lonGc = GlobalConfigurationBuilder.defaultClusteredBuilder();
lonGc.sites().localSite("LON");
```

The names of the site (case sensitive) should match the name of a site as defined within JGroups' RELAY2 protocol configuration file.
Besides the global configuration, each cache specifies its backup policy in the "site" element:

```
<namedCache name="users">
   <sites>
     <backups backupSites="NYC,SFO">
       <backup site="NYC" backupFailurePolicy="WARN" strategy="SYNC" timeout="12000"/>
       <backup site="SFO" backupFailurePolicy="IGNORE" strategy="ASYNC"/>
       <backup site="LON" strategy="SYNC"/>
     </backups>
  </sites>
</namedCache>
```

The "users" cache backups its data to the "NYC" and "SFO" sites. Even though the "LON" appears as a backup site, only the sites specified in the "backupSItes" attributes of the "backups" element are considered when it comes to backing up, all other defined backups being ignored.
For each site backup, the following configuration attributes can be specified:

- strategy - the strategy used for backing up data, either "SYNC" or "ASYNC". Defaults to "ASYNC"
- backupFailurePolicy - Decides what the system would do in case of failure during backup. Possible values are:
    - IGNORE - allow the local operation/transaction to succeed
    - WARN - same as IGNORE but also logs an warning message. Default.
    - FAIL - only in effect if "strategy" is "SYNC" - fails local cluster operation/transaction by throwing an exception to the user
    - CUSTOM - user provided, see "failurePolicyClass" below
- failurePolicyClass - If the 'backupFailurePolicy' is set to 'CUSTOM' then this attribute is required and should contain the fully qualified name of a class implementing org.infinispan.xsite.CustomFailurePolicy
- timeout - The timeout(milliseconds) to be used when backing up data remotely. Defaults to 10000 (10 seconds)

The same setup can be achieved programatically:

```
ConfigurationBuilder lon = new ConfigurationBuilder();
lon.sites().addBackup()
      .site("NYC")
      .backupFailurePolicy(BackupFailurePolicy.WARN)
      .strategy(BackupConfiguration.BackupStrategy.SYNC)
      .replicationTimeout(12000)
      .sites().addInUseBackupSite("NYC")
    .sites().addBackup()
      .site("SFO")
      .backupFailurePolicy(BackupFailurePolicy.IGNORE)
      .strategy(BackupConfiguration.BackupStrategy.ASYNC)
      .sites().addInUseBackupSite("SFO")
```

The "users" cache above doesn't know on which cache on the remote sites its data is being replicated. By default the remote site writes the backup data to a cache having the same name as the originator, i.e. "users". This behaviour can be overridden with an "backupFor" element. For example the following configuration in SFO makes the "usersLONBackup" cache act as the backup cache for the "users" cache defined above in the LON site:

```
<infinispan>
  <namedCache name="usersLONBackup">
    <sites>
      <backupFor remoteCache="users" remoteSite="LON"/>
    </sites>
  </namedCache>
</infinispan>
```

The same setup can be achieved programatically:

```
ConfigurationBuilder cb = new ConfigurationBuilder();
cb.sites().backupFor().remoteCache("users").remoteSite("LON");
```

65.2.2 Local cluster's jgroups .xml configuration

This is the configuration file for the local (intra-site) infinispan cluster. It is referred from the infinispan configuration file, see "configurationFile" below:

```
<infinispan>
..
  <global>
    <transport clusterName="infinispan-cluster">
      <properties>
        <property name="configurationFile" value="jgroups.xml"/>
      </properties>
    </transport>
  </global>
..
</infinispan>
```

In order to allow inter-site calls, the RELAY2 protocol needs to be added to the protocol stack defined in the jgroups configuration (see attached jgroups.xml for an example).

65.2.3 RELAY2 configuration file

The RELAY2 configuration file is linked from the jgroups.xml (see attached relay2.xml). It defines the sites seen by this cluster and also the jgroups configuration file that is used by RELAY2 in order to communicate with the remote sites.

65.3 Data replication

For both transactional and non-transactional caches, the backup calls are performed in parallel with local cluster calls, e.g. if we write data to node N1 in LON then replication to the local nodes N2 and N3 and remote backup sites SFO and NYC happen in parallel.

65.3.1 Non transactional caches

In the case of non-transactional caches the replication happens during each operation. Given that data is sent in parallel to backups and local caches, it is possible for the operations to succeed locally and fail remotely, or the other way, causing inconsistencies

65.3.2 Transactional caches

For synchronous transactional caches, Infinispan internally uses a two phase commit protocol: lock acquisition during the 1st phase (prepare) and apply changes during the 2nd phase (commit). For asynchronous caches the two phases are merged, the "apply changes" message being sent asynchronously to the owners of data. This 2PC protocol maps to 2PC received from the JTA transaction manager.
For transactional caches, both optimistic and pessimistic, the backup to remote sites happens during the prepare and commit phase only.

Synchronous local cluster with async backup

In this scenario the backup call happens during local commit phase(2nd phase). That means that if the local prepare fails, no remote data is being sent to the remote backup.

Synchronous local cluster with sync backup

In this case there are two backup calls:

- during prepare a message is sent across containing all the modifications that happened within this transaction
    - if the remote backup cache is transactional then a transaction is started remotely and all these modifications are being written within this transaction's scope. The transaction is not committed yet (see below)
    - if the remote backup cache is not transactional, then the changes are applied remotely
- during the commit/rollback, a commit/rollback message is sent across
    - if the remote backups cache is transactional then the transaction started at the previous phase is committed/rolled back
    - if the remote backup is not transactional then this call is ignored

Both the local and the backup call(if the "backupFailurePolicy" is set to "FAIL") can veto transaction's prepare outcome

Asynchronous local cluster

- In the case of asynchronous local clusters, the backup data is sent during the commit phase. If the backup call fails and the "backupFailurePolicy" is set to "FAIL" then the user is notified through an exception.

65.4 Taking a site offline

If backing up to a site fails for a certain number of times during an time interval, then it is possible to automatically mark that site as offline. When a site is marked as offline the local site won't try to backup data to it anymore. In order to be taken online a system administrator intervention being required.

65.4.1 Configuration

The taking offline of a site can be configured as follows:

```
<namedCache name="bestEffortBackup">
   ...
    <sites>
        <backups>
           <backup site="NYC" strategy="SYNC" backupFailurePolicy="FAIL">
               <takeOffline afterFailures="500" minTimeToWait="10000"/>
           </backup>
        </backups>
    </sites>
    ...
</namedCache>
```

The *takeOfline* element under the *backup* configures the taking offline of a site:
**afterFailure* - the number of failed backup operations after which this site should be taken offline. Defaults to 0 (never). A negative value would mean that the site will be taken offline after *minTimeToWait*
**minTimeToWait* - the number of milliseconds in which a site is not marked offline even if it is unreachable for 'afterFailures' number of times. If smaller or equal to 0, then only *afterFailures* is considered.

The equivalent programmatic configuration is:

```
lon.sites().addBackup()
      .site("NYC")
      .backupFailurePolicy(BackupFailurePolicy.FAIL)
      .strategy(BackupConfiguration.BackupStrategy.SYNC)
      .takeOffline()
         .afterFailures(500)
         .minTimeToWait(10000);
```

65.4.2 Taking a site back online

In order to bring a site back online after being taken offline, one can use the JMX console and invoke the "bringSiteOnline(siteName)" operation on the *XSiteAdmin* managed bean. At the moment this method would need to be invoked on all the nodes within the site(further releases will overcome this limitation).

65.5 Reference

This document (Sept 2012) describes the technical design of cross site replication in more detail.

66 Design of data versioning in Infinispan

66.1 Overview

Infinispan will offer three forms of data versioning, including simple, partition aware and external. Each case is described in detail below.

66.1.1 Simple versioning

The purpose of simple versioning is to provide a reliable mechanism of write skew checks when using optimistic transactions, REPEATABLE_READ and a clustered cache. Write skew checks are performed at prepare-time to ensure a concurrent transaction hasn't modified an entry while it was read and potentially updated based on the value read.

When operating in LOCAL mode, write skew checks rely on Java object references to compare differences and this is adequate to provide a reliable write-skew check, however this technique is useless in a cluster and a more reliable form of versioning is necessary to provide reliable write skew checks.

Simple versioning is an implementation of the proposed `EntryVersion` interface, backed by a `long` that is incremented each time the entry is updated.

66.1.2 Partition-aware versioning

This versioning scheme makes use of vector clocks to provide a network partition resilient form of versioning.

Unlike simple versioning, which is maintained per entry, a vector clock's node counter is maintained per-node.

66.1.3 External versioning

This scheme is used to encapsulate an external source of data versioning within Infinispan, such as when using Infinispan with Hibernate which in turn gets it's data version information directly from a database.

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

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

66.1.4 Tombstones

To deal with deletions of entries, tombstones will be maintained as `null` entries that have been deleted, so that version information of the deleted entry can be maintained and write skews can still be detected. However this is an expensive thing to do, and as such, is a configuration option, disabled by default. Further, tombstones will follow a strict lifespan and will be cleared from the system after a specific amount of time.

66.2 Configuration

By default versioning will be *disabled*. This will mean write skew checks when using transactions and **REPEATABLE_READ** as an isolation level will be unreliable when used in a cluster. Note that this doesn't affect single-node, LOCAL mode usage.

66.2.1 Via XML

```
<versioning enabled="false" type="SIMPLE|PARTITION_AWARE|EXTERNAL" useTombstones="false"
tombstoneLifespan="60000"/>
```

66.2.2 Via the programmatic API

```
fluent().versioning().type(SIMPLE).useTombstones(true).tombstoneLifespan(1, TimeUnit.MINUTES);
```

67 Infinispan modules for JBoss AS 7.x

Since Infinispan 5.2, the distribution includes a set of modules for JBoss AS 7.x. By installing these modules, it is possible to deploy user applications without packaging the Infinispan JARs within the deployments (WARs, EARs, etc), thus minimizing their size. In order not to conflict with the Infinispan modules which are already present within an AS installation, the modules provided by the Infinispan distribution are located within their own slot identified by the *major.minor* versions (e.g. slot="5.2").

In order to tell the AS deployer that we want to use the Infinispan APIs within our application, we need to add explicit dependencies to the deployment's MANIFEST:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:5.2 services
```

If you are using Maven to generate your artifacts, mark the Infinispan dependencies as *provided* and configure your artifact archiver to generate the appropriate MANIFEST.MF file:

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-core</artifactId>
    <version>5.2.0.CR2</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-cachestore-jdbc</artifactId>
    <version>5.2.0.CR2</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-war-plugin</artifactId>
       <configuration>
         <archive>
           <manifestEntries>
             <Dependencies>org.infinispan:5.2 services, org.infinispan.cachestore.jdbc:5.2
services</Dependencies>
           </manifestEntries>
         </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

68 Infinispan with Groovy

68.1 Introduction

The idea by this tutorial is to give an introduction in the use of the Infinispan API and its configuration file. As trying to do it in a more interactive fashion, the tutorial makes use of the Groovy dynamic language that will allow to interact with the API by using a console. So your first task should be to create the necessary environment to execute this tutorial, you can find the instructions here.

The tutorial will start by showing the basic usage of the Infinispan API and a use of a simple cache configuration, then it will walk through different configuration scenarios and use cases. By the end of the tutorial you should have a clear understanding of the use the Infinispan API and some of the various configuration options.

The scenarios and use cases shown are:

- Basic cache configuration
- Cache with transaction management configuration
- Cache with a cache store configuration
- Cache with eviction configuration
- Cache with eviction and cache store configuration
- Cache with REPL_SYNC & transaction management configuration.

All the sample configurations are in the sample-configurations.xml file attached to this tutorial, check the environment configuration to know how to make use of this configuration file. Lets get started:

NOTE: This document is part of the Infinispan Interactive Tutorial

68.1.1 Introduction

The Infinispan tutorial makes use of Groovy to get a more interactive experience when starting to learn about how to use the Infinispan API. So you will need to install a few prerequisites before getting started:

- The Groovy Platform, I used Groovy 1.6.3
- Java and Infinispan

Download those and extract/install where you feel appropriate, depending on your operating system and personal preferences you will either have installers or compressed distributions. You can read more about read installing Java and Infinispan in Installing Infinispan for the tutorials.

68.1.2 Installing Groovy

You can use the installer or compressed file to install the Groovy Platform, I used the compressed file and decompressed at C:\Program Files\groovy\groovy-1.6.3. Once you have installed the Groovy Platform you should set some environment variables:

```
GROOVY_HOME=C:\Program Files\groovy\groovy-1.6.3
```

and add to the PATH environment variable:

```
PATH=%PATH%;%GROOVY_HOME%\bin
```

test that everything is correct by executing in a Command Shell/Terminal the commands shown:

```
$> groovy -v
Groovy Version: 1.6.3 JVM: 1.6.0_14
```

If you get a similar result as shown, everything went well.

68.1.3 Installing Infinispan

Now you should add the Infinispan libraries to the Groovy Platform so you will able to access the API from the Groovy console. Add the infinispan-core.jar and its dependencies to the $USER_HOME/.groovy/lib directory, the jar is located in $INFINISPAN_HOME/modules/core and the dependencies at $INIFINISPAN_HOME/modules/core/lib.

For example, on Windows, you need to copy it to:

```
C:\Documents and Settings\Alejandro Montenegro\.groovy\lib
```

or on Linux:

```
/home/amontenegro/.groovy/lib
```

and $INFINISPAN_HOME is where you decompressed the Infinispan distribution.
To test the installation, download the attached file infinispantest.groovy and in a Command Shell/Terminal execute

```
$> groovy infinispantest
4.0.0.ALPHA5
```

68.1.4 Setting the classpath

The last thing to do is to add to the CLASSPATH environment variable the sample configuration file, this file contains definitions of cache's that will be used in the tutorial. I created the directory $USER_HOME/.groovy/cp and added it to the classpath

For example, on Windows:

```
CLASSPATH=%CLASSPATH%;C:\Documents and Settings\Alejandro Montenegro\.groovy\cp
```

or, on Linux:

```
CLASSPATH=$CLASSPATH:/home/amontenegro/.groovy/cp
```

finally add the sample-configurations.xml and infinispan-config-4.0.xsd files(attached) to the directory.

68.2 Loading the configuration file

The cache manager is the responsible to manage all the cache's, so you have to start by indicating where to get the cache definitions to the cache manager, remember that the cache definitions are in the sample-configurations.xml file. If no cache definitions are indicated, the cache manager will use a default cache.

Start by open a groovy console by typing groovy.sh in a command shell or terminal. You should now have something similar to:

```
Groovy Shell (1.6.3, JVM: 1.6.0_14)
Type 'help' or '\h' for help.
-------------------------------------------------------
groovy:000>
```

It's time to start typing some commands, first start by importing the necessary libraries

```
groovy:000> import org.infinispan.*
===> [import org.infinispan.*]
groovy:000> import org.infinispan.manager.*
===> [import org.infinispan.*, import org.infinispan.manager.*]
```

And now, create a cache manager indicating the file with the cache definitions.

```
groovy:000> manager = new DefaultCacheManager("sample-configurations.xml")
===> org.infinispan.manager.DefaultCacheManager@19cc1b@Address:null
```

the cache manager has now the knowledge of all the named caches defined in the configuration file and also has a no named cache that's used by default. You can now access any of the cache's by interacting with the cache manager as shown.

```
groovy:000> defaultCache = manager.getCache()
===> Cache 'org.infinispan.manager.DefaultCacheManager.DEFAULT_CACHE_NAME'@7359733
//TO GET A NAMED CACHE
groovy:000> cache = manager.getCache("NameOfCache")
```

68.3 Basic cache configuration

The basic configuration, is the simplest configuration that you can have, its make use of default settings for the properties of the cache configuration, the only thing you have to set is the name of the cache.

```
<namedCache name="Local"/>
```

That's all you have to add to the configuration file to have a simple named cache, now its time to interact with the cache by using the Infinispan API. Lets start by getting the named cache and put some objects inside it.

```
//START BY GETTING A REFERENCE TO THE NAMED CACHE
groovy:000> localCache = manager.getCache("Local")
===> Cache 'Local'@19521418
//THE INITIAL SIZE IS 0
groovy:000> localCache.size()
===> 0
//NOW PUT AN OBJECT INSIDE THE CACHE
groovy:000> localCache.put("aKey", "aValue")
===> null
//NOW THE SIZE IS 1
groovy:000> localCache.size()
===> 1
//CHECK IF IT HAS OUR OBJECT
groovy:000> localCache.containsKey("aKey")
===> true
//BY OBTAINING AN OBJECT DOESN'T MEAN TO REMOVE
groovy:000> localCache.get("aKey")
===> aValue
groovy:000> localCache.size()
===> 1
//TO REMOVE ASK IT EXPLICITLY
groovy:000> localCache.remove("aKey")
===> aValue
groovy:000> localCache.isEmpty()
===> true
```

So you have seen the basic of the Infinispan API, adding, getting and removing from the cache, there is more, but don't forget that you are working with a cache that are an extension of java.util.ConcurrentHasMap and the rest of the API is as simple as the one shown above, many of the cool things in Infinispan are totally transparent (that's actually the coolest thing about Infinispan) and depends only on the configuration of your cache.
If you check the Infinispan JavaDoc you will see that the Cache#put() method has been overridden several times.

```
//YOU WILL NEED TO IMPORT ANOTHER LIBRARY
groovy:000> import java.util.concurrent.TimeUnit
===> [import org.infinispan.*, import org.infinispan.manager.*, import
java.util.concurrent.TimeUnit]
//NOTHING NEW HERE JUST PUTTING A NEW OBJECT
groovy:000> localCache.put("bKey", "bValue")
===> null
//WOW! WHATS HAPPEN HERE? PUTTED A NEW OBJECT BUT IT WILL TIMEOUT AFTER A SECOND
groovy:000> localCache.put("timedKey", "timedValue", 1000, TimeUnit.MILLISECONDS)
===> null
//LETS CHECK THE SIZE
groovy:000> localCache.size()
===> 2
//NOW TRY TO GET THE OBJECT, OOPS ITS GONE! (IF NOT, IT'S BECAUSE YOU ARE A SUPERTYPER, CALL
GUINNESS!))
groovy:000> localCache.get("timedKey")
===> null
//LETS CHECK THE SIZE AGAIN, AS EXPECTED THE SIZE DECREASED BY 1
groovy:000> localCache.size()
===> 1
```

The Infinispan API also allows you to manage the life cycle of the cache, you can stop and start a cache but by default you will loose the content of the cache except if you configure a cache store, more about that later in the tutorial. lets check what happens when you restart the cache

```
groovy:000> localCache.size()
===> 1
//RESTARTING CACHE
groovy:000> localCache.stop()
===> null
groovy:000> localCache.start()
===> null
//DAMN! LOST THE CONTENT OF THE CACHE
groovy:000> localCache.size()
===> 0
```

Thats all related to the use of the Infinispan API, now lets check some different behaviors depending on the configuration of the cache.

68.4 Cache with transaction management

You are able to specify the cache to use a transaction manager, and even explicitly control the transactions. Start by configuring the cache to use a specific TransactionManagerLookup class. Infinispan implements a couple TransactionManagerLookup classes.

- org.infinispan.transaction.lookup.DummyTransactionManagerLookup
- org.infinispan.transaction.lookup.GenericTransactionManagerLookup
- org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup
- org.infinispan.transaction.lookup.JBossTransactionManagerLookup

Each use different methods to lookup the transaction manager, depending on the environment you are running Infinispan you should figure out which one to use. Check the JavaDoc for more details.

For the tutorial its enough to use:

```
<namedCache name="LocalTX">
    <transaction
transactionManagerLookupClass="org.infinispan.transaction.lookup.DummyTransactionManagerLookup"/>
```

Lets check how to interact with the Transaction Manager and to have the control over a transaction

```
groovy:000> import javax.transaction.TransactionManager
===> [import org.infinispan.*, import org.infinispan.manager.*, import
java.util.concurrent.TimeUnit, import javax.transaction.TransactionManager]
//GET A REFERENCE TO THE CACHE WITH TRANSACTION MANAGER
groovy:000> localTxCache = manager.getCache("LocalTX")
===> Cache 'LocalTX'@16075230
groovy:000> cr = localTxCache.getComponentRegistry()
===> org.infinispan.factories.ComponentRegistry@87e9bf
//GET A REFERENCE TO THE TRANSACTION MANAGER
groovy:000> tm = cr.getComponent(TransactionManager.class)
===> org.infinispan.transaction.tm.DummyTransactionManager@b5d05b
//STARTING A NEW TRANSACTION
groovy:000> tm.begin()
===> null
//PUTTING SOME OBJECTS INSIDE THE CACHE
groovy:000> localTxCache.put("key1", "value1")
===> null
//MMM SIZE DOESN'T INCREMENT
groovy:000> localTxCache.size()
===> 1
//LETS TRY AGAIN
groovy:000> localTxCache.put("key2", "value2")
===> null
//MMM NOTHING..
groovy:000> localTxCache.size()
===> 2
//OH! HAS TO DO THE COMMIT
groovy:000> tm.commit()
===> null
//AND THE SIZE IS AS EXPECTED.. HAPPY!
groovy:000> localTxCache.size()
===> 2
```

As shown in the example, the transaction is controlled explicitly and the changes in the cache wont be reflected until you make the commit.

68.5 Cache with a cache store

Infinispan allows you to configure a persistent store that can be used to persist the content of the cache, so if the cache is restarted the cache will be able to keep the content. It can also be used if you want to limit the size of the cache, then the cache will start putting the objects in the store to keep the size limit, more on that when looking at the eviction configuration.

Infinispan provides several cache store implementations:

- FileCacheStore
- JdbcBinaryCacheStore
- JdbcMixedCacheStore
- JdbcStringBasedCacheStore
- JdbmCacheStore
- S3CacheStore
- BdbjeCacheStore

The tutorial uses the FileCacheStore, that saves the objects in files in a configured directory, in this case the /tmp directory. If the directory is not set it defaults to Infinispan-FileCacheStore in the current working directory.

```
<namedCache name="CacheStore">
     <loaders passivation="false" shared="false" preload="true">
          <loader class="org.infinispan.loaders.file.FileCacheStore" fetchPersistentState="true"
               ignoreModifications="false" purgeOnStartup="false">
            <properties>
                 <property name="location" value="/tmp"/>
               </properties>
          </loader>
     </loaders>
</namedCache>
```

Now you have a cache with persistent store, lets try it to see how it works

```
//GETTING THE NEW CACHE
groovy:000> cacheCS = manager.getCache("CacheStore")
===> Cache 'CacheStore'@23240342
//LETS PUT AN OBJECT INSIDE THE CACHE
groovy:000> cacheCS.put("storedKey", "storedValue")
===> null
//LETS PUT THE SAME OBJECT IN OUR BASIC CACHE
groovy:000> localCache.put("storedKey", "storedValue")
===> storedValue
//RESTART BOTH CACHES
groovy:000> cacheCS.stop()
===> null
groovy:000> localCache.stop()
===> null
groovy:000> cacheCS.start()
===> null
groovy:000> localCache.start()
===> null
//LETS TRY GET THE OBJECT FROM THE RESTARTED BASIC CACHE.. NO LUCK
groovy:000> localCache.get("storedKey")
===> null
//INTERESTING CACHE SIZE IS NOT CERO
groovy:000> cacheCS.size()
===> 1
//WOW! JUST RESTARTED THE CACHE AND THE OBKECT KEEPS STAYING THERE!
groovy:000> cacheCS.get("storedKey")
===> storedValue
```

68.6 Cache with eviction

The eviction allow to define policy for removing objects from the cache when it reach its limit, as the true is that the caches doesn't has unlimited size because of many reasons. So the fact is that you normally will set a maximum number of objects in the cache and when that number is reached then the cache has to decide what to do when a new object is added. That's the whole story about eviction, to define the policy of removing object when the cache is full and want to keep putting objects. You have three eviction strategies:

- NONE
- FIFO
- LRU

Let check the configuration of the cache:

```
<namedCache name="Eviction">
   <eviction wakeUpInterval="500" maxEntries="2" strategy="FIFO"/>
</namedCache>
```

The strategy has been set to FIFO, so the oldest objects will be removed first and the maximum number of objects are only 2, so it will be easy to show how it works

```
//GETTING THE NEW CACHE
groovy:000> evictionCache = manager.getCache("Eviction")
===> Cache 'Eviction'@5132526
//PUT SOME OBJECTS
groovy:000> evictionCache.put("key1", "value1")
===> null
groovy:000> evictionCache.put("key2", "value2")
===> null
groovy:000> evictionCache.put("key3", "value3")
===> null
//HEY! JUST LOST AN OBJECT IN MY CACHE.. RIGHT, THE SIZE IS ONLY TWO
groovy:000> evictionCache.size()
===> 2
//LETS CHECK WHAT OBJECT WAS REMOVED
groovy:000> evictionCache.get("key3")
===> value3
groovy:000> evictionCache.get("key2")
===> value2
//COOL! THE OLDEST WAS REMOVED
groovy:000> evictionCache.get("key1")
===> null
```

Now you are sure that your cache wont consume all your memory and hang your system, but its an expensive price you have to pay for it, you are loosing objects in your cache. The good news is that you can mix cache store with the eviction policy and avoid loosing objects.

68.7 Cache with eviction and cache store

Ok, the cache has a limited size but you don't want to loose your objects in the cache. Infinispan is aware of these issues, so it makes it very simple for you combing the cache store with the eviction policy. When the cache is full it will persist an object and remove it from the cache, but if you want to recover an object that has been persisted the the cache transparently will bring it to you from the cache store.

The configuration is simple, just combine eviction and cache store configuration

```
<namedCache name="CacheStoreEviction">
     <loaders passivation="false" shared="false" preload="true">
          <loader class="org.infinispan.loaders.file.FileCacheStore" fetchPersistentState="true"
            ignoreModifications="false" purgeOnStartup="false">
               <properties>
                    <property name="location" value="/tmp"/>
               </properties>
          </loader>
     </loaders>
     <eviction wakeUpInterval="500" maxEntries="2" strategy="FIFO"/>
</namedCache>
```

Nothing new in the configuration, lets check how it works

```
//GETTING THE CACHE
groovy:000> cacheStoreEvictionCache = manager.getCache("CacheStoreEviction")
===> Cache 'CacheStoreEviction'@6208201
//PUTTING SOME OBJECTS
groovy:000> cacheStoreEvictionCache.put("cs1", "value1")
===> value1
groovy:000> cacheStoreEvictionCache.put("cs2", "value2")
===> value2
groovy:000> cacheStoreEvictionCache.put("cs3", "value3")
===> value3
///MMM SIZE IS ONLY TWO, LETS CHECK WHAT HAPPENED
groovy:000> cacheStoreEvictionCache.size()
===> 2
groovy:000> cacheStoreEvictionCache.get("cs3")
===> value3
groovy:000> cacheStoreEvictionCache.get("cs2")
===> value2
//WOW! EVEN IF THE CACHE SIZE IS 2, I RECOVERED THE THREE OBJECTS.. COOL!!
groovy:000> cacheStoreEvictionCache.get("cs1")
===> value1
```

68.7.1 Cache with REPL_SYNC & transaction management

TODO

69 Using Infinispan with Scala

69.1 Introduction

This article shows how to use Infinispan with Scala language. It uses the same commands and configurations used in the Groovy edition of interactive tutorial. For more details about the scenarios and steps please visit about page since here will will only focus on Scala compatibility.

69.2 Environment

Preparing the environment is almost similar to one described here, but with a minor difference that unlike Groovy which uses **~/.groovy/lib** folder to extend initial classpath, we will use classic **CLASSPATH** environment variable with Scala. Another issue is that with the recent edition of Infinispan core jar file is in the root folder of $INIFINISPAN_HOME, hence here a sample bash script to prepare CLASSPATH for our demo:

```
export INFINISPAN_HOME=~/build/infinispan/infinispan-4.2.1.CR1
for j in $INFINISPAN_HOME/lib/*.jar; do CLASSPATH=$CLASSPATH:$j; done
export CLASSPATH=$CLASSPATH:$INFINISPAN_HOME/infinispan-core.jar
export CLASSPATH=$CLASSPATH:[Path to folder containing sample-configurations.xml file]
```

Download*sample-configurations.xml* file from here.

69.3 Testing Setup

The following code shows how to start an Scala console that will allow commands to be entered interactively. To verify that the Infinispan classes have been imported correctly, an import for all Infinispan classes will be attempted and then a request will be made to print the version of Infinispan:

```
[z@dnb:~/Go/demos/interactive-infinispan-scala]% ./scala
Welcome to Scala version 2.8.1.final (Java HotSpot(TM) 64-Bit Server VM, Java 1.6.0_22).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import org.infinispan._
import org.infinispan._

scala> println(Version.version)
4.2.1.CR1
```

69.4 Loading the Configuration file

In this next example, a new cache manager will be created using the configuration file downloaded earlier:

```
scala> import org.infinispan.manager._
import org.infinispan.manager._

scala> val manager = new DefaultCacheManager("sample-configurations.xml")
manager: org.infinispan.manager.DefaultCacheManager =
org.infinispan.manager.DefaultCacheManager@38b58e73@Address:null
```

Retrieving cache instances from cache manager

In this example, the default cache is retrieved expecting keys and values to be of String type:

```
scala> val defaultCache = manager.getCache[String, String]()
defaultCache: org.infinispan.Cache[String,String] = Cache '___defaultcache'@1326840752
```

In this next example, a named cache is retrieved, again with keys and values expected to be String:

```
scala> val namedCache = manager.getCache[String, String]("NameOfCache")
namedCache: org.infinispan.Cache[String,String] = Cache 'NameOfCache'@394890130
```

69.5 Basic cache operations

In this section, several basic operations will be executed against the cache that show how it can be populated with data, how data can be retrieved and size can be checked, and finally how after removing the data entered, the cache is empty:

```
scala> val localCache = manager.getCache[String, String]("Local")
localCache: org.infinispan.Cache[String,String] = Cache 'Local'@420875876

scala> localCache.size()
res0: Int = 0

scala> localCache.put("aKey", "aValue")
res1: String = null
// This null was returned by put() indicating that
// the key was not associated with any previous value.

scala> localCache.size()
res2: Int = 1

scala> localCache.containsKey("aKey")
res3: Boolean = true

scala> localCache.get("aKey")
res4: String = aValue

scala> localCache.size()
res5: Int = 1

scala> localCache.remove("aKey")
res6: String = aValue

scala> localCache.isEmpty()
res7: Boolean = true
```

69.6 Basic cache operations with TTL

When a cache entry is stored, a maximum lifespan for the entry can be provided. So, when that time is exceeded, the entry will dissapear from the cache:

```
scala> localCache.put("bKey", "bValue")
res8: String = null

scala> import java.util.concurrent.TimeUnit
import java.util.concurrent.TimeUnit

scala> localCache.put("timedKey", "timedValue", 1000, TimeUnit.MILLISECONDS)
res9: String = null

scala> localCache.size()
res10: Int = 2

scala> localCache.get("timedKey")
res11: String = null

scala> localCache.size()
res12: Int = 1
```

69.7 Cache restarts

When caches are local and not configured with a persistent store, restarting them means that the data is gone. To avoid this issue you can either configure caches to be clustered so that if one cache dissapears, the data is not completely gone, or configure the cache with a persistent cache store. The latter option will be explained later on.

```
scala> localCache.size()
res13: Int = 1

scala> localCache.stop()

scala> localCache.start()

scala> localCache.size()
res16: Int = 0
```

69.8 Transactional cache operations

Infinispan caches can be operated within a transaction, in such way that operations can be grouped in order to be executed atomically. The key thing to understand about transactions is that within the transactions changes are visible, but to other non-transactional operations, or other transactions, these are not visible until the transaction is committed. The following example shows how within a transaction an entry can be stored but outside the transaction, this modification is not yet visible, and that once the transaction is committed, the modification is visible to all:

```
scala> import javax.transaction.TransactionManager
import javax.transaction.TransactionManager

scala> val localTxCache = manager.getCache[String, String]("LocalTX")
localTxCache: org.infinispan.Cache[String,String] = Cache 'LocalTX'@955386212

scala> val tm = localTxCache.getAdvancedCache().getTransactionManager()
tm: javax.transaction.TransactionManager =
org.infinispan.transaction.tm.DummyTransactionManager@81ee8c1

scala> tm.begin()

scala> localTxCache.put("key1", "value1")
res1: String = null

scala> localTxCache.size()
res2: Int = 1

scala> tm.suspend()
res3: javax.transaction.Transaction = DummyTransaction{xid=DummyXid{id=1}, status=0}

scala> localTxCache.size()
res4: Int = 0

scala> localTxCache.get("key1")
res5: String = null

scala> tm.resume(res3)

scala> localTxCache.size()
res7: Int = 1

scala> localTxCache.get("key1")
res8: String = value1

scala> tm.commit()

scala> localTxCache.size()
res10: Int = 1

scala> localTxCache.get("key1")
res11: String = value1
```

Note how this example shows a very interesting characteristic of the Scala console. Every operation's return value is stored in a temporary variable which can be referenced at a later stage, even if the user forgets to assign the result of a operation when the code was executed.

69.9 Persistent stored backed Cache operations

When a cache is backed by a persistent store, restarting the cache does not lead to data being lost. Upon restart, the cache can retrieve in lazy or prefetched fashion cache entries stored in the backend persistent store:

```
scala> val cacheWithStore = manager.getCache[String, String]("CacheStore")
cacheWithStore: org.infinispan.Cache[String,String] = Cache 'CacheStore'@2054925789

scala> cacheWithStore.put("storedKey", "storedValue")
res21: String = null

scala> localCache.put("storedKey", "storedValue")
res22: String = null

scala> cacheWithStore.stop()

scala> localCache.stop()

scala> cacheWithStore.start()

scala> localCache.start()

scala> localCache.get("storedKey")
res27: String = null

scala> cacheWithStore.size()
res28: Int = 1

scala> cacheWithStore.get("storedKey")
res29: String = storedValue
```

69.10 Operating against a size bounded cache

Infinispan caches can be configured with a max number of entries, so if this is exceeded certain cache entries are evicted from in-memory cache. Which cache entries get evicted is dependant on the eviction algorithm chosen. In this particular example, FIFO algorithm has been configured, so when a cache entry needs to be evicted, those stored first will go first:

```
scala> val evictionCache = manager.getCache[String, String]("Eviction")
evictionCache: org.infinispan.Cache[String,String] = Cache 'Eviction'@882725548

scala> evictionCache.put("key1", "value1")
res30: String = null

scala> evictionCache.put("key2", "value2")
res31: String = null

scala> evictionCache.put("key3", "value3")
res32: String = null

scala> evictionCache.size()
res33: Int = 2

scala> evictionCache.get("key3")
res34: String = value3

scala> evictionCache.get("key2")
res35: String = value2

scala> evictionCache.get("key1")
res36: String = null
```

69.11 Size bounded caches with persistent store

When caches configured with eviction are configured with a persistent store as well, when the cache exceeds certain size, apart from removing the corresponding cache entries from memory, these entries are stored in the persistent store. So, if they're requested by cache operations, these are retrieved from the cache store:

```
scala> val cacheStoreEvictionCache = manager.getCache[String, String]("CacheStoreEviction")
cacheStoreEvictionCache: org.infinispan.Cache[String,String] = Cache
'CacheStoreEviction'@367917752

scala> cacheStoreEvictionCache.put("cs1", "value1")
res37: String = null

scala> cacheStoreEvictionCache.put("cs2", "value2")
res38: String = null

scala> cacheStoreEvictionCache.put("cs3", "value3")
res39: String = null

scala> cacheStoreEvictionCache.size()
res40: Int = 2

scala> cacheStoreEvictionCache.get("cs3")
res41: String = value3

scala> cacheStoreEvictionCache.get("cs2")
res42: String = value2

scala> cacheStoreEvictionCache.get("cs1")
res43: String = value1
```