Ehcache

v. 2.4

User Guide

Terracotta, Inc. **2011-05-05**

Table of Contents

1 Table of Contents i
2 Preface 1
3 Introduction 2
4 Getting Started 9
5 Dependencies 11
6 Cache Concepts 12
7 Configuration 17
8 Storage Options 24
9 Cache Consistency Options 31
10 Cache Eviction Algorithms 39
11 Big Memory:Off-Heap Store 42
12 JDBC Caching 52
13 Spring Caching with Ehcache 56
14 Code Samples 58
15 Class loading and Class Loaders 65
16 Tuning Garbage Collection 67
17 Cache Decorators 68
18 Hibernate Caching 71
19 Web Caching 83
20 Using ColdFusion with Ehcache 88
21 Cache Topologies 91
22 Distributed Caching .....
23 Replicated Caching With RMI 94
24 Replicated Caching With JGroups 100
25 Replicated Caching With JMS 104
26 Shutting Down Ehcache 115
26 Logging 117
26 Remote Network replication debugging: RMI Replicated Caches 118
26 JMX Management And Monitoring 120
26 JTA And Transactions 127
26 Search 135
26 Ehcache Monitor 145
26 Bulk Loading .....
26 CacheManager Event Listeners 151
26 Cache Event Listeners 154
26 Cache Exception Handlers 158
26 Cache Extensions 161
26 Cache Loaders 164

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

26 **Write-through and write-behind caching with CacheWriters** 168
26 **Cache Server with SOAP and RESTful Web Services** 177
26 **Explicit Locking API** 195
26 **BlockingCache and SelfPopulatingCache** 198
26 **OpenJPA Caching** 199
26 **Grails Caching** 200
26 **JRuby Caching** 202
26 **Glassfish HowTo** 206
26 **Google App Engine Caching** 208
26 **Tomcat Issues and Best Practices** 211
26 **JSR107 (JCACHE) Support** 212
26 **Building From Source** 218
26 **FAQ** 220

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

1 Preface

1.1 Preface

This is a book about Ehcache, a widely used open source Java cache. Ehcache has grown in size and scope since it was introduced in October 2003. As people used it they often noticed it was missing a feature they wanted. Over time, the features that were repeatedly asked for, and make sense for a Cache, have been added.

Ehcache is now used for Hibernate caching, data access object caching, security credential caching, web caching, SOAP and RESTful server caching, application persistence and distributed caching.

In August 2009, Ehcache was acquired by Terracotta, Inc. and has been continously enhanced since then.

1.1.1 Version

This book is for Ehcache version 2.4.1.

1.1.2 Audience

The intended audience for this book is developers who use ehcache. It should be able to be used to start from scratch, get up and running quickly, and also be useful for the more complex options.

Ehcache is about performance and load reduction of underlying resources. Another natural audience is performance specialists.

It is also intended for application and enterprise architects. Some of the features of ehcache, such as distributed caching and Java EE caching, are alternatives to be considered along with other ways of solving those problems. This book discusses the trade-offs in Ehcache's approach to help make a decision about appropriateness of use.

1.1.3 Acknowledgements

Ehcache has had many contributions in the form of forum discussions, feature requests, bug reports, patches and code commits.

Rather than try and list the many hundreds of people who have contributed to Ehcache in some way it is better to link to the web site where contributions are acknowledged in the following ways:

- Bug reports and features requests appear in the changes report here:
- Patch contributors generally end up with an author tag in the source they contributed to.
- Team members appear on the team list page here:

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

2 Introduction

2.1 Introduction

Ehcache is a cache library. Before getting into ehcache, it is worth stepping back and thinking about caching generally.

2.1.1 About Caches

Wiktionary defines a cache as *A store of things that will be required in future, and can be retrieved rapidly*. That is the nub of it.

In computer science terms, a cache is a collection of temporary data which either duplicates data located elsewhere or is the result of a computation. Once in the cache, the data can be repeatedly accessed inexpensively.

2.1.2 Why caching works

2.1.2.1 Locality of Reference

While Ehcache concerns itself with Java objects, caching is used throughout computing, from CPU caches to the DNS system. Why? Because many computer systems exhibit *locality of reference*. Data that is near other data or has just been used is more likely to be used again.

2.1.2.2 The Long Tail

Chris Anderson, of Wired Magazine, coined the term *The Long Tail* to refer to Ecommerce systems. The idea that a small number of items may make up the bulk of sales, a small number of blogs might get the most hits and so on. While there is a small list of popular items, there is a long tail of less popular ones.

The Long Tail

The Long Tail is itself a vernacular term for a Power Law probability distribution. They don't just appear in ecommerce, but throughout nature. One form of a Power Law distribution is the Pareto distribution, commonly know as the 80:20 rule.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

This phenomenon is useful for caching. If 20% of objects are used 80% of the time and a way can be found to reduce the cost of obtaining that 20%, then the system performance will improve.

2.1.3 Will an Application Benefit from Caching?

The short answer is that it often does, due to the effects noted above.

The medium answer is that it often depends on whether it is CPU bound or I/O bound. If an application is I/O bound then then the time taken to complete a computation depends principally on the rate at which data can be obtained. If it is CPU bound, then the time taken principally depends on the speed of the CPU and main memory.

While the focus for caching is on improving performance, it it also worth realizing that it reduces load. The time it takes something to complete is usually related to the expense of it. So, caching often reduces load on scarce resources.

2.1.3.1 Speeding up CPU bound Applications

CPU bound applications are often sped up by:

- improving algorithm performance
- parallelizing the computations across multiple CPUs (SMP) or multiple machines (Clusters).
- upgrading the CPU speed.

  The role of caching, if there is one, is to temporarily store computations that may be reused again.

  An example from Ehcache would be large web pages that have a high rendering cost. Another caching of authentication status, where authentication requires cryptographic transforms.

2.1.3.2 Speeding up I/O bound Applications

Many applications are I/O bound, either by disk or network operations. In the case of databases they can be limited by both.

There is no Moore's law for hard disks. A 10,000 RPM disk was fast 10 years ago and is still fast. Hard disks are speeding up by using their own caching of blocks into memory.

Network operations can be bound by a number of factors:

- time to set up and tear down connections
- latency, or the minimum round trip time
- throughput limits
- marshalling and unmarhshalling overhead

  The caching of data can often help a lot with I/O bound applications. Some examples of Ehcache uses are:

- Data Access Object caching for Hibernate
- Web page caching, for pages generated from databases.

2.1.3.3 Increased Application Scalability

The flip side of increased performance is increased scalability. Say you have a database which can do 100 expensive queries per second. After that it backs up and if connections are added to it it slowly dies.

In this case, caching may be able to reduce the workload required. If caching can cause 90 of that 100 to be cache hits and not even get to the database, then the database can scale 10 times higher than otherwise.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

2.1.4 How much will an application speed up with Caching?

2.1.4.1 The short answer

The short answer is that it depends on a multitude of factors being:

- how many times a cached piece of data can and is reused by the application
- the proportion of the response time that is alleviated by caching

  In applications that are I/O bound, which is most business applications, most of the response time is getting data from a database. Therefore the speed up mostly depends on how much reuse a piece of data gets.

  In a system where each piece of data is used just once, it is zero. In a system where data is reused a lot, the speed up is large.

The long answer, unfortunately, is complicated and mathematical. It is considered next.

2.1.4.2 Applying Amdahl's Law

Amdahl's law, after Gene Amdahl, is used to find the system speed up from a speed up in part of the system.

```
1 / ((1 - Proportion Sped Up) + Proportion Sped Up / Speed up)
```

The following examples show how to apply Amdahl's law to common situations. In the interests of simplicity, we assume:

- a single server
- a system with a single thing in it, which when cached, gets 100% cache hits and lives forever.

2.Persistent Object Relational Caching

A Hibernate Session.load() for a single object is about 1000 times faster from cache than from a database.

A typical Hibernate query will return a list of IDs from the database, and then attempt to load each. If Session.iterate() is used Hibernate goes back to the database to load each object.

Imagine a scenario where we execute a query against the database which returns a hundred IDs and then load each one.

The query takes 20% of the time and the roundtrip loading takes the rest (80%). The database query itself is 75% of the time that the operation takes. The proportion being sped up is thus 60% (75% * 80%).

The expected system speedup is thus:

```
1 / ((1 - .6) + .6 / 1000)
= 1 / (.4 + .006)
= 2.5 times system speedup
```

2.Web Page Caching

An observed speed up from caching a web page is 1000 times. Ehcache can retrieve a page from its SimplePageCachingFilter in a few ms.

Because the web page is the end result of a computation, it has a proportion of 100%.

The expected system speedup is thus:

```
1 / ((1 - 1) + 1 / 1000)
= 1 / (0 + .001)
= 1000 times system speedup
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

2.Web Page Fragment Caching

Caching the entire page is a big win. Sometimes the liveness requirements vary in different parts of the page. Here the SimplePageFragmentCachingFilter can be used.

Let's say we have a 1000 fold improvement on a page fragment that taking 40% of the page render time.

The expected system speedup is thus:

```
1 / ((1 - .4) + .4 / 1000)
= 1 / (6 + .004)
= 1.6 times system speedup
```

2.1.4.3 Cache Efficiency

In real life cache entrie do not live forever. Some examples that come close are "static" web pages or fragments of same, like page footers, and in the database realm, reference data, such as the currencies in the world.

Factors which affect the efficiency of a cache are:

liveness

how live the data needs to be. The less live the more it can be cached

proportion of data cached

what proportion of the data can fit into the resource limits of the machine. For 32 bit Java systems, there was a hard limit of 2GB of address space. While now relaxed, garbage collection issues make it harder to go a lot large. Various eviction algorithms are used to evict excess entries.

Shape of the usage distribution

If only 300 out of 3000 entries can be cached, but the Pareto distribution applies, it may be that 80% of the time, those 300 will be the ones requested. This drives up the average request lifespan.

Read/Write ratio

The proportion of times data is read compared with how often it is written. Things such as the number of rooms left in a hotel will be written to quite a lot. However the details of a room sold are immutable once created so have a maximum write of 1 with a potentially large number of reads.

Ehcache keeps these statistics for each Cache and each element, so they can be measured directly rather than estimated.

2.1.4.4 Cluster Efficiency

Also in real life, we generally do not find a single server?

Assume a round robin load balancer where each hit goes to the next server.

The cache has one entry which has a variable lifespan of requests, say caused by a time to live. The following table shows how that lifespan can affect hits and misses.

```
Server 1     Server 2    Server 3     Server 4
 M             M            M            M
 H             H            H            H
 H             H            H            H
 H             H            H            H
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
  H             H           H           H
  ...           ...         ...         ...
```

The cache hit ratios for the system as a whole are as follows:

Entry Lifespan in Hits	Hit Ratio 1 Server	Hit Ratio 2 Servers	Hit Ratio 3 Servers	Hit Ratio 4 Servers
2	1/2	0/2	0/2	0/2
4	3/4	2/4	1/4	0/4
10	9/10	8/10	7/10	6/10
20	19/20	18/20	17/20	16/10
50	49/50	48/50	47/20	46/50

The efficiency of a cluster of standalone caches is generally:

```
(Lifespan in requests - Number of Standalone Caches) / Lifespan in requests
```

Where the lifespan is large relative to the number of standalone caches, cache efficiency is not much affected.

However when the lifespan is short, cache efficiency is dramatically affected.

(To solve this problem, Ehcache supports distributed caching, where an entry put in a local cache is also propagated to other servers in the cluster.)

2.1.4.5 A cache version of Amdahl's law

From the above we now have:

```
1 / ((1 - Proportion Sped Up * effective cache efficiency) +
(Proportion Sped Up  * effective cache efficiency)/ Speed up)
```

effective cache efficiency = cache efficiency * cluster efficiency

2.1.4.6 Web Page example

Applying this to the earlier web page cache example where we have cache efficiency of 35% and average request lifespan of 10 requests and two servers:

```
cache efficiency = .35
cluster efficiency = .(10 - 1) / 10
                   = .9
effective cache efficiency = .35 * .9
                           = .315
      1 / ((1 - 1 * .315) + 1 * .315 / 1000)
      = 1 / (.685 + .000315)
      = 1.45 times system speedup
```

What if, instead the cache efficiency is 70%; a doubling of efficiency. We keep to two servers.

```
cache efficiency = .70
cluster efficiency = .(10 - 1) / 10
                   = .9
effective cache efficiency = .70 * .9
                           = .63
      1 / ((1 - 1 * .63) + 1 * .63 / 1000)
      = 1 / (.37 + .00063)
      = 2.69 times system speedup
```

What if, instead the cache efficiency is 90%; a doubling of efficiency. We keep to two servers.

```
cache efficiency = .90
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
cluster efficiency = .(10 - 1) / 10
                   = .9
effective cache efficiency = .9 * .9
                           = .81
      1 / ((1 - 1 * .81) + 1 * .81 / 1000)
      = 1 / (.19 + .00081)
      = 5.24 times system speedup
```

Why is the reduction so dramatic? Because Amdahl's law is most sensitive to the proportion of the system that is sped up.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

3 Getting Started

3.1 Getting Started

Firstly, if you have not downloaded Ehcache, you can download it here.

Ehcache can be used directly. It can also be used with the popular Hibernate Object/Relational tool. Finally, it can be used for Java EE Servlet Caching.

This quick guide gets you started on each of these. The rest of the documentation can be explored for a deeper understanding.

3.1.1 General Purpose Caching

- Make sure you are using a supported Java version.
- Place the Ehcache jar into your classpath.
- Ensure that any libraries required to satisfy dependencies are also in the classpath.
- Configure ehcache.xml and place it in your classpath.
- Optionally, configure an appropriate logging level.
  See the Code Samples chapter for more information on direct interaction with ehcache.

3.1.2 Hibernate

- Perform the same steps as for General Purpose Caching.
- Create caches in ehcache.xml.
  See the Hibernate Caching chapter for more information.

3.1.3 Distributed Caching

Ehcache supports distributed caching with two lines of configuration.

- Download the ehcache-distribution package.
- Add ehcache-core jar to your classpath
- Add ehcache-terracotta jar to your classpath
- Add a 'terracotta' element to your 'cache' stanza(s) in ehcache.xml
- Add a 'terracottaConfig' element to your 'ehcache' stanza in ehcache.xml.
- See the Distributed Caching With Terracotta chapter for more information.

3.1.4 Java EE Servlet Caching

- Perform the same steps as for General Purpose Caching.
- Configure a cache for your web page in ehcache.xml.
- To cache an entire web page, either use SimplePageCachingFilter or create your own subclass of CachingFilter
- To cache a jsp:Include or anything callable from a RequestDispatcher, either use SimplePageFragmentCachingFilter or create a subclass of PageFragmentCachingFilter.
- Configure the web.xml. Declare the filters created above and create filter mapping associating the filter with a URL.
  See the Web Caching chapter for more information.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

3.1.5 RESTful and SOAP Caching with the Cache Server

- Download the ehcache-standalone-server from https://sourceforge.net/projects/ehcache/files/ehcache-server.
- cd to the bin directory
- Type `startup.sh` to start the server with the log in the foreground.

  By default it will listen on port 8080, will have both RESTful and SOAP web services enabled, and will use a sample Ehcache configuration from the WAR module.
- See the code samples in the Cache Server chapter. You can use Java or any other programming language to the use the Cache Server.

  See the Cache Server chapter for more information.

3.1.6 JCache style caching

Ehcache contains an early draft implementation of JCache contained in the net.sf.ehcache.jcache package.

See the JSR107 chapter for usage.

3.1.7 Spring, Cocoon, Acegi and other frameworks

Usually, with these, you are using Ehcache without even realising it. The first steps in getting more control over what is happening are:

- discover the cache names used by the framework
- create your own ehcache.xml with settings for the caches and place it in the application classpath.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

4 Dependencies

4.1 Java Requirements and Dependencies

4.1.1 Java Requirements

Current Ehcache releases require Java 1.5 and 1.6 at runtime.

Ehcache 1.5 requires Java 1.4.

The ehcache-monitor module, which provides management and monitoring, will work with Ehcache 1.2.3 but only for Java 1.5 or higher.

4.1.2 Mandatory Dependencies

Ehcache core 1.6 through to 1.7.0 has no dependencies.

Ehcache core 1.7.1 depends on SLF4J (http://www.slf4j.org), an increasingly commonly used logging framework which provides a choice of concrete logging implementation. See the chapter on Logging for configuration details.

Other modules have dependencies as specified in their maven poms.

4.1.3 Maven Snippet

To include Ehcache in your project use:

```
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>2.3.1</version>
    <type>pom</type>
</dependency>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

5 Cache Concepts

5.1 Key Ehcache Concepts

5.1.1 Definitions

- **cache-hit**: When a data element is requested of the cache and the element exists for the given key, it is referrred to as a cache hit (or simply 'hit').
- **cache-miss**: When a data element is requested of the cache and the element does not exist for the given key, it is referred to as a cache miss (or simply 'miss').
- **system-of-record**: The core premise of caching assumes that there is a source of truth for the data. This is often referred to as a system-of-record (SOR). The cache acts as a local copy of data retrieved from or stored to the system-of-record.
- **SOR**: See system-of-record.

5.1.2 Key Ehcache Classes

Ehcache consists of a `CacheManager`, which manages caches. Caches contain elements, which are essentially name value pairs. Caches are physically implemented either in-memory, or on disk.

5.1.2.1 CacheManager

The `CacheManager` comprises Caches which in turn comprise Elements.

Creation of, access to and removal of caches is controlled by the `CacheManager`.

5.CacheManager Creation Modes

`CacheManager` supports two creation modes: singleton and instance.

5.Singleton Mode

Ehcache-1.1 supported only one `CacheManager` instance which was a singleton. CacheManager can still be used in this way using the static factory methods.

5.Instance Mode

From ehcache-1.2, CacheManager has constructors which mirror the various static create methods. This enables multiple CacheManagers to be created and used concurrently. Each CacheManager requires its own configuration.

If the Caches under management use only the MemoryStore, there are no special considerations. If Caches use the DiskStore, the diskStore path specified in each CacheManager configuration should be unique. When a new CacheManager is created, a check is made that there are no other CacheManagers using the same diskStore path. If there are, a CacheException is thrown. If a CacheManager is part of a cluster, there will also be listener ports which must be unique.

5.Mixed Singleton and Instance Mode

If an application creates instances of CacheManager using a constructor, and also calls a static create method, there will exist a singleton instance of CacheManager which will be returned each time the create method is called together with any other instances created via constructor. The two types will coexist peacefully.

5.1.2.2 Ehcache

All caches implement the `Ehcache` interface. A cache has a name and attributes. Each cache contains Elements.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

A Cache in Ehcache is analogous to a cache region in other caching systems.

Cache elements are stored in the `MemoryStore`. Optionally they also overflow to a `DiskStore`.

5.1.2.3 Element

An element is an atomic entry in a cache. It has a key, a value and a record of accesses. Elements are put into and removed from caches. They can also expire and be removed by the Cache, depending on the Cache settings.

As of ehcache-1.2 there is an API for Objects in addition to the one for Serializable. Non-serializable Objects can use all parts of Ehcache except for DiskStore and replication. If an attempt is made to persist or replicate them they are discarded without error and with a DEBUG level log message.

The APIs are identical except for the return methods from Element. Two new methods on Element: getObjectValue and getKeyValue are the only API differences between the Serializable and Object APIs. This makes it very easy to start with caching Objects and then change your Objects to Seralizable to participate in the extra features when needed. Also a large number of Java classes are simply not Serializable.

5.1.3 Cache Usage Patterns

There are several common access patterns when using a cache. Ehcache supports the following patterns:

- cache-aside (or direct manipulation)
- cache-as-sor (a combination of read-through and write-through or write-behind patterns)
- read-through
- write-through
- write-behind (or write-back) []

5.1.3.1 cache-aside

Here, application code uses the cache directly.

This means that application code which accesses the system-of-record (SOR) should consult the cache first, and if the cache contains the data, then return the data directly from the cache, bypassing the SOR.

Otherwise, the application code must fetch the data from the system-of-record, store the data in the cache, and then return it.

When data is written, the cache must be updated with the system-of-record.

This results in code that often looks like the following pseudo-code:

```
public class MyDataAccessClass
{
    private final Ehcache cache;
    public MyDataAccessClass(Ehcache cache)
    {
        this.cache = cache;
    }

    /* read some data, check cache first, otherwise read from sor */
    public V readSomeData(K key)
    {
        Element element;
        if ((element = cache.get(key)) != null) {
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
            return element.getValue();
        }
        // note here you should decide whether your cache
        // will cache 'nulls' or not
        if (value = readDataFromDataStore(key)) != null) {
            cache.put(new Element(key, value));
        }
        return value;
    }
    /* write some data, write to sor, then update cache */
    public void writeSomeData(K key, V value)
    {
        writeDataToDataStore(key, value);
        cache.put(new Element(key, value);
    }
}
```

5.1.3.2 cache-as-sor

The cache-as-sor pattern implies using the cache as though it were the primary system-of-record (SOR). The pattern delegates SOR reading and writing activies to the cache, so that application code is absolved of this responsibility.

To implement the cache-as-sor pattern, use a combination of the following read and write patterns:

- read-through
- write-through or write-behind

Advantages of using the cache-as-sor pattern are:

- less cluttered application code (improved maintainability)
- easily choose between write-through or write-behind strategies on a per-cache basis (use only configuration)
- allow the cache to solve the "thundering-herd" problem

Disadvantages are:

- less directly visible code-path

5.1.3.3 read-through

The read-through pattern mimics the structure of the cache-aside pattern when reading data. The difference is that you must implement the `CacheEntryFactory` interface to instruct the cache how to read objects on a cache miss, and you must wrap the Ehcache instance with an instance of `SelfPopulatingCache`.

Compare the appearance of the read-through pattern code to the code provided in the cache-aside pattern. (The full example is provided at the end of this document that includes a read-through and write-through implementation).

5.1.3.4 write-through

The write-through pattern mimics the structure of the cache-aside pattern when writing data. The difference is that you must implement the `CacheWriter` interface and configure the cache for write-through or write-behind.

A write-through cache writes data to the system-of-record in the same thread of execution, therefore in the common scenario of using a database transaction in context of the thread, the write to the database is covered by the transaction in scope.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

More details (including configuration settings) can be found in the User Guide chapter on Write-through and Write-behind Caching.

5.1.3.5 write-behind

The write-behind pattern changes the timing of the write to the system-of-record. Rather than writing to the System of Record in the same thread of execution, write-behind queues the data for write at a later time.

The consequences of the change from write-through to write-behind are that the data write using write-behind will occur outside of the scope of the transaction.

This often-times means that a new transaction must be created to commit the data to the system-of-record that is separate from the main transaction.

More details (including configuration settings) can be found in the User Guide chapter on Write-through and Write-behind Caching.

5.1.3.6 cache-as-sor example

```
public class MyDataAccessClass
{
    private final Ehcache cache;
    public MyDataAccessClass(Ehcache cache)
    {
        cache.registerCacheWriter(new MyCacheWriter());
        this.cache = new SelfPopulatingCache(cache);
    }
    /* read some data - notice the cache is treated as an SOR.
     * the application code simply assumes the key will always be available
     */
    public V readSomeData(K key)
    {
        return cache.get(key);
    }
    /* write some data - notice the cache is treated as an SOR, it is
     * the cache's responsibility to write the data to the SOR.
     */
    public void writeSomeData(K key, V value)
    {
        cache.put(new Element(key, value);
    }
    /**
     * Implement the CacheEntryFactory that allows the cache to provide
     * the read-through strategy
     */
    private class MyCacheEntryFactory implements CacheEntryFactory
    {
        public Object createEntry(Object key) throws Exception
        {
            return readDataFromDataStore(key);
        }
    }
    /**
     * Implement the CacheWriter interface which allows the cache to provide
     * the write-through or write-behind strategy.
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
     */
    private class MyCacheWriter implements CacheWriter
        public CacheWriter clone(Ehcache cache) throws CloneNotSupportedException;
        {
            throw new CloneNotSupportedException();
        }
        public void init() { }
        void dispose() throws CacheException { }
        void write(Element element) throws CacheException;
        {
            writeDataToDataStore(element.getKey(), element.getValue());
        }
        void writeAll(Collection<Element> elements) throws CacheException
        {
            for (Element element : elements) {
                write(element);
            }
        }
        void delete(CacheEntry entry) throws CacheException
        {
            deleteDataFromDataStore(element.getKey());
        }
        void deleteAll(Collection<CacheEntry> entries) throws CacheException
        {
            for (Element element : elements) {
                delete(element);
            }
        }
    }
}
```

5.1.3.7 Copy Cache

A Copy Cache can have two behaviors: it can copy Element instances it returns, when `copyOnRead` is true and copy elements it stores, when `copyOnWrite` to true.

A copy on read cache can be useful when you can't let multiple threads access the same Element instance (and the value it holds) concurrently. For example, where the programming model doesn't allow it, or you want to isolate changes done concurrently from each other.

Copy on write also lets you determine exactly what goes in the cache and when. i.e. when the value that will be in the cache will be in state it was when it actually was put in cache. *All mutations to the value, or the element, after the put operation will not be reflected in the cache*.

A concrete example of a copy cache is a Cache configured for `XA`. It will always be configured `copyOnRead` and `copyOnWrite` to provide proper transaction isolation and clear transaction boundaries (the state the objects are in at commit time is the state making it into the cache).

By default, the copy operation will be performed using standard Java object serialization. We do recognize though that for some applications this might not be good (or fast) enough. You can configure your own `CopyStrategy` which will be used to perform these copy operations. For example, you could easily implement use cloning rather than Serialization.

More information on configuration can be found here: copyOnRead and copyOnWrite cache configuration.

6 Configuration

6.1 Cache Configuration

Caches can be configured in Ehcache either declaratively, in xml, or by creating them programmatically and specifying their parameters in the constructor.

While both approaches are fully supported it is generally a good idea to separate the cache configuration from runtime use. There are also these benefits:

- It is easy if you have all of your configuration in one place. Caches consume memory, and disk space. They need to be carefully tuned. You can see the total effect in a configuration file. You could do this code, but it would not as visible.
- Cache configuration can be changed at deployment time.
- Configuration errors can be checked for at start-up, rather than causing a runtime error.

This chapter covers XML declarative configuration.

Ehcache is redistributed by lots of projects. They may or may not provide a sample Ehcache XML configuration file. If one is not provided, download Ehcache from http://ehcache.org. It, and the ehcache.xsd is provided in the distibution.

6.1.1 Dynamically Changing Cache Configuration

After a Cache has been started its configuration is not generally changeable. However, since Ehcache 2.0, certain aspects of cache configuration can modified dynamically at runtime, namely:

- timeToLive
- timeToIdle
- maxElementsInMemory
- maxElementsOnDisk
- memory store eviciton policy
- CacheEventListeners can be added and removed dynamically []

Note that the `eternal` attribute, when set to "true", overrides `timeToLive` and `timeToIdle` so that no expiration can take place.

This example shows how to dynamically modify the cache configuration of an already running cache:

```
Cache cache = manager.getCache("sampleCache");
CacheConfiguration config = cache.getCacheConfiguration();
config.setTimeToIdleSeconds(60);
config.setTimeToLiveSeconds(120);
config.setMaxElementsInMemory(10000);
config.setMaxElementsOnDisk(1000000);
```

Dynamic cache configurations can also be frozen to prevent future changes:

```
Cache cache = manager.getCache("sampleCache");
cache.disableDynamicFeatures();
```

6.1.2 Memory Based Cache Sizing (Ehcache 2.5 and higher)

Historically Ehcache has only permitted sizing of caches by maxElementsInMemory for the the OnHeap Store and maxElementsOnDisk for the DiskStore. The OffHeap Store introduced sizing in terms of memory use.

From Ehcache 2.5, we are extending sizing based on bytes consumed to all stores.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

The new cache attributes are:

- maxBytesOnHeap
- maxBytesOffHeap (formerly maxMemoryOffHeap)
- maxBytesOnDisk

  Size may be expressed in bytes using the convention for specifying -Xmx (e.g. 200k, 30m, 5g etc.)

  For added simplicity you can also specify these attributes at the ehcache level, which then applies them to the whole CacheManager, leaving each cache to share in one large pool of memory.

  If you specify a CacheManager wide sizes, you can also use percentages at the cache level. e.g maxBytesOnHeap="20%".

  For completeness we also add cache pinning and rules for cache-level configuration to override CacheManager level configuration.

6.1.2.1 Example Configuration

An example is shown below. It allocates 1GB on heap and 4GB off heap at the CacheManager level.

It also demonstrates some finer points which we will conver in the following sections.

```
<ehcache maxBytesOnHeap="1g" maxBytesOffHeap="4g" maxBytesOnDisk="100g" >
   <cache name="explicitlyAllocatedCache1"
          maxBytesOnHeap="50m"
          maxBytesOffHeap="200m"
          timeToLiveSeconds="100"
   </cache>
   <cache name="explicitlyAllocatedCache2"
          maxBytesOnHeap="10%"
          maxBytesOffHeap="200m"
          timeToLiveSeconds="100"
   </cache>
   <cache name="automaticallyAllocatedCache1"
          timeToLiveSeconds="100"
          overflowToDisk="true"
   </cache>
   <cache name="automaticallyAllocatedCache2"
          timeToLiveSeconds="100"
   </cache>
   <cache name="pinnedCache"
          timeToLiveSeconds="100"
          <pinning storage="inMemory"/>
   </cache>
</ehcache>
```

6.1.2.2 CacheManager versus Cache level configuration

Caches without specific configuration participate in the general storage pools. And caches with specific configuration take either a fixed amount (e.g. 200m) or a percentage (e.g. 5%).

When managing storage out of CacheManager level pools, element are evicted across all caches using an LRU (possibly clock based) algorithm.

The CacheManager level storage pool attributes are:

- maxBytesOnHeap="size"

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- maxBytesOffHeap="size"
- maxBytesOnDisk="size"

  where size is the Java -Xmx syntax. e.g. 4g

  If a store is configured using a CacheManager level pool, the maxElements form of configuration cannot be used.

6.Cache level overrides

There will be times when the developer knows more about the tuning of each cache than and can outperform CacheManager level tuning. In this case it is recommended to provide cache specific configuration.

Cache specific configuration always overrides CacheManager allocations.

The Cache level storage pool attributes are:

- maxBytesOnHeap="size | %"
- maxBytesOffHeap="size | %"
- maxBytesOnDisk="size | %"

  where size is the Java -Xmx syntax. e.g. 4g and % is simply a positive number between 0 and 100. e.g. 5%

6.Overallocation Rules

To prevent overallocation of CacheManager level pools by cache level overrdies we perform a number of checks on startup:

- We convert percentages to fixed amounts
- We then add the those to any other fixed allocations
- If the sum exceeds the CacheManager allocation, we throw an `InvalidConfigurationException`.
- If the sum equals the CacheManager allocation, we issue a warning, as there will not be memory left for caches without overrides

  Overallocations can only be detected at configuration time. For this reason we do not permit the use of max element count (e.g. maxElementsInMemory) configuration with CacheManager storage pools.

6.1.2.3 Pinning of Caches and Elements in Memory

6.Pinning of Caches

Caches may be pinned using the new pinning sub-element:

```
<cache name="pinnedCache"
   timeToLiveSeconds="100"
   <pinning storage="onHeap | inMemory | inCache" />
</cache>
```

Pinning means that cache Elements are never evicted due to space. They cache will continue to grow as elements are added to it. Elements will only be evicted unless the Element has expired.

Pinning is possible at three different levels:

- onHeap - retain the elements in the Java heap
- inMemory - retain the elements in either the OnHeap or the OffHeap stores, depending on what stores there are and how much is space is available in each.
- inCache - retain the elements in the cache. This allows further off loading to either the DiskStore in a standalone cache, or the L2 in a Terracotta backed Distributed Ehcache.

  The recommended use is reference data, where you always want the whole dataset in memory.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

Pinning cannot be used with either maxElementsInMemory or maxBytesOnHeap - it is unbounded.

Caution: It is possible to cause an OutOfMemory error with pinned caches. They may even look like a memory leak in the application. They are meant to be a convenience. They should not be used with potentially unbounded data sets.

6.Pinning of Elements

Some APIs like OpenJPA and Hibernate require pinning of specific Elements.

A new method on Element, Element.setPinned(true|false, onHeap|inMemory|inCache) has been added. When a pinned Element is placed in the cache it will not be evicted from the On-Heap store. Element level pinning is a noop when the whole cache is pinned.

6.1.3 Cache Warming for multi-tier Caches (Ehcache 2.5 and higher)

When a cache starts up, the On-Heap and Off-Heap stores are always empty. Ehcache provides a BootstrapCacheLoader mechanism to overcome this. The BootstrapCacheLoader is run before the cache is set to alive. If synchronous, loading completes before the CacheManager starts, or if asynchronous, the CacheManager starts but loading continues agressively rather than waiting for elements to be requested, which is a lazy loading approach.

Replicated caches provide a boot strap mechanism which populates them. For example following is the JGroups bootstrap cache loader:

```
<bootstrapCacheLoaderFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsBootstrapCacheLoaderFactory"
    properties="bootstrapAsynchronously=true"/>
```

We have two new bootstrapCacheLoaderFactory implementations: one for standalone caches with DiskStores, and one for Terracotta Distributed caches.

6.1.3.1 DiskStoreBootstrapCacheLoaderFactory

The DiskStoreBootstrapCacheLoaderFactory loads elements from the DiskStore to the On-Heap Store and the Off-Heap store until either:

- the memory stores are full
- the DiskStore has been completely loaded

6.Configuration

The DiskStoreBootstrapCacheLoaderFactory is configured as follows:

```
<bootstrapCacheLoaderFactory
    class="net.sf.ehcache.store.DiskStoreBootstrapCacheLoaderFactory"
    properties="bootstrapAsynchronously=true"/>
```

6.1.3.2 TerracottaBootstrapCacheLoaderFactory

The TerracottaBootstrapCacheLoaderFactory loads elements from the Terracotta L2 to the On-Heap Store and the Off-Heap store until either:

- the memory stores are full
- the L2 has been completely loaded

The TerracottaBootstrapCacheLoader uses knowledge of what Elements other L1s in the cluster have to predict the likely hot set for this L1. If this L1 is the first L1 in the cluster, then there is no guidance. The loader will then only load Elements from the L2 if the combined capacity of the memory stores exceeds 50% of the size in memory of the cache in the L2.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

6.Configuration

The TerracottaStoreBootstrapCacheLoaderFactory is configured as follows:

```
<bootstrapCacheLoaderFactory
    class="net.sf.ehcache.store.TerracottaStoreBootstrapCacheLoaderFactory"
    properties="bootstrapAsynchronously=true"/>
```

6.1.4 copyOnRead and copyOnWrite cache configuration

A cache can be configured to copy the data, rather than return reference to it on get or put. This is configured using the `copyOnRead` and `copyOnWrite` attributes of cache and defaultCache elements in your configuration or programmatically as follows:

```
CacheConfiguration config = new CacheConfiguration("copyCache", 1000).copyOnRead(tr
Cache copyCache = new Cache(config);
```

The default configuration will be false for both options.

In order to copy elements on put()-like and/or get()-like operations, a CopyStrategy is being used. The default implementation uses serialization to copy elements. You can provide your own implementation of `net.sf.ehcache.store.compound.CopyStrategy` like this:

```
<cache name="copyCache"
       maxElementsInMemory="10"
       eternal="false"
       timeToIdleSeconds="5"
       timeToLiveSeconds="10"
       overflowToDisk="false"
       copyOnRead="true"
       copyOnWrite="true">
    <copyStrategy class="com.company.ehcache.MyCopyStrategy"/>
</cache>
```

Per cache, a single instance of your `CopyStrategy` will be use, hence your implementation of CopyStrategy.copy(T): T has to be thread-safe.

6.1.5 Special System Properties

6.1.5.1 net.sf.ehcache.disabled

Setting this System Property to `true` disables caching in ehcache. If disabled no elements will be added to a cache. i.e. puts are silently discarded.

e.g. `java -Dnet.sf.ehcache.disabled=true` in the Java command line.

6.1.5.2 net.sf.ehcache.use.classic.lru

Set this System property to `true` to use the older LruMemoryStore implementation when LRU is selected as the eviction policy.

This is provided for ease of migration.

e.g. `java -Dnet.sf.ehcache.use.classic.lru=true` in the Java command line.

6.1.6 ehcache.xsd

Ehcache configuration files must be comply with the Ehcache XML schema, ehcache.xsd.

It can be downloaded from http://ehcache.org/ehcache.xsd.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

6.1.7 ehcache-failsafe.xml

If the CacheManager default constructor or factory method is called, Ehcache looks for a file called ehcache.xml in the top level of the classpath. Failing that it looks for ehcache-failsafe.xml in the classpath. ehcache-failsafe.xml is packaged in the Ehcache jar and should always be found.

ehcache-failsafe.xml provides an extremely simple default configuration to enable users to get started before they create their own ehcache.xml.

If it used Ehcache will emit a warning, reminding the user to set up a proper configuration.

The meaning of the elements and attributes are explained in the section on ehcache.xml.

```
<ehcache>
    <diskStore path="java.io.tmpdir"/>
    <defaultCache
            maxElementsInMemory="10000"
            eternal="false"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            overflowToDisk="true"
            maxElementsOnDisk="10000000"
            diskPersistent="false"
            diskExpiryThreadIntervalSeconds="120"
            memoryStoreEvictionPolicy="LRU"
            />
</ehcache>
```

6.1.8 Update Checker

The update checker is used to see if you have the latest version of Ehcache. It is also used to get non-identifying feedback on the OS architectures using Ehcache.

To disable the check, do one of the following:

6.1.8.1 By System Property

```
-Dnet.sf.ehcache.skipUpdateCheck=true
```

6.1.8.2 By Configuration

The outer `ehcache` element takes an `updateCheck` attribute, which is set to false as in the following example.

```
-->
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd"
         updateCheck="false" monitoring="autodetect"
         dynamicConfig="true">
```

6.1.9 ehcache.xml and other configuration files

Prior to ehcache-1.6, Ehcache only supported ASCII ehcache.xml configuration files. Since ehcache-1.6, UTF8 is supported, so that configuration can use Unicode. As UTF8 is backwardly compatible with ASCII, no conversion is necessary.

If the CacheManager default constructor or factory method is called, Ehcache looks for a file called ehcache.xml in the top level of the classpath.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

The non-default creation methods allow a configuration file to be specified which can be called anything.

One XML configuration is required for each CacheManager that is created. It is an error to use the same configuration, because things like directory paths and listener ports will conflict. Ehcache will attempt to resolve conflicts and will emit a warning reminding the user to configure a separate configuration for multiple CacheManagers with conflicting settings.

The sample ehcache.xml is included in the Ehcache distribution. It contains full commentary required to configure each element. Further information can be found in specific chapters in the Guide.

It can also be downloaded from http://ehcache.org/ehcache.xml.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

7 Storage Options

7.1 Storage Options

Ehcache has three stores:

- a MemoryStore
- an OffHeapStore (BigMemory, Enterprise Ehcache only) and
- a DiskStore (two versions: open source and Ehcache Enterprise)

7.1.1 Memory Store

The `MemoryStore` is always enabled. It is not directly manipulated, but is a component of every cache.

- Suitable Element Types

  All Elements are suitable for placement in the MemoryStore.

  It has the following characteristics:

  - Safety

    Thread safe for use by multiple concurrent threads.

    Tested for memory leaks. See MemoryCacheTest#testMemoryLeak. This test passes for Ehcache but exploits a number of memory leaks in JCS. JCS will give an OutOfMemory error with a default 64M in 10 seconds.
  - Backed By JDK

    LinkedHashMap The `MemoryStore` for JDK1.4 and JDK 5 it is backed by an extended LinkedHashMap. This provides a combined linked list and a hash map, and is ideally suited for caching. Using this standard Java class simplifies the implementation of the memory cache. It directly supports obtaining the least recently used element.
  - Fast

    The memory store, being all in memory, is the fastest caching option.

7.1.1.1 Memory Use, Spooling and Expiry Strategy

All caches specify their maximum in-memory size, in terms of the number of elements, at configuration time.

When an element is added to a cache and it goes beyond its maximum memory size, an existing element is either deleted, if overflowToDisk is false, or evaluated for spooling to disk, if overflowToDisk is true. In the latter case, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled. The eviction of an item from the memory store is based on the MemoryStoreEvictionPolicy setting specified in the configuration file.

memoryStoreEvictionPolicy is an optional attribute in ehcache.xml introduced since 1.2. Legal values are LRU (default), LFU and FIFO.

LRU, LFU and FIFO eviction policies are supported. LRU is the default, consistent with all earlier releases of ehcache.

- Least Recently Used (LRU) - Default

  The eldest element, is the Least Recently Used (LRU). The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.
- Less Frequently Used (LFU)

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element with least number of hits, the Less Frequently Used element, is evicted.

- First In First Out (FIFO)

  Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

  For all the eviction policies there are also `putQuiet` and `getQuiet` methods which do not update the last used timestamp.

  When there is a `get` or a `getQuiet` on an element, it is checked for expiry. If expired, it is removed and null is returned.

  Note that at any point in time there will usually be some expired elements in the cache. Memory sizing of an application must always take into account the maximum size of each cache. There is a convenience method which can provide an estimate of the size in bytes of the `MemoryStore`. See calculateInMemorySize(). It returns the serialized size of the cache. Do not use this method in production. It is very slow. It is only meant to provide a rough estimate.

  The alternative would have been to have an expiry thread. This is a trade-off between lower memory use and short locking periods and cpu utilisation. The design is in favour of the latter. For those concerned with memory use, simply reduce the `maxElementsInMemory`.

7.1.2 Off-Heap Store

Terracotta BigMemory is an add-on to Enterprise Ehcache that permits caches to use an additional type of memory store outside the object heap.

This off-heap store, which is not subject to Java GC, is 100 times faster than the DiskStore and allows very large caches to be created (we have tested this up to 350GB).

Because off-heap data is stored in bytes, there are two implications:

- Only Serializable cache keys and values can be placed in the store, similar to DiskStore.
- Serialization and deserialization take place on putting and getting from the store. This means that the off-heap store is slower in an absolute sense (around 10 times slower than the MemoryStore), but this theoretical difference disappears due to two effects:
  - the MemoryStore holds the hottest subset of data from the off-heap store, already in deserialized form
  - when the GC involved with larger heaps is taken into account, the off-heap store is faster on average

7.1.2.1 Suitable Element Types

Only `Elements` which are `Serializable` can be placed in the `OffHeapMemoryStore`. Any non serializable `Elements` which attempt to overflow to the `OffHeapMemoryStore` will be removed instead, and a WARNING level log message emitted.

See the Off-Heap Store chapter for more details.

7.1.3 DiskStore

The `DiskStore` provides a disk spooling facility.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

7.1.3.1 `DiskStore`s are Optional

The diskStore element in ehcache.xml is now optional (as of 1.5). If all caches use only `MemoryStore`s, then there is no need to configure a diskStore. This simplifies configuration, and uses less threads. It is also good where where multiple CacheManagers are being used, and multiple disk store paths would need to be configured.

If one or more caches requires a DiskStore, and none is configured, java.io.tmpdir will be used and a warning message will be logged to encourage explicity configuration of the diskStore path.

7.Turning off disk stores

To turn off disk store path creation, comment out the diskStore element in ehcache.xml.

The `ehcache-failsafe.xml` configuration uses a disk store. This will remain the case so as to not affect existing Ehcache deployments. So, if you do not wish to use a disk store make sure you specify your own ehcache.xml and comment out the diskStore element.

7.1.3.2 Suitable Element Types

Only `Elements` which are `Serializable` can be placed in the DiskStore. Any non serializable `Elements` which attempt to overflow to the `DiskStore` will be removed instead, and a WARNING level log message emitted.

7.1.3.3 Enterprise DiskStore

The commercial version of Ehcache 2.4 introduced an upgraded disk store. Improvements include:

- Upgraded fragmentation control/management to be the same as offheap
- No Heap used for fragmentation management or keys
- Much more predictable write latency up to caches over half a terabyte.
- SSD aware and optimised.

  Throughput is approximately 110,000 operations/s which translates to around 60MB/sec on a 10k rpm hard drive with even higher rates on SSD drives, for which the Disk

7.1.3.4 Storage

7.Files

The disk store creates a data file for each cache on startup called " *cache_name*.data". If the `DiskStore` is configured to be persistent, an index file called " **cache name**.index" is created on flushing of the `DiskStore` either explicitly using `Cache.flush` or on `CacheManager` shutdown.

7.Storage Location

Files are created in the directory specified by the diskStore configuration element. The diskStore configuration for the ehcache-failsafe.xml and bundled sample configuration file ehcache.xml is "java.io.tmpdir", which causes files to be created in the system's temporary directory.

7. *diskStore Element*

The `diskStore` element is has one attribute called `path`. --- *diskStore path="java.io.tmpdir"/* --- Legal values for the path attibute are legal file system paths. e.g.for Unix

```
/home/application/cache
```

The following system properties are also legal, in which case they are translated:

- user.home - User's home directory
- user.dir - User's current working directory
- java.io.tmpdir - Default temp file path

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- ehcache.disk.store.dir - A system property you would normally specify on the command line e.g. java -Dehcache.disk.store.dir=/u01/myapp/diskdir ...

  Subdirectories can be specified below the system property e.g.

  ```
  java.io.tmpdir/one
  becomes, on a Unix system,
  /tmp/one
  ```

7.1.3.5 Expiry

One thread per cache is used to remove expired elements. The optional attribute `diskExpiryThreadIntervalSeconds` sets the interval between runs of the expiry thread. Warning: setting this to a low value is not recommended. It can cause excessive `DiskStore` locking and high cpu utilisation. The default value is 120 seconds.

7.1.3.6 Eviction

If the `maxElementsOnDisk` attribute is set, elements will be evicted from the `DiskStore` when it exceeds that amount. The LFU algorithm is used for these evictions. It is not configurable to use another algorithm.

7.1.3.7 Serializable Objects

Only Serializable objects can be stored in a `DiskStore`. A NotSerializableException will be thrown if the object is not serializable.

7.1.3.8 Safety

`DiskStores` are thread safe.

7.1.3.9 Persistence

`DiskStore` persistence is controlled by the diskPersistent configuration element. If false or omitted, `DiskStores` will not persist between `CacheManager` restarts. The data file for each cache will be deleted, if it exists, both on shutdown and startup. No data from a previous instance `CacheManager` is available.

If diskPersistent is true, the data file, and an index file, are saved. Cache Elements are available to a new `CacheManager`. This `CacheManager` may be in the same VM instance, or a new one.

The data file is updated continuously during operation of the Disk Store if `overflowToDisk` is true. Otherwise it is not updated until either `cache.flush()` is called or the cache is disposed.

In all cases the index file is only written when dispose is called on the `DiskStore`. This happens when the CacheManager is shut down, a Cache is disposed, or the VM is being shut down. It is recommended that the CacheManager shutdown() method be used. See Virtual Machine Shutdown Considerations for guidance on how to safely shut the Virtual Machine down.

When a `DiskStore` is persisted, the following steps take place:

- Any non-expired Elements of the `MemoryStore` are flushed to the DiskStore
- Elements awaiting spooling are spooled to the data file
- The free list and element list are serialized to the index file

On startup the following steps take place:

- An attempt is made to read the index file. If it does not exist or cannot be read successfully, due to disk corruption, upgrade of ehcache, change in JDK version etc, then the data file is deleted and the `DiskStore` starts with no Elements in it.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- If the index file is read successfully, the free list and element list are loaded into memory. Once this is done, the index file contents are removed. This way, if there is a dirty shutdown, when restarted, Ehcache will delete the dirt index and data files.
- The `DiskStore` starts. All data is available.
- The expiry thread starts. It will delete Elements which have expired.

These actions favour safety over persistence. Ehcache is a cache, not a database. If a file gets dirty, all data is deleted. Once started there is further checking for corruption. When a get is done, if the Element cannot be successfully derserialized, it is deleted, and null is returned. These measures prevent corrupt and inconsistent data being returned.

- Fragmentation

  Expiring an element frees its space on the file. This space is available for reuse by new elements. The element is also removed from the in-memory index of elements.
- Serialization

  Writes to and from the disk use ObjectInputStream and the Java serialization mechanism. This is not required for the MemoryStore. As a result the DiskStore can never be as fast as the MemoryStore.

  Serialization speed is affected by the size of the objects being serialized and their type. It has been found in the ElementTest test that:

  - The serialization time for a Java object being a large Map of String arrays was 126ms, where the a serialized size was 349,225 bytes.
  - The serialization time for a byte[] was 7ms, where the serialized size was 310,232 bytes

  Byte arrays are 20 times faster to serialize. Make use of byte arrays to increase DiskStore performance.
- RAMFS

  One option to speed up disk stores is to use a RAM file system. On some operating systems there are a plethora of file systems to choose from. For example, the Disk Cache has been successfully used with Linux' RAMFS file system. This file system simply consists of memory. Linux presents it as a file system. The Disk Cache treats it like a normal disk - it is just way faster. With this type of file system, object serialization becomes the limiting factor to performance.

  - Operation of a Cache where overflowToDisk is false and diskPersistent is true

    In this configuration case, the disk will be written on `flush` or `shutdown`.

    The next time the cache is started, the disk store will initialise but will not permit overflow from the MemoryStore. In all other respects it acts like a normal disk store.

    In practice this means that persistent in-memory cache will start up with all of its elements on disk. As gets cause cache hits, they will be loaded up into the `MemoryStore`. The oher thing that may happen is that the elements will expire, in which case the `DiskStore` expiry thread will reap them, (or they will get removed on a get if they are expired).

    So, the Ehcache design does not load them all into memory on start up, but lazily loads them as required.

7.1.4 Some Configuration Examples

These examples show how to allocate 8GB of machine memory to different stores. It assumes a data set of 7GB - say for a cache of 7M items (each 1kb in size).

Those who want minimal application response time variance (ie minimizing GC pause times), will likely want all the cache to be off-heap.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

Assuming that 1GB of heap is needed for the rest of the app, they will set their Java config as follows:

```
java -Xms1G -Xmx1G -XX:maxDirectMemorySize=7G
```

And their Ehcache config as

```
<cache
maxElementsInMemory=100
overflowToOffHeap="true"
maxMemoryOffHeap="7G"
... />
```

Those who want best possible performance for a hot set of data while still reducing overall application repsonse time variance will likely want a combination of on-heap and off-heap. The heap will be used for the hot set, the offheap for the rest. So, for example if the hot set is 1M items (or 1GB) of the 7GB data. They will set their Java config as follows

```
java -Xms2G -Xmx2G -XX:maxDirectMemorySize=6G
```

And their Ehcache config as

```
<cache
maxElementsInMemory=1M
overflowToOffHeap="true"
maxMemoryOffHeap="6G"
...>
```

This configuration will compare VERY favorably against the alternative of keeping the less-hot set in a database (100x slower) or caching on local disk (20x slower).

Where pauses are not a problem, the whole data set can be kept on heap:

```
<cache
maxElementsInMemory=1
overflowToOffHeap="false"
...>
Where latency isn't an issue overflow to disk can be used:
```

cache maxElementsInMemory=1M overflowToOffDisk="true" ... ---

7.1.5 Performance Considerations

7.1.5.1 Relative Speeds

Ehcache comes with a `MemoryStore` and a `DiskStore`. The `MemoryStore` is approximately an order of magnitude faster than the `DiskStore`. The reason is that the `DiskStore` incurs the following extra overhead:

- Serialization of the key and value
- Eviction from the `MemoryStore` using an eviction algorithm
- Reading from disk

Note that writing to disk is not a synchronous performance overhead because it is handled by a separate thread.

7.1.5.2 Always use some amount of Heap

A Cache should alway have its `maximumSize` attribute set to 1 or higher. A Cache with a maximum size of 1 has twice the performance of a disk only cache, i.e. one where the `maximumSize` is set to 0. For this reason a warning will be issued if a Cache is created with a 0 `maximumSize`.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

And when using the Offheap Store, frequently accessed elements can be held in heap in derserialized form if an Onheap (configured with maxElementsInMemory) store is used

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

8 Cache Consistency Options

8.1 Cache Consistency Options

The purpose of this chapter is to explain Distributed Ehcache's consistency models in terms of standard distributed systems theory.

8.1.1 Ehcache Topologies

Ehcache is available with the following clustered caching topologies:

- Standalone - the cached data set is held in the application node. Any other application nodes are independent with no communication between them. If standalone caching is being used where there are multiple application nodes running the same application, then there is Weak Consistency between them. Indeed they will only reflect the same values for immutable data or after the time to live on an Element has completed and the Element needs to be reloaded.
- Replicated - the cached data set is held in each application node and data is copied or invalidated across the cluster without locking. Replication can be either asynchronous or synchronous, where the writing thread blocks while progagation occurs. The only consistency mode available in this topology is Weak Consistency.
- Distributed Ehcache - the data is held in a Terracotta Server Array ("SA") with a subset of recently used data held in each application cache node.

The distributed topology supports a very rich set of consistency modes which will be explored in this chapter.

8.1.2 Server Side Consistency

Leaving aside the issue of data also held in the Ehcache nodes, let us look at the server side consistency of the Terracotta Server Array.

8.1.2.1 Server Deployment Topology

Large datasets are handled with partitions which are managed automatically using a consistent hashing algorithm once a set of "stripes" are defined in the *tcconfig*. There is no dynamic resizing of clusters, so the consistenct hash always resolves to the same stripe.

The TSA is typically deployed with a pair of servers per partition of data, which is known in the *tcconfig* as a Mirror Group.

A mirror group has an active server which handles all requests for that partition and a passive or warm standby which does not service any requests. The active server propagates changes to the passive server.

In the language of consistency protocols, the active and passive are replicas - they should contain the same data.

8.1.2.2 How writes are written

Regardless of the consistency model being used, data is written to the TSA the same way.

- Within an Ehcache node, a write is done to an in-process Transaction Buffer (a LinkedBlockingQueue). Within the Java process the write is thread-safe. Any local threads in Ehcache A will have immediate visibility of the change.
- When a write hits the Transaction Buffer, a notify occurs, and the Transaction Buffer initiates sending the write asynchronously to the Terracotta Server Array. The write stays in the Transaction Buffer until an acknowledgement from the TSA has been received.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- Consistent hashing is used to identify which stripe in the TSA to write to. The client maintains knowledge of which replica is the Active server using an election protocol. The write is done to the Active server. The Active server has knowledge of the *tcconfig* and knows to replicate the change to the passive. The write is then written to the Passive. The passive then acknowledges the write to the Active, the Active then acknowledges the write to the Ehcache node. Once received, the write is removed from the Transaction Buffer.

8.1.2.3 Restating in terms of Quorum based replicated-write protocols

To use the terminology from Gifford (1979) a storage system has N storage replicas. A write is a W. A read is an R.

The server side storage system will be strongly consistent if:

- R + W > N.
- W > N/2

  In Terracotta, there is one Active and one Passive. The acknowledgement is not sent until all have been written to. We always read from only one replica, the Active.

  So, R = 1, W = 2, N = 2.

  Substituing the terms of R + W > N, we get 1 + 2 > 2, which is clearly true.

  And for W > N/2 we get 2 > 2/2 => 2 > 1 which is clearly true.

  Therefore we are strongly consistent server side.

8.1.3 Client-Side Consistency

Because data is also held in Ehcache nodes, and Ehcache nodes are what application code interact with, there is more to the story than consistency in the TSA.

Werner Vogel's seminal Eventually Consistent paper presented standard terms for client-side consistency and a way of reasoning about whether that consistency can be achieved in a distributed system. This paper in turn referenced Tannenbaum's Distributed Systems: Principles and Paradigms (2nd Edition).

He was popularising research work done on Bayou, a database system. See Page 290 of Distributed Systems, Principles and Paradigms by Tannenbaum and Van Steen for detailed coverage of this material.

8.1.3.1 Model Components

Before explaining our consistency modes, we need to expain the standard components of the the reference model which is an abstract model of a distributed system that can be used for studying interactions.

- A storage system. The storage system consists of data stored durably in one server or multiple servers connected via a network. In Ehcache durability is optional and the storage system might simply be in memory.
- Client Process A. This is a process that writes to and reads from the storage system.
- Client Processes B and C. These two processes are independent of process A and write to and read from the storage system. It is irrelevant whether these are really processes or threads within the same process; what is important is that they are independent and need to communicate to share information. Client-side consistency has to do with how and when observers (in this case the processes A, B, or C) see updates made to a data object in the storage systems.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

8.1.3.2 Mapping the Model to Distributed Ehcache

The model maps to Distributed Ehcache as follows:

- there is a Terracotta Server Array ("TSA") which is the 'storage system'
- there are three nodes connected to the TSA: Ehcache A, B and C, mapping to the processes in the standard model
- a "write" in the standard model is a "put" or "remove" in Ehcache.

8.1.3.3 Standard Client Side Consistency Modes

It then goes on to define the following consistencies where process A has made an update to a data object:

- Strong consistency. After the update completes, any subsequent access (by A, B, or C) will return the updated value.
- Weak consistency. The system does not guarantee that subsequent accesses will return the updated value.
- Eventual consistency. This is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme.

  Within eventual consistency there are a number of desirable properties:
- Read-your-writes consistency. This is an important model where process A, after it has updated a data item, always accesses the updated value and will never see an older value. This is a special case of the causal consistency model.
- Session consistency. This is a practical version of the previous model, where a process accesses the storage system in the context of a session. As long as the session exists, the system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session needs to be created and the guarantees do not overlap the sessions.
- Monotonic read consistency. If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.
- Monotonic write consistency. In this case the system guarantees to serialize the writes by the same process. Systems that do not guarantee this level of consistency are notoriously hard to program.

  Finally, in eventual consistency, the period between the update and the moment when it is guaranteed that any observer will always see the updated value is dubbed the inconsistency window.

8.1.4 Consistency Modes in Distributed Ehcache

8.1.4.1 Strong Consistency

In the distributed cache, strong consistency is configured as follows:

```
<cache name="sampleCache1"
     ...
     />
     <terracotta consistency="strong" />
</cache>
```

We will walk through how a write is done and show that it is strongly consistent.

1 A thread in Ehcache A performs a write.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

2 Before the write is done, a write lock is obtained from the Terracotta Server (storage system). The write lock is granted only after all read locks have been surrendered.
3 The write is done to an in-process Transaction Buffer. Within the Java process the write is thread-safe. Any local threads in Ehcache A will have immediate visibility of the change.
4 Once the change has hit the Transaction Buffer which is a LinkedBlockingQueue, a notify occurs, and the Transaction Buffer initiates sending the write (update) asynchronously to the Terracotta Server Array (storage system).
5 The Terracotta Server is generally configured with multiple replicas forming a Mirror Group. Within the mirror group there is an Active server, and one or more Passive servers. The write is to the Active server. The Active server does not acknowledge the write until it has written it to each of the passive servers in the Mirror Group. It then sends back an acknowledgement to Ehcache A which then deletes the write from the Transaction Buffer.
6 A read or write request from Ehcache A is immediately available because a read lock is automatically granted when a write lock has already been acquired. A read or write request in Ehcache B or C requires the acquisition of a read or write lock respectively which will block until step 5 has occurred, and in addition, if you have a stale copy locally it is updated first. When the lock is granted the write is present in all replicas. Because Ehcache also maintains copies of Elements in-process in potentially each node, if any of Ehcache A, B or C have a copy they are also updated before Step 5 completes.

Note: This analysis assumes that if the `nonstop` is being used, it is configured with the default of Exception, so that on a `clusterOffline` event no cache operations happen locally. (Nonstop allows fine-grained tradeoffs to be made in the event of a network partition, including dropping consistency)

8.1.4.2 Eventual Consistency

Distributed Ehcache may be configured with `consistency="eventual"`. There is also a bulk loading mode which may additionally be set programmatically with `setNodeBulkLoadEnabled(boolean)`. Finally there is *UnlockedReadsView*, a `CacheDecorator` that can be created like a view on a cache which shows the latest write visible to the local Ehcache node without respect for any locks.

Regardless, Ehcache B and C will eventually see the change made by Ehcache A. This occurs as follows:

- With no partitions or interruptions, B and C will see the change generally within 5ms. The *inconsistency window* is therefore usually less than 5ms.
- If a GC happens on a Terracotta Server Array node, or Ehcache A or B, the inconsistency window is increased by the length of the GC.
- `setNodeBulkLoadEnabled(true)` changes things so that the Terracotta Server Array does not update Ehcache B and C. Instead they are set to a 5 minute fixed TTL. The *inconsistency window* thus increases to 5 minutes plus the above.

If a network partition occurs, the only configurable option is to discard on rejoin, so once this happens Ehcache A or B gets the write.

From the perspective of other threads in Ehcache A, all writes are thread-safe.

In all modes the *happens-before* requirement of the Java Memory Model is honored. As a result the following is true:

- A thread in Ehcache A will see any writes made by another thread. => Read your writes consistency.
- Monotonic Read Consistency in Ehcache A is true.
- Monotonic Write Consistency is Ehcache A is true.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

It should be noted that desirable characteristics of eventual consistency are from the point of view of Ehcache A. From the context of a web application, if order for an end user interacting with a whole application to see this behaviour, either:

- sticky sessions should be used, so that the use interacts with the same node (i.e. Ehcache A) for each step. If an application node falls over, a new session will be established. The time between the last write, failure, detection by the load balancer and allocation to a new application node will take longer than the 5ms plus that it takes for all Ehcache nodes in the cluster to get the write. So when the new application node is switched to, eventual consistency has occurred and no loss of consistency is observed by the user.
- do not use sticky sessions but rely on the time gap between a click or submit and the next one in a *click path* being much larger than the 5ms plus that it takes for other nodes to become eventually consistent. In an Internet context the user is sufficiently distant from the server so that the response time is at least an order of magnitude greater than the *inconsistency window*. Probabilistically it is therefore unlikely that a user would see inconsistency.

8.1.5 Other Safety Features

Ehcache offers a rich set of data safety features. In this section we look at some of the others and how they interact with the `strong` and `eventual` consistency.

8.1.5.1 CAS Cache Operations

We support three CAS (#Compare and Swap#) operations:

- `cache.replace(Element old, Element new)`
- `cache.putIfAbsent(Element)`
- `cache.remove(Element)`

In each case the Terracotta Server Array will only perform the write if the old value is the same as that presented. This is guaranteed to be done atomically as required by the CAS pattern.

CAS achieves strong consistency between A, B and C. The key difference is that it achieves it with optimistic locking rather than pessimistic locking. As with all optimistic locking approaches, the operations are not guaranteed to succeed. If someone else got in and changed the Element ahead of you, the methods will return `false`. You should read the new value, take that into account in your business logic and then retry your mutation.

CAS will work with both `strong` and `eventual` consistency modes, but because it does not use the locks it does not need `strong`.

8.1.6 Use Cases And Recommended Practices

In this section we look at some common use cases and give advice on what consistency and safety options should be used. These serve as a useful starting point for your own analysis.

We welcome commentary and further discussion on these use cases. Please post to the ehcache mailing list or post your questions on the forums.

8.1.6.1 Shopping Cart - optimistic inventory

8.Problem

A user adds items to a shopping cart. Do not decrement inventory until checkout.

8.Solution

Use eventual consistency.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

8.1.6.2 Shopping Cart with Inventory Decrementing

8.Problem

A user adds items to a shopping cart. There is limited inventory and the business policy is that the first user to add the inventory to their shopping cart can buy it. If the user does not proceed to checkout a timer will release the inventory back. As a result, inventory must be decremented at the time the item is added to the shopping cart.

8.Solution

Use strong consistency with one of:

- explicit locking
- local transactions
- XA transactions

The key thing here is that two resources have to be updated: the shopping cart, which is only visible to one user, and on it's own has low consistency requirements, and an inventory which is transactiional in nature.

8.1.6.3 Financial Order Processing - write to cache and database

8.Problem

An order processing system sends a series of messages in a workflow, perhaps using Business Process Management software. The system involves multiple servers and the next step in the processing of an order may occur on any server. Let's say there are 5 steps in the process.

To avoid continual re-reading from a database, the processing results are also written to a distributed cache. The next step could execute in a few ms to minutes depending on what other orders are going through and how busy the hardware is.

8.Solution

Use strong consistency plus XA transactions.

Because the execution step cannot be replayed once completed, and may be under the control of a BPM, it is very important that the change in state gets to the cache cluster. Synchronous writes can also be used (at a high performance cost) so that the put to the cache does not return until the data has been applied. If an executing node failed before the data was transferred, the locks would still be in place preventing readers from reading stale data, but that will not help the next step in the process.

XA transactions are needed because we want to keep the database and the cache in sync.

8.1.6.4 Immutable Data

8.Problem

The application uses data that once it comes into existence is immutable. Nothing is immutable forever. The key point is that it is immutable up until the time of the next software release.

Some examples are:

- application constants
- reference data - zip and post codes, countries etc.

If you analyse database traffic commonly used reference data turns out to be a big hitter.

As they are immutable they can only be appended or read, never updated.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

8.Solution

In concurrent programming, immutable data never needs further concurrency protection. So we simply want to use the fastest mode.

Here we would always use eventual consistency.

8.1.6.5 Financial Order Processing - write to cache as SOR

8.Problem

An order processing system sends a series of messages in a workflow, perhaps using Business Process Management software. The system involves multiple servers and the next step in the processing of an order may occur on any server. Let's say there are 50 steps in the process.

To avoid overloading a database the processing results at each step only written to a distributed cache. The next step could execute in a few ms to minutes depending on what other orders are going through and how busy the hardware is.

8.Solution

Use one of:

- strong consistency and local transactions (if changes are needed to be applied to multiple caches or entries).

  Because the execution step, once completed cannot be replayed, and may be under the control of a BPM, it is very important that the change in state gets to the cache cluster. Synchronous writes can also be used (at a high performance cost) so that the put to the cache does not return until the data has been applied. If an executing node failed before the data was transferred, the locks would still be in place preventing readers from reading stale data, but that will not help the next step in the process.
- CAS operations with eventual consistency. The CAS methods will not return until the data has been applied to the server, so it is not necessary to use synchronous writes.

In a 50 step process it is likely there are key milestones. Often it is desirable to record these in a database with the non-milestone steps recorded in the cache. For these key milestones use the "Financial Order Processing - write to cache and database" pattern.

8.1.6.6 E-commerce web app with Non-sticky sessions

Here a user makes reads and writes to a web application cluster. There are n servers where n > 1. The load balancer is non-sticky so any of the n servers can be hit on the next HTTP operation.

When a user submits using a HTML form, either a GET or POST is done based on the form action. And of course if it is an AJAx app then requests are being done with `XMLHttpRequest` and any HTTP request method can be sent. If POST (form and AJAX) or PUT (AJAX) is used no content is returned and a separate GET is required to refresh the view or AJAX app. The key point is that sending a change and getting a view may happen with one request or two. If it happens with two, then the same server might respond to the second request or not. The probability that the second server will be the same as the first is 1/n.

AJAX apps can further exacebate this situation. A page may make multiple request to fill different panels. This opens up the possibility of, within a single page, having data come from multiple servers. Any lack of consistency could be glaring indeed.

8.Solution

Use one of:

- strong consistency
- CAS

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

Other options can be added depending on what is needed for the request. e.g. XA if a database plus the cache is updated.

8.1.6.7 E-commerce web app with sticky sessions

8.Problem

Here a user makes reads and writes to a web application cluster. There are n servers where n > 1. The load balancer is sticky so any of the n servers can be hit on the next HTTP operation. There are different ways of configuring sticky sessions. The same server might be used for the length of a session, which is the standard meaning, or a browser's IP can permanently hash to a server. In any case each request is guaranteed to hit the same server.

8.Solution

The same server is always hit. The consistency mode depends on whether only the user making the changes needs to see them applied (read your writes, monotonic reads, monotonic writes) or whether they are mutating shared state like inventory where write - write conflicts might occur.

For mutating user only consistency use eventual consistency.

For multi-user shared state use strong consistency at a minimum plus further safety mechanisms depending on the type of mutation.

8.1.6.8 E-commerce Catalog

8.Problem

Catalogues display inventory. There are product details and pricing. There may be also be an inventory status of available or sold out.

Catalogue changes are usually made by one user or process (for example a daily update load from a supplier) and are usually do not have write-write conflicts. While the catalogue is often non-sticky, admin users are typically configured sticky.

There is often tolerance for the displayed catalogue to lag behind the change made. Users are usually less tolerance for a user following a click path to see inconsistencies.

8.Solution

The person making the changes can see a consistent view by virtue of the sticky session. So eventual consistency will often be enough.

For end users following a click path, they need a consistent view. However the network or Internet time plus their think time to move along the path adds up to seconds to minutes, while eventual consistency will propagate in the order of 2+ milliseconds. With eventual consistency it is very unlikely they will see inconsistency. The general recommendation is therefore to use eventual consistency.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

9 Cache Eviction Algorithms

9.1 Cache Eviction Algorithms

A cache eviction algorithm is a way of deciding which `Element` to evict when the cache is full.

In Ehcache the `MemoryStore` has a fixed limited size set by `maxElementsInMemory` (unless the maxElementsInMemory is 0, in which case the capacity is unlimited). When the store gets full, `elements` are evicted. The eviction algorithms in Ehcache determines which elements is evicted. The default is LRU.

What happens on eviction depends on the cache configuration. If a `DiskStore` is configured, the evicted element will overflow to disk, otherwise it will be removed.

The `DiskStore` size by default is unbounded. But a maximum size can be set using the `maxElementsOnDisk` cache attribute. If the `DiskStore` is full, then adding an element will cause one to be evicted. The `DiskStore` eviction algorithm is not configurable. It uses LFU.

9.1.1 Provided `MemoryStore` Eviction Algorithms

The idea here is, given a limit on the number of items to cache, how to choose the thing to evict that gives the *best* result.

In 1966 Laszlo Belady showed that the most efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This it a theoretical result that is unimplementable without domain knowledge. The Least Recently Used ("LRU") algorithm is often used as a proxy. It works pretty well because of the locality of reference phenonemon. As a result, LRU is the default eviction algorithm in Ehcache, as it is in most caches.

Ehcache users may sometimes have a good domain knowledge. Accordingly, Ehcache provides three eviction algorithms to choose from for the `MemoryStore`.

9.1.1.1 Less Recently Used (LRU)

This is the default.

The eldest element, is the Least Recently Used (LRU). The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.

It takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

If probabilistic eviction does not suit your application, a true Least Recently Used deterministic algorithm is available by setting `java -Dnet.sf.ehcache.use.classic.lru=true`.

9.1.1.2 Less Frequently Used (LFU)

For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached) the element with least number of hits, the Less Frequently Used element, is evicted.

If cache element use follows a pareto distribution, this algorithm may give better results than LRU.

LFU is an algorithm unique to Ehcache. It takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

9.1.1.3 First In First Out (FIFO)

Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

This algorithm is used if the use of an element makes it less likely to be used in the future. An example here would be an authentication cache.

It takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

9.1.2 Plugging in your own Eviction Algorithm

Ehcache 1.6 and higher allows you to plugin in your own eviction algorithm. You can utilise any Element metadata which makes possible some very interesting approaches. For example, evict an Element if it has been hit more than 10 times.

```
/**
 * Sets the eviction policy strategy. The Cache will use a policy at startup. There
 * are three policies which can be configured: LRU, LFU and FIFO. However many other
 * policies are possible. That the policy has access to the whole element enables polic
 * based on the key, value, metadata, statistics, or a combination of any of the above.
 * It is safe to change the policy of a store at any time. The new policy takes effect
 * immediately.
 *
 * @param policy the new policy
 */
public void setMemoryStoreEvictionPolicy(Policy policy) {
    memoryStore.setEvictionPolicy(policy);
}
```

A Policy must implement the following interface:

```
public interface Policy {
 /**
  * @return the name of the Policy. Inbuilt examples are LRU, LFU and FIFO.
  */
 String getName();
 /**
  * Finds the best eviction candidate based on the sampled elements. What distinguis
  * this approach from the classic data structures approach is that an Element conta
  * metadata (e.g. usage statistics) which can be used for making policy decisions,
  * while generic data structures do not. It is expected that implementations will t
  * advantage of that metadata.
  *
  * @param sampledElements this should be a random subset of the population
  * @param justAdded       we probably never want to select the element just added.
  * It is provided so that it can be ignored if selected. May be null.
  * @return the selected Element
  */
 Element selectedBasedOnPolicy(Element[] sampledElements, Element justAdded);
 /**
  * Compares the desirableness for eviction of two elements
  *
  * @param element1 the element to compare against
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
     * @param element2 the element to compare
     * @return true if the second element is preferable for eviction to the first eleme
     * under ths policy
     */
    boolean compare(Element element1, Element element2);
}
```

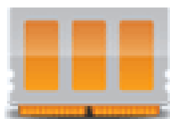
9.1.3 `DiskStore` Eviction Algorithms

The `DiskStore` uses the Less Frequently Used algorithm to evict an element when it is full.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

10 Big Memory:Off-Heap Store

10.1 BigMemory: Off-heap Store



Terracotta BigMemory is an add-on to Enterprise Ehcache that permits caches to use an additional type of memory store outside the object heap.

This off-heap store, which is not subject to Java GC, is 100 times faster than the DiskStore and allows very large caches to be created (we have tested this up to 350GB).

Because off-heap data is stored in bytes, there are two implications:

- Only Serializable cache keys and values can be placed in the store, similar to DiskStore.
- Serialization and deserialization take place on putting and getting from the store. This means that the off-heap store is slower in an absolute sense (around 10 times slower than the MemoryStore), but this theoretical difference disappears due to two effects:
    - the MemoryStore holds the hottest subset of data from the off-heap store, already in deserialized form
    - when the GC involved with larger heaps is taken into account, the off-heap store is faster on average

10.1.1 Configuration

10.1.1.1 Configuring caches to overflow to off-heap.

Configuring a cache to use an off-heap store can be done either through XML or programmatically.

10.Declarative Configuration

The following XML configuration creates an off-heap cache with an in-heap store (maxElementsInMemory) of 10,000 elements which overflow to a 1-gigabyte off-heap area.

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache updateCheck="false" monitoring="off"
         dynamicConfig="false">
    <defaultCache maxElementsInMemory="10000"
                  eternal="true"
                  memoryStoreEvictionPolicy="LRU"
                  statistics="false" />
    <cache name="sample-offheap-cache"
           maxElementsInMemory="10000"
           eternal="true"
           memoryStoreEvictionPolicy="LRU"
           overflowToOffHeap="true"
           maxMemoryOffHeap="1G"/>
</ehcache>
```

The configuration options are:

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

10.overflowToOffHeap

Values may be true or false.

When set to true, enables the cache to utilize off-heap memory storage to improve performance. Off-heap memory is not subject to Java GC cycles and has a size limit set by the Java property MaxDirectMemorySize. The default value is false.

10.maxMemoryOffHeap

Sets the amount of off-heap memory available to the cache. This attribute's values are given as *number*k|K|m|M|g|G|t|T for kilobytes (k|K), megabytes (m|M), gigabytes (g|G), or terabytes (t|T). For example, maxMemoryOffHeap="2g" allots 2 gigabytes to off-heap memory. In effect only if overflowToOffHeap is true.

The minimum amount that can be allocated is 128MB. There is no maximum.

Note that it is recommended to set maxElementsInMemory to at least 100 elements when using an off-heap store, otherwise performance will be seriously degraded, and a warning will be logged.

10.Programmatic Configuration

The equivalent cache can be created using the following programmatic configuration:

```
public Cache createOffHeapCache() {
  CacheConfiguration config = new CacheConfiguration("sample-offheap-
cache", 10000)
    .overflowToOffHeap(true).maxMemoryOffHeap("1G");
  Cache cache = new Cache(config);
  manager.addCache(cache);
  return cache;
}
```

10.1.1.2 Add The License

The Ehcache Enterprise trial download (available here - http://www.terracotta.org/bigmemory) comes with a trial license key which must be added to activate the off-heap store.

It can be added to the classpath or via a system property.

10.Configuring the License in the Classpath

Add the `terracotta-license.key` to the root of your classpath, which is also where you add ehcache.xml. It will be automatically found.

10.Configuring the License as a Java system property

Add a `com.tc.productkey.path=/path/to/key` system property which points to the key location.

e.g.

```
java -Dcom.tc.productkey.path=/path/to/key
```

10.1.1.3 Allocating Direct Memory in the JVM

In order to use these configurations you must then use the ehcache-core-ee jar on your classpath, and modify your JVM command-line to increase the amount of direct memory allowed by the JVM. You must allocate at least 32MB more to direct memory than the total off-heap memory allocated to caches.

e.g. to allocate 2GB of memory in the JVM.

```
java -XX:MaxDirectMemorySize=2G ..."
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

10.1.2 Advanced Configuration Options

There are some rarer configuration options which can be used for fine grained control

10.1.2.1 -XX:+UseLargePages

This is a JVM flag which is meant to improve performance of memory-hungry applications. In testing, this option gives a 5% speed improvement with a 1Gb off-heap cache.

See http://andrigoss.blogspot.com/2008/02/jvm-performance-tuning.html for a discussion.

10.1.2.2 Increasing the maximum serialized size of an Element that can be stored in the OffHeapStore

Firstly, the MemoryStore and the DiskStore do not have any limits.

By default, the OffHeapStore has a 4MB limit for classes with high quality hashcodes, and 256KB for those with pathologically bad hashcodes. The built-in classes such as the `java.lang.Number` subclasses such as Long, Integer etc and and `String` have high quality hashcodes.

You can increase the size by setting a system property net.sf.ehcache.offheap.cache_name.config.idealMaxSegmentSize to the size you require.

e.g. net.sf.ehcache.offheap.com.company.domain.State.config.idealMaxSegmentSize=30M

10.1.2.3 Avoiding OS Swapping

Operating systems use swap partitions for virtual memory and are free to move less frequently used pages of memory to the swap partition. This is generally not what you want when using the OffHeapStore, as the time it takes to swap a page back in when demanded will add to cache latency.

It is recommended that you minimise swap use for maximum performance.

On Linux, you can set `/proc/sys/vm/swappiness` to reduce the risk of memory pages being swapped out. See http://lwn.net/Articles/83588/ for details of tuning this parameter. Note that there are bugs in this which were fixed in kernel 2.6.30 and higher.

Another option is to configure HugePages. See http://unixfoo.blogspot.com/2007/10/hugepages.html

This kind of problem bit us several times in the past in Linux. Although there's a swappiness kernel parameter that can be set to zero, it is usually not enough to avoid swapping altogether. The only surefire way to avoid any kind of swapping is either (a) disabling the swap partition, with the undesirable consequences which that may bring, or (b) using HugePages, which are always mapped to physical memory and cannot be swapped out to disk.

10.1.2.4 -XX:UseCompressedOops

This setting applies to the HotSpot JVM. It's use should be considered to make the most efficient use of memory in 64 bit mode. See http://wikis.sun.com/display/HotSpotInternals/CompressedOops for details.

10.1.2.5 Controlling Overallocation of Memory to the OffHeapStore

If the memory use is dramatically overallocated, you may end up trying to use more than the physical and even virtual memory available on your OS. We attempt to detect this situation. If it takes more than 3 seconds to allocate a 1GB chunk of memory we will log an error message and call `System.exit(1)` to protect the stability of your OS.

If you wish to force Ehcache to wait set the system property `net.sf.ehcache.offheap.DoNotHaltOnCriticalAllocationDelay` to `true`.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

10.1.3 Sample application

The easiest way to get started is to play with a simple sample app.

Download here a simple Maven-based application that uses the ehcache off-heap functionality.

Note: You will need to get a license key and install it as discussed above to run this.

10.1.4 Performance Comparisons

Checkout https://svn.terracotta.org/repo/forge/offHeap-test/ terracotta_community_login a Maven-based performance comparisons between different store configurations.

Note: You will need to get a demo license key and install it as discussed above to run the test.

Here are some charts from tests we have run on the release candidate of BigMemory.

The test machine was a Cisco UCS box running with Intel(R) Xeon(R) Processors. It had 6 2.93Ghz Xeon(R) cpus for a total of 24 cores, with 128GB of RAM, running RHEL5.1 with Sun JDK 1.6.0_21 in 64 bit mode.

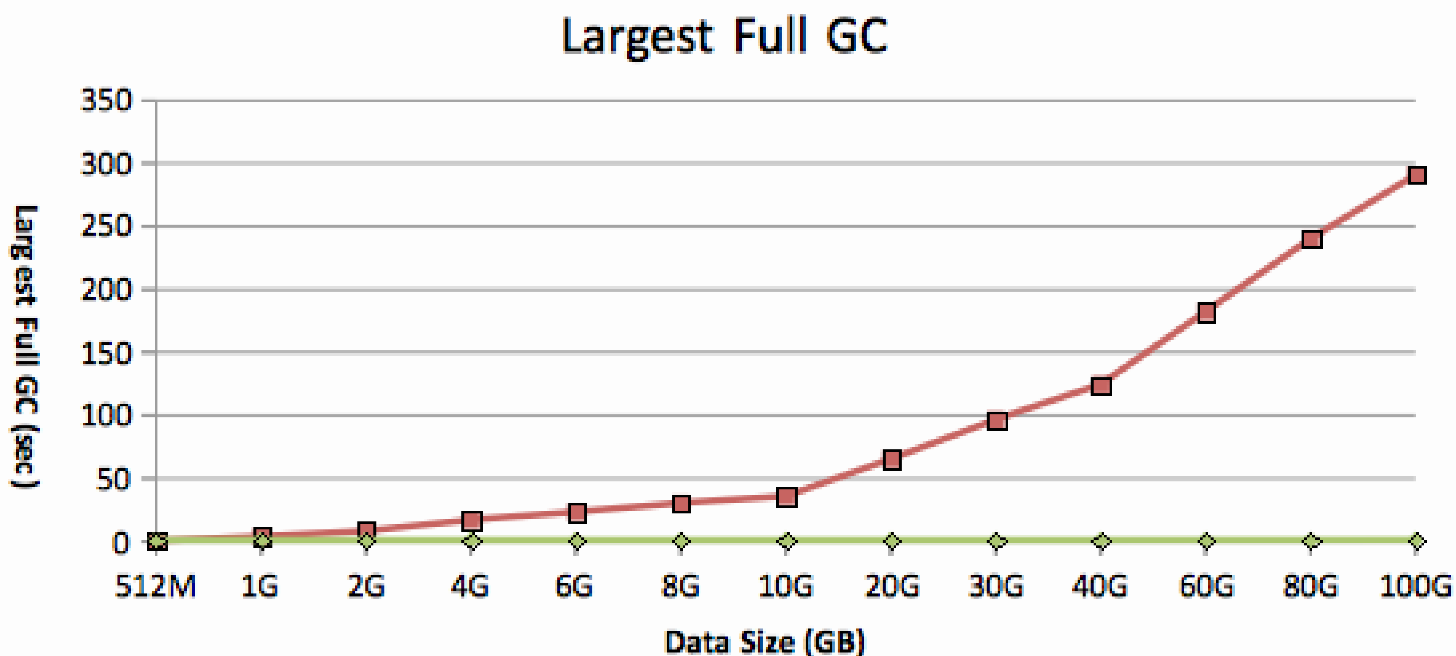
We used 50 threads doing an even mix of reads and writes with 1KB elements. We used the default garbage collection settings.

The tests all go through a load/warmup phase then start a performance run. You can use the tests in your own environments and extend them to cover different read/write ratios, data sizes, -Xmx settings and hot sets. The full suite, which is done with `run.sh` takes 4-5 hours to complete.

The following charts show the most common caching use case. The read/write ratio is 90% reads and 10% writes. The hot set is that 90% of the time `cache.get()` will access 10% of the key set. This is representative of the the familiar Pareto distribution that is very commonly observed.

There are of course many other caching use cases. Further performance results are covered on the Further Performance Analysis page.

10.1.4.1 Largest Full GC

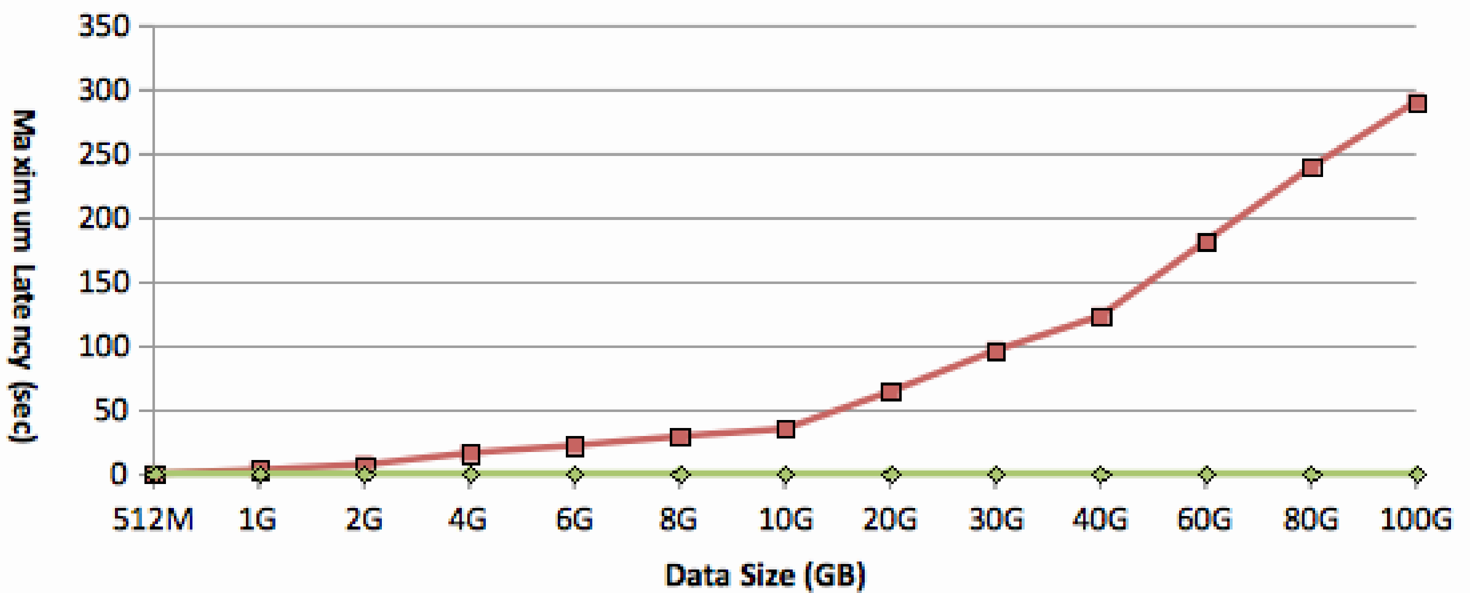


©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

This chart shows the largest observed full GC duration which occurred during the performance run. Most non-batch applications have maximum response time SLAs. As can be seen in the chart, as data sizes grow the full GC gets worse and worse for cache held on heap, whereas off-heap remains a low constant.

The off-heap store will therefore enable applications with maximum response time SLAs to reliably meet those SLAs.

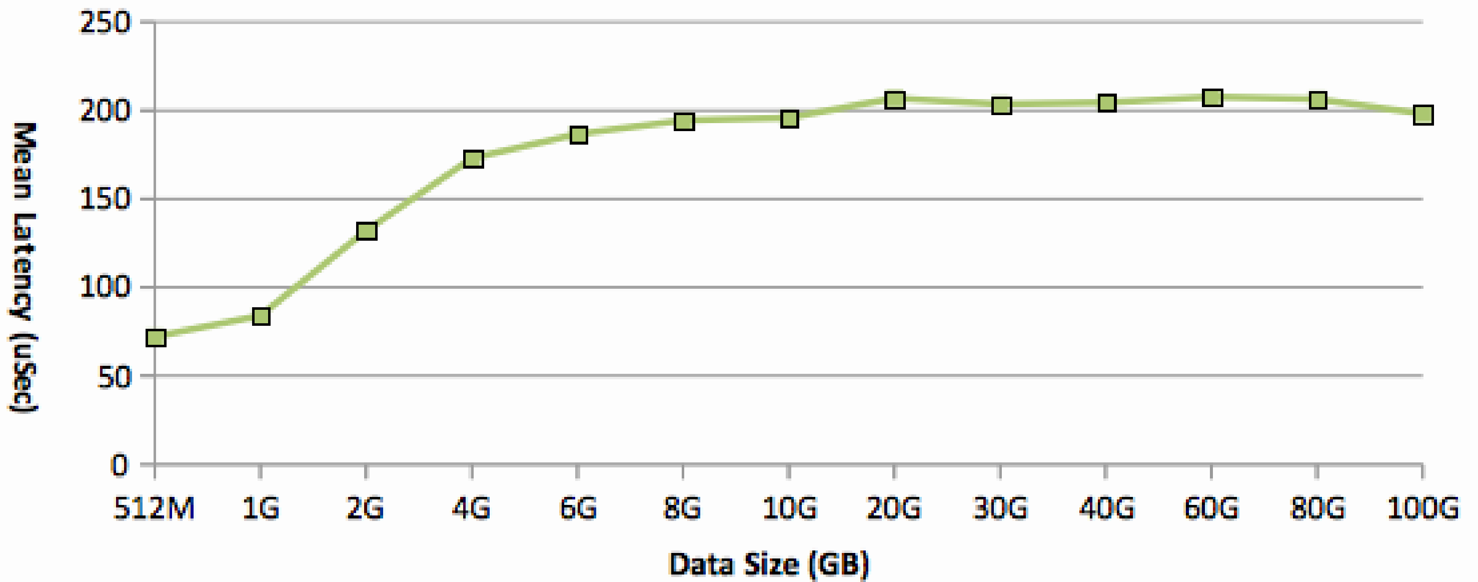
10.1.4.2 Latency



This chart shows the maximum observed latency while perfomring either a `cache.put()` or a `cache.get()`. It is very similar to the Full GC chart because the reason the on-heap latencies blow out is full GCs, where all threads in the test app get frozen.

Once again the off-heap store can be observed to have a flat, low maximum latency, because any full GCs are tiny, and the cache has excellent concurrency properties.

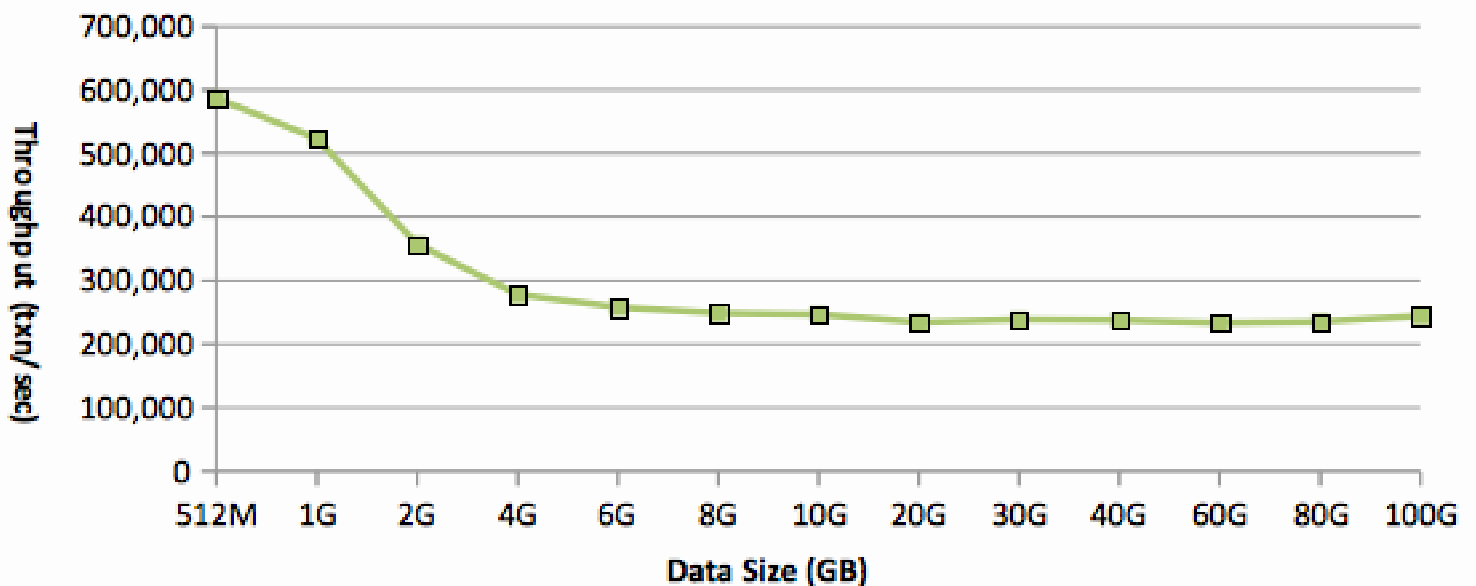
©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.



This chart shows the off-heap mean latency in microseconds. It can be observed to be flat from 2GB up to 40GB. Further in-house testing shows that the it remains flat up to the limits we have tested to which is currently 350GB.

Lower latencies are observed at smaller data set sizes because we use a `maxElementsInMemory` setting which approximates to 200MB of on-heap store. On-heap, excluding GC effects is faster than off-heap because there is no deserialization on gets. At lower data sizes there is a higher probability that the small on-heap store will be hit, which is reflected in the lower average latencies.

10.1.4.3 Throughput



©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

This chart shows the cache operations per second achieved with off-heap. It is the inverse of average latency and shows much the same thing. Once the effect of the on-heap store becomes marginal, throughput remains constant, regardless of cache size. Once again we have verified this constancy up to 350GB.

Note that much larger throughputs than those shown in this chart are achievable. Throughput is affected by:

- the number of threads (more threads -> more throughput)
- the read/write ratio (reads are slightly faster)
- data payload per operation (more data implies a lower throughput in tps but similar in bytes)
- cpu cores available and their speed (our testing shows that the cpu is always the limiting factor with enough threads. In other words cache throughput can be increased by adding threads until all cores are utilised and then adding cpu cores - an ideal situation where the software can use as much hardware as you can throw at it.)

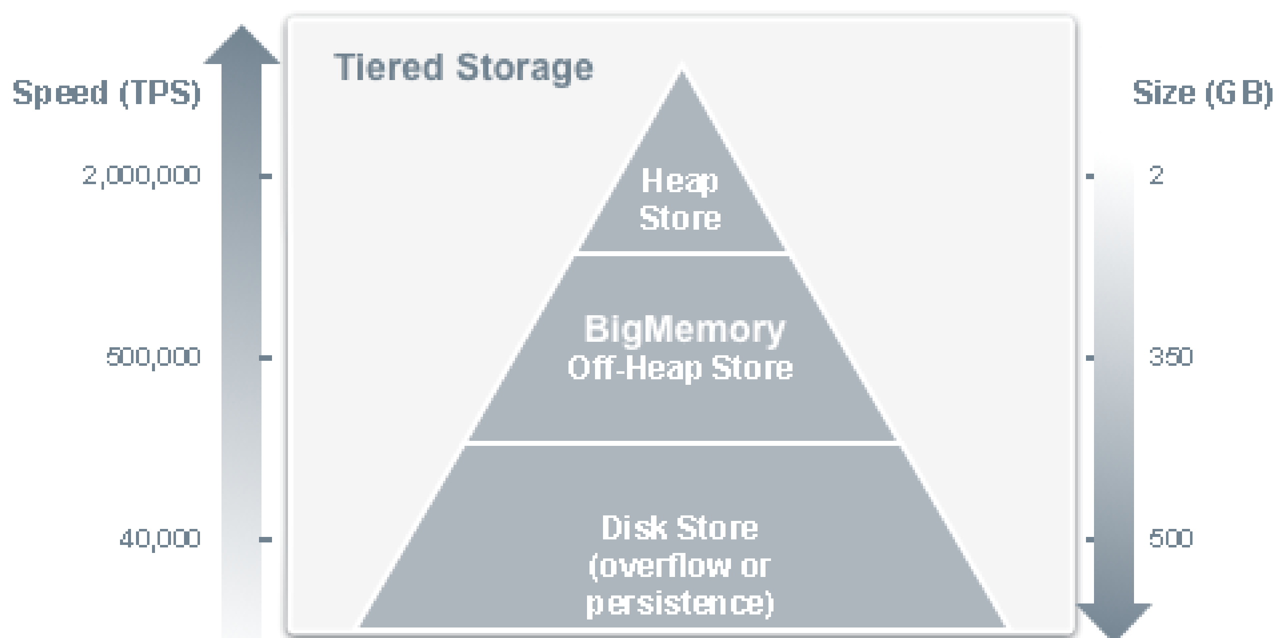
10.1.5 Storage

10.1.5.1 Storage Hierarchy

With the OffHeapStore, Ehcache Enterprise has three stores:

- MemoryStore - very fast storage of Objects on heap. Limited by the size of heap you can comfortably garbage collect
- OffHeapStore - fast (one order of magnitude slower than MemoryStore) storage of Serialized objects off heap. Limited only by the amount of RAM on your hardware and address space. You need a 64 bit OS to address higher than 2-4GB.
- DiskStore - speedy storage on disk. It is two orders of magnitude slower than the OffHeapStore but still much faster than a database or a distributed cache

  The relationship between speed and size for each store is illustrated below:



©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

10.1.5.2 Memory Use in each Store

As a performance optimisation, and because storage gets much cheaper as you drop down through the hierarchy, we write each put to as many stores as are configured. So, if all three are configured, the Element may be present in MemoryStore, OffHeapStore and DiskStore.

The result is that each store consumes storage for itself and the other stores higher up the hierarchy. So, if the MemoryStore has 1,000,000 Elements which consume 2Gb, and the OffHeapStore is configured for 8GB, then 2GB of that will be duplicate of what is in the MemoryStore. And the 8GB will also be duplicated on the DiskStore plus the DiskStore will have what cannot fit in any of the other stores.

This needs to be taken into account when configuring the OffHeap and Disk stores.

It has the great benefit, which pays for the duplication, of not requiring copy on eviction. On eviction from a store, an Element can simply be removed. It is already in the next store down.

One further twist: the MemoryStore is only populated on a read. Puts go to the OffHeapStore and then when read, are held in the MemoryStore. The MemoryStore thus holds hot items of the OffHeapStore. This will result in a difference in what can be expected to be in the MemoryStore between this implementation and the open source one. A "usage" for the purposes of the eviction algorithms is either a put or a get. As only gets are counted in this implementation, some differences will be observed.

10.1.6 Handling JVM startup and shutdown

So you can have a huge in-process cache. But this is not a distributed cache, so when you shut down you will lose what is in the cache. And when you start up, how long will it take to load the cache?

In caches up to a GB or two, these issues are not hugely problematic. You can often pre-load the cache on start-up before you bring the application online. Provided this only takes a few minutes, there is minimal operations impact.

But when we go to tens of GBs, these startup times are O(n), and what took 2 minutes now takes 20 minutes.

To solve this problem, we provide a new implementation of Ehcache's DiskStore, available in the enterprise version.

You simply mark the cache `diskPersistent=true` as you normally would for a disk persistent cache.

It works as follows:

- on startup, which is immediate, the cache will get elements from disk and gradually fill the MemoryStore and the OffHeapStore.
- when running elements are written to the OffHeapStore, they are already serialized. We write these to the DiskStore asynchronously in a write-behind pattern. Tests show they can be written at a rate of 20MB/s on server-class machines with fast disks. If writes get behind, they will back up and once they reach the `diskSpoolBufferSizeMB` cache puts will be slowed while the DiskStore writer catches up. By default this buffer is 30MB but can be increased through configuration.
- When the Cache is disposed, only a final sync is required to shut the DiskStore down.

10.1.7 Using OffHeapStore with 32 bit JVMs

On a 32 bit operating system, Java will always start with a 32 bit data model. On 64 bit OSs, it will default to 64 bit, but can be forced into 32 bit mode with the Java command-line option `-d32`. The problem is that this limits the size of the process to 4GB. Because garbage collection problems are

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

generally manageable up to this size, there is not much point in using the OffHeapStore, as it will simply be slower.

If you are suffering GC issues with a 32 bit JVM, then OffHeapStore can help. There are a few points to keep in mind.

- Everything has to fit in 4GB of addressable space. If you allocate 2GB of heap (with `-Xmx2g`) then you have at most 2GB left for your off-heap caches.
- Don't expect to be able to use all of the 4GB of addressable space for yourself. The JVM process requires some of it for its code and shared libraries plus any extra Operating System overhead.
- If you allocate a 3GB heap with -Xmx as well as 2047MB of off-heap memory the virtual machine certainly won't complain at startup but when it's time to grow the heap you will get an OutOfMemoryError.
- If you use both -Xms3G and -Xmx3G with 2047MB of off-heap memory the virtual machine will start but then complain as soon as the OffHeapStore tries to allocate the off-heap buffers.
- Some APIs, such as java.util.zip.ZipFile on Sun 1.5 JVMs, may *mmap* files in memory. This will also use up process space and may trigger an OutOfMemoryError.

  For these reasons we issue a warning to the log when OffHeapStore is used with 32 bit JVMs.

10.1.8 Slow off-heap allocation

Off-heap allocation time is measured to avoid allocating buffers too large to fit in memory. If it takes more than 1.5s to allocate a buffer a warning is issued as it could very well be that the OS has started paging to disk. If it takes more than 15s then the JVM is halted (with `System.exit()`, but different things are tried when the Security Manager prevents this) unless the `net.sf.ehcache.offheap.DoNotHaltOnCriticalAllocationDelay` system property is set to true.

This mechanism was built in because allocating an off-heap buffer too large to fit in RAM can quickly and easily deplete critical system resources like RAM and swap space and crash the host operating system. Linux and Mac OS X will crash in these circumstances.

10.1.9 Reducing Cache Misses

While the MemoryStore holds a hotset (a subset) of the entire data set, the off-heap store should be large enough to hold the entire data set. The frequency of cache misses begins to rise when the data is too large to fit into off-heap memory, forcing gets to fetch data from the DiskStore. More misses in turn raise latency and lower performance.

For example, tests with a 4GB data set and a 5GB off-heap store recorded no misses. With the off-heap store reduced to 4GB, 1.7 percent of cache operations resulted in misses. With the off-heap store at 3GB, misses reached 15 percent.

10.1.10 FAQ

10.1.10.1 The DiskStore Access stripes configuration no longer has effect. Why?

This has been reimplemented for Ehcache Enterprise and will get added back into the core in the future.

10.1.10.2 What Eviction Algorithms are supported?

The pluggable MemoryStore eviction algorithms work as normal. The OffHeapStore and DiskStore use a Clock Cache, a standard paging algorithm which is an approximation of LRU.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

10.1.10.3 Why do I see performance slow down and speed up in a cyclical pattern when I am filling a cache?

This is due to repartitioning in the OffHeapStore which is normal. Once the cache is fully filled the performance slow-downs cease.

10.1.10.4 What is the maximum serialized size of an object when using OffHeapStore?

Firstly, the MemoryStore and the DiskStore do not have any limits.

By default, the OffHeapStore has a 4MB limit for classes with high quality hashcodes, and 256KB for those with pathologically bad hashcodes. The built-in classes such as the `java.lang.Number` subclasses such as Long, Integer etc and and `String` have high quality hashcodes.

You can increase the size by setting the system property net.sf.ehcache.offheap.cache_name.config.idealMaxSegmentSize to the size you require.

e.g. net.sf.ehcache.offheap.com.company.domain.State.config.idealMaxSegmentSize=30M

10.1.10.5 Why is my application startup slower?

On startup the CacheManager will calculate the amount of off-heap storage required for all caches using off-heap stores. The memory will be allocated from the OS and zeroed out by Java. The time taken will depend on the OS. A server-class machine running Linux will take approximately half a second per GB.

We print out log messages for each 10% allocated, and also report the total time taken.

This time is incurred only once at startup. The pre-allocation of memory from the OS is one of the reasons that runtime performance is so fast.

10.1.10.6 How can I do Maven testing with BigMemory?

Maven starts java for you. You cannot add the required `-XX` switch in as a `mvn` argument.

Maven provides you with a MAVEN_OPTS environment variable you can use for this on Unix systems.

e.g. to specify 1GB of MaxDirectMemorySize and then to run jetty:

```
export MAVEN_OPTS=-XX:MaxDirectMemorySize=1G
mvn jetty:run-war
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

11 JDBC Caching

11.1 JDBC Caching

Ehcache can easily be combined with your existing JDBC code. Whether you access JDBC directly, or have a DAO/DAL layer, Ehcache can be combined with your existing data access pattern to speed up frequently accessed data to reduce page load times, improve performance, and reduce load from your database.

This document discusses how to add caching to a JDBC application using the commonly used DAO/ DAL layer patterns:

11.1.1 Adding JDBC caching to a DAO/DAL layer

If your application already has a DAO/DAL layer, this is a natural place to add caching. To add caching, follow these steps:

- identify methods which can be cached
- instantiate a cache and add a member variable to your DAO to hold a reference to it
- Put and get values from the cache

11.1.1.1 Identifying methods which can be cached

Normally, you will want to cache the following kinds of method calls:

- Any method which retrieves entities by an Id
- Any queries which can be tolerate some inconsistent or out of date data

Example methods that are commonly cacheable:

```
public V getById(final K id);
public Collection<V> findXXX(...);
```

11.1.1.2 Instantiate a cache and add a member variable

Your DAO is probably already being managed by Spring or Guice, so simply add a setter method to your DAO layer such as `setCache(Cache cache)`. Configure the cache as a bean in your Spring or Guice config, and then use the the frameworks injection methodology to inject an instance of the cache.

If you are not using a DI framework such as Spring or Guice, then you will need to instantiate the cache during the bootstrap of your application. As your DAO layer is being instantiated, pass the cache instance to it.

11.1.1.3 Put and get values from the cache

Now that your DAO layer has a cache reference, you can start to use it. You will want to consider using the cache using one of two standard cache access patterns:

- cache-aside
- cache-as-sor

You can read more about these in the Concepts cache-aside and Concepts cache-as-sor sections.

11.1.2 Putting it all together - an example

Here is some example code that demonstrates a DAO based cache using a cache aside methodology wiring it together with Spring..

This code implements a PetDao modeled after the Spring Framework PetClinic sample application.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

It implements a standard pattern of creating an abstract GenericDao implementation which all Dao implementations will extend.

It also uses Spring's SimpleJdbcTemplate to make the job of accessing the database easier.

Finally, to make Ehcache easier to work with in Spring, it implements a wrapper that holds the cache name.

11.1.2.1 The example files

The following are relevant snippets from the example files. A full project will be available shortly.

11.CacheWrapper.java

Simple get/put wrapper interface.

```
public interface CacheWrapper<K, V>
{
  void put(K key, V value);
  V get(K key);
}
```

11.EhcacheWrapper.java

The wrapper implementation. Holds the name so caches can be named.

```
public class EhCacheWrapper<K, V> implements CacheWrapper<K, V>
{
    private final String cacheName;
    private final CacheManager cacheManager;
    public EhCacheWrapper(final String cacheName, final CacheManager cacheManager)
    {
        this.cacheName = cacheName;
        this.cacheManager = cacheManager;
    }
    public void put(final K key, final V value)
    {
        getCache().put(new Element(key, value));
    }
    public V get(final K key, CacheEntryAdapter<V> adapter)
    {
        Element element = getCache().get(key);
        if (element != null) {
            return (V) element.getValue();
        }
        return null;
    }
    public Ehcache getCache()
    {
        return cacheManager.getEhcache(cacheName);
    }
}
```

11.GenericDao.java

The Generic Dao. It implements most of the work.

```
public abstract class GenericDao<K, V extends BaseEntity> implements Dao<K, V>
{
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
    protected DataSource datasource;
    protected SimpleJdbcTemplate jdbcTemplate;
    /* Here is the cache reference */
    protected CacheWrapper<K, V> cache;
    public void setJdbcTemplate(final SimpleJdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    public void setDatasource(final DataSource datasource) {
       this.datasource = datasource;
    }
    public void setCache(final CacheWrapper<K, V> cache) {
        this.cache = cache;
    }
    /* the cacheable method */
    public V getById(final K id) {
        V value;
        if ((value = cache.get(id)) == null) {
          value = this.jdbcTemplate.queryForObject(findById, mapper, id);
          if (value != null) {
            cache.put(id, value);
          }
        }
        return value;
    }
    /** rest of GenericDao implementation here **/
    /** ... **/
    /** ... **/
    /** ... **/
}
```

11.PetDaoImpl.java

The Pet Dao implementation, really it doesn't need to do anything unless specific methods not available via GenericDao are cacheable.

```
public class PetDaoImpl extends GenericDao<Integer, Pet> implements PetDao
{
    /** ... **/
}
```

We need to configure the JdbcTemplate, Datasource, CacheManager, PetDao, and the Pet cache using the spring configuration file.

11.application.xml

The Spring configuration file.

```
<!-- datasource and friends -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.FasterLazyConnectionDa
    <property name="targetDataSource" ref="dataSourceTarget"/>
</bean>
<bean id="dataSourceTarget" class="com.mchange.v2.c3p0.ComboPooledDataSource"
        destroy-method="close">
    <property name="user" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
    <property name="driverClass" value="${jdbc.driverClassName}"/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
    <property name="initialPoolSize" value="50"/>
    <property name="maxPoolSize" value="300"/>
    <property name="minPoolSize" value="30"/>
    <property name="acquireIncrement" value="2"/>
    <property name="acquireRetryAttempts" value="0"/>
</bean>
<!-- jdbctemplate -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate"
    <constructor-arg ref="dataSource"/>
</bean>
<!-- the cache manager -->
<bean id="cacheManager" class="EhCacheManagerFactoryBean">
    <property name="configLocation" value="classpath:${ehcache.config}"/>
</bean>
<!-- the pet cache to be injected into the pet dao -->
<bean name="petCache" class="EhCacheWrapper">
    <constructor-arg value="pets"/>
    <constructor-arg ref="cacheManager"/>
</bean>
<!-- the pet dao -->
<bean id="petDao" class="PetDaoImpl">
    <property name="cache" ref="petCache"/>
    <property name="jdbcTemplate" ref="jdbcTemplate"/>
    <property name="datasource" ref="dataSource"/>
</bean>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

12 Spring Caching with Ehcache

12.1 Using Spring and Ehcache

Ehcache has had excellent Spring integration for many years. More recently there are two new ways of using Ehcache with Spring

12.1.1 Spring 3.1

Spring Framework 3.1 added a new generic cache abstraction for transparently applying caching to Spring applications.

It adds caching support for classes and methods using two annotations:

12.1.1.1 @Cacheable

Cache a method call.

In the following example, the value is the return type, a Manual. The key is extracted from the ISBN argument using the id.

```
@Cacheable(value="manual", key="#isbn.id")
public Manual findManual(ISBN isbn, boolean checkWarehouse)
```

12.1.1.2 @CacheEvict

Clears the cache when called.

```
@CacheEvict(value = "manuals", allEntries=true)
public void loadManuals(InputStream batch)
```

Spring 3.1 includes an Ehcache implementation. See the Spring 3.1 JavaDoc.

It also does much more with SpEL expressions. See for an excellent blog post covering this material in more detail.

12.1.2 Spring 2.5 - 3.1: Ehcache Annotations For Spring

This open source, led by Eric Dalquist, predates the Spring 3.1 project. You can use it with earlier versions of Spring or you can use it with 3.1.

12.1.2.1 @Cacheable

As with Spring 3.1 it uses an @Cacheable annotation to cache a method. In this example calls to findMessage are stored in a cache named "messageCache". The values are of type `Message`. The id for each entry is the `id` argument given.

```
@Cacheable(cacheName = "messageCache")
public Message findMessage(long id)
```

12.1.2.2 @TriggersRemove

And for cache invalidation, there is the @TriggersRemove annotation.

In this example, `cache.removeAll()` is called after the method is invoked.

```
@TriggersRemove(cacheName = "messagesCache",
    when = When.AFTER_METHOD_INVOCATION, removeAll = true)
public void addMessage(Message message)
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

See for a blog post explaining it's user and providing further links.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

13 Code Samples

13.1 Recipes and Code Samples

The Recipes and Code Samples page contains recipes - short concise examples for specific use cases - and a set of code samples that will help you get started with Ehcache.

If you have a suggestion or an idea for a recipe or more code samples, please tell us about it using the mailing list or forums.

13.2 Recipes

Recipe	Description
Web Page and Fragment Caching	How to use inluded Servlet Filters to Cache Web Page and Web Page Fragments
Configure a Grails App for Clustering	How to configure a Grails Application for clustered Hibernate 2nd Level Cache
Data Freshness and Expiration	How to maintain cache "freshness" by configuring TTL and data expiration properly
Enable Terracotta Programmatically	How to enable Terracotta support for Ehcache programmatically
WAN Replication	3 Strategies for configuring WAN replication
Caching Empty Values	Why caching empty values can be desirable to deflect load from the database
Database Read Overload	When many readers simultaneously request the same data element it is called the "Thundering Herd" problem. How to prevent it in a single jvm or clustered configuration
Database Write Overload	Writing to the Database is a Bottleneck. Configure write-behind to offload database writes.
Caching methods with Spring Annotations	Adding caching to methods using Ehcache Annotations for Spring project
Cache Wrapper	A simple class to make accessing Ehcache easier for simple use cases

13.3 Code Samples

13.3.1 Using the CacheManager

All usages of Ehcache start with the creation of a CacheManager.

13.3.1.1 Singleton versus Instance

As of ehcache-1.2, Ehcache CacheManagers can be created as either singletons (use the create factory method) or instances (use new).

Create a singleton CacheManager using defaults, then list caches.

```
CacheManager.create();
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
String[] cacheNames = CacheManager.getInstance().getCacheNames();
```

Create a CacheManager instance using defaults, then list caches.

```
CacheManager manager = new CacheManager();
String[] cacheNames = manager.getCacheNames();
```

Create two CacheManagers, each with a different configuration, and list the caches in each.

```
CacheManager manager1 = new CacheManager("src/config/ehcache1.xml");
CacheManager manager2 = new CacheManager("src/config/ehcache2.xml");
String[] cacheNamesForManager1 = manager1.getCacheNames();
String[] cacheNamesForManager2 = manager2.getCacheNames();
```

13.3.1.2 Ways of loading Cache Configuration

When the CacheManager is created it creates caches found in the configuration.

Create a CacheManager using defaults. Ehcache will look for ehcache.xml in the classpath.

```
CacheManager manager = new CacheManager();
```

Create a CacheManager specifying the path of a configuration file.

```
CacheManager manager = new CacheManager("src/config/ehcache.xml");
```

Create a CacheManager from a configuration resource in the classpath.

```
URL url = getClass().getResource("/anotherconfigurationname.xml");
CacheManager manager = new CacheManager(url);
```

Create a CacheManager from a configuration in an InputStream.

```
InputStream fis = new FileInputStream(new File("src/config/
ehcache.xml").getAbsolutePath());
try {
    CacheManager manager = new CacheManager(fis);
} finally {
    fis.close();
}
```

13.3.1.3 Adding and Removing Caches Programmatically

You are not just stuck with the caches that were placed in the configuration. You can create and remove them programmatically.

Add a cache using defaults, then use it. The following example creates a cache called *testCache*, which will be configured using defaultCache from the configuration.

```
CacheManager singletonManager = CacheManager.create();
singletonManager.addCache("testCache");
Cache test = singletonManager.getCache("testCache");
```

Create a Cache and add it to the CacheManager, then use it. Note that Caches are not usable until they have been added to a CacheManager.

```
CacheManager singletonManager = CacheManager.create();
Cache memoryOnlyCache = new Cache("testCache", 5000, false, false, 5, 2);
manager.addCache(memoryOnlyCache);
Cache test = singletonManager.getCache("testCache");
```

See the cache constructor for the full parameters for a new Cache:

Remove cache called sampleCache1

```
CacheManager singletonManager = CacheManager.create();
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
singletonManager.removeCache("sampleCache1");
```

13.3.1.4 Shutdown the CacheManager

Ehcache should be shutdown after use. It does have a shutdown hook, but it is best practice to shut it down in your code.

Shutdown the singleton CacheManager

```
CacheManager.getInstance().shutdown();
```

Shutdown a CacheManager instance, assuming you have a reference to the CacheManager called *manager*

```
manager.shutdown();
```

See the CacheManagerTest for more examples.

13.3.2 Creating Caches Programmatically

13.3.2.1 Creating a new cache from defaults

A new cache with a given name can be created from defaults very simply:

```
manager.addCache(cache name);
```

13.3.2.2 Creating a new cache with custom parameters

The configuration for a Cache can be specified programmatically as an argument to the Cache constructor:

```
public Cache(CacheConfiguration cacheConfiguration) {
    ...
   }
```

Here is an example which creates a cache called test.

```
   //Create a CacheManager using defaults
   CacheManager manager = CacheManager.create();
   //Create a Cache specifying its configuration.
   Cache testCache = new Cache(
     new CacheConfiguration("test", maxElements)
       .memoryStoreEvictionPolicy(MemoryStoreEvictionPolicy.LFU)
       .overflowToDisk(true)
       .eternal(false)
       .timeToLiveSeconds(60)
       .timeToIdleSeconds(30)
       .diskPersistent(false)
       .diskExpiryThreadIntervalSeconds(0));
   manager.addCache(cache);
```

Once the cache is created, add it to the list of caches managed by the CacheManager:

```
     manager.addCache(testCache);
```

The cache is not usable until it has been added.

13.3.3 Using Caches

All of these examples refer to *manager*, which is a reference to a CacheManager, which has a cache in it called *sampleCache1*.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

13.3.3.1 Obtaining a reference to a Cache

Obtain a Cache called "sampleCache1", which has been preconfigured in the configuration file

```
Cache cache = manager.getCache("sampleCache1");
```

13.3.3.2 Performing CRUD operations

Put an element into a cache

```
Cache cache = manager.getCache("sampleCache1");
Element element = new Element("key1", "value1");
cache.put(element);
```

Update an element in a cache. Even though `cache.put()` is used, Ehcache knows there is an existing element, and considers the put an update for the purpose of notifying cache listeners.

```
Cache cache = manager.getCache("sampleCache1");
cache.put(new Element("key1", "value1"));
//This updates the entry for "key1"
cache.put(new Element("key1", "value2"));
```

Get a Serializable value from an element in a cache with a key of "key1".

```
Cache cache = manager.getCache("sampleCache1");
Element element = cache.get("key1");
Serializable value = element.getValue();
```

Get a NonSerializable value from an element in a cache with a key of "key1".

```
Cache cache = manager.getCache("sampleCache1");
Element element = cache.get("key1");
Object value = element.getObjectValue();
```

Remove an element from a cache with a key of "key1".

```
Cache cache = manager.getCache("sampleCache1");
cache.remove("key1");
```

13.3.3.3 Disk Persistence on demand

sampleCache1 has a persistent diskStore. We wish to ensure that the data and index are written immediately.

```
Cache cache = manager.getCache("sampleCache1");
cache.flush();
```

13.3.3.4 Obtaining Cache Sizes

Get the number of elements currently in the `Cache`.

```
Cache cache = manager.getCache("sampleCache1");
int elementsInMemory = cache.getSize();
```

Get the number of elements currently in the `MemoryStore`.

```
Cache cache = manager.getCache("sampleCache1");
long elementsInMemory = cache.getMemoryStoreSize();
```

Get the number of elements currently in the `DiskStore`.

```
Cache cache = manager.getCache("sampleCache1");
long elementsInMemory = cache.getDiskStoreSize();
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

13.3.3.5 Obtaining Statistics of Cache Hits and Misses

These methods are useful for tuning cache configurations.

Get the number of times requested items were found in the cache. i.e. cache hits

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getHitCount();
```

Get the number of times requested items were found in the `MemoryStore` of the cache.

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getMemoryStoreHitCount();
```

Get the number of times requested items were found in the `DiskStore` of the cache.

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getDiskStoreCount();
```

Get the number of times requested items were not found in the cache. i.e. cache misses.

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getMissCountNotFound();
```

Get the number of times requested items were not found in the cache due to expiry of the elements.

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getMissCountExpired();
```

These are just the most commonly used methods. See CacheTest for more examples. See Cache for the full API.

13.3.3.6 Dynamically Modifying Cache Configurations

This example shows how to dynamically modify the cache configuration of an already running cache:

```
Cache cache = manager.getCache("sampleCache");
CacheConfiguration config = cache.getCacheConfiguration();
config.setTimeToIdleSeconds(60);
config.setTimeToLiveSeconds(120);
config.setMaxElementsInMemory(10000);
config.setMaxElementsOnDisk(1000000);
```

Dynamic cache configurations can also be frozen to prevent future changes:

```
Cache cache = manager.getCache("sampleCache");
cache.disableDynamicFeatures();
```

13.3.3.7 JTA

A cache will automatically participate in the ongoing UserTransaction when configured in transactionalMode XA. This can be done programmatically:

```
//Create a CacheManager using defaults
CacheManager manager = CacheManager.create();
//Create a Cache specifying its configuration.
Cache xaCache = new Cache(
        new CacheConfiguration("test", 1000)
            .overflowToDisk(true)
            .eternal(false)
            .transactionalMode(CacheConfiguration.TransactionalMode.XA)
            .terracotta(new TerracottaConfiguration().clustered(true)));
manager.addCache(xaCache);
```

Or in your CacheManager's configuration file :

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
<cache name="xaCache"
    maxElementsInMemory="500"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="false"
    diskPersistent="false"
    diskExpiryThreadIntervalSeconds="1"
    transactionalMode="xa">
  <terracotta clustered="true"/>
    </cache>
```

Please note that XA Transactional caches are supported for standalone Ehcache and also when clustered with Terracotta, but with the replicating cluster architectures such as RMI|JMS|JGroups as there is no locking in those architectures.

The Cache can then be used without any special requirement. Changes will only become visible to others, once the transaction has been committed.

```
Ehcache cache = cacheManager.getEhcache("xaCache");
transactionManager.begin();
try {
  Element e = cache.get(key);
  Object result = complexService.doStuff(element.getValue());
  // This put will be rolled back should complexService.doMoreStuff throw an Except
  cache.put(new Element(key, result));
  // Any changes to result in that call, will be visible to others when the Transac
  complexService.doMoreStuff(result);
  transactionManager.commit();
} catch (Exception e) {
  transactionManager.rollback();
}
```

13.3.4 Using Distributed Caches

13.3.4.1 Terracotta Example

See the fully worked examples in the Terracotta Clustering Chapter.

13.3.5 Cache Statistics and Monitoring

13.3.5.1 Registering CacheStatistics in an MBeanServer

This example shows how to register CacheStatistics in the JDK1.5 platform MBeanServer, which works with the JConsole management agent.

```
CacheManager manager = new CacheManager();
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
ManagementService.registerMBeans(manager, mBeanServer, false, false, false, true);
```

13.3.6 More examples

13.3.6.1 JCache Examples

See the examples in the JCache Chapter.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

13.3.6.2 Cache Server Examples

See the examples in the Cache Server Chapter.

13.3.6.3 Browse the JUnit Tests

Ehcache comes with a comprehensive JUnit test suite, which not only tests the code, but shows you how to use ehcache.

A link to browsable unit test source code for the major Ehcache classes is given per section. The unit tests are also in the src.zip in the Ehcache tarball.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

14 Class loading and Class Loaders

14.1 Class loading and Class Loaders

Class loading within the plethora of environments Ehcache can be running is a somewhat vexed issue.

Since ehcache-1.2 all classloading is done in a standard way in one utility class: `ClassLoaderUtil`.

14.1.1 Plugin class loading

Ehcache allows plugins for events and distribution. These are loaded and created as follows:

```
/**
 * Creates a new class instance. Logs errors along the way. Classes are loaded using th
 * Ehcache standard classloader.
 *
 * @param className a fully qualified class name
 * @return null if the instance cannot be loaded
 */
public static Object createNewInstance(String className) throws CacheException {
    Class clazz;
    Object newInstance;
    try {
        clazz = Class.forName(className, true, getStandardClassLoader());
    } catch (ClassNotFoundException e) {
        //try fallback
        try {
            clazz = Class.forName(className, true, getFallbackClassLoader());
        } catch (ClassNotFoundException ex) {
            throw new CacheException("Unable to load class " + className +
                    ". Initial cause was " + e.getMessage(), e);
        }
    }
    try {
        newInstance = clazz.newInstance();
    } catch (IllegalAccessException e) {
        throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    } catch (InstantiationException e) {
        throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    }
    return newInstance;
}
/**
 * Gets the <code>ClassLoader</
code> that all classes in ehcache, and extensions, should
 * use for classloading. All ClassLoading in Ehcache should use this one. This is the o
 * thing that seems to work for all of the class loading situations found in the wild.
 * @return the thread context class loader.
 */
public static ClassLoader getStandardClassLoader() {
    return Thread.currentThread().getContextClassLoader();
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
}
/**
 * Gets a fallback <code>ClassLoader</
code> that all classes in ehcache, and extensions,
 * should use for classloading. This is used if the context class loader does not work.
 * @return the <code>ClassLoaderUtil.class.getClassLoader();</code>
 */
public static ClassLoader getFallbackClassLoader() {
    return ClassLoaderUtil.class.getClassLoader();
}
```

If this does not work for some reason a CacheException is thrown with a detailed error message.

14.1.2 Loading of ehcache.xml resources

If the configuration is otherwise unspecified, Ehcache looks for a configuration in the following order:

- Thread.currentThread().getContextClassLoader().getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache-failsafe.xml")

Ehcache uses the first configuration found.

Note the use of "/ehcache.xml" which requires that ehcache.xml be placed at the root of the classpath, i.e. not in any package.

14.1.3 Classloading with Terracotta clustering

If Terracotta clustering is being used with valueMode="serialization" then keys and values will be moved across the cluster in byte[] and deserialized on other nodes.

The classloaders used (in order) to instantiate those classes will be:

- Thread.currentThread().getContextClassLoader()
- The classloader that defined the CacheManager initially

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

15 Tuning Garbage Collection

15.1 Tuning Garbage Collection

Applications which use Ehcache can be expected to have larger heaps. Some Ehcache applications have heap sizes greater than 6GB.

Ehcache works well at this scale. However large heaps or long held object, which is what a cache is, can place strain on the default Garbage Collector.

Note. The following documentation relates to Sun JDK 1.5.

Finally Ehcache 2.3 introduced the Big Memory Offheap Store which adds an additional store outside of the heap so solve this problem.

15.1.1 Detecting Garbage Collection Problems

A full garbage collection event pauses all threads in the JVM. Nothing happens during the pause. If this pause takes more than a few seconds it will become noticeable.

The clearest way to see if this is happening is to run `jstat`. The following command will produce a log of garbage collection statistics, updated each ten seconds.

```
jstat -gcutil <pid> 10 1000000
```

The thing to watch for is the Full Garbage Collection Time. The difference between the total time for each reading is the time the system spends time paused. If there is a jump more than a few seconds this will not be acceptable in most application contexts.

15.1.2 Garbage Collection Tuning

The Sun core garbage collection team has offered the following tuning suggestion for virtual machines with large heaps using caching:

```
java ... -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC
    -XX:NewSize=<1/4 of total heap size> -XX:SurvivorRatio=16
```

The reasoning for each setting is as follows:

- -XX:+DisableExplicitGC - some libs call System.gc(). This is usually a bad idea and could explain some of what we saw.
- -XX:+UseConcMarkSweepGC - use the low pause collector
- -XX:NewSize= *1/4 of total heap size* -XX:SurvivorRatio=16

15.1.3 Distributed Caching Garbage Collection Tuning

Some users have reported that enabling distributed caching causes a full GC each minute. This is an issue with RMI generally, which can be worked around by increasing the interval for garbage collection. The effect that RMI is having is similar to a user application calling `System.gc()` each minute. In the settings above this is disabled, but it does not disable the full GC initiated by RMI.

The default in JDK6 was increased to 1 hour. The following system properties control the interval.

```
-Dsun.rmi.dgc.client.gcInterval=60000
-Dsun.rmi.dgc.server.gcInterval=60000
```

See http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4403367 for the bug report and detailed instructions on workarounds.

Increase the interval as required in your application.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

16 Cache Decorators

16.1 Cache Decorators

Ehcache 1.2 introduced the Ehcache interface, of which Cache is an implementation. It is possible and encouraged to create Ehcache decorators that are backed by a Cache instance, implement Ehcache and provide extra functionality.

The Decorator pattern is one of the the well known Gang of Four patterns.

Decorated caches are accessed from the CacheManager using `CacheManager.getEhcache(String name)`.

Note that, for backward compatibility, `CacheManager.getCache(String name)` has been retained. However only `CacheManager.getEhcache(String name)` returns the decorated cache.

16.1.1 Creating a Decorator

16.1.1.1 Programmatically

Cache decorators are created as follows:

```
BlockingCache newBlockingCache = new BlockingCache(cache);
```

The class must implement Ehcache.

16.1.1.2 By Configuration

Cache decorators can be configured directly in ehcache.xml. The decorators will be created and added to the CacheManager. It accepts the name of a concrete class that extends net.sf.ehcache.constructs.CacheDecoratorFactory The properties will be parsed according to the delimiter (default is comma ',') and passed to the concrete factory's `createDecoratedEhcache(Ehcache cache, Properties properties)` method along with the reference to the owning cache.

It is configured as per the following example:

```
<cacheDecoratorFactory
                    class="com.company.SomethingCacheDecoratorFactory"
                    properties="property1=36 ..." />
```

Note that from version 2.2, decorators can be configured against the `defaultCache`. This is very useful for frameworks like Hibernate that add caches based on the `defaultCache`.

16.1.2 Adding decorated caches to the CacheManager

Having created a decorator programmatically it is generally useful to put it in a place where multiple threads may access it.

Note that decorators created via configuration in ehcache.xml have already been added to the `CacheManager`.

16.1.2.1 Using `CacheManager.replaceCacheWithDecoratedCache()`

A built-in way is to replace the Cache in CacheManager with the decorated one. This is achieved as in the following example:

```
cacheManager.replaceCacheWithDecoratedCache(cache, newBlockingCache);
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

The `CacheManager replaceCacheWithDecoratedCache` method requires that the decorated cache be built from the underlying cache from the same name.

Note that any overwritten Ehcache methods will take on new behaviours without casting, as per the normal rules of Java. Casting is only required for new methods that the decorator introduces.

Any calls to get the cache out of the CacheManager now return the decorated one.

A word of caution. This method should be called in an appropriately synchronized init style method before multiple threads attempt to use it. All threads must be referencing the same decorated cache. An example of a suitable init method is found in `CachingFilter`:

```
/**
 * The cache holding the web pages. Ensure that all threads for a given cache name
 * are using the same instance of this.
 */
private BlockingCache blockingCache;
/**
 * Initialises blockingCache to use
 *
 * @throws CacheException The most likely cause is that a cache has not been
 *                        configured in Ehcache's configuration file ehcache.xml for th
 *                        filter name
 */
public void doInit() throws CacheException {
    synchronized (this.getClass()) {
        if (blockingCache == null) {
            final String cacheName = getCacheName();
            Ehcache cache = getCacheManager().getEhcache(cacheName);
            if (!(cache instanceof BlockingCache)) {
                //decorate and substitute
                BlockingCache newBlockingCache = new BlockingCache(cache);
                getCacheManager().replaceCacheWithDecoratedCache(cache, newBlockingCach
            }
            blockingCache = (BlockingCache) getCacheManager().getEhcache(getCacheName()
        }
    }
}
Ehcache blockingCache = singletonManager.getEhcache("sampleCache1");
```

The returned cache will exhibit the decorations.

16.1.2.2 Using `CacheManager.addDecoratedCache()`

Sometimes you want to add a decorated cache but retain access to the underlying cache.

The way to do this is to create a decorated cache and then call `cache.setName(new_name)` and then add it to `CacheManager` with `CacheManager.addDecoratedCache()`.

```
    /**
     * Adds a decorated {@link Ehcache} to the CacheManager. This method neither create
disk store
     * nor initializes the cache. It only adds the cache reference to the map of caches
     * cacheManager.
     * <p/>
     * It is generally required that a decorated cache, once constructed, is made avail
     * threads. The simplest way of doing this is to either add it to the cacheManager 
     * substitute the original cache with the decorated one.
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
     * <p/>
     * This method adds the decorated cache assuming it has a different name. If anothe
     * with the same name already exists, it will throw {@link ObjectExistsException}. I
     * cache with another decorated cache having same name, please use
     * {@link #replaceCacheWithDecoratedCache(Ehcache, Ehcache)}
     * <p/>
     * Note that any overridden Ehcache methods by the decorator will take on new behav
     * Casting is only required for new methods that the decorator introduces. For more
     * known Gang of Four Decorator pattern.
     *
     * @param decoratedCache
     * @throws ObjectExistsException
     *             if another cache with the same name already exists.
     */
    public void addDecoratedCache(Ehcache decoratedCache) throws ObjectExistsException
```

16.1.3 Built-in Decorators

16.1.3.1 BlockingCache

A blocking decorator for an Ehcache, backed by a @link Ehcache.

It allows concurrent read access to elements already in the cache. If the element is null, other reads will block until an element with the same key is put into the cache.

This is useful for constructing read-through or self-populating caches.

BlockingCache is used by `CachingFilter`.

16.1.3.2 SelfPopulatingCache

A selfpopulating decorator for Ehcache that creates entries on demand.

Clients of the cache simply call it without needing knowledge of whether the entry exists in the cache. If null the entry is created.

The cache is designed to be refreshed. Refreshes operate on the backing cache, and do not degrade performance of get calls.

SelfPopulatingCache extends BlockingCache. Multiple threads attempting to access a null element will block until the first thread completes. If refresh is being called the threads do not block - they return the stale data.

This is very useful for engineering highly scalable systems.

16.1.3.3 Caches with Exception Handling

These are decorated. See Cache Exception Handlers for full details.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

17 Hibernate Caching

17.1 Hibernate Second Level Cache

IMPORTANT NOTICES - PLEASE READ

Users of Ehcache and/or Terracotta Ehcache for Hibernate prior to Ehcache 2.0 should read:

- Upgrade Notes for Ehcache versions prior to 2.0

These instructions are for Hibernate 3. For older instructions on how to use Hibernate 2.1, please refer to:

- Guide for Version 1.1

17.1.1 Overview

Ehcache easily integrates with the Hibernate Object/Relational persistence and query service. Gavin King, the maintainer of Hibernate, is also a committer to the Ehcache project. This ensures Ehcache will remain a first class cache for Hibernate.

Configuring Ehcache for Hibernate is simple. The basic steps are:

- Download and install Ehcache into your project
- Configure Ehcache as a cache provider in your project's Hibernate configuration.
- Configure second level caching in your project's Hibernate configuration.
- Configure Hibernate caching for each entity, collection, or query you wish to cache.
- Configure ehcache.xml as necessary for each entity, collection, or query configured for caching.

For more information regarding cache configuration in Hibernate see the Hibernate documentation.

17.1.2 Downloading and Installing Ehcache

The Hibernate provider is in the ehcache-core module. Download:

- the latest version of the Ehcache core module here

For Terracotta clustering, download:

- a full Ehcache distribution here

17.1.3 Maven Dependency versions vary with the specific kit you intend to use. Since kits are guaranteed to contain compatible artifacts, find the artifact versions you need by downloading a kit.

Configure or add the following repository to your build (pom.xml):

```
<repository>
    <id>terracotta-releases</id>
    <url>http://www.terracotta.org/download/reflector/releases</url>
    <releases><enabled>true</enabled></releases>
    <snapshots><enabled>false</enabled></snapshots>
</repository>
```

Configure or add the the ehcache core module defined by the following dependency to your build (pom.xml):

```
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache-core</artifactId>
    <version>${ehcacheVersion}</version>
</dependency>
```

If you are configuring Hibernate and Ehcache for Terracotta clustering, add the following dependencies to your build (pom.xml):

```
    <dependency>
        <groupId>net.sf.ehcache</groupId>
        <artifactId>ehcache-terracotta</artifactId>
        <version>${ehcacheVersion}</version>
    </dependency>
    <dependency>
        <groupId>org.terracotta</groupId>
        <artifactId>terracotta-toolkit-${toolkitAPIversion}-runtime</
artifactId>
        <version>${toolkitVersion}</version>
    </dependency>
```

17.1.4 Configure Ehcache as the Second Level Cache Provider

To configure Ehcache as a Hibernate second level cache, set the region factory property (for Hibernate 3.3 and above) or the factory class property (Hibernate 3.2 and below) to one of the following in the Hibernate configuration.

Hibernate configuration is configured either via hibernate.cfg.xml, hibernate.properties or Spring. The format given is for hibernate.cfg.xml.

17.1.4.1 Hibernate 3.3 and higher

ATTENTION HIBERNATE 3.2 USERS: *Make sure to note the change to BOTH the property name and value.*

Use:

```
    <property name="hibernate.cache.region.factory_class">
              net.sf.ehcache.hibernate.EhCacheRegionFactory</property>
```

for instance creation, or

```
    <property name="hibernate.cache.region.factory_class">
              net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory</
property>
```

to force Hibernate to use a singleton of Ehcache CacheManager.

17.1.4.2 Hibernate 3.0 - 3.2

Use:

```
    <property name="hibernate.cache.provider_class">
              net.sf.ehcache.hibernate.EhCacheProvider</property>
```

for instance creation, or

```
    <property name="hibernate.cache.provider_class">
              net.sf.ehcache.hibernate.SingletonEhCacheProvider</property>
```

to force Hibernate to use a singleton Ehcache CacheManager.

17.1.5 Enable Second Level Cache and Query Cache Settings

In addition to configuring the second level cache provider setting, you will need to turn on the second level cache (by default it is configured to off - 'false' - by Hibernate). This is done by setting the following property in your hibernate config:

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

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

You may also want to turn on the Hibernate query cache. This is done by setting the following property in your hibernate config:

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

17.1.6 Optional

The following settings or actions are optional.

17.1.6.1 Ehcache Configuration Resource Name

The `configurationResourceName` property is used to specify the location of the ehcache configuration file to be used with the given Hibernate instance and cache provider/region-factory.

The resource is searched for in the root of the classpath. It is used to support multiple CacheManagers in the same VM. It tells Hibernate which configuration to use. An example might be "ehcache-2.xml".

When using multiple Hibernate instances it is therefore recommended to use multiple non-singleton providers or region factories, each with a dedicated Ehcache configuration resource.

```
net.sf.ehcache.configurationResourceName=/name_of_ehcache.xml
```

17.1.6.2 Set the Hibernate cache provider programmatically

The provider can also be set programmatically in Hibernate by adding necessary Hibernate property settings to the configuration before creating the SessionFactory:

```
Configuration.setProperty("hibernate.cache.region.factory_class",
                          "net.sf.ehcache.hibernate.EhCacheRegionFactory")
```

17.1.7 Putting it all together

If you are using Hibernate 3.3 and enabling both second level caching and query caching, then your hibernate config file should contain the following:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property name="hibernate.cache.region.factory_class">net.sf.ehcache.hibernate.EhCacheR
property>
```

An equivalent Spring configuration file would contain:

```
<prop key="hibernate.cache.use_second_level_cache">true</prop>
<prop key="hibernate.cache.use_query_cache">true</prop>
<prop key="hibernate.cache.region.factory_class">net.sf.ehcache.hibernate.EhCacheRegion
prop>
```

17.1.8 Configure Hibernate Entities to use Second Level Caching

In addition to configuring the Hibernate second level cache provider, Hibernate must also be told to enable caching for entities, collections, and queries.

For example, to enable cache entries for the domain object com.somecompany.someproject.domain.Country there would be a mapping file something like the following:

```
<hibernate-mapping>
<class
    name="com.somecompany.someproject.domain.Country"
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
    table="ut_Countries"
    dynamic-update="false"
    dynamic-insert="false"
>
...
</class>
</hibernate-mapping>
```

To enable caching, add the following element.

```
<cache usage="read-write|nonstrict-read-write|read-only" />
```

e.g.

```
<hibernate-mapping>
<class
    name="com.somecompany.someproject.domain.Country"
    table="ut_Countries"
    dynamic-update="false"
    dynamic-insert="false"
>
  <cache usage="read-write" />
...
</class>
</hibernate-mapping>
```

This can also be achieved using the @Cache annotation, e.g.

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Country {
    ...
}
```

17.1.8.1 Definition of the different cache strategies

17.read-only

Caches data that is never updated.

17.nonstrict-read-write

Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database. Configure your cache timeout accordingly!

17.read-write

Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read", this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes.

17.1.9 Configure ehcache.xml

Because ehcache.xml has a defaultCache, caches will always be created when required by Hibernate. However more control can be exerted by specifying a configuration per cache, based on its name.

In particular, because Hibernate caches are populated from databases, there is potential for them to get very large. This can be controlled by capping their maxElementsInMemory and specifying whether to overflowToDisk beyond that.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

Hibernate uses a specific convention for the naming of caches of Domain Objects, Collections, and Queries.

17.1.9.1 Domain Objects

Hibernate creates caches named after the fully qualified name of Domain Objects.

So, for example to create a cache for com.somecompany.someproject.domain.Country create a cache configuration entry similar to the following in ehcache.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
  <cache
    name="com.somecompany.someproject.domain.Country"
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true"
  />
</ehcache>
```

17.1.9.2 Hibernate

CacheConcurrencyStrategy read-write, nonstrict-read-write and read-only policies apply to Domain Objects.

17.1.9.3 Collections

Hibernate creates collection caches named after the fully qualified name of the Domain Object followed by "." followed by the collection field name.

For example, a Country domain object has a set of advancedSearchFacilities. The Hibernate doclet for the accessor looks like:

```
 /**
  * Returns the advanced search facilities that should appear for this country.
  * @hibernate.set cascade="all" inverse="true"
  * @hibernate.collection-key column="COUNTRY_ID"
  * @hibernate.collection-one-to-
many class="com.wotif.jaguar.domain.AdvancedSearchFacility"
  * @hibernate.cache usage="read-write"
  */
 public Set getAdvancedSearchFacilities() {
    return advancedSearchFacilities;
 }
```

You need an additional cache configured for the set. The ehcache.xml configuration looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
  <cache name="com.somecompany.someproject.domain.Country"
     maxElementsInMemory="50"
     eternal="false"
     timeToLiveSeconds="600"
     overflowToDisk="true"
 />
  <cache
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
      name="com.somecompany.someproject.domain.Country.advancedSearchFacilities"
      maxElementsInMemory="450"
      eternal="false"
      timeToLiveSeconds="600"
      overflowToDisk="true"
 />
</ehcache>
```

17.1.9.4 Hibernate CacheConcurrencyStrategy

read-write, nonstrict-read-write and read-only policies apply to Domain Object collections.

17.1.9.5 Queries

Hibernate allows the caching of query results using two caches.

"net.sf.hibernate.cache.StandardQueryCache" and "net.sf.hibernate.cache.UpdateTimestampsCache" in versions 2.1 to 3.1 and "org.hibernate.cache.StandardQueryCache" and "org.hibernate.cache.UpdateTimestampsCache" in version 3.2. are always used.

17.1.9.6 StandardQueryCache

This cache is used if you use a query cache without setting a name. A typical ehcache.xml configuration is:

```
<cache
    name="org.hibernate.cache.StandardQueryCache"
    maxElementsInMemory="5"
    eternal="false"
    timeToLiveSeconds="120"
    overflowToDisk="true"/>
```

17.1.9.7 UpdateTimestampsCache

Tracks the timestamps of the most recent updates to particular tables. It is important that the cache timeout of the underlying cache implementation be set to a higher value than the timeouts of any of the query caches. In fact, it is recommend that the the underlying cache not be configured for expiry at all.

A typical ehcache.xml configuration is:

```
<cache
    name="org.hibernate.cache.UpdateTimestampsCache"
    maxElementsInMemory="5000"
    eternal="true"
    overflowToDisk="true"/>
```

17.1.9.8 Named Query Caches

In addition, a QueryCache can be given a specific name in Hibernate using Query.setCacheRegion(String name). The name of the cache in ehcache.xml is then the name given in that method. The name can be whatever you want, but by convention you should use "query." followed by a descriptive name.

E.g.

```
<cache name="query.AdministrativeAreasPerCountry"
    maxElementsInMemory="5"
    eternal="false"
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
    timeToLiveSeconds="86400"
    overflowToDisk="true"/>
```

17.1.9.9 Using Query Caches

For example, let's say we have a common query running against the Country Domain.

Code to use a query cache follows:

```
public List getStreetTypes(final Country country) throws HibernateException {
 final Session session = createSession();
 try {
        final Query query = session.createQuery(
        "select st.id, st.name"
        + " from StreetType st "
        + " where st.country.id = :countryId "
        + " order by st.sortOrder desc, st.name");
        query.setLong("countryId", country.getId().longValue());
        query.setCacheable(true);
        query.setCacheRegion("query.StreetTypes");
        return query.list();
     } finally {
        session.close();
     }
 }
```

The `query.setCacheable(true)` line caches the query.

The `query.setCacheRegion("query.StreetTypes")` line sets the name of the Query Cache.

Alex Miller has a good article on the query cache here.

17.1.9.10 Hibernate CacheConcurrencyStrategy

None of read-write, nonstrict-read-write and read-only policies apply to Domain Objects. Cache policies are not configurable for query cache. They act like a non-locking read only cache.

17.1.10 Demo Apps

We have demo applications showing how to use the Hibernate 3.3 CacheRegionFactory.

17.1.10.1 Hibernate Tutorial

Check out from https://svn.terracotta.org/repo/forge/projects/hibernate-tutorial-web/trunk terracotta_community_login

17.1.10.2 Examinator

Examinator is our complete application that shows many aspects of caching, all using the Terracotta Server Array.

Check out from http://svn.terracotta.org/svn/forge/projects/exam/trunk terracotta_community_login * Performance Tips

17.1.10.3 Session.load

Session.load will always try to use the cache.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

17.1.10.4 Session.find and Query.find

Session.find does not use the cache for the primary object. Hibernate will try to use the cache for any associated objects. Session.find does however cause the cache to be populated.

Query.find works in exactly the same way.

Use these where the chance of getting a cache hit is low.

17.1.10.5 Session.iterate and Query.iterate

Session.iterate always uses the cache for the primary object and any associated objects.

Query.iterate works in exactly the same way.

Use these where the chance of getting a cache hit is high.

17.1.11 How to Scale

Configuring each Hibernate instance with a standalone ehcache will dramatically improve performance. However most production applications use multiple application instances for redundancy and for scalability. Ideally applications are horizontally scalable, where adding more application instances linearly improves throughput.

With an application deployed on multiple nodes, using standalone Ehcache means that each instance holds its own data. On a cache miss on any node, Hibernate will read from the database. This generally results in N reads where N is the number of nodes in the cluster. As each new node gets added database workload goes up. Also, when data is written in one node, the other nodes are unaware of the data write, and thus subsequent reads of this data on other nodes will result in stale reads.

The solution is to turn on distributed caching or replicated caching.

Ehcache comes with native cache distribution using the following mechanism:

- Terracotta

Ehcache supports the following methods of cache replication:

- RMI
- JGroups
- JMS replication

Selection of the distributed cache or replication mechanism may be made or changed at any time. There are no changes to the application. Only changes to ehcache.xml file are required. This allows an application to easily scale as it grows without expensive re-architecting.

17.1.12 Configuring Ehcache for distributed caching using Terracotta

Ehcache provides built-in support for Terracotta distributed caching. The following are the key considerations when selecting this option:

- Simple snap-in configuration with one line of configuration
- Simple to scale up to as much performance as you need -- no application changes required
- Wealth of "CAP" configuration options allow you to configure your cache for whatever it needs - fast, coherent, asynchronous updates, dirty reads etc.
- The fastest coherent option for caches with reads and writes
- Store as much data as you want - 20GB -> 1TB
- Commercial products and support available from http://www.terracotta.org

Configuring Terracotta replication is described in the Terracotta Documentation. A sample cache configuration is provided here:

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
  <terracottaConfig url="localhost:9510" />
  <cache
    name="com.somecompany.someproject.domain.Country"
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true">
    <terracotta/>
  </cache>
</ehcache>
```

17.1.13 Configuring Replicated Caching using RMI, JGroups, or JMS

Ehcache can use JMS, JGroups or RMI as a cache replication scheme. The following are the key considerations when selecting this option:

- The consistency is weak. Nodes might be stale, have different versions or be missing an element that other nodes have. Your application should be tolerant of weak consistency.
- `session.refresh()` should be used to check the cache against the database before performing a write that must be correct. This can have a performance inmpact on the database.
- Each node in the cluster stores all data, thus the cache size is limited to memory size, or disk if disk overflow is selected.

17.1.13.1 Configuring for RMI Replication

RMI configuration is described in the Ehcache User Guide - RMI Distributed Caching. A sample cache configuration (using automatic discovery) is provided here:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
  <cacheManagerPeerProviderFactory
   class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
   properties="peerDiscovery=automatic, multicastGroupAddress=230.0.0.1,
   multicastGroupPort=4446, timeToLive=32"/>
  <cache
    name="com.somecompany.someproject.domain.Country"
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true">
    <cacheEventListenerFactory
            class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>
  </cache>
</ehcache>
```

17.1.13.2 Configuring for JGroups Replication

Configuraging JGroups replication is described in the Ehcache User Guide - Distributed Caching with JGroups. A sample cache configuration is provided here:

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

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
  <cacheManagerPeerProviderFactory class="net.sf.ehcache.distribution.jgroups
    .JGroupsCacheManagerPeerProviderFactory"
     properties="connect=UDP(mcast_addr=231.12.21.132;mcast_port=45566;ip_ttl=32;
     mcast_send_buf_size=150000;mcast_recv_buf_size=80000):
     PING(timeout=2000;num_initial_members=6):
     MERGE2(min_interval=5000;max_interval=10000):
     FD_SOCK:VERIFY_SUSPECT(timeout=1500):
     pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):
     UNICAST(timeout=5000):
     pbcast.STABLE(desired_avg_gossip=20000):
     FRAG:
     pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;
     shun=false;print_local_addr=true)"
 propertySeparator="::"
     />
  <cache
    name="com.somecompany.someproject.domain.Country"
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true">
    <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"
    properties="replicateAsynchronously=true, replicatePuts=true,
replicateUpdates=true, replicateUpdatesViaCopy=false,
replicateRemovals=true" />
  </cache>
</ehcache>
```

17.1.13.3 Configuring for JMS Replication

Configuring JMS replication is described in the Ehcache User Guide - JMS Distributed Caching. A sample cache configuration (for ActiveMQ) is provided here:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
  <cacheManagerPeerProviderFactory
            class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"
            properties="initialContextFactoryName=ExampleActiveMQInitialContextFactory,
                providerURL=tcp://localhost:61616,
                topicConnectionFactoryBindingName=topicConnectionFactory,
                topicBindingName=ehcache"
            propertySeparator=","
            />
  <cache
    name="com.somecompany.someproject.domain.Country"
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true">
    <cacheEventListenerFactory
          class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
```

```
          properties="replicateAsynchronously=true,
                        replicatePuts=true,
                        replicateUpdates=true,
                        replicateUpdatesViaCopy=true,
                        replicateRemovals=true,
                        asynchronousReplicationIntervalMillis=1000"
           propertySeparator=","/>
  </cache>
</ehcache>
```

17.1.14 FAQ

17.1.14.1 Should I use the provider in the Hibernate distribution or in Ehcache?

Since Hibernate 2.1, Hibernate has included an Ehcache `CacheProvider`. That provider is periodically synced up with the provider in the Ehcache Core distribution. New features are generally added in to the Ehcache Core provider and then the Hibernate one.

17.1.14.2 What is the relationship between the Hibernate and Ehcache projects?

Gavin King and Greg Luck cooperated to create Ehcache and include it in Hibernate. Since 2009 Greg Luck has been a committer on the Hibernate project so as to ensure Ehcache remains a first-class 2nd level cache for Hibernate.

17.1.14.3 Does Ehcache support the new Hibernate 3.3 2nd level caching SPI?

Yes. Ehcache 2.0 supports this new API.

17.1.14.4 Does Ehcache support the transactional strategy?

Yes. It was introduced in Ehcache 2.1.

17.1.14.5 Is Ehcache Cluster Safe?

hibernate.org maintains a table listing the providers. While ehcache works as a distributed cache for Hibernate, it is not listed as "Cluster Safe". What this means is that `Hibernate's lock and unlock methods are not implemented. Changes in one node will be applied without locking. This may or may not be a noticeable problem.

In Ehcache 1.7 when using Terracotta, this cannot happen as access to the clustered cache itself is controlled with read locks and write locks.

In Ehcache 2.0 when using Terracotta, the lock and unlock methods tie-in to the underlying clustered cache locks. We expect Ehcache 2.0 to be marked as cluster safe in new versions of the Hibernate documentation.

17.1.14.6 How are Hibernate Entities keyed?

Hibernate identifies cached Entities via an object id. This is normally the primary key of a database row.

17.1.14.7 Can you use Identity mode with the Terracotta Server Array

You cannot use identity mode clustered cache with Hibernate. If the cache is exclusively used by Hibernate we will convert identity mode caches to serialization mode. If the cache cannot be determined to be exclusively used by Hibernate (i.e. generated from a singleton cache manager) then

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

an exception will be thrown indicating the misconfigured cache. Serialization mode is in any case the default for Terracotta clustered caches.

17.1.14.8 I get `org.hibernate.cache.ReadWriteCache - An item was expired by the cache while it was locked` error messages. What is it?

Soft locks are implemented by replacing a value with a special type that marks the element as locked, thus indicating to other threads to treat it differently to a normal element. This is used in the Hibernate Read/Write strategy to force fall-through to the database during the two-phase commit - since we don't know exactly what should be returned by the cache while the commit is in process (but the db does).

If a soft-locked Element is evicted by the cache during the 2 phase commit, then once the 2 phase commit completes the cache will fail to update (since the soft-locked Element was evicted) and the cache entry will be reloaded from the database on the next read of that object. This is obviously non-fatal (we're a cache failure here so it should not be a problem).

The only problem it really causes would I imagine be a small rise in db load.

So, in summary the Hibernate messages are not problematic.

The underlying cause is the probabilistic evictor can theoretically evict recently loaded items. This evictor has been tuned over successive ehcache releases. As a result this warning will happen most often in 1.6, less often in 1.7 and very rarely in 1.8.

You can also use the deterministic evictor to avoid this problem. Specify the `java -Dnet.sf.ehcache.use.classic.lru=true` system property to turn on classic LRU which contains a deterministic evictor.

17.1.14.9 I get java.lang.ClassCastException: org.hibernate.cache.ReadWriteCache$Item incompatible with net.sf.ehcache.hibernate.strategy.AbstractReadWriteEhcacheAccessStrategy$Lockable

This is the tell-tale error you get if you are:

- using a Terracotta cluster with Ehcache
- you have upgraded part of the cluster to use net.sf.ehcache.hibernate.EhCacheRegionFactory but part of it is still using the old SPI of EhCacheProvider.
- you are upgrading a Hibernate version

  Ensure you have changed all nodes and that you clear any caches during the upgrade.

17.1.14.10 Are compound keys supported?

For standalone caching yes. With Terracotta a larger amount of memory is used.

17.1.14.11 Why do I not see replicated data when using nonstrict mode?

You may thing that Hibernate's *nonstrict* mode is just like *read-write* but with dirty reads. The truth is far more complex than that. Suffice to say, in *nonstrict* mode, Hibernate puts the object in the appropriate cache but then IMMEDIATELY removes it. The PUT and the REMOVE are BOTH replicated by ehcache so the net effect of that is the new object is copied to remote cache but then it's immediately followed by a replicated remove # so the next time you try get the object it's not in cache and hibernate goes back to the DB.

So, practically there is no point using nonstrict mode with replicated or distributed caches. If you want the updated entry to be replicated or distributed use *readwrite* or *transactional*.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

18 Web Caching

18.1 Web Caching

Ehcache provides a set of general purpose web caching filters in the `ehcache-web` module.

Using these can make an amazing difference to web application performance. A typical server can deliver 5000+ pages per second from the page cache. With built-in gzipping, storage and network transmission is highly efficient. Cache pages and fragments make excellent candidates for `DiskStore` storage, because the object graphs are simple and the largest part is already a `byte[]`.

18.1.1 SimplePageCachingFilter

This is a simple caching filter suitable for caching compressable HTTP responses such as HTML, XML or JSON.

It uses a Singleton CacheManager created with the default factory method. Override to use a different CacheManager

It is suitable for:

- complete responses i.e. not fragments.
- A content type suitable for gzipping. e.g. text or text/html
  For fragments see the SimplePageFragmentCachingFilter.

18.1.2 Keys

Pages are cached based on their key. The key for this cache is the URI followed by the query string. An example is `/admin/SomePage.jsp?id=1234&name=Beagle`.

This key technique is suitable for a wide range of uses. It is independent of hostname and port number, so will work well in situations where there are multiple domains which get the same content, or where users access based on different port numbers.

A problem can occur with tracking software, where unique ids are inserted into request query strings. Because each request generates a unique key, there will never be a cache hit. For these situations it is better to parse the request parameters and override `calculateKey(javax.servlet.http.HttpServletRequest)` with an implementation that takes account of only the significant ones.

18.1.3 Configuring the cacheName

A cache entry in ehcache.xml should be configured with the name of the filter.

Names can be set using the init-param `cacheName`, or by sub-classing this class and overriding the name.

18.1.4 Concurrent Cache Misses

A cache miss will cause the filter chain, upstream of the caching filter to be processed. To avoid threads requesting the same key to do useless duplicate work, these threads block behind the first thread.

The thead timeout can be set to fail after a certain wait by setting the init-param `blockingTimeoutMillis`. By default threads wait indefinitely. In the event upstream processing never returns, eventually the web server may get overwhelmed with connections it has not responded to. By setting a timeout, the waiting threads will only block for the set time, and then throw a @link

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

net.sf.ehcache.constructs.blocking.LockTimeoutException. Under either scenario an upstream failure will still cause a failure.

18.1.5 Gzipping

Significant network efficiencies, and page loading speedups, can be gained by gzipping responses.

Whether a response can be gzipped depends on:

- Whether the user agent can accept GZIP encoding. This feature is part of HTTP1.1. If a browser accepts GZIP encoding it will advertise this by including in its HTTP header: All common browsers except IE 5.2 on Macintosh are capable of accepting gzip encoding. Most search engine robots do not accept gzip encoding.
- Whether the user agent has advertised its acceptance of gzip encoding. This is on a per request basis. If they will accept a gzip response to their request they must include the following in the HTTP request header:

  ```
  Accept-Encoding: gzip
  ```

  Responses are automatically gzipped and stored that way in the cache. For requests which do not accept gzip encoding the page is retrieved from the cache, ungzipped and returned to the user agent. The ungzipping is high performance.

18.1.6 Caching Headers

The `SimpleCachingHeadersPageCachingFilter` extends `SimplePageCachingFilter` to provide the HTTP cache headers: ETag, Last-Modified and Expires. It supports conditional GET.

Because browsers and other HTTP clients have the expiry information returned in the response headers, they do not even need to request the page again. Even once the local browser copy has expired, the browser will do a conditional GET.

So why would you ever want to use SimplePageCachingFilter, which does not set these headers? The answer is that in some caching scenarios you may wish to remove a page before its natural expiry. Consider a scenario where a web page shows dynamic data. Under Ehcache the Element can be removed at any time. However if a browser is holding expiry information, those browsers will have to wait until the expiry time before getting updated. The caching in this scenario is more about defraying server load rather than minimising browser calls.

18.1.7 Init-Params

The following init-params are supported:

- `cacheName` - the name in ehcache.xml used by the filter.
- `blockingTimeoutMillis` - the time, in milliseconds, to wait for the filter chain to return with a response on a cache miss. This is useful to fail fast in the event of an infrastructure failure.
- `varyHeader` - set to true to set Vary:Accept-Encoding in the response when doing Gzip. This header is needed to support HTTP proxies however it is off by default.

  ```
  <init-param>
     <param-name>varyHeader</param-name>
     <param-value>true</param-value>
  </init-param>
  ```

18.1.8 Re-entrance

Care should be taken not to define a filter chain such that the same `CachingFilter` class is reentered. The `CachingFilter` uses the `BlockingCache`. It blocks until the thread which did a get which results in a null does a put. If reentry happens a second get happens before the

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

first put. The second get could wait indefinitely. This situation is monitored and if it happens, an IllegalStateException will be thrown.

18.1.9 SimplePageFragmentCachingFilter

The SimplePageFragmentCachingFilter does everyting that SimplePageCachingFilter does, except it never gzips, so the fragments can be combined. There is variant of this filter which sets browser caching headers, because that is only applicable to the entire page.

18.1.10 Example web.xml configuration

```
<web-app 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/web-app_2_5.xsd "
     version="2.5">
  <filter>
    <filter-name>CachePage1CachingFilter</filter-name>
    <filter-
class>net.sf.ehcache.constructs.web.filter.SimplePageCachingFilter
    </filter-class>
    <init-param>
      <param-name>suppressStackTraces</param-name>
      <param-value>false</param-value>
    </init-param>
    <init-param>
      <param-name>cacheName</param-name>
      <param-value>CachePage1CachingFilter</param-value>
    </init-param>
  </filter>
  <filter>
    <filter-name>SimplePageFragmentCachingFilter</filter-name>
    <filter-
class>net.sf.ehcache.constructs.web.filter.SimplePageFragmentCachingFilter
    </filter-class>
    <init-param>
      <param-name>suppressStackTraces</param-name>
      <param-value>false</param-value>
    </init-param>
    <init-param>
      <param-name>cacheName</param-name>
      <param-value>SimplePageFragmentCachingFilter</param-value>
    </init-param>
  </filter>
  <filter>
    <filter-name>SimpleCachingHeadersPageCachingFilter</filter-name>
    <filter-
class>net.sf.ehcache.constructs.web.filter.SimpleCachingHeadersPageCachingFilter
    </filter-class>
    <init-param>
      <param-name>suppressStackTraces</param-name>
      <param-value>false</param-value>
    </init-param>
    <init-param>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
      <param-name>cacheName</param-name>
      <param-value>CachedPage2Cache</param-value>
    </init-param>
  </filter>
  <!-- This is a filter chain. They are executed in the order below.
  Do not change the order. -->
  <filter-mapping>
    <filter-name>CachePage1CachingFilter</filter-name>
    <url-pattern>/CachedPage.jsp</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>FORWARD</dispatcher>
  </filter-mapping>
  <filter-mapping>
    <filter-name>SimplePageFragmentCachingFilter</filter-name>
    <url-pattern>/include/Footer.jsp</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>SimplePageFragmentCachingFilter</filter-name>
    <url-pattern>/fragment/CachedFragment.jsp</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>SimpleCachingHeadersPageCachingFilter</filter-name>
    <url-pattern>/CachedPage2.jsp</url-pattern>
  </filter-mapping>
```

An ehcache.xml configuration file, matching the above would then be:

```
<Ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="../../main/config/ehcache.xsd">
    <diskStore path="java.io.tmpdir"/>
    <defaultCache
            maxElementsInMemory="10"
            eternal="false"
            timeToIdleSeconds="5"
            timeToLiveSeconds="10"
            overflowToDisk="true"
            />
    <!-- Page and Page Fragment Caches -->
    <cache name="CachePage1CachingFilter"
           maxElementsInMemory="10"
           eternal="false"
           timeToIdleSeconds="10000"
           timeToLiveSeconds="10000"
           overflowToDisk="true">
    </cache>
    <cache name="CachedPage2Cache"
           maxElementsInMemory="10"
           eternal="false"
           timeToLiveSeconds="3600"
           overflowToDisk="true">
    </cache>
    <cache name="SimplePageFragmentCachingFilter"
           maxElementsInMemory="10"
           eternal="false"
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
            timeToIdleSeconds="10000"
            timeToLiveSeconds="10000"
            overflowToDisk="true">
    </cache>
    <cache name="SimpleCachingHeadersTimeoutPageCachingFilter"
            maxElementsInMemory="10"
            eternal="false"
            timeToIdleSeconds="10000"
            timeToLiveSeconds="10000"
            overflowToDisk="true">
    </cache>
</ehcache>
```

18.1.11 CachingFilter Exceptions

Additional exception types have been added to the Caching Filter.

18.1.11.1 FilterNonReentrantException

Thrown when it is detected that a caching filter's doFilter method is reentered by the same thread. Reentrant calls will block indefinitely because the first request has not yet unblocked the cache. Nasty.

18.1.11.2 AlreadyGzippedException

The web package performs gzipping operations. One cause of problems on web browsers is getting content that is double or triple gzipped. They will either get gobblydeegook or a blank page. This exception is thrown when a gzip is attempted on already gzipped content.

18.1.11.3 ResponseHeadersNotModifiableException

A gzip encoding header needs to be added for gzipped content. The HttpServletResponse#setHeader() method is used for that purpose. If the header had already been set, the new value normally overwrites the previous one. In some cases according to the servlet specification, setHeader silently fails. Two scenarios where this happens are:

- The response is committed.
- RequestDispatcher#include method caused the request.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

19 Using ColdFusion with Ehcache

19.1 Using Ehcache with ColdFusion

19.1.1 Which version of Ehcache comes with which version of ColdFusion?

ColdFusion now ships with Ehcache. Here are the versions shipped:

- ColdFusion 9.0.1 includes Ehcache 2.0 out-of-the-box
- ColdFusion 9 includes Ehcache 1.6 out-of-the-box
- ColdFusion 8 caching was not built on Ehcache, but Ehcache can easily be integrated with a CF8 application (see below).

19.1.2 Which version of Ehcache should I use if I want a distributed cache?

Ehcache is designed so that applications written to use it can easily scale out. A standalone cache (the default in ColdFusion 9) can easily be distributed. A distributed cache solves database bottleneck problems, cache drift (where the data cached in individual application server nodes becomes out of sync), and also (when using the recommended 2-tier Terracotta distributed cache) provides the ability to have a highly available, coherent in-memory cache that is far larger than can fit in any single JVM heap. See http://ehcache.org/documentation/distributed_caching.html for details.

Note: Ehcache 1.7 and higher support the Terracotta distributed cache out of the box. Due to Ehcache's API backward compatibility, it is easy to swap out older versions of ehcache with newer ones to leverage the features of new releases.

19.1.3 Using Ehcache with ColdFusion 9 and 9.0.1

The ColdFusion community has actively engaged with Ehcache and have put out lots of great blogs. Here are three to get you started.

For a short introduction - check out Raymond Camden's blog: http://www.coldfusionjedi.com/index.cfm/2009/7/18/ColdFusion-9-and-Caching-Enhancements

For more in-depth analysis read Rob Brooks-Bilson's awesome 9 part Blog Series: http://www.brooks-bilson.com/blogs/rob/index.cfm/2009/7/21/Caching-Enhancements-in-ColdFusion-9--Part-1-Why-Cache

14 days of ColdFusion caching, by Aaron West, covering a different topic each day: http://www.aaronwest.net/blog/index.cfm/2009/11/17/14-Days-of-ColdFusion-9-Caching-Day-1--Caching-a-Full-Page

19.1.4 Switching from a local cache to a distributed cache with ColdFusion 9.0.1

1. http://www.terracotta.org/dl. Click the link to the open-source kit if you are using open source and get `terracotta-<version>-installer.jar`.

Install the kit with 'java -jar terracotta-<version>-installer.jar'. We will refer to the directory you installed it into as TCHOME. Similarly, we will refer to the location of ColdFusion as CFHOME. These instructions assume you are working with a standalone server install of ColdFusion; if working with a EAR/WAR install you will need to modify the steps accordingly (file locations may vary and additional steps may be needed to rebuild the EAR/WAR).

Before integrating the distributed cache with ColdFusion, you may want to follow the simple self-contained tutorial which uses one of the samples in the kit to demonstrate distributed caching: http://www.terracotta.org/start/distributed-cache-tutorial

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

2. Copy TCHOME/ehcache/lib/ehcache-terracotta-<version>.jar into CFHOME/lib

3. Edit the CFHOME/lib/ehcache.xml to add the necessary two lines of config to turn on distributed caching

```
<terracottaConfig url="localhost:9510"/>
<defaultCache
        ...
        >
                <terracotta clustered="true" />
</defaultCache>
```

4. Edit jvm.config (typically in CFHOME/runtime/bin) properties to ensure that coldfusion.classPath (set with -Dcoldfusion.classPath= in java.args) DOES NOT include any relative paths (ie ../) - ie replace the relative paths with full paths (This is to work around a known issue in ehcache-terracotta-2.0.0.jar).

5. Start the Terracotta server in a *NIX shell or Microsoft Windows:

```
$TCHOME/bin/start-tc-server.sh
start-tc-server.bat
```

Note: In production, you would run your servers on a set of separate machines for fault tolerance and performance.

6. Start ColdFusion, access your application, and see the distributed cache in action.

7. This is just the tip of the iceberg & you'll probably have lots of questions. Drop us an email to info@terracottatech.com to let us know how you got on, and if you have questions you'd like answers to.

19.1.5 Using Ehcache with ColdFusion 8

To integrate Ehcache with ColdFusion 8, first add the ehcache-core jar (and its dependent jars) to your web application lib directory.

The following code demonstrates how to call Ehcache from ColdFusion 8. It will cache a CF object in Ehcache and the set expiration time to 30 seconds. If you refresh the page many times within 30 seconds, you will see the data from cache. After 30 seconds, you will see a cache miss, then the code will generate a new object and put in cache again.

```
<CFOBJECT type="JAVA" class="net.sf.ehcache.CacheManager" name="cacheManager">
<cfset cache=cacheManager.getInstance().getCache("MyBookCache")>
<cfset myBookElement=#cache.get("myBook")#>
<cfif IsDefined("myBookElement")>
        <cfoutput>
         myBookElement: #myBookElement#<br />
        </cfoutput>
        <cfif IsStruct(myBookElement.getObjectValue())>
                <strong>Cache Hit</strong><p/>
                <!-- Found the object from cache -->
                <cfset myBook = #myBookElement.getObjectValue()#>
        </cfif>
</cfif>
<cfif IsDefined("myBook")>
<cfelse>
<strong>Cache Miss</strong>
        <!-- object not found in cache, go ahead create it -->
        <cfset myBook = StructNew()>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
        <cfset a = StructInsert(myBook, "cacheTime", LSTimeFormat(Now(), 'hh:mm:sstt'),
        <cfset a = StructInsert(myBook, "title", "EhCache Book", 1)>
        <cfset a = StructInsert(myBook, "author", "Greg Luck", 1)>
        <cfset a = StructInsert(myBook, "ISBN", "ABCD123456", 1)>
        <CFOBJECT type="JAVA" class="net.sf.ehcache.Element" name="myBookElement">
        <cfset myBookElement.init("myBook", myBook)>
        <cfset cache.put(myBookElement)>
</cfif>
<cfoutput>
 Cache time: #myBook["cacheTime"]#<br />
 Title: #myBook["title"]#<br />
 Author: #myBook["author"]#<br />
 ISBN: #myBook["ISBN"]#
</cfoutput>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

20 Cache Topologies

20.1 Distributed and Replicated Caching

Many production applications are deployed in clusters of multiple instances for availability and scalability. However, without a distributed or replicated cache, application clusters exhibit a number of undesirable behaviors, such as:

- **Cache Drift**--if each application instance maintains its own cache, updates made to one cache will not appear in the other instances. This also happens to web session data. A distributed or replicated cache ensures that all of the cache instances are kept in sync with each other.
- **Database Bottlenecks**--In a single-instance application, a cache effectively shields a database from the overhead of redundant queries. However, in a distributed application environment, each instance much load and keep its own cache fresh. The overhead of loading and refreshing multiple caches leads to database bottlenecks as more application instances are added. A distributed or replicated cache eliminates the per-instance overhead of loading and refreshing multiple caches from a database.

20.1.1 Distributed Caching

Ehcache comes bundled with a distributed caching mechanism using Terracotta that enables multiple CacheManagers and their caches in multiple JVMs to share data with each other. Adding distributed caching to Ehcache takes only two lines of configuration.

Using Terracotta for Ehcache distributed caching is the recommended method of operating Ehcache in a distributed or scaled-out application environment. It provides the highest level of performance, availability, and scalability. As the maintainers of Ehcache, the Terracotta development team has invested million of hours developing Ehcache and its distributed cache capabilities.

To get started, see the Distributed Caching With Terracotta chapter.

20.1.2 Replicated Caching

In addition to the built-in distributed caching, Ehcache has a pluggable cache replication scheme which enables the addition of cache replication mechanisms.

The following additional replicated caching mechanisms are available:

- RMI
- JGroups
- JMS
- Cache Server

Each of the is covered in its own chapter.

One solution is to replicate data between the caches to keep them consistent, or coherent. Typical operations which Applicable operations include:

- put
- update (put which overwrites an existing entry)
- remove

Update supports updateViaCopy or updateViaInvalidate. The latter sends the a remove message out to the cache cluster, so that other caches remove the Element, thus preserving coherency. It is typically a lower cost option than a copy.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

20.1.2.1 Using a Cache Server

Ehcache 1.5 supports the Ehcache Cache Server.

To achieve shared data, all JVMs read to and write from a Cache Server, which runs it in its own JVM.

To achieve redundancy, the Ehcache inside the Cache Server can be set up in its own cluster.

This technique will be expanded upon in Ehcache 1.6.

20.1.2.2 Notification Strategies

The best way of notifying of put and update depends on the nature of the cache.

If the Element is not available anywhere else then the Element itself should form the payload of the notification. An example is a cached web page. This notification strategy is called copy.

Where the cached data is available in a database, there are two choices. Copy as before, or invalidate the data. By invalidating the data, the application tied to the other cache instance will be forced to refresh its cache from the database, preserving cache coherency. Only the Element key needs to be passed over the network.

Ehcache supports notification through copy and invalidate, selectable per cache.

20.1.2.3 Potential Issues with Replicated Caching

20.Potential for Inconsistent Data

Timing scenarios, race conditions, delivery, reliability constraints and concurrent updates to the same cached data can cause inconsistency (and thus a lack of coherency) across the cache instances.

This potential exists within the Ehcache implementation. These issues are the same as what is seen when two completely separate systems are sharing a database; a common scenario.

Whether data inconsistency is a problem depends on the data and how it is used. For those times when it is important, Ehcache provides for synchronous delivery of puts and updates via invalidation. These are discussed below:

20.Synchronous Delivery

Delivery can be specified to be synchronous or asynchronous. Asynchronous delivery gives faster returns to operations on the local cache and is usually preferred. Synchronous delivery adds time to the local operation, however delivery of an update to all peers in the cluster happens before the cache operation returns.

20.Put and Update via Invalidation

The default is to update other caches by copying the new value to them. If the replicatePutsViaCopy property is set to false in the replication configuration, puts are made by removing the element in any other cache peers. If the replicateUpdatesViaCopy property is set to false in the replication configuration, updates are made by removing the element in any other cache peers.

This forces the applications using the cache peers to return to a canonical source for the data.

A similar effect can be obtained by setting the element TTL to a low value such as a second.

Note that these features impact cache performance and should not be used where the main purpose of a cache is performance boosting over coherency.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

20.Use of Time To Idle

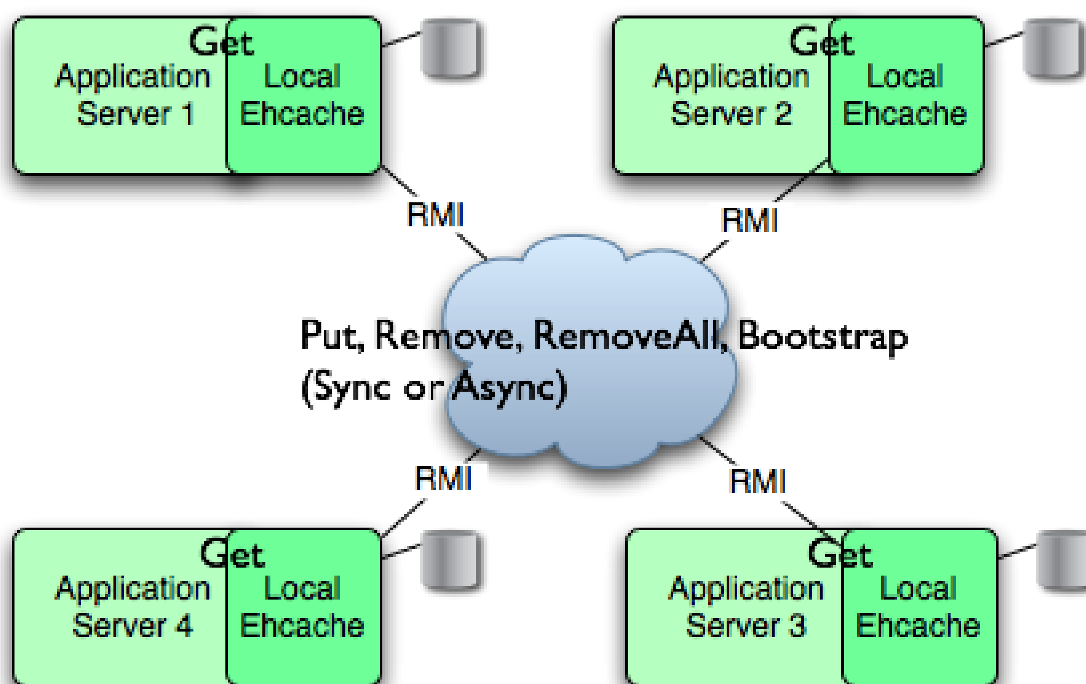
Time To Idle is inconsistent with replicated caching. Time-to-idle makes some entries live longer on some nodes than in others because of cache usage patterns. However, the cache entry "last touched" timestamp is not replicated across the distributed cache.

Do not use Time To Idle with distributed caching, unless you do not care about inconsistent data across nodes.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

21 Replicated Caching With RMI

21.1 RMI Replicated Caching



Since version 1.2, Ehcache has provided replicated caching using RMI.

An RMI implementation is desirable because:

- it itself is the default remoting mechanism in Java
- it is mature
- it allows tuning of TCP socket options
- Element keys and values for disk storage must already be Serializable, therefore directly transmittable over RMI without the need for conversion to a third format such as XML.
- it can be configured to pass through firewalls
- RMI had improvements added to it with each release of Java, which can then be taken advantage of.

While RMI is a point-to-point protocol, which can generate a lot of network traffic, Ehcache manages this through batching of communications for the asynchronous replicator.

To set up RMI replicated caching you need to configure the CacheManager with:

- a PeerProvider
- a CacheManagerPeerListener

  The for each cache that will be replicated, you then need to add one of the RMI cacheEventListener types to propagate messages.

  You can also optionally configure a cache to bootstrap from other caches in the cluster.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

21.1.1 Suitable Element Types

Only Serializable Elements are suitable for replication.

Some operations, such as remove, work off Element keys rather than the full Element itself. In this case the operation will be replicated provided the key is Serializable, even if the Element is not.

21.1.2 Configuring the Peer Provider

21.1.2.1 Peer Discovery

Ehcache has the notion of a group of caches acting as a replicated cache. Each of the caches is a peer to the others. There is no master cache. How do you know about the other caches that are in your cluster? This problem can be given the name Peer Discovery.

Ehcache provides two mechanisms for peer discovery, just like a car: manual and automatic.

To use one of the built-in peer discovery mechanisms specify the class attribute of `cacheManagerPeerProviderFactory` as `net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory` in the ehcache.xml configuration file.

21.1.2.2 Automatic Peer Discovery

Automatic discovery uses TCP multicast to establish and maintain a multicast group. It features minimal configuration and automatic addition to and deletion of members from the group. No a priori knowledge of the servers in the cluster is required. This is recommended as the default option.

Peers send heartbeats to the group once per second. If a peer has not been heard of for 5 seconds it is dropped from the group. If a new peer starts sending heartbeats it is admitted to the group.

Any cache within the configuration set up as replicated will be made available for discovery by other peers.

To set automatic peer discovery, specify the properties attribute of `cacheManagerPeerProviderFactory` as follows:

peerDiscovery=automatic multicastGroupAddress=multicast address | multicast host name multicastGroupPort=port timeToLive=0-255 (See below in common problems before setting this) hostName= the hostname or IP of the interface to be used for sending and receiving multicast packets (relevant to mulithomed hosts only)

21.Example

Suppose you have two servers in a cluster. You wish to distribute sampleCache11 and sampleCache12. The configuration required for each server is identical:

Configuration for server1 and server2

```
<cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
properties="peerDiscovery=automatic, multicastGroupAddress=230.0.0.1,
multicastGroupPort=4446, timeToLive=32"/>
```

21.1.2.3 Manual Peer Discovery

Manual peer configuration requires the IP address and port of each listener to be known. Peers cannot be added or removed at runtime. Manual peer discovery is recommended where there are technical difficulties using multicast, such as a router between servers in a cluster that does not propagate

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

multicast datagrams. You can also use it to set up one way replications of data, by having server2 know about server1 but not vice versa.

To set manual peer discovery, specify the properties attribute of `cacheManagerPeerProviderFactory` as follows: peerDiscovery=manual rmiUrls=//server:port/cacheName, ...

The rmiUrls is a list of the cache peers of the server being configured. Do not include the server being configured in the list.

21.Example

Suppose you have two servers in a cluster. You wish to distribute sampleCache11 and sampleCache12. Following is the configuration required for each server:

Configuration for server1

```
<cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
properties="peerDiscovery=manual,
rmiUrls=//server2:40001/sampleCache11|//server2:40001/sampleCache12"/>
```

Configuration for server2

```
<cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
properties="peerDiscovery=manual,
rmiUrls=//server1:40001/sampleCache11|//server1:40001/sampleCache12"/>
```

21.1.3 Configuring the CacheManagerPeerListener

A CacheManagerPeerListener listens for messages from peers to the current CacheManager.

You configure the CacheManagerPeerListener by specifiying a CacheManagerPeerListenerFactory which is used to create the CacheManagerPeerListener using the plugin mechanism.

The attributes of cacheManagerPeerListenerFactory are:

- class - a fully qualified factory class name * properties - comma separated properties having meaning only to the factory.

  Ehcache comes with a built-in RMI-based distribution system. The listener component is RMICacheManagerPeerListener which is configured using RMICacheManagerPeerListenerFactory. It is configured as per the following example:

  ```
  <cacheManagerPeerListenerFactory
  class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"
  properties="hostName=localhost, port=40001,
  socketTimeoutMillis=2000"/>
  ```

  Valid properties are:

- hostName (optional) - the hostName of the host the listener is running on. Specify where the host is multihomed and you want to control the interface over which cluster messages are received.

  The hostname is checked for reachability during CacheManager initialisation.

  If the hostName is unreachable, the CacheManager will refuse to start and an CacheException will be thrown indicating connection was refused.

  If unspecified, the hostname will use `InetAddress.getLocalHost().getHostAddress()`, which corresponds to the default host network interface.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

Warning: Explicitly setting this to localhost refers to the local loopback of 127.0.0.1, which is not network visible and will cause no replications to be received from remote hosts. You should only use this setting when multiple CacheManagers are on the same machine.

- port (mandatory) - the port the listener listens on.
- socketTimeoutMillis (optional) - the number of seconds client sockets will wait when sending messages to this listener until they give up. By default this is 2000ms.

21.1.4 Configuring Cache Replicators

Each cache that will be replicated needs to set a cache event listener which then replicates messages to the other CacheManager peers. This is done by adding a cacheEventListenerFactory element to each cache's configuration.

```
<!-- Sample cache named sampleCache2. -->
<cache name="sampleCache2"
   maxElementsInMemory="10"
   eternal="false"
   timeToIdleSeconds="100"
   timeToLiveSeconds="100"
   overflowToDisk="false">
 <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"
    properties="replicateAsynchronously=true, replicatePuts=true, replicateUpdates=true
    replicateUpdatesViaCopy=false, replicateRemovals=true "/>
</cache>
```

class - use net.sf.ehcache.distribution.RMICacheReplicatorFactory

The factory recognises the following properties:

- replicatePuts=true | false - whether new elements placed in a cache are replicated to others. Defaults to true.
- replicateUpdates=true | false - whether new elements which override an element already existing with the same key are replicated. Defaults to true.
- replicateRemovals=true - whether element removals are replicated. Defaults to true.
- replicateAsynchronously=true | false - whether replications are asyncrhonous (true) or synchronous (false). Defaults to true.
- replicateUpdatesViaCopy=true | false - whether the new elements are copied to other caches (true), or whether a remove message is sent. Defaults to true.

To reduce typing if you want default behaviour, which is replicate everything in asynchronous mode, you can leave off the `RMICacheReplicatorFactory` properties as per the following example:

```
<!-- Sample cache named sampleCache4. All missing RMICacheReplicatorFactory propert
     default to true -->
<cache name="sampleCache4"
       maxElementsInMemory="10"
       eternal="true"
       overflowToDisk="false"
       memoryStoreEvictionPolicy="LFU">
    <cacheEventListenerFactory
        class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>
</cache>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

21.1.5 Configuring Bootstrap from a Cache Peer

When a peer comes up, it will be incoherent with other caches. When the bootstrap completes it will be partially coherent. Bootstrap gets the list of keys from a random peer, and then loads those in batches from random peers. If bootstrap fails then the Cache will not start (not like this right now). However if a cache replication operation occurs which is then overwritten by bootstrap there is a chance that the cache could be inconsistent.

Here are some scenarios:

Delete overwritten by boostrap put --- Cache A keys with values: 1, 2, 3, 4, 5

Cache B starts bootstrap

Cache A removes key 2

Cache B removes key 2 and then bootstrap puts it back

Put overwritten by boostrap put --- Cache A keys with values: 1, 2, 3, 4, 5

Cache B starts bootstrap

Cache A updates the value of key 2

Cache B updates the value of key 2 and then bootstrap overwrites it with the old value

---

The solution is for bootstrap to get a list of keys and write them all before committing transactions.

This could cause synchronous transaction replicates to back up. To solve this problem, commits will be accepted, but not written to the cache until after bootstrap. Coherency is maintained because the cache is not available until bootstrap has completed and the transactions have been completed.

21.1.6 Full Example

Ehcache's own integration tests provide complete examples of RMI-based replication. The best example is the integration test for cache replication. You can see it online here: http://ehcache.org/xref-test/net/sf/ehcache/distribution/RMICacheReplicatorTest.html

The test uses 5 ehcache.xml's representing 5 CacheManagers set up to replicate using RMI.

21.1.7 Common Problems

21.1.7.1 Tomcat on Windows

There is a bug in Tomcat and/or the JDK where any RMI listener will fail to start on Tomcat if the installation path has spaces in it. See http://archives.java.sun.com/cgi-bin/wa?A2=ind0205&L=rmi-users&P=797 and http://www.ontotext.com/kim/doc/sys-doc/faq-howto-bugs/known-bugs.html.

As the default on Windows is to install Tomcat in "Program Files", this issue will occur by default.

21.1.7.2 Multicast Blocking

The automatic peer discovery process relies on multicast. Multicast can be blocked by routers. Virtualisation technologies like Xen and VMWare may be blocking multicast. If so enable it. You may also need to turn it on in the configuration for your network interface card.

An easy way to tell if your multicast is getting through is to use the Ehcache remote debugger and watch for the heartbeat packets to arrive.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

21.1.7.3 Multicast Not Propagating Far Enough or Propagating Too Far

You can control how far the multicast packets propagate by setting the badly misnamed time to live. Using the multicast IP protocol, the timeToLive value indicates the scope or range in which a packet may be forwarded. By convention:

```
0 is restricted to the same host
1 is restricted to the same subnet
32 is restricted to the same site
64 is restricted to the same region
128 is restricted to the same continent
255 is unrestricted
```

The default value in Java is 1, which propagates to the same subnet. Change the timeToLive property to restrict or expand propagation.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

22 Replicated Caching With JGroups

22.1 Replicated Caching using JGroups

JGroups can be used as the underlying mechanism for the replication operations in ehcache. JGroups offers a very flexible protocol stack, reliable unicast and multicast message transmission. On the down side JGroups can be complex to configure and some protocol stacks have dependencies on others.

To set up replicated caching using JGroups you need to configure a PeerProviderFactory of type JGroupsCacheManagerPeerProviderFactory which is done globally for a CacheManager For each cache that will be replicated, you then need to add a cacheEventListenerFactory of type JGroupsCacheReplicatorFactory to propagate messages.

22.1.1 Suitable Element Types

Only Serializable Elements are suitable for replication.

Some operations, such as remove, work off Element keys rather than the full Element itself. In this case the operation will be replicated provided the key is Serializable, even if the Element is not.

22.1.2 Peer Discovery

If you use the UDP multicast stack there is no additional configuration. If you use a TCP stack you will need to specify the initial hosts in the cluster.

22.1.3 Configuration

There are two things to configure:

- The JGroupsCacheManagerPeerProviderFactory which is done once per CacheManager and therefore once per ehcache.xml file.
- The JGroupsCacheReplicatorFactory which is added to each cache's configuration.

  The main configuration happens in the JGroupsCacheManagerPeerProviderFactory connect sub-property. A connect property is passed directly to the JGroups channel and therefore all the protocol stacks and options available in JGroups can be set.

22.1.4 Example configuration using UDP Multicast

Suppose you have two servers in a cluster. You wish to replicated sampleCache11 and sampleCache12 and you wish to use UDP multicast as the underlying mechanism.

The configuration for server1 and server2 are identical and will look like this:

```
<cacheManagerPeerProviderFactory
     class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory"
     properties="connect=UDP(mcast_addr=231.12.21.132;mcast_port=45566;):PING:
     MERGE2:FD_SOCK:VERIFY_SUSPECT:pbcast.NAKACK:UNICAST:pbcast.STABLE:FRAG:pbcast.GMS"
     propertySeparator="::"
     />
```

22.1.5 Example configuration using TCP Unicast

The TCP protocol requires the IP address of all servers to be known. They are configured through the `TCPPING` protocol of Jgroups.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

Suppose you have 2 servers host1 and host2, then the configuration is:

```
 <cacheManagerPeerProviderFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory"
    properties="connect=TCP(start_port=7800):
        TCPPING(initial_hosts=host1[7800],host2[7800];port_range=10;timeout=3000;
        num_initial_members=3;up_thread=true;down_thread=true):
        VERIFY_SUSPECT(timeout=1500;down_thread=false;up_thread=false):
        pbcast.NAKACK(down_thread=true;up_thread=true;gc_lag=100;retransmit_timeout=300
        pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;shun=false;
        print_local_addr=false;down_thread=true;up_thread=true)"
    propertySeparator="::" />
```

22.1.6 Protocol considerations.

You should read the JGroups documentation to configure the protocols correctly.

See http://www.jgroups.org/javagroupsnew/docs/manual/html_single/index.html.

If using UDP you should at least configure PING, FD_SOCK (Failure detection), VERIFY_SUSPECT, pbcast.NAKACK (Message reliability), pbcast.STABLE (message garbage collection).

22.1.7 Configuring CacheReplicators

Each cache that will be replicated needs to set a cache event listener which then replicates messages to the other CacheManager peers. This is done by adding a cacheEventListenerFactory element to each cache's configuration. The properties are identical to the one used for RMI replication.

The listener factory *MUST* be of type `JGroupsCacheReplicatorFactory`.

```
<!-- Sample cache named sampleCache2. -->
<cache name="sampleCache2"
   maxElementsInMemory="10"
   eternal="false"
   timeToIdleSeconds="100"
   timeToLiveSeconds="100"
   overflowToDisk="false">
   <cacheEventListenerFactory
   class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"
   properties="replicateAsynchronously=true, replicatePuts=true,
   replicateUpdates=true, replicateUpdatesViaCopy=false, replicateRemovals=true" /
>
</cache>
```

The configuration options are explained below:

class - use net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory

The factory recognises the following properties:

- replicatePuts=true | false - whether new elements placed in a cache are replicated to others. Defaults to true.
- replicateUpdates=true | false - whether new elements which override an element already existing with the same key are replicated. Defaults to true.
- replicateRemovals=true - whether element removals are replicated. Defaults to true.
- replicateAsynchronously=true | false - whether replications are asyncrhonous (true) or synchronous (false). Defaults to true.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- replicateUpdatesViaCopy=true | false - whether the new elements are copied to other caches (true), or whether a remove message is sent. Defaults to true.
- asynchronousReplicationIntervalMillis default 1000ms Time between updates when replication is asynchroneous

22.1.8 Complete Sample configuration

A typical complete configuration for one replicated cache configured for UDP will look like:

```
<Ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../main/config/ehcache.xsd">
<diskStore path="java.io.tmpdir/one"/>
<cacheManagerPeerProviderFactory class="net.sf.ehcache.distribution.jgroups
    .JGroupsCacheManagerPeerProviderFactory"
     properties="connect=UDP(mcast_addr=231.12.21.132;mcast_port=45566;ip_ttl=32;
     mcast_send_buf_size=150000;mcast_recv_buf_size=80000):
     PING(timeout=2000;num_initial_members=6):
     MERGE2(min_interval=5000;max_interval=10000):
     FD_SOCK:VERIFY_SUSPECT(timeout=1500):
     pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):
     UNICAST(timeout=5000):
     pbcast.STABLE(desired_avg_gossip=20000):
     FRAG:
     pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;
     shun=false;print_local_addr=true)"
 propertySeparator="::"
     />
<cache name="sampleCacheAsync"
   maxElementsInMemory="1000"
   eternal="false"
   timeToIdleSeconds="1000"
   timeToLiveSeconds="1000"
   overflowToDisk="false">
    <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"
    properties="replicateAsynchronously=true, replicatePuts=true,
replicateUpdates=true, replicateUpdatesViaCopy=false,
replicateRemovals=true" />
</cache>
</ehcache>
```

22.1.9 Common Problems

If replication using JGroups doesn't work the way you have it configured try this configuration which has been extensively tested:

```
<cacheManagerPeerProviderFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory"/
>
<cache name="sampleCacheAsync"
   maxElementsInMemory="1000"
   eternal="false"
   timeToIdleSeconds="1000"
   timeToLiveSeconds="1000"
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
   overflowToDisk="false">
   <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"
    properties="replicateAsynchronously=true, replicatePuts=true,
    replicateUpdates=true, replicateUpdatesViaCopy=false,
    replicateRemovals=true" />
</cache>
```

If this fails to replicate, try to get the example programs from JGroups to run:

http://www.jgroups.org/javagroupsnew/docs/manual/html/ch02.html#d0e451

and

http://www.jgroups.org/javagroupsnew/docs/manual/html/ch02.html#ItDoesntWork

Once you have figured out the connection string that works in your network for JGroups, you can directly paste it in the connect property of `JGroupsCacheManagerPeerProviderFactory`.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

23 Replicated Caching With JMS

23.1 Replicated Caching using JMS

As of version 1.6, JMS can be used as the underlying mechanism for the replicated operations in Ehcache with the jmsreplication module.

JMS, ("Java Message Service") is an industry standard mechanism for interacting with message queues. Message queues themselves are a very mature piece of infrastructure used in many enterprise software contexts. Because they are a required part of the Java EE specification, the large enterprise vendors all provide their own implementations. There are also several open source choices including Open MQ and Active MQ. Ehcache is integration tested against both of these.

The Ehcache jmsreplication module lets organisations with a message queue investment leverage it for caching.

It provides:

- replication between cache nodes using a replication topic, in accordance with ehcache's standard replication mechanism
- pushing of data directly to cache nodes from external topic publishers, in any language. This is done by sending the data to the replication topic, where it automatically picked up by the cache subscribers.
- a JMSCacheLoader, which sends cache load requests to a queue. Either an Ehcache cluster node, or an external queue receiver can respond.

23.1.1 Ehcache Replication and External Publishers

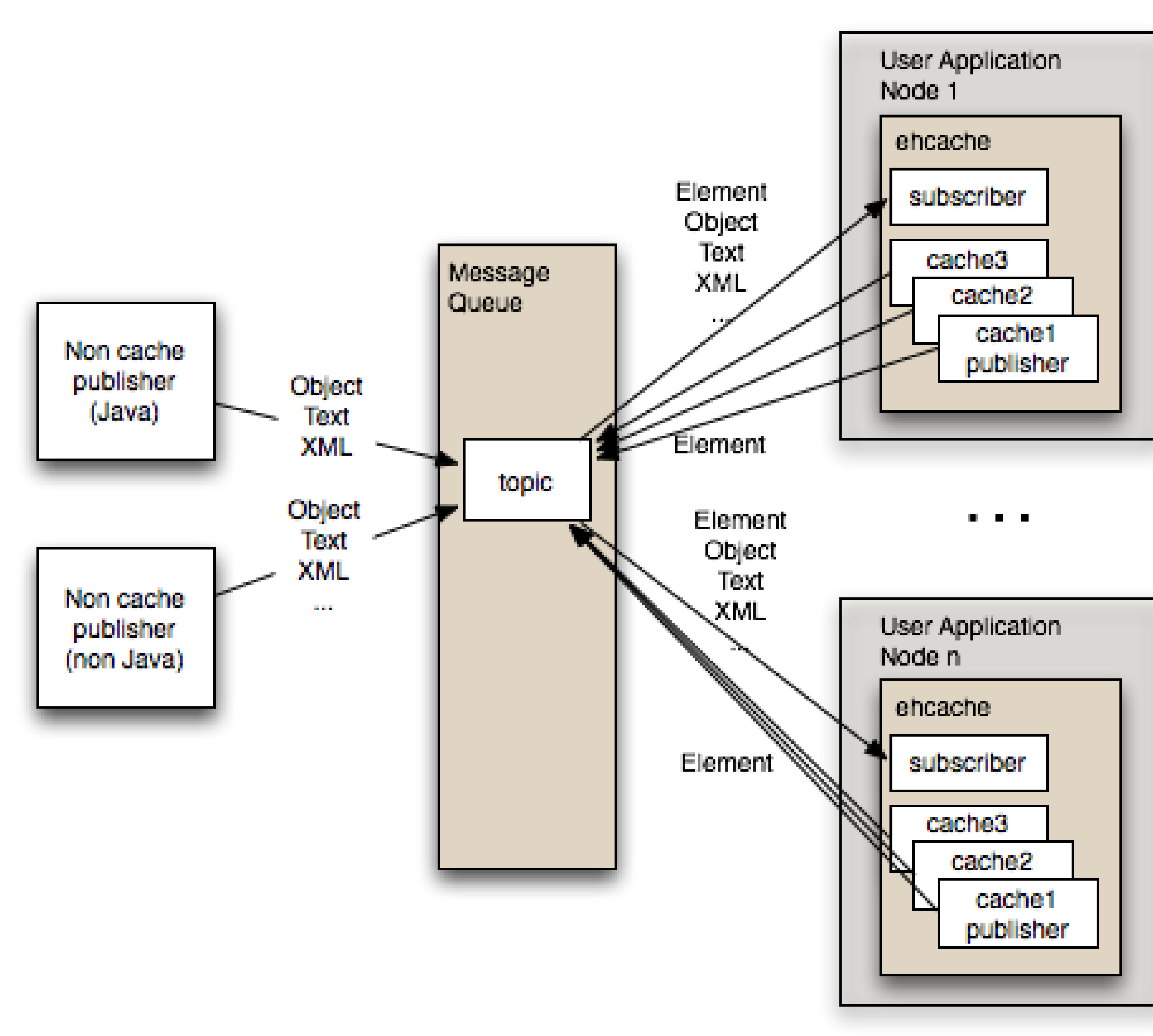
Ehcache replicates using JMS as follows:

- Each cache node subscribes to a predefined topic, configured as the *topicBindingName* in ehcache.xml.
- Each replicated cache publishes cache `Elements` to that topic. Replication is configured per cache.

  To set up replicated caching using JMS you need to configure a JMSCacheManagerPeerProviderFactory which is done globally for a CacheManager.

  For each cache that wishing to replicate, you add a JGroupsCacheReplicatorFactory element to the cache element.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.



23.1.1.1 Configuration

23.Message Queue Configuration

Each cluster needs to use a fixed topic name for replication. Set up a topic using the tools in your message queue. Out of the box, both ActiveMQ and Open MQ support auto creation of destinations, so this step may be optional.

23.Ehcache Configuration

Configuration is done in the ehcache.xml.

There are two things to configure:

- The JMSCacheManagerPeerProviderFactory which is done once per CacheManager and therefore once per ehcache.xml file.
- The JMSCacheReplicatorFactory which is added to each cache's configuration if you want that cache replicated.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

The main configuration happens in the JGroupsCacheManagerPeerProviderFactory connect sub-property. A connect property is passed directly to the JGroups channel and therefore all the protocol stacks and options available in JGroups can be set.

23.Configuring the JMSCacheManagerPeerProviderFactory

Following is the configuration instructions as it appears in the sample ehcache.xml shipped with ehcache:

```
{Configuring JMS replication}.
===========================
<cacheManagerPeerProviderFactory
        class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"
        properties="..."
        propertySeparator=","
        />
The JMS PeerProviderFactory uses JNDI to maintain message queue independence.
Refer to the manual for full configuration examples using ActiveMQ and Open Message Que
Valid properties are:
* initialContextFactoryName (mandatory) - the name of the factory used to create the
    message queue initial context.
* providerURL (mandatory) - the JNDI configuration information for the service provider
use.
* topicConnectionFactoryBindingName (mandatory) - the JNDI binding name for the
    TopicConnectionFactory
* topicBindingName (mandatory) - the JNDI binding name for the topic name
* securityPrincipalName - the JNDI java.naming.security.principal
* securityCredentials - the JNDI java.naming.security.credentials
* urlPkgPrefixes - the JNDI java.naming.factory.url.pkgs
* userName - the user name to use when creating the TopicConnection to the Message Queu
* password - the password to use when creating the TopicConnection to the Message Queue
* acknowledgementMode - the JMS Acknowledgement mode for both publisher and subscriber.
    The available choices are
                        AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE and SESSION_TRANSACTED.
                        The default is AUTO_ACKNOWLEDGE.
* listenToTopic - true or false. If false, this cache will send to the JMS topic but wi
    not listen for updates.
* Default is true.
```

23.Example Configurations

Usage is best illustrated with concrete examples for Active MQ and Open MQ.

23.Configuring the JMSCacheManagerPeerProviderFactory for Active MQ

This configuration works with Active MQ out of the box.

```
<cacheManagerPeerProviderFactory
            class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"
            properties="initialContextFactoryName=ExampleActiveMQInitialContextFactory,
                providerURL=tcp://localhost:61616,
                topicConnectionFactoryBindingName=topicConnectionFactory,
                topicBindingName=ehcache"
            propertySeparator=","
            />
```

You need to provide your own ActiveMQInitialContextFactory for the initialContextFactoryName.

An example which should work for most purposes is:

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
public class ExampleActiveMQInitialContextFactory extends ActiveMQInitialContextFactory
/**
 * {@inheritDoc}
 */
@Override
@SuppressWarnings("unchecked")
public Context getInitialContext(Hashtable environment) throws NamingException {
    Map<String, Object> data = new ConcurrentHashMap<String, Object>();
    String factoryBindingName = (String)environment.get(JMSCacheManagerPeerProviderFact
    .TOPIC_CONNECTION_FACTORY_BINDING_NAME);
    try {
        data.put(factoryBindingName, createConnectionFactory(environment));
    } catch (URISyntaxException e) {
        throw new NamingException("Error initialisating ConnectionFactory with message
        + e.getMessage());
    }
    String topicBindingName = (String)environment.get(JMSCacheManagerPeerProviderFactor
    .TOPIC_BINDING_NAME);
    data.put(topicBindingName, createTopic(topicBindingName));
    return createContext(environment, data);
}
}
```

23.Configuring the JMSCacheManagerPeerProviderFactory for Open MQ

This configuration works with an out of the box Open MQ.

```
<cacheManagerPeerProviderFactory
            class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"
            properties="initialContextFactoryName=com.sun.jndi.fscontext.RefFSContextFa
                providerURL=file:///tmp,
                topicConnectionFactoryBindingName=MyConnectionFactory,
                topicBindingName=ehcache"
            propertySeparator=","
            />
```

To set up the Open MQ file system initial context to work with this example use the following `imqobjmgr` commands to create the requires objects in the context.

```
imqobjmgr add -t tf -l 'MyConnectionFactory' -j java.naming.provider.url \
=file:///tmp -
j java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory -f
imqobjmgr add -t t -l 'ehcache' -o 'imqDestinationName=EhcacheTopicDest'
-j java.naming.provider.url\
=file:///tmp -
j java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory -f
```

23.Configuring the JMSCacheReplicatorFactory

This is the same as configuring any of the cache replicators. The class should be `net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory`.

See the following example:

```
<cache name="sampleCacheAsync"
      maxElementsInMemory="1000"
      eternal="false"
      timeToIdleSeconds="1000"
      timeToLiveSeconds="1000"
      overflowToDisk="false">
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
    <cacheEventListenerFactory
            class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
            properties="replicateAsynchronously=true,
                         replicatePuts=true,
                         replicateUpdates=true,
                         replicateUpdatesViaCopy=true,
                         replicateRemovals=true,
                         asynchronousReplicationIntervalMillis=1000"
             propertySeparator=","/>
</cache>
```

23.1.1.2 External JMS Publishers

Anything that can publish to a message queue can also add cache entries to ehcache. These are called non-cache publishers.

23.Required Message Properties

Publishers need to set up to four String properties on each message: cacheName, action, mimeType and key.

23. `cacheName` Property

A JMS message property which contains the name of the cache to operate on.

If no cacheName is set the message will be *ignored*. A warning log message will indicate that the message has been ignored.

23. `action` Property

A JMS message property which contains the action to perform on the cache.

Available actions are strings labeled `PUT`, `REMOVE` and `REMOVE_ALL`.

If not set no action is performed. A warning log message will indicate that the message has been ignored.

23. `mimeType` Property

A JMS message property which contains the mimeType of the message. Applies to the `PUT` action. If not set the message is interpreted as follows:

ObjectMessage - if it is an net.sf.ehcache.Element, then it is treated as such and stored in the cache.

For other objects, a new Element is created using the object in the ObjectMessage as the value and the key property as a key. Because objects are already typed, the mimeType is ignored.

TextMessage - Stored in the cache as value of MimeTypeByteArray. The mimeType should be specified. If not specified it is stored as type `text/plain`.

BytesMessage - Stored in the cache as value of MimeTypeByteArray. The mimeType should be specified. If not specified it is stored as type `application/octet-stream`.

Other message types are not supported.

To send XML use a TextMessage or BytesMessage and set the mimeType to `application/xml`.It will be stored in the cache as a value of MimeTypeByteArray.

The `REMOVE` and `REMOVE_ALL` actions do not require a `mimeType` property.

23. `key` Property

The key in the cache on which to operate on. The key is of type String.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

The REMOVE_ALL action does not require a key property.

If an ObjectMessage of type net.sf.ehcache.Element is sent, the key is contained in the element. Any key set as a property is ignored.

If the key is required but not provided, a warning log message will indicate that the message has been ignored.

23. Code Samples

These samples use Open MQ as the message queue and use it with out of the box defaults. They are heavily based on Ehcache's own JMS integration tests. See the test source for more details.

Messages should be sent to the topic that Ehcache is listening on. In these samples it is EhcacheTopicDest.

All samples get a Topic Connection using the following method:

```
private TopicConnection getMQConnection() throws JMSException {
    com.sun.messaging.ConnectionFactory factory = new com.sun.messaging.ConnectionFacto
    factory.setProperty(ConnectionConfiguration.imqAddressList, "localhost:7676");
    factory.setProperty(ConnectionConfiguration.imqReconnectEnabled, "true");
    TopicConnection myConnection = factory.createTopicConnection();
    return myConnection;
}
```

23.PUT a Java Object into an Ehcache JMS Cluster

```
String payload = "this is an object";
TopicConnection connection = getMQConnection();
connection.start();
TopicSession publisherSession =
    connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
ObjectMessage message = publisherSession.createObjectMessage(payload);
message.setStringProperty(ACTION_PROPERTY, "PUT");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");
//don't set. Should work.
//message.setStringProperty(MIME_TYPE_PROPERTY, null);
//should work. Key should be ignored when sending an element.
message.setStringProperty(KEY_PROPERTY, "1234");
Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);
connection.stop();
```

Ehcache will create an Element in cache "sampleCacheAsync" with key "1234" and a Java class String value of "this is an object".

23.PUT XML into an Ehcache JMS Cluster

```
TopicConnection connection = getMQConnection();
connection.start();
TopicSession publisherSession = connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
String value = "<?xml version=\"1.0\"?>\n" +
        "<oldjoke>\n" +
        "<burns>Say <quote>goodnight</quote>,\n" +
        "Gracie.</burns>\n" +
        "<allen><quote>Goodnight, \n" +
        "Gracie.</quote></allen>\n" +
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
        "<applause/>\n" +
        "</oldjoke>";
TextMessage message = publisherSession.createTextMessage(value);
message.setStringProperty(ACTION_PROPERTY, "PUT");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");
message.setStringProperty(MIME_TYPE_PROPERTY, "application/xml");
message.setStringProperty(KEY_PROPERTY, "1234");
Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);
connection.stop();
```

Ehcache will create an Element in cache "sampleCacheAsync" with key "1234" and a value of type MimeTypeByteArray.

On a get from the cache the MimeTypeByteArray will be returned. It is an Ehcache value object from which a mimeType and byte[] can be retrieved. The mimeType will be "application/xml". The byte[] will contain the XML String encoded in bytes, using the platform's default charset.

23.PUT arbitrary bytes into an Ehcache JMS Cluster

```
byte[] bytes = new byte[]{0x34, (byte) 0xe3, (byte) 0x88};
TopicConnection connection = getMQConnection();
connection.start();
TopicSession publisherSession = connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
BytesMessage message = publisherSession.createBytesMessage();
message.writeBytes(bytes);
message.setStringProperty(ACTION_PROPERTY, "PUT");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");
message.setStringProperty(MIME_TYPE_PROPERTY, "application/octet-stream");
message.setStringProperty(KEY_PROPERTY, "1234");
Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);
```

Ehcache will create an Element in cache "sampleCacheAsync" with key "1234" in and a value of type MimeTypeByteArray.

On a get from the cache the MimeTypeByteArray will be returned. It is an Ehcache value object from which a mimeType and byte[] can be retrieved. The mimeType will be "application/octet-stream". The byte[] will contain the original bytes.

23.REMOVE

```
TopicConnection connection = getMQConnection();
connection.start();
TopicSession publisherSession = connection.createTopicSession(false, Session.AUTO_ACKNO
ObjectMessage message = publisherSession.createObjectMessage();
message.setStringProperty(ACTION_PROPERTY, "REMOVE");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");
message.setStringProperty(KEY_PROPERTY, "1234");
Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);
```

Ehcache will remove the Element with key "1234" from cache "sampleCacheAsync" from the cluster.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

23.REMOVE_ALL

```
TopicConnection connection = getMQConnection();
connection.start();
TopicSession publisherSession = connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
ObjectMessage message = publisherSession.createObjectMessage();
message.setStringProperty(ACTION_PROPERTY, "REMOVE_ALL");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");
Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);
connection.stop();
```

Ehcache will remove all Elements from cache "sampleCacheAsync" in the cluster.

23.1.2 Using the JMSCacheLoader

The JMSCacheLoader is a CacheLoader which loads objects into the cache by sending requests to a JMS Queue.

The loader places an ObjectMessage of type JMSEventMessage on the getQueue with an Action of type GET.

It is configured with the following String properties, `loaderArgument`.

The defaultLoaderArgument, or the loaderArgument if specified on the load request. To work with the JMSCacheManagerPeerProvider this should be the name of the cache to load from. For custom responders, it can be anything which has meaning to the responder.

A queue responder will respond to the request. You can either create your own or use the one built-into the JMSCacheManagerPeerProviderFactory, which attempts to load the queue from its cache.

The JMSCacheLoader uses JNDI to maintain message queue independence. Refer to the manual for full configuration examples using ActiveMQ and Open Message Queue.

It is configured as per the following example:

```
<cacheLoaderFactory class="net.sf.ehcache.distribution.jms.JMSCacheLoaderFactory"
   properties="initialContextFactoryName=com.sun.jndi.fscontext.RefFSContextFactory,
providerURL=file:///tmp,
replicationTopicConnectionFactoryBindingName=MyConnectionFactory,
replicationTopicBindingName=ehcache,
getQueueConnectionFactoryBindingName=queueConnectionFactory,
getQueueBindingName=ehcacheGetQueue,
timeoutMillis=20000
defaultLoaderArgument=/>
```

Valid properties are:

- initialContextFactoryName (mandatory) - the name of the factory used to create the message queue initial context.
- providerURL (mandatory) - the JNDI configuration information for the service provider to use.
- getQueueConnectionFactoryBindingName (mandatory) - the JNDI binding name for the QueueConnectionFactory
- getQueueBindingName (mandatory) - the JNDI binding name for the queue name used to do make requests.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- defaultLoaderArgument - (optional) - an application specific argument. If not supplied as a cache.load() parameter this default value will be used. The argument is passed in the JMS request as a StringProperty called loaderArgument.
- timeoutMillis - time in milliseconds to wait for a reply.
- securityPrincipalName - the JNDI java.naming.security.principal
- securityCredentials - the JNDI java.naming.security.credentials
- urlPkgPrefixes - the JNDI java.naming.factory.url.pkgs
- userName - the user name to use when creating the TopicConnection to the Message Queue
- password - the password to use when creating the TopicConnection to the Message Queue
- acknowledgementMode - the JMS Acknowledgement mode for both publisher and subscriber. The available choices are AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE and SESSION_TRANSACTED. The default is AUTO_ACKNOWLEDGE.

23.1.2.1 Example Configuration Using Active MQ

```
<cache name="sampleCacheNorep"
   maxElementsInMemory="1000"
   eternal="false"
   timeToIdleSeconds="1000"
   timeToLiveSeconds="1000"
   overflowToDisk="false">
   <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
        properties="replicateAsynchronously=false, replicatePuts=false,
        replicateUpdates=false, replicateUpdatesViaCopy=false,
        replicateRemovals=false, loaderArgument=sampleCacheNorep"
        propertySeparator=","/>
   <cacheLoaderFactory class="net.sf.ehcache.distribution.jms.JMSCacheLoaderFactory"
    properties="initialContextFactoryName=net.sf.ehcache.distribution.jms.
        TestActiveMQInitialContextFactory,
            providerURL=tcp://localhost:61616,
            replicationTopicConnectionFactoryBindingName=topicConnectionFactory,
            getQueueConnectionFactoryBindingName=queueConnectionFactory,
            replicationTopicBindingName=ehcache,
            getQueueBindingName=ehcacheGetQueue,
            timeoutMillis=10000"/>
</cache>
```

23.1.2.2 Example Configuration Using Open MQ

```
<cache name="sampleCacheNorep"
   maxElementsInMemory="1000"
   eternal="false"
   timeToIdleSeconds="100000"
   timeToLiveSeconds="100000"
   overflowToDisk="false">
   <cacheEventListenerFactory
   class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
                        properties="replicateAsynchronously=false, replicatePuts=fal
                          replicateUpdates=false, replicateUpdatesViaCopy=false,
                          replicateRemovals=false"
                        propertySeparator=","/>
   <cacheLoaderFactory class="net.sf.ehcache.distribution.jms.JMSCacheLoaderFactory"
```

```
        properties="initialContextFactoryName=com.sun.jndi.fscontext.RefFSContextFactory,
        providerURL=file:///tmp,
        replicationTopicConnectionFactoryBindingName=MyConnectionFactory,
        replicationTopicBindingName=ehcache,
        getQueueConnectionFactoryBindingName=queueConnectionFactory,
        getQueueBindingName=ehcacheGetQueue,
        timeoutMillis=10000,
        userName=test,
        password=test"/>
</cache>
```

23.1.3 Configuring Clients for Message Queue Reliability

Ehcache replication and cache loading is designed to gracefully degrade if the message queue infrastructure stops. Replicates and loads will fail. But when the message queue comes back, these operations will start up again.

For this to work, the ConnectionFactory used with the specific message queue needs to be configured correctly.

For example, with Open MQ, reconnection is configured as follows:

- imqReconnect='true' - without this reconnect will not happen
- imqPingInterval='5' - Consumers will not reconnect until they notice the connection is down. The ping interval
- does this. The default is 30. Set it lower if you want the Ehcache cluster to reform more quickly.
- Finally, unlimited retry attempts are recommended. This is also the default.

  For greater reliability consider using a message queue cluster. Most message queues support clustering. The cluster configuration is once again placed in the ConnectionFactory configuration.

23.1.4 Tested Message Queues

23.1.4.1 Sun Open MQ

This open source message queue is tested in integration tests. It works perfectly.

23.1.4.2 Active MQ

This open source message queue is tested in integration tests. It works perfectly other than having a problem with temporary reply queues which prevents the use of JMSCacheLoader. JMSCacheLoader is not used during replication.

23.1.4.3 Oracle AQ

Versions up to an including 0.4 do not work, due to Oracle not supporting the unified API (send) for topics.

23.1.4.4 JBoss Queue

Works as reported by a user.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

23.1.5 Known JMS Issues

23.1.5.1 Active MQ Temporary Destinatons

ActiveMQ seems to have a bug in at least ActiveMQ 5.1 where it does not cleanup temporary queues, even though they have been deleted. That bug appears to be long standing but was though to have been fixed.

See:

- http://www.nabble.com/Memory-Leak-Using-Temporary-Queues-td11218217.html#a11218217
- http://issues.apache.org/activemq/browse/AMQ-1255

  The JMSCacheLoader uses temporary reply queues when loading. The Active MQ issue is readily reproduced in Ehcache integration testing. Accordingly, use of the JMSCacheLoader with ActiveMQ is not recommended. Open MQ tests fine.

  Active MQ works fine for replication.

23.1.5.2 WebSphere 5 and 6

Websphere Application Server prevents MessageListeners, which are not MDBs, from being created in the container. While this is a general Java EE limitation, most other app servers either are permissive or can be configured to be permissive. WebSphere 4 worked, but 5 and 6 enforce the restriction.

Accordingly the JMS replicator cannot be used with WebSphere 5 and 6.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

24 Shutting Down Ehcache

24.1 Shutting Down Ehcache

If you are using persistent disk stores, or distributed caching, care should be taken to shutdown ehcache.

Note that Hibernate automatically shuts down its Ehcache `CacheManager`.

The recommended way to shutdown the Ehcache is:

- to call `CacheManager.shutdown()`
- in a web app, register the Ehcache `ShutdownListener`

  Though not recommended, Ehcache also lets you register a JVM shutdown hook.

24.1.1 ServletContextListener

Ehcache proivdes a ServletContextListener that shutsdown CacheManager. Use this when you want to shutdown Ehcache automatically when the web application is shutdown.

To receive notification events, this class must be configured in the deployment descriptor for the web application.

To do so, add the following to web.xml in your web application:

```
     <listener>
          <listener-class>net.sf.ehcache.constructs.web.ShutdownListener</
listener-class>
     </listener>
```

24.1.2 The Shutdown Hook

Ehcache CacheManager can optionally register a shutdown hook.

To do so, set the system property `net.sf.ehcache.enableShutdownHook=true`.

This will shutdown the CacheManager when it detects the Virtual Machine shutting down and it is not already shut down.

24.1.2.1 When to use the shutdown hook

Use the shutdown hook where:

- you need guaranteed orderly shutdown, when for example using persistent disk stores, or distributed caching.
- CacheManager is not already being shutdown by a framework you are using or by your application.

  Having said that, shutdown hooks are inherently dangerous. The JVM is shutting down, so sometimes things that can never be null are. Ehcache guards against as many of these as it can, but the shutdown hook should be the last option to use.

24.1.2.2 What the shutdown hook does

The shutdown hook is on CacheManager. It simply calls the shutdown method.

The sequence of events is:

- call dispose for each registered CacheManager event listener

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- call dispose for each Cache.
  Each Cache will:
  - shutdown the MemoryStore. The MemoryStore will flush to the DiskStore
  - shutdown the DiskStore. If the DiskStore is persistent, it will write the entries and index to disk.
  - shutdown each registered CacheEventListener
  - set the Cache status to shutdown, preventing any further operations on it.
- set the CacheManager status to shutdown, preventing any further operations on it

24.1.2.3 When a shutdown hook will run, and when it will not

The shutdown hook runs when:

- a program exists normally. e.g. System.exit() is called, or the last non-daemon thread exits
- the Virtual Machine is terminated. e.g. CTRL-C. This corresponds to `kill -SIGTERM pid` or `kill -15 pid` on Unix systems.

The shutdown hook will not run when:

- the Virtual Machine aborts
- A SIGKILL signal is sent to the Virtual Machine process on Unix systems. e.g. `kill -SIGKILL pid` or `kill -9 pid`
- A `TerminateProcess` call is sent to the process on Windows systems.

24.1.3 Dirty Shutdown

If Ehcache is shutdown dirty then any persistent disk stores will be corrupted. They will be deleted, with a log message, on the next startup.

Replications waiting to happen to other nodes in a distributed cache will also not get written.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

25 Logging

25.1 Logging

25.1.1 SLF4J Logging

As of 1.7.1, Ehcache uses the the slf4j (http://www.slf4j.org) logging facade. Plug in your own logging framework.

25.1.1.1 Concrete Logging Implementation Use in Maven

With slf4j, users must choose a concrete logging implementation at deploy time.

The maven dependency declarations are reproduced here for convenience. Add *one* of these to your Maven pom.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>1.5.8</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.5.8</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.5.8</version>
</dependency>
```

25.1.1.2 Concrete Logging Implemenation Use in the Download Kit

We provide the slf4j-api and slf4j-jdk14 jars in the kit along with the ehcache jars so that, if the app does not already use SLF4J, you have everything you need.

Additional concrete logging implementations can be downloaded from http://www.slf4j.org.

25.1.2 Recommended Logging Levels

Ehcache seeks to trade off informing production support developers or important messages and cluttering the log.

ERROR ERROR messages should not occur in normal production and indicate that action should be taken.

WARN WARN messages generally indicate a configuration change should be made or an unusual event has occurred.

DEBUG DEBUG and TRACE messages are for development use. All DEBUG level statements are surrounded with a guard so that no performance cost is incurred unless the logging level is set.

Setting the logging level to DEBUG should provide more information on the source of any problems. Many logging systems enable a logging level change to be made without restarting the application.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

26 Remote Network replication debugging: RMI Replicated Caches

26.1 Remote Network debugging and monitoring for Distributed Caches

26.1.1 Introduction

The ehcache-1.x-remote-debugger.jar} can be used to debug replicated cache operations. When started with the same configuration as the cluster, it will join the cluster and then report cluster events for the cache of interest. By providing a window into the cluster it can help to identify the cause of cluster problems.

26.1.2 Packaging

From version 1.5 it is packaged in its own distribution tarball along with a maven module.

It is provided as an executable jar.

26.1.3 Limitations

This version of the debugger has been tested only with the default RMI based replication.

26.1.4 Usage

It is invoked as follows:

```
java -classpath [add your application jars here]
 -jar ehcache-debugger-1.5.0.jar ehcache.xml sampleCache1
    path_to_ehcache.xml [cacheToMonitor]
```

Note: Add to the classpath any libraries your project uses in addition to these above, otherwise RMI will attempt to load them remotely which requires specific security policy settings that surprise most people.

It takes one or two arguments:

- the first argument, which is mandatory, is the Ehcache configuration file e.g. app/config/ehcache.xml. This file should be configured to allow Ehcache to joing the cluster. Using one of the existing ehcache.xml files from the other nodes normally is sufficient.
- the second argument, which is optional, is the name of the cache e.g. distributedCache1

  If only the first argument is passed, it will print our a list of caches with replication configured from the configuration file, which are then available for monitoring.

  If the second argument is also provided, the debugger will monitor cache operations received for the given cache.

  This is done by registering a CacheEventListener which prints out each operation.

26.1.4.1 Output

When monitoring a cache it prints a list of caches with replication configured, prints notifications as they happen, and periodically prints the cache name, size and total events received. See sample output below which is produced when the RemoteDebuggerTest is run.

```
Caches with replication configured which are available for monitoring are:
sampleCache19 sampleCache20 sampleCache26 sampleCache42 sampleCache33
sampleCache51 sampleCache40 sampleCache32 sampleCache18 sampleCache25
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
sampleCache9 sampleCache15 sampleCache56 sampleCache31 sampleCache7
sampleCache12 sampleCache17 sampleCache45 sampleCache41 sampleCache30
sampleCache13 sampleCache46 sampleCache4 sampleCache36 sampleCache29
sampleCache50 sampleCache37 sampleCache49 sampleCache48 sampleCache38
sampleCache6 sampleCache2 sampleCache55 sampleCache16 sampleCache27
sampleCache11 sampleCache3 sampleCache54 sampleCache28 sampleCache10
sampleCache8 sampleCache47 sampleCache5 sampleCache53 sampleCache39
sampleCache23 sampleCache34 sampleCache22 sampleCache44 sampleCache52
sampleCache24 sampleCache35 sampleCache21 sampleCache43 sampleCache1
Monitoring cache: sampleCache1
Cache: sampleCache1 Notifications received: 0 Elements in cache: 0
Received put notification for element [ key = this is an id, value=this is
a value, version=1, hitCount=0, CreationTime = 1210656023456,
LastAccessTime = 0 ]
Received update notification for element [ key = this is an id, value=this
is a value, version=1210656025351, hitCount=0, CreationTime =
1210656024458, LastAccessTime = 0 ]
Cache: sampleCache1 Notifications received: 2 Elements in cache: 1
Received remove notification for element this is an id
Received removeAll notification.
```

26.1.4.2 Providing more Detailed Logging

If you see nothing happening, but cache operations should be going through, enable trace (LOG4J) or finest (JDK) level logging on *code*net.sf.ehcache.distribution */code* in the logging configuration being used by the debugger. A large volume of log messages will appear. The normal problem is that the CacheManager has not joined the cluster. Look for the list of cache peers.

26.1.4.3 Yes, but I still have a cluster problem

Check the FAQ where a lot of commonly reported errors and their solutions are provided. Beyond that, post to the forums or mailing list or contact Ehcache for support.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

27 JMX Management And Monitoring

27.1 JMX Management and Monitoring

27.1.1 Terracotta Monitoring Products

An extensive monitoring product, available in Enterprise Ehcache, provides a monitoring server with probes supporting Ehcache-1.2.3 and higher for standalone and clustered Ehcache. It comes with a web console and a RESTful API for operations integration.

See the ehcache-monitor documentation for more information.

When using Ehcache 1.7 with Terracotta clustering, the Terracotta Developer Console shows statistics for Ehcache.

27.1.2 JMX Overview

JMX, part of JDK1.5, and available as a download for 1.4, creates a standard way of instrumenting classes and making them available to a management and monitoring infrastructure.

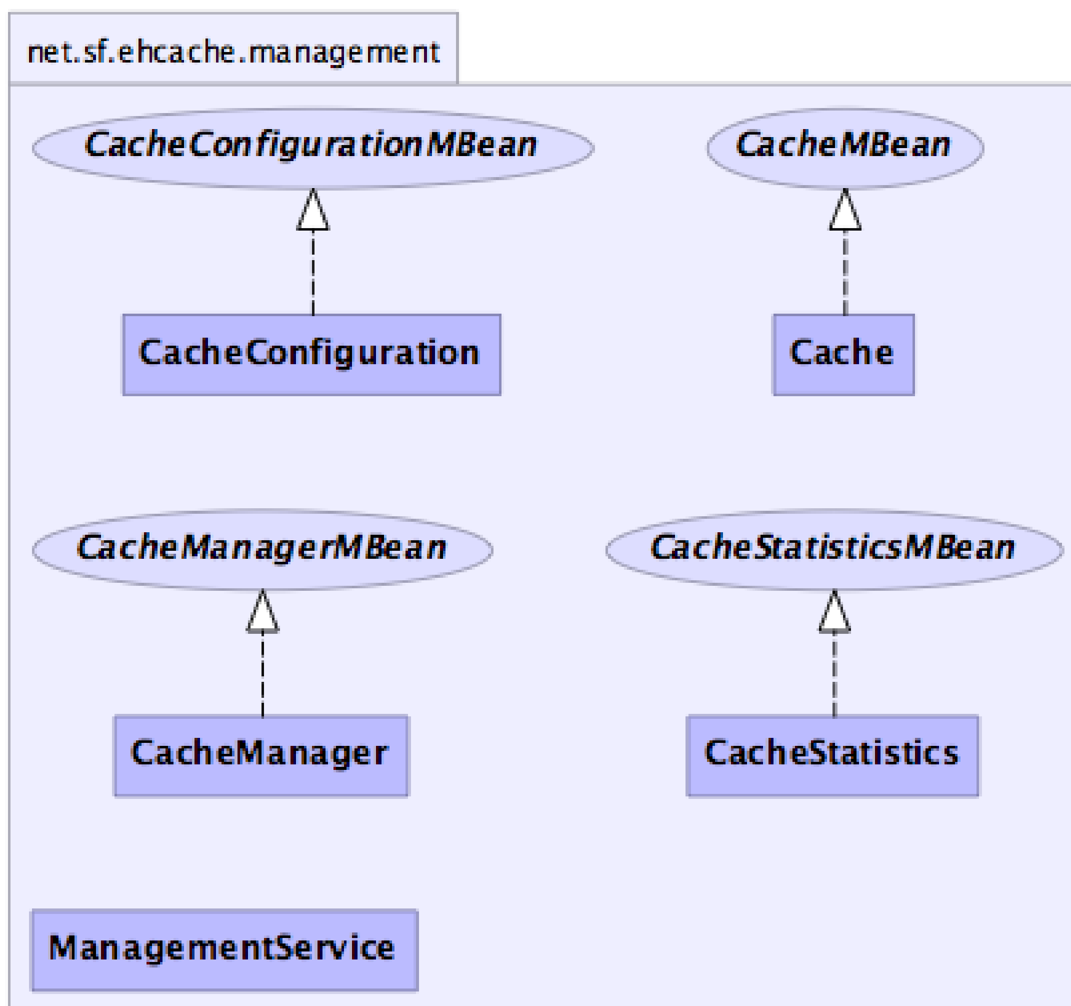
The `net.sf.ehcache.management` package contains MBeans and a `ManagementService` for JMX management of ehcache. It is in a separate package so that JMX libraries are only required if you wish to use it - there is no leakage of JMX dependencies into the core Ehcache package.

This implementation attempts to follow Sun's JMX best practices. See http://java.sun.com/javase/technologies/core/mntr-mgmt/ javamanagement/best-practices.jsp.

Use `net.sf.ehcache.management.ManagementService.registerMBeans(...)` static method to register a selection of MBeans to the MBeanServer provided to the method.

If you wish to monitor Ehcache but not use JMX, just use the existing public methods on `Cache` and `CacheStatistics`.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.



The Management Package

27.1.3 MBeans

Ehcache uses Standard MBeans. MBeans are available for the following:

- CacheManager
- Cache
- CacheConfiguration
- CacheStatistics

  All MBean attributes are available to a local MBeanServer. The CacheManager MBean allows traversal to its collection of Cache MBeans. Each Cache MBean likewise allows traversal to its CacheConfiguration MBean and its CacheStatistics MBean.

27.1.4 JMX Remoting

The JMX Remote API allows connection from a remote JMX Agent to an MBeanServer via an `MBeanServerConnection`.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

Only `Serializable` attributes are available remotely. The following Ehcache MBean attributes are available remotely:

- limited CacheManager attributes
- limited Cache attributes
- all CacheConfiguration attributes
- all CacheStatistics attributes

  Most attributes use built-in types. To access all attributes, you need to add ehcache.jar to the remote JMX client's classpath e.g. `jconsole -J-Djava.class.path=ehcache.jar`.

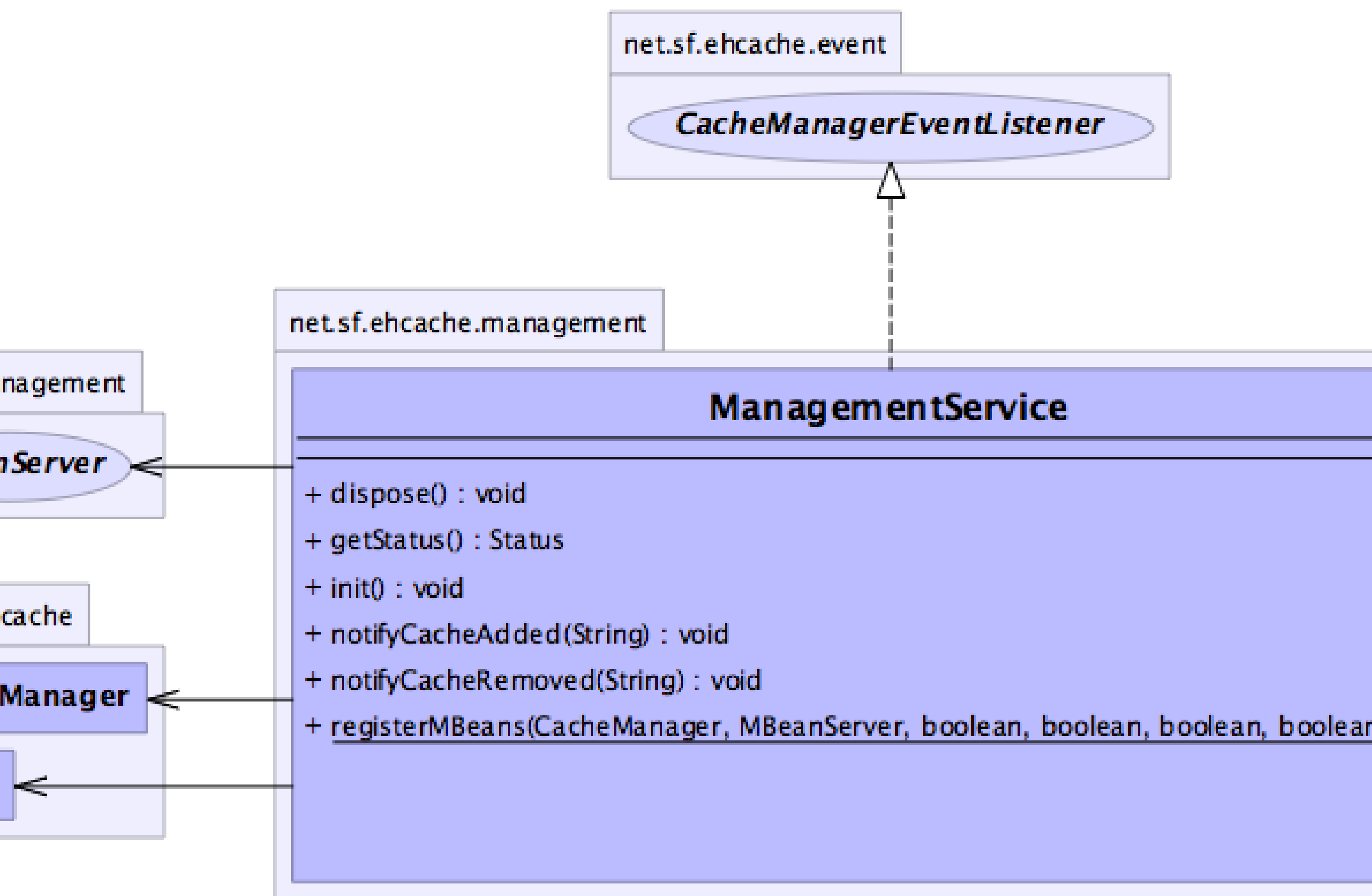
27.1.5 `ObjectName` naming scheme

- CacheManager - "net.sf.ehcache:type=CacheManager,name= *CacheManager*"
- Cache - "net.sf.ehcache:type=Cache,CacheManager= *cacheManagerName*,name= *cacheName*"
- CacheConfiguration - "net.sf.ehcache:type=CacheConfiguration,CacheManager= *cacheManagerName*,name= *cacheName*"
- CacheStatistics - "net.sf.ehcache:type=CacheStatistics,CacheManager= *cacheManagerName*,name= *cacheName*"

27.1.6 The Management Service

The `ManagementService` class is the API entry point.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.



ManagementService

There is only one method, `ManagementService.registerMBeans` which is used to initiate JMX registration of an Ehcache CacheManager's instrumented MBeans.

The `ManagementService` is a `CacheManagerEventListener` and is therefore notified of any new Caches added or disposed and updates the MBeanServer appropriately.

Once initiated the MBeans remain registered in the MBeanServer until the CacheManager shuts down, at which time the MBeans are deregistered. This behaviour ensures correct behaviour in application servers where applications are deployed and undeployed.

```
/**
 * This method causes the selected monitoring options to be be registered
 * with the provided MBeanServer for caches in the given CacheManager.
 * <p/>
 * While registering the CacheManager enables traversal to all of the other
 *  items,
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 * this requires programmatic traversal. The other options allow entry points closer
 * to an item of interest and are more accessible from JMX management tools like JConso
 * Moreover CacheManager and Cache are not serializable, so remote monitoring is not
 * possible * for CacheManager or Cache, while CacheStatistics and CacheConfiguration a
 * Finally * CacheManager and Cache enable management operations to be performed.
 * <p/>
 * Once monitoring is enabled caches will automatically added and removed from the
 * MBeanServer * as they are added and disposed of from the CacheManager. When the
 * CacheManager itself * shutsdown all registered MBeans will be unregistered.
 *
 * @param cacheManager the CacheManager to listen to
 * @param mBeanServer the MBeanServer to register MBeans to
 * @param registerCacheManager Whether to register the CacheManager MBean
 * @param registerCaches Whether to register the Cache MBeans
 * @param registerCacheConfigurations Whether to register the CacheConfiguration MBeans
 * @param registerCacheStatistics Whether to register the CacheStatistics MBeans
 */
public static void registerMBeans(
        net.sf.ehcache.CacheManager cacheManager,
        MBeanServer mBeanServer,
        boolean registerCacheManager,
        boolean registerCaches,
        boolean registerCacheConfigurations,
        boolean registerCacheStatistics) throws CacheException {
```

27.1.7 JConsole Example

This example shows how to register CacheStatistics in the JDK1.5 platform MBeanServer, which works with the JConsole management agent.

```
CacheManager manager = new CacheManager();
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
ManagementService.registerMBeans(manager, mBeanServer, false, false, false, true);
```

CacheStatistics MBeans are then registered.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

CacheStatistics MBeans in JConsole

27.1.8 Hibernate statistics

If you are running Terracotta clustered caches as hibernate second-level cache provider, it is possible to access the hibernate statistics + ehcache stats etc via jmx.

`EhcacheHibernateMBean` is the main interface that exposes all the API's via jmx. It basically extends two interfaces -- `EhcacheStats` and `HibernateStats`. And as the name implies `EhcacheStats` contains methods related with Ehcache and `HibernateStats` related with Hibernate.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

You can see cache hit/miss/put rates, change config element values dynamically -- like maxElementInMemory, TTI, TTL, enable/disable statistics collection etc and various other things. Please look into the specific interface for more details.

27.1.9 JMX Tutorial

See http://weblogs.java.net/blog/maxpoon/archive/2007/06/extending_the_n_2.html for an online tutorial.

27.1.10 Performance

Collection of cache statistics is not entirely free of overhead. In production systems where monitoring is not required statistics can be disabled. This can be done either programatically by calling setStatisticsEnabled(false) on the cache instance, or in configuration by setting the statistics="false" attribute of the relevant cache configuration element.

From Ehcache 2.1.0 statistics are off by default.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

28 JTA And Transactions

28.1 Transactions in Ehcache

28.1.1 Introduction

Transactions are supported in versions of Ehcache 2.0 and higher.

The 2.3.x or lower releases only support XA. However since ehcache 2.4 support for both Global Transactions with `xa_strict` and `xa` modes, and Local Transactions with `local` mode has been added.

28.1.1.1 All or nothing

If a cache is enabled for transactions, all operations on it must happen within a transaction context otherwise a `TransactionException` will be thrown.

28.1.1.2 Change Visibility

The isolation level offered in Ehcache is `READ_COMMITTED`. Ehcache can work as an XAResource in which case, full two-phase commit is supported.

Specifically:

- All mutating changes to the cache are transactional including `put`, `remove`, `putWithWriter`, `removeWithWriter` and `removeAll`.
- Mutating changes are not visible to other transactions in the local JVM or across the cluster until `COMMIT` has been called.
- Until then, read such as by `cache.get(...)` by other transactions will return the old copy. Reads do not block.

28.1.2 When to use transactional modes

28.1.2.1 Transactional modes are a powerful extension of Ehcache allowing you to perform atomic operations on your caches and potentially other data stores, eg: to keep your cache in sync with your database.

- `local` When you want your changes across multiple caches to be performed atomically.

  Use this mode when you need to update your caches atomically, ie: have all your changes be committed or rolled back using a straight simple API. This mode is most useful when a cache contains data calculated out of other cached data.
- `xa`

  Use this mode when you cache data from other data stores (eg: DBMS, JMS) and want to do it in an atomic way under the control of the JTA API but don't want to pay the price of full two-phase commit. In this mode, your cached data can get out of sync with the other resources participating in the transactions in case of a crash so only use it if you can afford to live with stale data for a brief period of time.
- `xa_strict`

  Same as xa but use it only if you need strict XA disaster recovery guarantees. In this mode, the cached data can never get out of sync with the other resources participating in the transactions, even in case of a crash but you pay a high price in performance to get that extra safety.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

28.1.3 Requirements

The objects you are going to store in your transactional cache must:

- implement `java.io.Serializable`

  This is required to store cached objects when the cache is clustered with Terracotta but it's also required by the copy on read / copy on write mechanism used to implement isolation.
- override `equals` and `hashcode`

  Those must be overridden as the transactional stores rely on element value comparison, see: `ElementValueComparator` and the `elementValueComparator` configuration setting.

28.1.4 Configuration

Transactions are enabled on a cache by cache basis with the `transactionalMode` cache attribute.

The allowed values are:

- `xa_strict`
- `xa`
- `local`
- `off`

  The default value is off.

  Enabling a cache for `xa_strict` transactions is shown in the following example:

```
<cache name="xaCache"
    maxElementsInMemory="500"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="false"
    diskPersistent="false"
    diskExpiryThreadIntervalSeconds="1"
    transactionalMode="xa_strict">
</cache>
```

28.1.4.1 Considerations when using clustered caches with Terracotta

For Terracotta clustered caches, `transactionalMode` can only be used where `terracotta consistency="strong"`. Because caches can be dynamically changed into consistencyType=eventual mode for bulk loading, any attempt to perform a transaction when this is the case will throw a `CacheException`.

Note that transactions do not work with Terracotta's `identity` mode. An attempt to initialise a transactional cache when this mode is set will result in a `CacheException` being thrown. The default mode is `serialization` mode.

Also note that all transactional modes are currently sensitive to the ABA problem.

28.1.5 Global Transactions

Global Transactions are supported by Ehcache. Ehcache can act as an XAResouce to participate in JTA ("Java Transaction API") transactions under the control of a Transaction Manager. This is typically provided by your application server, however you may also use a third party transaction manager such as Bitronix.

To use Global Transactions, select either `xa_strict` or `xa` mode. The differences are explained in the sections below.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

28.1.5.1 Implementation

Global transactions support is implemented at the Store level, through XATransactionStore and JtaLocalTransactionStore. The former actually decorates the underlying MemoryStore implementation, augmenting it with transaction isolation and two-phase commit support through an *XAResouce* implementation. The latter decorates a LocalTransactionStore-decorated cache to make it controllable by the standard JTA API instead of the proprietary TransactionController API.

During its initialization, the Cache will lookup the TransactionManager using the provided TransactionManagerLookup implementation. Then, using the `TransactionManagerLookup.register(XAResouce)`, the newly created XAResource will be registered.

The store is automatically configured to copy every Element read from the cache or written to it. Cache is copy-on-read and copy-on-write.

28.1.6 Failure Recovery

As specified by the JTA specification, only *prepared* transaction data is recoverable.

Prepared data is persisted onto the cluster and locks on the memory are held. This basically means that non-clustered caches cannot persist transactions data, so recovery errors after a crash may be reported by the transaction manager.

28.1.6.1 Recovery

At any time after something went wrong, an `XAResource` may be asked to recover. Data that has been prepared may either be committed or rolled back during recovery. In accordance with XA, data that has not yet been `prepared` is discarded.

The recovery guarantee differs depending on the xa mode.

28.xa Mode

With `xa`, the cache doesn't get registered as an XAResource with the transaction manager but merely can follow the flow of a JTA transaction by registering a JTA Synchronization. The cache can end up inconsistent with the other resources if there is a JVM crash in the mutating node.

In this mode, some inconsistency may occur between a cache and other XA resources (such as databases) after a crash. However, the cache's data remains consistent because the transaction is still fully atomic on the cache itself.

28.xa_strict Mode

If `xa_strict` is used the cache will always respond to the TransactionManager's recover calls with the list of prepared XIDs of failed transactions. Those transaction branches can then be committed or rolled back by the transaction manager. This is the standard XA mechanism in strict compliance with the JTA specification.

28.1.7 Sample Apps

We have three sample applications showing how to use XA with a variety of technologies.

28.1.7.1 XA Sample App

This sample application uses JBoss application server. It shows an example using User managed transactions. While we expect most people will use JTA from within Spring or EJB where the container rather than managing it themselves, it clearly shows what is going on.

The following snippet from our SimpleTX servlet shows a complete transaction.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
Ehcache cache = cacheManager.getEhcache("xaCache");
UserTransaction ut = getUserTransaction();
printLine(servletResponse, "Hello...");
try {
    ut.begin();
    int index = serviceWithinTx(servletResponse, cache);
    printLine(servletResponse, "Bye #" + index);
    ut.commit();
} catch(Exception e) {
    printLine(servletResponse,
        "Caught a " + e.getClass() + "! Rolling Tx back");
    if(!printStackTrace) {
        PrintWriter s = servletResponse.getWriter();
        e.printStackTrace(s);
        s.flush();
    }
    rollbackTransaction(ut);
}
```

The source code for the demo can be checked out from http://svn.terracotta.org/svn/forge/projects/ehcache-jta-sample/trunk

A README.txt explains how to get the JTA Sample app going.

28.1.7.2 XA Banking Application

The Idea of this application is to show a real world scenario. AwWeb app reads *account transfer* messages from a queue and tries to execute these account transfers.

With JTA turned on, failures are rolled back so that the cached account balance is always the same as the true balance summed from the database.

This app is a Spring-based Java web app running in a Jetty container. It has (embedded) the following components:

- A JMS Server (ActiveMQ)
- 2 databases (embedded Derby XA instances)
- 2 caches (JTA Ehcache)

  All XA Resources are managed by Atomikos TransactionManager. Transaction demarcation is done using Spring AOP's `@Transactional` annotation.

  You can run it with: `mvn clean jetty:run`. Then point your browser at: http://localhost:9080.

  To see what happens without XA transactions:

  `mvn clean jetty:run -Dxa=no`

  The source code for the demo can be checked out from http://svn.terracotta.org/svn/forge/projects/ehcache-jta-banking/trunk

  A README.txt explains how to get the JTA Sample app going.

28.1.7.3 Examinator

Examinator is our complete application that shows many aspects of caching in one web based Exam application, all using the Terracotta Server Array.

Check out from http://svn.terracotta.org/svn/forge/projects/exam/

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

28.1.8 Transaction Managers

28.1.8.1 Automatically Detected Transaction Managers

Ehcache automatically detects and uses the following transaction managers in the following order:

- GenericJNDI (e.g. Glassfish, JBoss, JTOM and any others that register themselves in JNDI at the standard location of java:/TransactionManager
- Weblogic (since 2.4.0)
- Bitronix
- Atomikos

No configuration is required; they work out of the box.

The first found is used.

28.1.8.2 Configuring a Transaction Manager

If your Transaction Manager is not in the above list or you wish to change the priority you need to configure your own lookup class and specify it in place of the `DefaultTransactionManagerLookup` in the ehcache.xml config:

```
<transactionManagerLookup
 class= "net.sf.ehcache.transaction.manager.DefaultTransactionManagerLookup"
 properties="" propertySeparator=":"/>
```

You can also provide a different location for the JNDI lookup by providing the jndiName property to the DefaultTransactionManagerLookup.

The example below provides the proper location for the TransactionManager in GlassFish v3:

```
<transactionManagerLookup
 class="net.sf.ehcache.transaction.manager.DefaultTransactionManagerLookup"
 properties="jndiName=java:appserver/
TransactionManager" propertySeparator=";"/>
```

28.1.9 Local Transactions

Local Transactions allow single phase commit across multiple cache operations, across one or more caches, and in the same CacheManager, whether distributed with Terracotta or standalone.

This lets you apply multiple changes to a CacheManager all in your own transaction. If you also want to apply changes to other resources such as a database then you need to open a transaction to them and manually handle commit and rollback to ensure consistency.

Local transactions are not controlled by a Transaction Manager. Instead there is an explicit API where a reference is obtained to a `TransactionController` for the CacheManager using `cacheManager.getTransactionController()` and the steps in the transaction are called explicitly.

The steps in a local transaction are:

- `transactionController.begin()` - This marks the beginning of the local transaction on the current thread. The changes are not visible to other threads or to other transactions.
- `transactionController.commit()` - Commits work done in the current transaction on the calling thread.
- `transactionController.rollback()` - Rolls back work done in the current transaction on the calling thread. The changes done since begin are not applied to the cache.

These steps should be placed in a try-catch block which catches `TransactionException`. If any exceptions are thrown, rollback() should be called.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

Local Transactions has it's own exceptions that can be thrown, which are all subclasses of `CacheException`. They are:

- `TransactionException` - a general exception
- `TransactionInterruptedException` - if Thread.interrupt() got called while the cache was processing a transaction.
- `TransactionTimeoutException` - if a cache operation or commit is called after the transaction timeout has elapsed.

28.1.9.1 Introduction Video

Ludovic Orban the primary author of Local Transactions presents an introductory video on Local Transactions.

28.1.9.2 Configuration

Local transactions are configured as follows:

```
<cache name="sampleCache"
    ...
    transactionalMode="local"
</cache>
```

28.1.9.3 Isolation Level

As with the other transaction modes, the isolation level is READ_COMMITTED.

28.1.9.4 Transaction Timeouts

If a transaction cannot complete within the timeout period, then a `TransactionTimeoutException` will be thrown. To return the cache to a consistent state you need to call `transactionController.rollback()`.

Because `TransactionController` is at the level of the CacheManager, a default timeout can be set which applies to all transactions across all caches in a CacheManager. If not set, it is 15 seconds.

To change the defaultTimeout:

```
transactionController.setDefaultTransactionTimeout(int defaultTransactionTimeoutSec
```

The countdown starts straight after `begin()` is called. You might have another local transaction on a JDBC connection and you may be making multiple changes. If you think it could take longer than 15 seconds for an individual transaction, you can override the default when you begin the transaction with:

```
transactionController.begin(int transactionTimeoutSeconds) {
```

28.1.9.5 Sample Code

This example shows a transaction which performs multiple operations across two caches.

```
CacheManager cacheManager = CacheManager.getInstance();
try {
    cacheManager.getTransactionController().begin();
    cache1.put(new Element(1, "one"));
    cache2.put(new Element(2, "two"));
    cache1.remove(4);
    cacheManager.getTransactionController().commit();
} catch (CacheException e) {
    cacheManager.getTransactionController().rollback()
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
}
```

28.1.9.6 What can go wrong

28.JVM crash between begin and commit

On restart none of the changes applied after begin are there.

On restart, nothing needs to be done. Under the covers in the case of a Terracotta cluster, the Element's new value is there but not applied. It's will be lazily removed on next access.

28.1.10 Performance

28.1.10.1 Managing Contention

If two transactions, either standalone or across the cluster, attempt to perform a cache operation on the same element then the following rules apply:

- The first transaction gets access
- The following transactions will block on the cache operation until either the first transaction completes or the transaction timeout occurs.

Under the covers, when an element is involved in a transaction, it is replaced with a new element with a marker that is locked, along with the transaction ID. The normal cluster semantics are used.

Because transactions only work with consistency=strong caches, the first transaction will be the thread that manages to atomically place a soft lock on the Element. (Up to Terracotta 3.4 this was done with write locks. After that it is done with the CAS based putIfAbsent and replace methods).

28.1.10.2 What granularity of locking is used?

Ehcache standalone up to 2.3 used page level locking, where each segment in the `CompoundStore` is locked. From 2.4, it is one with soft locks stored in the Element itself and is on a key basis.

Terracotta clustered caches are locked on the key level.

28.1.10.3 Performance Comparisons

Any transactional cache adds an overhead which is significant for writes and nearly negligible for reads.

Within the modes the relative time take to perform writes, where off = 1, is:

- off - no overhead
- xa_strict - 20 times slower
- xa - 3 times slower
- local - 3 times slower

  The relative read performance is:
- off - no overhead
- xa_strict - 20 times slower
- xa - 30% slower
- local - 30% slower

Accordingly, xa_strict should only be used where it's full guarantees are required, othewise one of the other modes should be used.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

28.1.11 FAQ

28.1.11.1 Is IBM Websphere Transaction Manager supported?

Mostly. xa_strict is not supported due to each version of Websphere essentially being a custom implementation i.e. no stable interface to implement against. However, `xa`, which uses TransactionManager callbacks and `local` are supported.

28.1.11.2 How do transactions interact with Write-behind and Write-through caches?

If your transactional enabled cache is being used with a writer, write operations will be queued until transaction commit time. Solely a Write-through approach would have its potential XAResource participate in the same transaction. Write-behind, while supported, should probably not be used with an XA transactional Cache, as the operations would never be part of the same transaction. Your writer would also be responsible for obtaining a new transaction...

Using Write-through with a non XA resource would also work, but there is no guarantee the transaction will succeed after the write operations have been executed successfully. On the other hand, any thrown exception during these write operations would cause the transaction to be rolled back by having UserTransaction.commit() throw a RollbackException.

28.1.11.3 Are Hibernate Transactions supported?

Ehcache is a "transactional" cache for Hibernate purposes. The `net.sf.ehcache.hibernate.EhCacheRegionFactory` has support for Hibernate entities configured with *cache usage="transactional"/.*

28.1.11.4 How do I make WebLogic 10 work with Ehcache JTA?

WebLogic uses an optimization that is not supported by our implementation. By default WebLogic 10 will spawn threads to start the Transaction on each XAResource in parallel. As we need transaction work to be performed on the same Thread, you will have to turn this optimization off by setting `parallel-xa-enabled` option to `false` in your domain configuration :

```
<jta>
  ...
  <checkpoint-interval-seconds>300</checkpoint-interval-seconds>
  <parallel-xa-enabled>false</parallel-xa-enabled>
  <unregister-resource-grace-period>30</unregister-resource-grace-period>
  ...
</jta>
```

28.1.11.5 How do I make Atomikos work with Ehcache JTA's `xa` mode?

Atomikos has a bug which makes the `xa` mode's normal transaction termination mechanism unreliable, see: http://fogbugz.atomikos.com/default.asp?community.6.802.3. There is an alternative termination mechanism built in that transaction mode that is automatically enabled when `net.sf.ehcache.transaction.xa.alternativeTerminationMode` is set to true or when Atomikos is detected as the controlling transaction manager.

This alternative termination mode has strict requirement on the way threads are used by the transaction manager and Atomikos's default settings won't work. You have to configure this property to make it work: --- com.atomikos.icatch.threaded_2pc=false ---

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

29 Search

29.1 Search

29.1.1 Ehcache Search API

The Ehcache Search API allows you to execute arbitrarily complex queries against either a standalone cache or a Terracotta clustered cache with pre-built indexes. Searchable attributes may be extracted from both keys and values. Keys, values, or summary values (Aggregators) can all be returned.

Here is a simple example: Search for 32 year old males and return the cache values.

```
Results results = cache.createQuery().includeValues()
   .addCriteria(age.eq(32).and(gender.eq("male"))).execute();
```

29.1.2 What is searchable?

Searches can be performed against Element keys and values.

Element keys and values are made searchable by extracting attributes with supported search types out of the values.

It is the attributes themelves which are searchable.

29.1.3 How to make a cache searchable

29.1.3.1 By Configuration

Caches are made searchable by adding a <searchable/> tag to the ehcachel.xml.

```
<cache name="cache2" maxElementsInMemory="10000" eternal="true" overflowToDisk="false">
    <searchable/>
</cache>
```

This configuration will scan keys and vales and if they are of supported search types, add them as attributes called "key" and "value" respectively. If you do not want automatic indexing of keys and values you can disable it with:

```
<cache name="cache3" ...>
    <searchable keys="false" values="false">
        ...
    </searchable>
</cache>
```

You might want to do this if you have a mix of types for your keys or values. The automatic indexing will throw an exception if types are mixed.

Lots of times keys or values will not be directly searchable and instead you will need to extract searchable attributes out of them. The following example shows this more typical case. Attribute Extractors are explained in more detail in the following section.

```
<cache name="cache3" maxElementsInMemory="10000" eternal="true" overflowToDisk="false">
    <searchable>
        <searchAttribute name="age" class="net.sf.ehcache.search.TestAttributeExtractor
>
        <searchAttribute name="gender" expression="value.getGender()"/>
    </searchable>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
</cache>
```

29.1.3.2 Programmatically

The following example shows how to programmatically create the cache configuration, with search attributes.

```
Configuration cacheManagerConfig = new Configuration();
CacheConfiguration cacheConfig = new CacheConfiguration("myCache", 0).eternal(true);
Searchable searchable = new Searchable();
cacheConfig.addSearchable(searchable);

// Create attributes to use in queries.
searchable.addSearchAttribute(new SearchAttribute().name("age"));

// Use an expression for accessing values.
searchable.addSearchAttribute(new SearchAttribute()
        .name("first_name")
        .expression("value.getFirstName()"));

searchable.addSearchAttribute(new SearchAttribute().name("last_name").expression("value
         searchable.addSearchAttribute(new SearchAttribute().name("zip_code").expressio

cacheManager = new CacheManager(cacheManagerConfig);
cacheManager.addCache(new Cache(cacheConfig));

Ehcache myCache = cacheManager.getEhcache("myCache");

// Now create the attributes and queries, then execute.
...
```

To learn more about the Ehcache Search API, see the `net.sf.ehcache.search*` packages in this Javadoc.

29.1.4 Attribute Extractors

Attributes are extracted from keys or values. This is done on `put()` into the cache using `AttributeExtractors` in the clustered implementation and during search in the standalone implementation

Extracted attributes must be one of the following supported types:

- Boolean
- Byte
- Character
- Double
- Float
- Integer
- Long
- Short
- String
- java.util.Date
- java.sql.Date
- Enum

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

If an attribute cannot be extracted due to not being found or of being the wrong type an AttributeExtractorException is thrown during the `put()` in the clustered implementation and on search execution in the standalone implementation

29.1.4.1 Well-known Attributes

The parts of an Element are well-known attributes that can be referenced by some predefined, well-known names.

If a keys and/or value is of a supported search type, they are added automatically as attributes with the names "key" amd "value".

These well-known attributes have convenience constant attributes made available on the `Query` class. So, for example, the attribute for "key" may be referenced in a query by `Query.KEY`. For even greater readability it is recommended to statically import so that in this example you would just use `KEY`.

Well-known Attribute Name	Attribute Constant
key	Query.KEY
value	Query.VALUE

29.1.4.2 ReflectionAttributeExtractor

The ReflectionAttributeExtractor is a built-in search attribute extractor which uses JavaBean conventions and also understands a simple form of expression.

Where a JavaBean property is available and it is of a searchable type, it can be simply declared using:

```
<cache>
    <searchable>
        <searchAttribute name="age"/>
    </searchable>
</cache>
```

Finally, when things get more complicated, we have an expression language using method/value dotted expression chains.

The expression chain must start with one of either "key", "value", or "element". From the starting object a chain of either method calls or field names follows. Method calls and field names can be freely mixed in the chain.

Some more examples:

```
<cache>
    <searchable>
        <searchAttribute name="age" expression="value.person.getAge()"/>
    </searchable>
</cache>
<cache>
    <searchable>
        <searchAttribute name="name" expression="element.toString()"/>
    </searchable>
</cache>
```

The method and field name portions of the expression are case sensitive.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

29.1.4.3 Custom AttributeExtractor

In more complex situations you can create your own attribute extractor by implementing the AttributeExtractor interface. Providing your extractor class is shown in the following example:

```
<cache name="cache2" maxElementsInMemory="0" eternal="true" overflowToDisk="false">
    <searchable>
        <searchAttribute name="age" class="net.sf.ehcache.search.TestAttributeExtractor
>
    </searchable>
</cache>
```

If you need to pass state to your custom extractor you may do so with properties as shown in the following example:

```
<cache>
    <searchable>
        <searchAttribute name="age"
             class="net.sf.ehcache.search.TestAttributeExtractor"
             properties="foo=this,bar=that,etc=12" />
    </searchable>
</cache>
```

If properties are provided then the attribute extractor implementation must have a public constructor that accepts a single java.util.Properties instance

29.1.5 Query API

Ehcache Search introduces a fluent Object Oriented query API, following DSL principles, which should feel familiar and natural to Java programmers.

Here is a simple example:

```
Query query = cache.createQuery().addCriteria(age.eq(35)).includeKeys().end();
Results results = query.execute();
```

29.1.5.1 Using attributes in queries

If declared and available, the well-known attributes are referenced by their name or the convenience attributes are used directly as shown in this example:

```
Results results = cache.createQuery().addCriteria(Query.KEY.eq(35)).execute();
Results results = cache.createQuery().addCriteria(Query.VALUE.lt(10)).execute();
```

Other attributes are referenced by the names given them in the configuration. E.g.

```
Attribute<Integer> age = cache.getSearchAttribute("age");
Attribute<String> gender = cache.getSearchAttribute("gender");
Attribute<String> name = cache.getSearchAttribute("name");
```

29.1.5.2 Expressions

The Query to be searched for is built up using Expressions.

Expressions include logical operators such as *and* and *or*. It also includes comparison operators such as *ge* (>=), *between* and *like*

`addCriteria(...)` is used to add a clause to a query. Adding a further clause automatically *and*s the clauses

```
query = cache.createQuery().includeKeys().addCriteria(age.le(65)).add(gender.eq("male")
```

Both logical and comparison operators implement the `Criteria` interface.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

To add a criteria with a different logical operator, you need to explicitly nest it within a new logical operator Criteria Object.

e.g. to check for age = 35 or gender = female, do the following:

```
query.addCriteria(new Or(age.eq(35),
                   gender.eq(Gender.FEMALE))
                  );
```

More complex compound expressions can be further created with extra nesting.

See the Expression JavaDoc for a complete list.

29.1.5.3 List of Operators

Operators are available as methods on attributes, so they are used by adding a ".". So, "lt" means less than and is used as `age.lt(10)`, which is a shorthand way of saying `new LessThan(10)`.

The full listing of operator shorthand is shown below.

Shorthand	Criteria Class	Description
and	And	The Boolean AND logical operator
between	Between	A comparison operator meaning between two values
eq	EqualTo	A comparison operator meaning Java "equals to" condition
gt	GreaterThan	A comparison operator meaning greater than.
ge	GreaterThanOrEqual	A comparison operator meaning greater than or equal to.
in	InCollection	A comparison operator meaning in the collection given as an argument
lt	LessThan	A comparison operator meaning less than.
le	LessThanOrEqual	A comparison operator meaning less than or equal to
ilike	ILike	A regular expression matcher. '?' and "*" may be used. Note that placing a wildcard in front of the expression will cause a table scan. ILike is always case insensitive.
not	Not	The Boolean NOT logical operator
ne	NotEqualTo	A comparison operator meaning not the Java "equals to" condition
or	Or	The Boolean OR logical operator

29.1.5.4 Making queries immutable

By default a query can be executed and then modified and re-executed. If `end` is called the query is made immutable.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

29.1.5.5 Ordering Results

Query results may be ordered in ascending or descending order by adding an `addOrderBy` clause to the query, which takes as parameters the attribute to order by and the ordering direction.

e.g. to order the results by ages in ascending order --- query.addOrderBy(age, Direction.ASCENDING); ---

29.1.5.6 Limiting the size of Results

By default a query will return an unlimited number of results. For example the following query will return all keys in the cache.

```
Query query = cache.createQuery();
query.includeKeys();
query.execute();
```

If too many results are returned it could cause an OutOfMemoryError

The `maxResults` clause is used to limit the size of the results.

e.g. to limit the above query to the first 100 elements found:

```
Query query = cache.createQuery();
query.includeKeys();
query.maxResults(100);
query.execute();
```

If a returns a very large result, you can get it in chunks with `Results.range()`.

29.1.6 Search Results

Queries return a `Results` object which contains a list of objects of class `Result`

29.1.6.1 Results

Either all results can be returned using `results.all()` to get the all in one chunk, or a range of results using `results.range(int start, int count)` to achieve paging.

When you are done with the results, it is recommended to call `discard()`. This allows resources to be freed. In the distributed implementation with Terracotta, resources may be used to hold results for paging or return.

To determine what was returned by the query use one of the interrogation methods on `Results`:

- `hasKeys()`
- `hasValues()`
- `hasAttributes()`
- `hasAggregators()`

29.1.6.2 Result

Each `Element` in the cache found with a query will be represented as a `Result` object. So if a query finds 350 elements there will be 350 `Result` objects. An exception to this if no keys or attributes are included but aggregators are -- In this case there will be exactly one `Result` present

A Result object can contain:

- the Element key - when `includeKeys()` was added to the query
- the Element value - when `includeValues()` was added to the query

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- predefined attribute(s) extracted from an Element value - when `includeAttribute(...)` was added to the query. To access an attribute from Result, use `getAttribute(Attribute<T> attribute`.
- aggregator results

Aggregator results are summaries computed for the search. They are available `Result.getAggregatorResults` which returns a list of `Aggregators` in the same order in which they were used in the `Query`.

29.1.6.3 Aggregators

Aggregators are added with `query.includeAggregator(<attribute>.<aggregator>)`.

E.g. to find the sum of the age attribute:

```
query.includeAggregator(age.sum());
```

See the Aggregators JavaDoc for a complete list.

29.1.7 Sample Application

We have created a simple standalone sample application with few dependencies for you to easily get started with Ehcache Search.

or check out the source:

```
git clone git://github.com/sharrissf/Ehcache-Search-Sample.git
```

The Ehcache Test Sources show lots of further examples on how to use each Ehcache Search feature.

29.1.8 Scripting Environments

Ehcache Search is readily amenable to scripting. The following example shows how to use it with BeanShell:

```
Interpreter i = new Interpreter();
//
Auto discover the search attributes and add them to the interpreter's context
Map<String, SearchAttribute> attributes = cache.getCacheConfiguration().getSearchAttrib
for (Map.Entry<String, SearchAttribute> entry : attributes.entrySet()) {
    i.set(entry.getKey(), cache.getSearchAttribute(entry.getKey()));
    LOG.info("Setting attribute " + entry.getKey());
}
//
Define the query and results. Add things which would be set in the GUI i.e.
//includeKeys and add to context
Query query = cache.createQuery().includeKeys();
Results results = null;
i.set("query", query);
i.set("results", results);
//This comes from the freeform text field
String userDefinedQuery = "age.eq(35)";
//Add the stuff on that we need
String fullQueryString = "results = query.addCriteria(" + userDefinedQuery + ").execute
i.eval(fullQueryString);
results = (Results) i.get("results");
assertTrue(2 == results.size());
for (Result result : results.all()) {
    LOG.info("" + result.getKey());
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

}

29.1.9 Concurrency Considerations

Unlike cache operations which has selectable concurrency control and/or transactions, the Search API does not. This may change in a future release, however our survey of prospective users showed that concurrency control in search indexes was not sought after.

The indexes are eventually consistent with the caches.

29.1.9.1 Index updating

Indexes will be updated asynchronously, so there state will lag slightly behind the state of the cache. The only exception is when the updating thread then performs a search.

For caches with concurrency control, an index will not reflect the new state of the cache until:

- The change has been applied to the cluster.
- For a cache with transactions, when `commit` has been called.

29.1.9.2 Query Results

There are several ways unexpected results could present:

- A search returns an Element reference which no longer exists.
- Search criteria select an Element, but the Element has been updated and a new Search would no longer match the Element.
- Aggregators, such as sum(), might disagree with the same calculation done by redoing the calculation yourself by re-accessing the cache for each key and repeating the calculation.
- `includeValues` returns values. Under the covers the index contains a server value reference. The reference gets returned with the search and Terracotta supplies the matching value. Because the cache is always updated before the search index it is possible that a value reference may refer to a value that has been removed from the cache. If this happens the value will be null but the key and attributes which were supplied by the now stale cache index will be non-null. Because values in Ehcache are also allowed to be null, you cannot tell whether your value is null because it has been removed from the cache since the index was last updated or because it is a null value.

29.1.9.3 Recommendations

Because the state of the cache can change between search executions it is recommended to add all of the Aggregators you want for a query at once so that the returned aggregators are consistent.

Use null guards when accessing a cache with a key returned from a search.

29.1.10 Implementations

29.1.10.1 Standalone Ehcache

The standalone Ehcache implementation does not use indexes. It uses fast iteration of the cache instead, relying on the very fast access to do the equivalent of a table scan for each query. Each element in the cache is only visited once.

Attributes are not extracted ahead of time. They are done during query execution.

29.Performance

Search operations perform in O(n) time.

Checkout https://svn.terracotta.org/repo/forge/offHeap-test/ terracotta_community_login, a Maven-based performance test showing standalone cache performance. This test shows search performance

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

of of an average of representative queries at 10ms per 10,000 entries. So, a typical query would take 1 second for a 1,000,000 entry cache.

Accordingly, standalone implementation is suitable for development and testing. For production it is recommended to only standalone search for caches that are less than 1 million elements.

Performance of different `Criteria` vary. For example, here are some queries and their execute times on a 200,000 element cache. (Note that these results are all faster than the times given above because they execute a single Criteria).

```
final Query intQuery = cache.createQuery();
intQuery.includeKeys();
intQuery.addCriteria(age.eq(35));
intQuery.end();
Execute Time: 62ms
final Query stringQuery = cache.createQuery();
stringQuery.includeKeys();
stringQuery.addCriteria(state.eq("CA"));
stringQuery.end();
Execute Time: 125ms
final Query iLikeQuery = cache.createQuery();
iLikeQuery.includeKeys();
iLikeQuery.addCriteria(name.ilike("H*"));
iLikeQuery.end();
Execute Time: 180ms
```

29.1.10.2 Ehcache backed by the Terracotta Server Array

This implementation uses indexes which are maintained on each Terracotta server. In Ehcache EX the index is on a single active server. In Ehcache FX the cache is sharded across the number of active nodes in the cluster. The index for each shard is maintained on that shard's server.

Searches are performed using the Scatter-Gather pattern. The query executes on each node and the results are then aggregated back in the Ehcache that initiated the search.

29.Performance

Search operations perform in O(log n / number of shards) time.

Performance is excellent and can be improved simply by adding more servers to the FX array.

29.Network Effects

Search results are returned over the network. The data returned could potentially be very large, so techniques to limit return size are recommended such as:

- limiting the results with `maxResults` or using the paging API `Results.range(int start, int length)`
- Only including the data you need. Specifically only use `includeKeys()` and/or `includeAttribute()` if those values are actually required for your application logic
- using a built-in `Aggregator` function when you only need a summary statistic

  `includeValues` rates a special mention. Once a query requiring values is executed we push the values from the server to the Ehcache CacheManager which requested it in batches for network efficiency. This is done ahead as soon as possible reducing the risk that `Result.getValue()` might have to wait for data over the network.
- turn off key and value indexing if you are not going to search against them as they will just chew up space on the server.

  You do this as follows:

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
<cache name="cache3" ...>
    <searchable keys="false" values="false">
        ...
    </searchable>
</cache>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

30 Ehcache Monitor

30.1 Ehcache Monitor

This add-on tool for Ehcache provides enterprise-class monitoring and management capabilities for use in both development and production. It is intended to help understand and tune cache usage, detect errors, and provide an easy to use access point to integrate with production management systems. It also provides administrative functionality such as the ability to forcefully remove items from caches.

Simply install the Monitor on an Operations server, add the Monitor Probe jar to your app, add a few lines of config in ehcache.xml and your done.

The package contains a probe and a server. The probe installs with your existing Ehcache cache instance, and communicates to a central server. The server aggregates data from multiple probes. It can be accessed via a simple web UI, as well as a scriptable API. In this way, it is easy to integrate with common third party systems management tools (such as Hyperic, Nagios etc). The probe is designed to be compatible with all versions of Ehcache from 1.5 and requires JDK 1.5 or 1.6.

Get the Ehcache Monitor now.

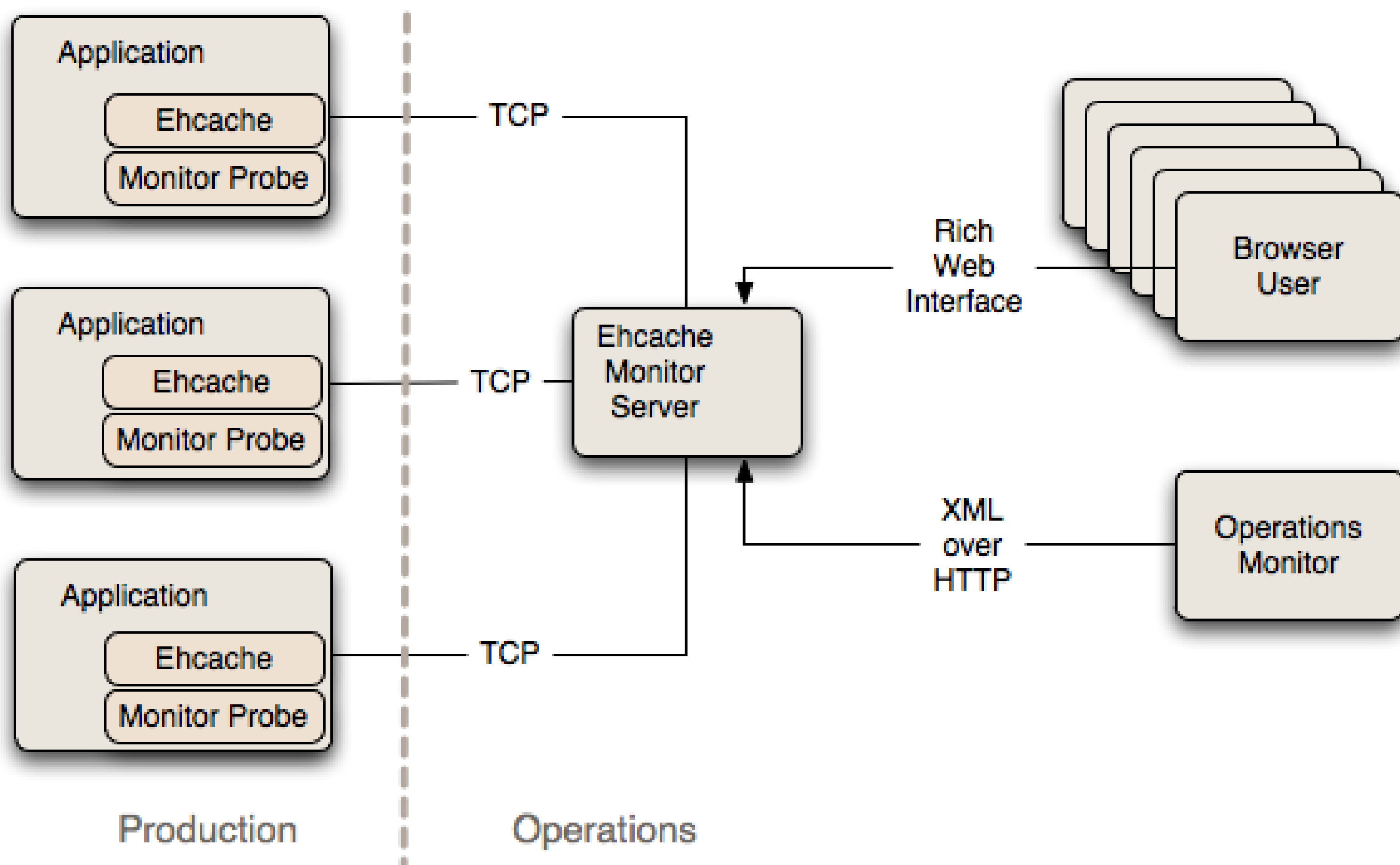
30.1.1 Installation And Configuration

First download and extract the Ehcache Monitor package.

The package consists of a lib directory with the probe and monitor server jars, a bin directory with startup and shutdown scripts for the monitor server and an etc directory with an example monitor server configuration file and a Jetty Server configuration file.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

30.1.2 Recommended Deployment Topology



It is recommended to place the Monitor on an Operations server separate to production. The Monitor acts as an aggregation point for access by end users and for scripted connection from Operations tools for data feeds and set up of alerts.

30.1.2.1 Probe

To include the probe in your Ehcache application, you need to perform two steps:

1 Add the ehcache-probe- *version*.jar to your application classpath (or war file). Do this in the same way you added the core ehcache jar to your application.

If you are Maven based, the probe module is in the Terracotta public repository for easy integration.

```
        <repository>
            <id>terracotta-releases</id>
            <url>http://www.terracotta.org/download/reflector/
releases</url>
        </repository>
        <dependency>
         <groupId>org.terracotta</groupId>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
        <artifactId>ehcache-probe</artifactId>
        <version>[version]</version>
      </dependency>
```

2 Configure Ehcache to communicate with the probe by specifying the class name of the probe, the address (or hostname), the port that the monitor will be running on and whether to do memory measurement. This is done by adding the following to ehcache.xml:

```
<cacheManagerPeerListenerFactory
    class="org.terracotta.ehcachedx.monitor.probe.ProbePeerListenerFactory"
    properties="monitorAddress=localhost, monitorPort=9889, memoryMeasurement=tr
```

3 Include required SLF4J logging jars.

Ehcache 1.7.1 and above require SLF4J. Earlier versions used commons logging. The probe, like all new Ehcache modules, uses SLF4J, which is becoming a new standard in open source projects.

If you are using Ehcache 1.5 to 1.7.0, you will need to add slf4j-api and one concrete logger.

If you are using Ehcache 1.7.1 and above you should not need to do anything because you will already be using slf4j-api and one concrete logger.

More information on SLF4J is available from http://www.slf4j.org.

4 Ensure that statistics capture in each cache is turned on for the probe to gather statistics. Statistics were turned off by default from Ehcache 2.1 onwards.

```
<cache name="sampleCache2"
           maxElementsInMemory="1000"
           eternal="true"
           overflowToDisk="false"
           memoryStoreEvictionPolicy="FIFO"
           statistics="true"
            />
```

30.1.2.2 Monitor

Copy the monitor package to a monitoring server.

To start the monitor, run the startup script provided in the bin directory: startup.sh on Unix and startup.bat on Microsoft Windows. The monitor port selected in this script should match the port specified in ehcache.xml.

The monitor can be configured, including interface, port and simple security settings, in the etc/ ehcache-monitor.conf.

Note that if you are using the commercial version, you need to specify in ehcache-monitor.conf the location of your license file.

e.g.

```
    license_file=/Users/karthik/Documents/workspace/lib/license/terracotta-
license.key
```

The monitor connection timeout can also be configured. If the monitor is frequently timing out while attempting to connect to a node (due to long GC cycles, for example), then the default timeout value may not be suitable for your environment. You can set the monitor timeout using the system property `ehcachedx.connection.timeout.seconds`. For example, `-Dehcachedx.connection.timeout.seconds=60` sets the timeout to 60 seconds.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

30.1.3 Securing the Monitor

The Monitor can be secured in a variety of ways. The simplest method involves simply editing ehcache-monitor.conf to specify a single user name and password. This method has the obvious drawbacks that (1) it provides only a single login identity, and (2) the credentials are stored in clear-text.

A more comprehensive security solution can be achieved by configuring the Jetty Server with one ore more `UserRealms` as described by Jetty and JAAS. Simply edit etc/jetty.xml to use the appropriate `UserRealm` implementation for your needs. To configure the Monitor to authenticate against an existing LDAP server, first ensure that you have defined and properly registered a `LoginConfig`, such as the following example:

```
MyExistingLDAPLoginConfig {
  com.sun.security.auth.module.LdapLoginModule REQUIRED
  java.naming.security.authentication="simple"
  userProvider="ldap://ldap-host:389"
  authIdentity="uid={USERNAME},ou=People,dc=myorg,dc=org"
  useSSL=false
  bindDn="cn=Manager"
  bindCredential="secretBindCredential"
  bindAuthenticationType="simple"
  debug=true;
};
```

Note: the `LdapLoginModule` is new with JDK 1.6.

JAAS supports many different types of login modules and it is up to the reader to provide a valid, working JAAS environment. For more information regarding JAAS refer to JAAS Reference Guide.

For information on how to register your LoginConfig refer to $JAVA_HOME/jre/lib/security/java.security.

Next, edit etc/jetty.xml like so:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
  "http://jetty.mortbay.org/configure.dtd">
<Configure id="Server" class="org.terracotta.ehcachedx.org.mortbay.jetty.Server">
    <Set name="UserRealms">
      <Array type="org.terracotta.ehcachedx.org.mortbay.jetty.security.UserRealm">
        <Item>
          <New class="org.terracotta.ehcachedx.org.mortbay.jetty.plus.jaas.JAASUser
           <Set name="Name">MyArbitraryLDAPRealmName</Set>
           <Set name="LoginModuleName">MyExistingLDAPLoginConfig</Set>
          </New>
        </Item>
      </Array>
    </Set>
</Configure>
```

The `LoginModuleName` you specify as the second constructor parameter to the `JAASUserRealm` class must exactly match the name of your `LoginModule`. The realm name specified as the first constructor parameter can be an arbitrary value.

Note: the version of Jetty used in the Monitor has been repackaged so be sure to prefix any standard Jetty class names with `org.terracotta.ehcachedx`.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

If the Jetty Server is found to have been configured with any security realms, the simple user name and password from ehcache-monitor.conf is ignored.

30.1.4 Using the Web GUI

The web-based GUI is available by pointing your browser at http:// *monitor-host-name*: *monitor-port*/ monitor. For a default installation on the local machine, this would be http://localhost:9889/monitor

The GUI contains six tabs, described as follows:

30.1.4.1 Cache Managers

This tab shows aggregate statistics for the cache managers being monitored by probes connected to the monitor server. Double-clicking on any cache manager drills down to the detailed Statistics tab for that manager.

30.1.4.2 Statistics

This tab shows the statistics being gathered for each cache managed by the selected cache manager.

The Settings button permits you to add additional statistics fields to the display. Note: only displayed fields are collected and aggregated by the probe. Adding additional display fields will increase the processing required for probe and the monitor. The selected settings are stored in a preferences cookie in your browser.

Double-clicking on any cache drills down to the Contents tab for that cache.

30.1.4.3 Configuration

This tab shows the key configuration information for each cache managed by the selected cache manager.

30.1.4.4 Contents

This tab enables you to look inside the cache, search for elements via their keys and remove individual or groups of elements from the cache.

The GUI is set to refresh at the same frequency that the probes aggregate their statistic samples which is every 10 seconds by default. The progress bar at the bottom of the screen indicates the time until the next refresh.

30.1.4.5 Charts

This tab contains various live charts of cache statistics. It gives you a feel for the trending of the each statistic, rather than just the latest value.

30.Estimated Memory Use Chart

This chart shows the estimated memory use of the Cache.

Memory is estimated by sampling. The first 15 puts or updates are measured and then every 100th put or update. Most caches contain objects of similar size. If this is not the case for your cache, then the estimate will not be accurate.

Measurements are performed by walking the object graph of sampled elements through reflection. In some cases such as classes not visible to the classloader, the measurement fails and 0 is recorded for cache size. If you see a chart with 0 memory size values but the cache has data in it, then this is the cause. For this release, caches distributed via Terracotta server show as 0.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

30.1.4.6 API

This tab contains a listing of the API methods. Each is a hyperlink, which may be clicked on. Some will display data and some will require additional arguments. If additional arguments are required an error message will be displayed with the details. This tab is meant for interative testing of the API.

30.1.5 Using the API

The Monitor provides a API over HTTP on the same port as the Web GUI.

The list of functions supported by the API can be accessed by pointing your browser at http://*monitor-host-name*: *monitor-port*/monitor/list. For a default installation on the local machine, this would be http://localhost:9889/monitor/list

The API returns data as either structured XML or plan text. The default format is txt.

For example, the getVersion function returns the software version of the monitor server. It can be called as follows:

http://localhost:9889/monitor/getVersion

or, to receive the results as XML:

http://localhost:9889/monitor/getVersion?format=xml

To query the data collected by the monitor server from scripts that can then be used to pass the data to enterprise system management frameworks, commands such as `curl` or `wget` can be used.

For example, on a Linux system, to query the list of probes that a local monitor on the default port is currently aware of, and return the data in XML format, the following command could be used:

```
$ curl http://localhost:9889/monitor/listProbes?format=xml
```

30.1.6 Licensing

Unless otherwise indicated, this module is licensed for usage in development.

For details see the license terms in the appropriate LICENSE.txt. To obtain a commercial license for use in production, please contact sales@terracottatech.com

30.1.7 Limitations

30.1.7.1 History not Implemented

This release has server side history configuration in place, however history is not implemented. It is anticipated it will be implemented in the next release. In the meantime, the charts with their recent history provide trending.

30.1.7.2 Memory Measurement limitations

Unfortunately in Java, there is no JSR for memory measurement of objects. Implementations, such as the sizeof one we use are subject to fragilities.

For example, Java 7 memory measurement is not supported at this time. You will get a `java.lang.NoSuchFieldException: header` exception message if you use memory measurement with Java 7.

Memory measurement can optionally be turned off by setting `memoryMeasurement=false` in the probe configuration.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

31 CacheManager Event Listeners

31.1 CacheManager Event Listeners

CacheManager event listeners allow implementers to register callback methods that will be executed when a `CacheManager` event occurs. Cache listeners implement the CacheManagerEventListener interface.

The events include:

- adding a `Cache`
- removing a `Cache`

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

31.1.1 Configuration

One CacheManagerEventListenerFactory and hence one CacheManagerEventListener can be specified per CacheManager instance.

The factory is configured as below:

```
<cacheManagerEventListenerFactory class="" properties=""/>
```

The entry specifies a CacheManagerEventListenerFactory which will be used to create a CacheManagerPeerProvider, which is notified when Caches are added or removed from the CacheManager.

The attributes of CacheManagerEventListenerFactory are:

- `class` - a fully qualified factory class name
- `properties` - comma separated properties having meaning only to the factory.

  Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

  If no class is specified, or there is no cacheManagerEventListenerFactory element, no listener is created. There is no default.

31.1.2 Implementing a CacheManagerEventListenerFactory and CacheManagerEventListener

CacheManagerEventListenerFactory is an abstract factory for creating cache manager listeners. Implementers should provide their own concrete factory extending this abstract factory. It can then be configured in ehcache.xml.

The factory class needs to be a concrete subclass of the abstract factory CacheManagerEventListenerFactory, which is reproduced below:

```
/**
 * An abstract factory for creating {@link CacheManagerEventListener}s. Implementers sh
 * provide their own concrete factory extending this factory. It can then be configured
 * ehcache.xml
 *
 * @author Greg Luck
 * @version $Id: cachemanager_event_listeners.apt 3744 2011-03-04 02:58:18Z gluck $
 * @see "http://ehcache.org/documentation/
cachemanager_event_listeners.html"
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 */
public abstract class CacheManagerEventListenerFactory {
    /**
     * Create a <code>CacheEventListener</code>
     *
     * @param properties implementation specific properties. These are configured as co
     *                   separated name value pairs in ehcache.xml. Properties may be n
     * @return a constructed CacheManagerEventListener
     */
    public abstract CacheManagerEventListener
            createCacheManagerEventListener(Properties properties);
}
```

The factory creates a concrete implementation of CacheManagerEventListener, which is reproduced below:

```
/**
 * Allows implementers to register callback methods that will be executed when a
 * <code>CacheManager</code> event occurs.
 * The events include:
 * <ol>
 * <li>adding a <code>Cache</code>
 * <li>removing a <code>Cache</code>
 * </ol>
 * <p/>
 * Callbacks to these methods are synchronous and unsynchronized. It is the responsibil
 * the implementer to safely handle the potential performance and thread safety issues
 * depending on what their listener is doing.
 * @author Greg Luck
 * @version $Id: cachemanager_event_listeners.apt 3744 2011-03-04 02:58:18Z gluck $
 * @since 1.2
 * @see CacheEventListener
 */
public interface CacheManagerEventListener {
/**
 * Called immediately after a cache has been added and activated.
 * <p/>
 * Note that the CacheManager calls this method from a synchronized method. Any attempt
 * call a synchronized method on CacheManager from this method will cause a deadlock.
 * <p/>
 * Note that activation will also cause a CacheEventListener status change notification
 * from {@link net.sf.ehcache.Status#STATUS_UNINITIALISED} to
 * {@link net.sf.ehcache.Status#STATUS_ALIVE}. Care should be taken on processing that
 * notification because:
 * <ul>
 * <li>the cache will not yet be accessible from the CacheManager.
 * <li>the addCaches methods whih cause this notification are synchronized on the
 * CacheManager. An attempt to call {@link net.sf.ehcache.CacheManager#getCache(String)
 * will cause a deadlock.
 * </ul>
 * The calling method will block until this method returns.
 * <p/>
 * @param cacheName the name of the <code>Cache</
code> the operation relates to
 * @see CacheEventListener
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 */
void notifyCacheAdded(String cacheName);
/**
 * Called immediately after a cache has been disposed and removed. The calling method w
 * block until this method returns.
 * <p/>
 * Note that the CacheManager calls this method from a synchronized method. Any attempt
 * call a synchronized method on CacheManager from this method will cause a deadlock.
 * <p/>
 * Note that a {@link CacheEventListener} status changed will also be triggered. Any
 * attempt from that notification to access CacheManager will also result in a deadlock
 * @param cacheName the name of the <code>Cache</
code> the operation relates to
 */
void notifyCacheRemoved(String cacheName);
}
```

The implementations need to be placed in the classpath accessible to ehcache. Ehcache uses the ClassLoader returned by `Thread.currentThread().getContextClassLoader()` to load classes.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

32 Cache Event Listeners

32.1 Cache Event Listeners

Cache listeners allow implementers to register callback methods that will be executed when a cache event occurs. Cache listeners implement the CacheEventListener interface.

The events include:

- an Element has been put
- an Element has been updated. Updated means that an Element exists in the Cache with the same key as the Element being put.
- an Element has been removed
- an Element expires, either because timeToLive or timeToIdle have been reached.

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

Listeners are guaranteed to be notified of events in the order in which they occurred.

Elements can be put or removed from a Cache without notifying listeners by using the putQuiet and removeQuiet methods.

In clustered environments event propagation can be configured to be propagated only locally, only remotely, or both. The default is both, to be backwardly compatible.

32.1.1 Configuration

Cache event listeners are configured per cache. Each cache can have multiple listeners.

Each listener is configured by adding a cacheEventListenerFactory element as follows:

```
<cache ...>
<cacheEventListenerFactory class="" properties="" listenFor=""/>
...
</cache>
```

The entry specifies a CacheManagerEventListenerFactory which is used to create a CachePeerProvider, which then receives notifications.

The attributes of CacheManagerEventListenerFactory are:

- class - a fully qualified factory class name
- properties - an optional comma separated properties having meaning only to the factory.
- listenFor - describes which events will be delivered in a clustered environment, defaults to 'all'.

  These are the possible values:

  - all - the default is to deliver all local and remote events
  - local - deliver only events originating in the current node
  - remote - deliver only events originating in other nodes

  Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

32.1.2 Implementing a CacheEventListenerFactory and CacheEventListener

CacheEventListenerFactory is an abstract factory for creating cache event listeners. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml

The factory class needs to be a concrete subclass of the abstract factory class CacheEventListenerFactory, which is reproduced below:

```
/**
 * An abstract factory for creating listeners. Implementers should provide their own
 * concrete factory extending this factory. It can then be configured in ehcache.xml
 *
 * @author Greg Luck
 * @version $Id: cache_event_listeners.apt 3789 2011-03-17 00:01:42Z gluck $
 */
public abstract class CacheEventListenerFactory {
/**
 * Create a <code>CacheEventListener</code>
 *
 * @param properties implementation specific properties. These are configured as comma
 *                   separated name value pairs in ehcache.xml
 * @return a constructed CacheEventListener
 */
public abstract CacheEventListener createCacheEventListener(Properties properties);
}
```

The factory creates a concrete implementation of the CacheEventListener interface, which is reproduced below:

```
/**
 * Allows implementers to register callback methods that will be executed when a cache e
 *  occurs.
 * The events include:
 * <ol>
 * <li>put Element
 * <li>update Element
 * <li>remove Element
 * <li>an Element expires, either because timeToLive or timeToIdle has been reached.
 * </ol>
 * <p/>
 * Callbacks to these methods are synchronous and unsynchronized. It is the responsibil
 * the implementer to safely handle the potential performance and thread safety issues
 * depending on what their listener is doing.
 * <p/>
 * Events are guaranteed to be notified in the order in which they occurred.
 * <p/>
 * Cache also has putQuiet and removeQuiet methods which do not notify listeners.
 *
 * @author Greg Luck
 * @version $Id: cache_event_listeners.apt 3789 2011-03-17 00:01:42Z gluck $
 * @see CacheManagerEventListener
 * @since 1.2
 */
public interface CacheEventListener extends Cloneable {
/**
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 * Called immediately after an element has been removed. The remove method will block u
 * this method returns.
 * <p/>
 * Ehcache does not chech for
 * <p/>
 * As the {@link net.sf.ehcache.Element} has been removed, only what was the key of the
 * element is known.
 * <p/>
 *
 * @param cache   the cache emitting the notification
 * @param element just deleted
 */
void notifyElementRemoved(final Ehcache cache, final Element element) throws CacheExcep
/**
 * Called immediately after an element has been put into the cache. The
 * {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
 * will block until this method returns.
 * <p/>
 * Implementers may wish to have access to the Element's fields, including value, so th
 * element is provided. Implementers should be careful not to modify the element. The
 * effect of any modifications is undefined.
 *
 * @param cache   the cache emitting the notification
 * @param element the element which was just put into the cache.
 */
void notifyElementPut(final Ehcache cache, final Element element) throws CacheException
/**
 * Called immediately after an element has been put into the cache and the element alre
 * existed in the cache. This is thus an update.
 * <p/>
 * The {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
 * will block until this method returns.
 * <p/>
 * Implementers may wish to have access to the Element's fields, including value, so th
 * element is provided. Implementers should be careful not to modify the element. The
 * effect of any modifications is undefined.
 *
 * @param cache   the cache emitting the notification
 * @param element the element which was just put into the cache.
 */
void notifyElementUpdated(final Ehcache cache, final Element element) throws CacheExcep
/**
 * Called immediately after an element is <i>found</i> to be expired. The
 * {@link net.sf.ehcache.Cache#remove(Object)} method will block until this method retu
 * <p/>
 * As the {@link Element} has been expired, only what was the key of the element is kno
 * <p/>
 * Elements are checked for expiry in Ehcache at the following times:
 * <ul>
 * <li>When a get request is made
 * <li>When an element is spooled to the diskStore in accordance with a MemoryStore
 * eviction policy
 * <li>In the DiskStore when the expiry thread runs, which by default is
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 * {@link net.sf.ehcache.Cache#DEFAULT_EXPIRY_THREAD_INTERVAL_SECONDS}
 * </ul>
 * If an element is found to be expired, it is deleted and this method is notified.
 *
 * @param cache   the cache emitting the notification
 * @param element the element that has just expired
 *     <p/>
 *     Deadlock Warning: expiry will often come from the <code>DiskStore</
code>
 *     expiry thread. It holds a lock to the DiskStorea the time the
 *     notification is sent. If the implementation of this method calls into a
 *     synchronized <code>Cache</
code> method and that subsequently calls into
 *     DiskStore a deadlock will result. Accordingly implementers of this method
 *     should not call back into Cache.
 */
void notifyElementExpired(final Ehcache cache, final Element element);
/**
 * Give the replicator a chance to cleanup and free resources when no longer needed
 */
void dispose();
/**
 * Creates a clone of this listener. This method will only be called by Ehcache before a
 * cache is initialized.
 * <p/>
 * This may not be possible for listeners after they have been initialized. Implementat
 * should throw CloneNotSupportedException if they do not support clone.
 * @return a clone
 * @throws CloneNotSupportedException if the listener could not be cloned.
 */
public Object clone() throws CloneNotSupportedException;
}
```

The implementations need to be placed in the classpath accessible to Ehcache.

See the chapter on Classloading for details on how classloading of these classes will be done.

32.1.3 FAQ

32.1.3.1 Can I add a listener to an already running cache?

Yes.

```
    cache.getCacheEventNotificationService().registerListener(myListener);
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

33 Cache Exception Handlers

33.1 Cache Exception Handlers

By default, most cache operations will propagate a runtime CacheException on failure. An interceptor, using a dynamic proxy, may be configured so that a CacheExceptionHandler can be configured to intercept Exceptions. Errors are not intercepted.

Caches with ExceptionHandling configured are of type Ehcache. To get the exception handling behaviour they must be referenced using `CacheManager.getEhcache()`, not `CacheManager.getCache()`, which returns the underlying undecorated cache.

CacheExceptionHandlers may be set either declaratively in the ehcache.xml configuration file or programmatically.

33.1.1 Declarative Configuration

Cache event listeners are configured per cache. Each cache can have at most one exception handler.

An exception handler is configured by adding a cacheExceptionHandlerFactory element as shown in the following example:

```
<cache ...>
      <cacheExceptionHandlerFactory
        class="net.sf.ehcache.exceptionhandler.CountingExceptionHandlerFactory"
        properties="logLevel=FINE"/>
</cache>
```

33.1.2 Implementing a CacheExceptionHandlerFactory and CacheExceptionHandler

CacheExceptionHandlerFactory is an abstract factory for creating cache exception handlers. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml

The factory class needs to be a concrete subclass of the abstract factory class CacheExceptionHandlerFactory, which is reproduced below:

```
/**
 * An abstract factory for creating <code>CacheExceptionHandler</
code>s at configuration
 * time, in ehcache.xml.
 * <p/>
 * Extend to create a concrete factory
 *
 * @author <a href="mailto:gluck@gregluck.com">Greg Luck</a>
 * @version $Id: cache_exception_handlers.apt 3744 2011-03-04 02:58:18Z gluck $
 */
public abstract class CacheExceptionHandlerFactory {
/**
 * Create an <code>CacheExceptionHandler</code>
 *
 * @param properties implementation specific properties. These are configured as comma
 *                   separated name value pairs in ehcache.xml
 * @return a constructed CacheExceptionHandler
 */
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
public abstract CacheExceptionHandler createExceptionHandler(Properties properties);
}
```

The factory creates a concrete implementation of the CacheExceptionHandler interface, which is reproduced below:

```
/**
 * A handler which may be registered with an Ehcache, to handle exception on Cache oper
 * <p/>
 * Handlers may be registered at configuration time in ehcache.xml, using a
 * CacheExceptionHandlerFactory, or *  set at runtime (a strategy).
 * <p/>
 * If an exception handler is registered, the default behaviour of throwing the excepti
 * will not occur. The handler * method <code>onException</
code> will be called. Of course, if
 * the handler decides to throw the exception, it will * propagate up through the call
 * If the handler does not, it won't.
 * <p/>
 * Some common Exceptions thrown, and which therefore should be considered when impleme
 * this class are listed below:
 * <ul>
 * <li>{@link IllegalStateException} if the cache is not
 * {@link net.sf.ehcache.Status#STATUS_ALIVE}
 * <li>{@link IllegalArgumentException} if an attempt is made to put a null
 * element into a cache
 * <li>{@link net.sf.ehcache.distribution.RemoteCacheException} if an issue occurs
 *  in remote synchronous replication
 * <li>
 * <li>
 * </ul>
 *
 * @author <a href="mailto:gluck@gregluck.com">Greg Luck</a>
 * @version $Id: cache_exception_handlers.apt 3744 2011-03-04 02:58:18Z gluck $
 */
public interface CacheExceptionHandler {
/**
 * Called if an Exception occurs in a Cache method. This method is not called
 * if an <code>Error</code> occurs.
 *
 * @param Ehcache   the cache in which the Exception occurred
 * @param key       the key used in the operation, or null if the operation does not us
 * key or the key was null
 * @param exception the exception caught
 */
void onException(Ehcache ehcache, Object key, Exception exception);
}
```

The implementations need to be placed in the classpath accessible to Ehcache.

See the chapter on Classloading for details on how classloading of these classes will be done.

33.1.3 Programmatic Configuration

The following example shows how to add exception handling to a cache then adding the cache back into cache manager so that all clients obtain the cache handling decoration.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
CacheManager cacheManager = ...
Ehcache cache = cacheManger.getCache("exampleCache");
ExceptionHandler handler = new ExampleExceptionHandler(...);
cache.setCacheLoader(handler);
Ehcache proxiedCache = ExceptionHandlingDynamicCacheProxy.createProxy(cache);
cacheManager.replaceCacheWithDecoratedCache(cache, proxiedCache);
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

34 Cache Extensions

34.1 Cache Extensions

CacheExtensions are a general purpose mechanism to allow generic extensions to a Cache.

CacheExtensions are tied into the Cache lifecycle. For that reason this interface has the lifecycle methods.

CacheExtensions are created using the CacheExtensionFactory which has a *code*createCacheCacheExtension() */code* method which takes as a parameter a Cache and properties. It can thus call back into any public method on Cache, including, of course, the load methods.

CacheExtensions are suitable for timing services, where you want to create a timer to perform cache operations. The other way of adding Cache behaviour is to decorate a cache.

See @link net.sf.ehcache.constructs.blocking.BlockingCache for an example of how to do this.

Because a CacheExtension holds a reference to a Cache, the CacheExtension can do things such as registering a CacheEventListener or even a CacheManagerEventListener, all from within a CacheExtension, creating more opportunities for customisation.

34.1.1 Declarative Configuration

Cache extension are configured per cache. Each cache can have zero or more.

A CacheExtension is configured by adding a cacheExceptionHandlerFactory element as shown in the following example:

```
<cache ...>
 <cacheExtensionFactory class="com.example.FileWatchingCacheRefresherExtensionFactory"
     properties="refreshIntervalMillis=18000, loaderTimeout=3000,
                 flushPeriod=whatever, someOtherProperty=someValue ..."/>
</cache>
```

34.1.2 Implementing a CacheExtensionFactory and CacheExtension

CacheExtensionFactory is an abstract factory for creating cache extension. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml

The factory class needs to be a concrete subclass of the abstract factory class CacheExtensionFactory, which is reproduced below:

```
/**
 * An abstract factory for creating <code>CacheExtension</
code>s. Implementers should
 * provide their own * concrete factory extending this factory. It can then be configur
 * in ehcache.xml.
 *
 * @author <a href="mailto:gluck@gregluck.com">Greg Luck</a>
 * @version $Id: cache_extensions.apt 3744 2011-03-04 02:58:18Z gluck $
 */
public abstract class CacheExtensionFactory {
/**
 * @param cache the cache this extension should hold a reference to, and to whose
 * lifecycle it should be bound.
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 * @param properties implementation specific properties configured as delimiter separat
 * name value pairs in ehcache.xml
 */
public abstract CacheExtension createCacheExtension(Ehcache cache, Properties propertie
}
```

The factory creates a concrete implementation of the CacheExtension interface, which is reproduced below:

```
/**
 * This is a general purpose mechanism to allow generic extensions to a Cache.
 * <p/>
 * CacheExtensions are tied into the Cache lifecycle. For that reason this interface ha
 *  lifecycle methods.
 * <p/>
 * CacheExtensions are created using the CacheExtensionFactory which has a
 * <code>createCacheCacheExtension()</
code> method which takes as a parameter a Cache and
 * properties. It can thus call back into any public method on Cache, including, of cou
 *  the load methods.
 * <p/>
 * CacheExtensions are suitable for timing services, where you want to create a timer t
 * perform cache operations. The other way of adding Cache behaviour is to decorate a c
 * See {@link net.sf.ehcache.constructs.blocking.BlockingCache} for an example of how t
 * this.
 * <p/>
 * Because a CacheExtension holds a reference to a Cache, the CacheExtension can do thi
 * such as registering a CacheEventListener or even a CacheManagerEventListener, all fr
 * within a CacheExtension, creating more opportunities for customisation.
 *
 * @author <a href="mailto:gluck@gregluck.com">Greg Luck</a>
 * @version $Id: cache_extensions.apt 3744 2011-03-04 02:58:18Z gluck $
 */
public interface CacheExtension {
/**
 * Notifies providers to initialise themselves.
 * <p/>
 * This method is called during the Cache's initialise method after it has changed it's
 * status to alive. Cache operations are legal in this method.
 *
 * @throws CacheException
 */
void init();
/**
 * Providers may be doing all sorts of exotic things and need to be able to clean up on
 * dispose.
 * <p/>
 * Cache operations are illegal when this method is called. The cache itself is partly
 * disposed when this method is called.
 *
 * @throws CacheException
 */
void dispose() throws CacheException;
/**
 * Creates a clone of this extension. This method will only be called by Ehcache before
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 * cache is initialized.
 * <p/>
 * Implementations should throw CloneNotSupportedException if they do not support clone
 * but that will stop them from being used with defaultCache.
 *
 * @return a clone
 * @throws CloneNotSupportedException if the extension could not be cloned.
 */
public CacheExtension clone(Ehcache cache) throws CloneNotSupportedException;
/**
 * @return the status of the extension
 */
public Status getStatus();
}
```

The implementations need to be placed in the classpath accessible to ehcache.

See the chapter on Classloading for details on how class loading of these classes will be done.

34.1.3 Programmatic Configuration

Cache Extensions may also be programmatically added to a Cache as shown.

```
TestCacheExtension testCacheExtension = new TestCacheExtension(cache, ...);
testCacheExtension.init();
cache.registerCacheExtension(testCacheExtension);
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

35 Cache Loaders

35.1 Cache Loaders

A `CacheLoader` is an interface which specifies `load` and `loadAll` methods with a variety of parameters.

CacheLoaders come from JCache, but are a frequently requested feature, so they have been incorporated into the core Ehcache classes and can be configured in ehcache.xml.

CacheLoaders are invoked in the following Cache methods:

- getWithLoader (synchronous)
- getAllWithLoader (synchronous)
- load (asynchronous)
- loadAll (asynchronous)

They are also invoked in similar (though slightly differently named) JCache methods.

The methods will invoke a CacheLoader if there is no entry for the key or keys requested. By implementing CacheLoader, an application form of loading can take place. The get... methods follow the pull-through cache pattern. The load... methods are useful as cache warmers.

CacheLoaders are similar to the CacheEntryFactory used in SelfPopulatingCache. However SelfPopulatingCache is a decorator to ehcache. The CacheLoader functionality is available right in a Cache, Ehcache or JCache and follows a more industry standard convention.

CacheLoaders may be set either declaratively in the ehcache.xml configuration file or programmatically. This becomes the default CacheLoader. Some of the methods invoking loaders enable an override CacheLoader to be passed in as a parameter.

More than one cacheLoader can be registered, in which case the loaders form a chain which are executed in order. If a loader returns null, the next in chain is called.

35.1.1 Declarative Configuration

cacheLoaderFactory - Specifies a CacheLoader, which can be used both asynchronously and synchronously to load objects into a cache. More than one cacheLoaderFactory element can be added, in which case the loaders form a chain which are executed in order. If a loader returns null, the next in chain is called.

```
<cache ...>
      <cacheLoaderFactory class="com.example.ExampleCacheLoaderFactory"
                                      properties="type=int,startCounter=10"/
>
</cache>
```

35.1.2 Implementing a CacheLoaderFactory and CacheLoader

CacheLoaderFactory is an abstract factory for creating CacheLoaders. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml

The factory class needs to be a concrete subclass of the abstract factory class CacheLoaderFactory, which is reproduced below:

```
/**
 * An abstract factory for creating cache loaders. Implementers should provide their ow
 * concrete factory extending this factory.
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 * <p/>
 * There is one factory method for JSR107 Cache Loaders and one for Ehcache ones. The E
 * loader is a sub interface of the JSR107 Cache Loader.
 * <p/>
 * Note that both the JCache and Ehcache APIs also allow the CacheLoader to be set
 * programmatically.
 * @author Greg Luck
 * @version $Id: cache_loaders.apt 3744 2011-03-04 02:58:18Z gluck $
 */
public abstract class CacheLoaderFactory {
/**
 * Creates a CacheLoader using the JSR107 creational mechanism.
 * This method is called from {@link net.sf.ehcache.jcache.JCacheFactory}
 *
 * @param environment the same environment passed into
 * {@link net.sf.ehcache.jcache.JCacheFactory}.
 * This factory can extract any properties it needs from the environment.
 * @return a constructed CacheLoader
 */
public abstract net.sf.jsr107cache.CacheLoader createCacheLoader(Map environment);
/**
 * Creates a CacheLoader using the Ehcache configuration mechanism at the time
 * the associated cache is created.
 *
 * @param properties implementation specific properties. These are configured as comma
 *                   separated name value pairs in ehcache.xml
 * @return a constructed CacheLoader
 */
public abstract net.sf.ehcache.loader.CacheLoader createCacheLoader(Properties properti
/**
 * @param cache the cache this extension should hold a reference to,
 * and to whose lifecycle it should be bound.
 * @param properties implementation specific properties configured as delimiter
 *  separated name value pairs in ehcache.xml
 * @return a constructed CacheLoader
 */
public abstract CacheLoader createCacheLoader(Ehcache cache, Properties properties);
}
```

The factory creates a concrete implementation of the CacheLoader interface, which is reproduced below.

A CacheLoader is bound to the lifecycle of a cache, so that `init()` is called during cache initialization, and `dispose()` is called on disposal of a cache.

```
/**
 * Extends JCache CacheLoader with load methods that take an argument in addition to a k
 * @author Greg Luck
 * @version $Id: cache_loaders.apt 3744 2011-03-04 02:58:18Z gluck $
 */
public interface CacheLoader extends net.sf.jsr107cache.CacheLoader {
/**
 * Load using both a key and an argument.
 * <p/>
 * JCache will call through to the load(key) method, rather than this method,
```

```
 * where the argument is null.
 *
 * @param key      the key to load the object for
 * @param argument can be anything that makes sense to the loader
 * @return the Object loaded
 * @throws CacheException
 */
Object load(Object key, Object argument) throws CacheException;
/**
 * Load using both a key and an argument.
 * <p/>
 * JCache will use the loadAll(key) method where the argument is null.
 *
 * @param keys     the keys to load objects for
 * @param argument can be anything that makes sense to the loader
 * @return a map of Objects keyed by the collection of keys passed in.
 * @throws CacheException
 */
Map loadAll(Collection keys, Object argument) throws CacheException;
/**
 * Gets the name of a CacheLoader
 *
 * @return the name of this CacheLoader
 */
String getName();
/**
 * Creates a clone of this extension. This method will only be called by Ehcache before
 * cache is initialized.
 * <p/>
 * Implementations should throw CloneNotSupportedException if they do not support clone
 * but that will stop them from being used with defaultCache.
 *
 * @return a clone
 * @throws CloneNotSupportedException if the extension could not be cloned.
 */
public CacheLoader clone(Ehcache cache) throws CloneNotSupportedException;
/**
 * Notifies providers to initialise themselves.
 * <p/>
 * This method is called during the Cache's initialise method after it has changed it's
 * status to alive. Cache operations are legal in this method.
 *
 * @throws net.sf.ehcache.CacheException
 */
void init();
/**
 * Providers may be doing all sorts of exotic things and need to be able to clean up on
 * dispose.
 * <p/>
 * Cache operations are illegal when this method is called. The cache itself is partly
 * disposed when this method is called.
 *
 * @throws net.sf.ehcache.CacheException
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 */
void dispose() throws net.sf.ehcache.CacheException;
/**
 * @return the status of the extension
 */
public Status getStatus();
}
```

The implementations need to be placed in the classpath accessible to ehcache.

See the chapter on Classloading for details on how classloading of these classes will be done.

35.1.3 Programmatic Configuration

The following methods on `Cache` allow runtime interrogation, registration and unregistration of loaders:

```
/**
 * Register a {@link CacheLoader} with the cache. It will then be tied into the cache
 *   lifecycle.
 * <p/>
 * If the CacheLoader is not initialised, initialise it.
 *
 * @param cacheLoader A Cache Loader to register
 */
public void registerCacheLoader(CacheLoader cacheLoader) {
    registeredCacheLoaders.add(cacheLoader);
}
/**
 * Unregister a {@link CacheLoader} with the cache. It will then be detached from the c
 * lifecycle.
 *
 * @param cacheLoader A Cache Loader to unregister
 */
public void unregisterCacheLoader(CacheLoader cacheLoader) {
    registeredCacheLoaders.remove(cacheLoader);
}
/**
 * @return the cache loaders as a live list
 */
public List<CacheLoader> getRegisteredCacheLoaders() {
    return registeredCacheLoaders;
}
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

36 Write-through and write-behind caching with CacheWriters

36.1 Write-through and Write-behind Caching with the CacheWriter

Write-through caching is a caching pattern where writes to the cache cause writes to an underlying resource. The cache acts as a facade to the underlying resource. With this pattern, it often makes sense to read through the cache too.

Write-behind caching uses the same client API; however, the write happens asynchronously.

Ehcache-2.0 introduced write-through and write-behind caching.

While file systems or a web-service clients can underlie the facade of a write-through cache, the most common underlying resource is a database. To simplify the discussion, we will use the database as the example resource.

36.1.1 Potential Benefits of Write-Behind

The major benefit of write-behind is database offload. This can be achieved in a number of ways:

- time shifting - moving writes to a specific time or time interval. For example, writes could be batched up and written overnight, or at 5 minutes past the hour, to avoid periods of peak contention.
- rate limiting - spreading writes out to flatten peaks. Say a Point of Sale network has an end-of-day procedure where data gets written up to a central server. All POS nodes in the same time zone will write all at once. A very large peak will occur. Using rate limiting, writes could be limited to 100 TPS, and the queue of writes are whittled down over several hours
- conflation - consolidate writes to create fewer transactions. For example, a value in a database row is updated by 5 writes, incrementing it from 10 to 20 to 31 to 40 to 45. Using conflation, the 5 transactions are replaced by one to update the value from 10 to 45.

  These benefits must be weighed against the limitations and constraints imposed.

36.1.2 Limitations & Constraints of Write-Behind

36.1.2.1 Transaction Boundaries

If the cache participates in a JTA transaction (ehcache-2.0 and higher), which means it is an XAResource, then the cache can be made consistent with the database. A write to the database, and a commit or rollback, happens with the transaction boundary.

In write-behind, the write to the resource happens after the write to the cache. The transaction boundary is the write to the outstanding queue, not the write behind. In write-through mode, commit can get called and both the cache and the underlying resource can get committed at once.

Because the database is being written to outside of the transaction, there is always a risk that a failure on the eventual write will occur. While this can be mitigated with retry counts and delays, compensating actions may be required.

36.1.2.2 Time delay

The obvious implication of asynchronous writes is that there is a delay between when the cache is updated and when the database is updated. This introduces an inconsistency between the cache and the database, where the cache holds the correct value and the database will be eventually consistent with the cache. The data passed into the CacheWriter methods is a snapshot of the cache entry at the time of the write to operation.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

A read against the database will result in incorrect data being loaded.

36.1.2.3 Applications Tolerant of Inconsistency

The application must be tolerant of inconsistent data. The following examples illustrate this requirement:

- The database is logging transactions and only appends are done.
- Reading is done by a part of the application that does not write, so there is no way that data can be corrupted. The application is tolerant of delays. For example, a news application where the reader displays the articles that are written.

Note if other applications are writing to the database, then a cache can often be inconsistent with the database.

36.1.2.4 Node time synchronisation

Ideally node times should be synchronised. The write-behind queue is generally written to the underlying resource in timestamp order, based on the timestamp of the cache operation, although there is no guaranteed ordering.

The ordering will be more consistent if all nodes are using the same time. This can easily be achieved by configuring your system clock to synchronise with a time authority using Network Time Protocol.

36.1.2.5 No ordering guarantees

The items on the write-behind queue are generally in order, but this isn't guaranteed. In certain situations and more particularly in clustered usage, the items can be processed out of order. Additionally, when batching is used, write and delete collections are aggregated separately and can be processed inside the CacheWriter in a different order than the order that was used by the queue.

Your application must be tolerant of item reordering or you need to compensate for this in your implementation of the CacheWriter. Possible examples are:

- Working with versioning in the cache elements.
- Verifications with the underlying resource to check if the scheduled write-behind operation is still relevant.

36.1.3 Using a combined Read-Through and Write-Behind Cache

For applications that are not tolerant of inconsistency, the simplest solution is for the application to always read through the same cache that it writes through. Provided all database writes are through the cache, consistency is guaranteed. And in the distributed caching scenario, using Terracotta clustering extends the same guarantee to the cluster.

If using transactions, the cache is the XAResource, and a commit is a commit to the cache.

The cache effectively becomes the System Of Record ("SOR"). Terracotta clustering provides HA and durability and can easily act as the SOR. The database then becomes a backup to the SOR.

The following aspects of read-through with write-behind should be considered:

36.1.3.1 Lazy Loading

The entire data set does not need to be loaded into the cache on startup. a read-through cache uses a `CacheLoader` that loads data into the cache on demand. In this way the cache can be populated lazily.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

36.1.3.2 Caching of a Partial Dataset

If the entire dataset cannot fit in the cache, then some reads will miss the cache and fall through to the `CacheLoader` which will in turn hit the database. If a write has occurred but has not yet hit the database due to write-behind, then the database will be inconsistent.

The simplest solution is to ensure that the entire dataset is in the cache. This then places some implications on cache configuration in the areas of expiry and eviction.

36.Eviction

Eviction occurs when the maximum elements for the cache have been exceeded. Ensure that the `maxElementsInMemory` and, if using the DiskStore or Terracotta clustering, the `maxElementsOnDisk` exceeds the required size, so that eviction does not not occur.

36.Expiry

Even if all of the dataset can fit in the cache, it could be evicted if Elements expire. Accordingly, both `timeToLive` and `timeToIdle` should be set to eternal ("0") to prevent this from happening.

36.1.4 Introduction Video

Alex Snaps the primary author of Write Behind presents an introductory video on Write Behind.

36.1.5 Sample Application

We have created a sample web application for a raffle which fully demonstrates how to use write behind.

You can also checkout the Ehcache Raffle application, that demonstrates Cache Writers and Cache Loaders from github.com.

36.1.6 Ehcache Versions

Both Ehcache standalone (DX) and with Terracotta Server Array (Ehcache EX and FX) are supported.

36.1.6.1 Ehcache DX (Standalone Ehcache)

The write-behind queue is stored locally in memory. It supports all configuration options, but any data in the queue will be lost on JVM shutdown.

36.1.6.2 Ehcache EX and FX

36.Durable HA write-behind Queue

EX and FX when used with the Terracotta Server Array will store the queue on the Terracotta Server Array and can thus be configured for durability and HA. The data is still kept in the originating node for performance.

36.1.7 Configuration

There are many configuration options. See the `CacheWriterConfiguration` for properties that may be set and their effect.

Below is an example of how to configure the cache writer in XML:

```
<cache name="cacheName" eternal="false" maxElementsInMemory="1000" overflowToDisk="false
    <cacheWriter writeMode="write_behind" maxWriteDelay="8" rateLimitPerSecond="5"
                 writeCoalescing="true" writeBatching="true" writeBatchSize="20"
                 retryAttempts="2" retryAttemptDelaySeconds="2">
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
        <cacheWriterFactory class="com.company.MyCacheWriterFactory"
                            properties="just.some.property=test; another.property=test2
>
    </cacheWriter>
</cache>
```

Further examples:

```
<cache name="writeThroughCache1" eternal="false" maxElementsInMemory="1000" overflowToD
>
<cache name="writeThroughCache2" eternal="false" maxElementsInMemory="1000" overflowToD
  <cacheWriter/>
</cache>
<cache name="writeThroughCache3" eternal="false" maxElementsInMemory="1000" overflowToD
  <cacheWriter writeMode="write_through" notifyListenersOnException="true" maxWriteDela
               rateLimitPerSecond="10" writeCoalescing="true" writeBatching="true" writ
               retryAttempts="20" retryAttemptDelaySeconds="60"/>
</cache>
<cache name="writeThroughCache4" eternal="false" maxElementsInMemory="1000" overflowToD
  <cacheWriter writeMode="write_through" notifyListenersOnException="false" maxWriteDel
               rateLimitPerSecond="0" writeCoalescing="false" writeBatching="false" wri
               retryAttempts="0" retryAttemptDelaySeconds="0">
      <cacheWriterFactory class="net.sf.ehcache.writer.WriteThroughTestCacheWriterFacto
>
  </cacheWriter>
</cache>
<cache name="writeBehindCache5" eternal="false" maxElementsInMemory="1000" overflowToDi
  <cacheWriter writeMode="write-
behind" notifyListenersOnException="true" maxWriteDelay="8" rateLimitPerSecond="5"
               writeCoalescing="true" writeBatching="false" writeBatchSize="20"
               retryAttempts="2" retryAttemptDelaySeconds="2">
      <cacheWriterFactory class="net.sf.ehcache.writer.WriteThroughTestCacheWriterFacto
                          properties="just.some.property=test; another.property=test2" 
>
  </cacheWriter>
</cache>
```

This configuration can also be achieved through the `Cache` constructor in Java:

```
Cache cache = new Cache(
     new CacheConfiguration("cacheName", 10)
      .cacheWriter(new CacheWriterConfiguration()
    .writeMode(CacheWriterConfiguration.WriteMode.WRITE_BEHIND)
    .maxWriteDelay(8)
    .rateLimitPerSecond(5)
    .writeCoalescing(true)
    .writeBatching(true)
    .writeBatchSize(20)
    .retryAttempts(2)
    .retryAttemptDelaySeconds(2)
    .cacheWriterFactory(new CacheWriterConfiguration.CacheWriterFactoryConfiguration()
        .className("com.company.MyCacheWriterFactory")
        .properties("just.some.property=test; another.property=test2")
        .propertySeparator(";"))));
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

Instead of relying on a `CacheWriterFactoryConfiguration>> to create a <<<CacheWriter`, it's also possible to explicitly register a `CacheWriter` instance from within Java code. This allows you to refer to local resources like database connections or file handles.

```
Cache cache = manager.getCache("cacheName");
MyCacheWriter writer = new MyCacheWriter(jdbcConnection);
cache.registerCacheWriter(writer);
```

36.1.7.1 Configuration Attributes

The CacheWriterFactory supports the following attributes:

36.All modes

- write-mode [write-through | write-behind] - Whether to run in write-behind or write-through mode. The default is write-through.

36.write-through mode only

- notifyListenersOnException - Whether to notify listeners when an exception occurs on a store operation. Defaults to false. If using cache replication, set this attribute to "true" to ensure that changes to the underlying store are replicated.

36.write-behind mode only

- writeBehindMaxQueueSize - The maximum number of elements allowed per queue, or per bucket (if the queue has multiple buckets). "0" means unbounded (default). When an attempt to add an element is made, the queue size (or bucket size) is checked, and if full then the operation is blocked until the size drops by one. Note that elements or a batch currently being processed (and coalesced elements) are not included in the size value. Programmatically, this attribute can be set with `net.sf.ehcache.config.CacheWriterConfiguration.setWriteBehindMaxQueueSize()`.
- writeBehindConcurrency - The number of thread-bucket pairs on the node for the given cache (default is 1). Each thread uses the settings configured for write-behind. For example, if rateLimitPerSecond is set to 100, each thread-bucket pair will perform up to 100 operations per second. In this case, setting writeBehindConcurrency="4" means that up to 400 operations per second will occur on the node for the given cache. Programmatically, this attribute can be set with `net.sf.ehcache.config.CacheWriterConfiguration.setWriteBehindConcurrency()`.
- maxWriteDelaySeconds - The maximum number of seconds to wait before writing behind. Defaults to 0. If set to a value greater than 0, it permits operations to build up in the queue to enable effective coalescing and batching optimisations.
- rateLimitPerSecond - The maximum number of store operations to allow per second.
- writeCoalescing - Whether to use write coalescing. Defaults to false. When set to true, if multiple operations on the same key are present in the write-behind queue, then only the latest write is done (the others are redundant). This can dramatically reduce load on the underlying resource.
- writeBatching - Whether to batch write operations. Defaults to false. If set to true, storeAll and deleteAll will be called rather than store and delete being called for each key. Resources such as databases can perform more efficiently if updates are batched to reduce load.
- writeBatchSize - The number of operations to include in each batch. Defaults to 1. If there are less entries in the write-behind queue than the batch size, the queue length size is used. Note that batching is split across operations. For example, if the batch size is 10 and there were 5 puts and 5 deletes, the CacheWriter is invoked. It does not wait for 10 puts or 10 deletes.
- retryAttempts - The number of times to attempt writing from the queue. Defaults to 1.
- retryAttemptDelaySeconds - The number of seconds to wait before retrying.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

36.1.8 API

CacheLoaders are exposed for API use through the `cache.getWithLoader(...)` method. CacheWriters are exposed with `cache.putWithWriter(...)` and `cache.removeWithWriter(...)` methods.

For example, following is the method signature for `cache.putWithWriter(...)`.

```
/**
 * Put an element in the cache writing through a CacheWriter. If no CacheWriter has bee
 * set for the cache, then this method has the same effect as cache.put().
 * <p/>
 * Resets the access statistics on the element, which would be the case if it has previ
 * been gotten from a cache, and is now being put back.
 * <p/>
 * Also notifies the CacheEventListener, if the writer operation succeeds, that:
 * <ul>
 * <li>the element was put, but only if the Element was actually put.
 * <li>if the element exists in the cache, that an update has occurred, even if the ele
 * would be expired if it was requested
 * </ul>
 *
 * @param element An object. If Serializable it can fully participate in replication an
 * DiskStore.
 * @throws IllegalStateException    if the cache is not {@link net.sf.ehcache.Status#ST
 * @throws IllegalArgumentException if the element is null
 * @throws CacheException
 */
void putWithWriter(Element element) throws IllegalArgumentException, IllegalStateExcept
 CacheException;
```

See the Cache JavaDoc for the complete API.

36.1.9 SPI

The Ehcache write-through SPI is the `CacheWriter` interface. Implementers perform writes to the underlying resource in their implementation.

```
/**
 * A CacheWriter is an interface used for write-through and write-
behind caching to a
 * underlying resource.
 * <p/>
 * If configured for a cache, CacheWriter's methods will be called on a cache operation
 * A cache put will cause a CacheWriter write
 * and a cache remove will cause a writer delete.
 * <p>
 * Implementers should create an implementation which handles storing and deleting to a
 * underlying resource.
 * </p>
 * <h4>Write-Through</h4>
 * In write-
through mode, the cache operation will occur and the writer operation will occur
 * before CacheEventListeners are notified. If
 * the write operation fails an exception will be thrown. This can result in a cache wh
 * is inconsistent with the underlying resource.
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
 * To avoid this, the cache and the underlying resource should be configured to partici
 * in a transaction. In the event of a failure
 * a rollback can return all components to a consistent state.
 * <p/>
 * <h4>Write-Behind</h4>
 * In write-behind mode, writes are written to a write-
behind queue. They are written by a
 * separate execution thread in a configurable
 * way. When used with Terracotta Server Array, the queue is highly available. In addit
 * any node in the cluster may perform the
 * write-behind operations.
 * <p/>
 * <h4>Creation and Configuration</h4>
 * CacheWriters can be created using the CacheWriterFactory.
 * <p/>
 * The manner upon which a CacheWriter is actually called is determined by the
 * {@link net.sf.ehcache.config.CacheWriterConfiguration} that is set up for cache
 * that is using the CacheWriter.
 * <p/>
 * See the CacheWriter chapter in the documentation for more information on how to use 
 *
 * @author Greg Luck
 * @author Geert Bevin
 * @version $Id: $
 */
public interface CacheWriter {
    /**
     * Creates a clone of this writer. This method will only be called by ehcache befor
     * cache is initialized.
     * <p/>
     * Implementations should throw CloneNotSupportedException if they do not support c
     * but that will stop them from being used with defaultCache.
     *
     * @return a clone
     * @throws CloneNotSupportedException if the extension could not be cloned.
     */
    public CacheWriter clone(Ehcache cache) throws CloneNotSupportedException;
    /**
     * Notifies writer to initialise themselves.
     * <p/>
     * This method is called during the Cache's initialise method after it has changed 
     * status to alive. Cache operations are legal in this method.
     *
     * @throws net.sf.ehcache.CacheException
     */
    void init();
    /**
     * Providers may be doing all sorts of exotic things and need to be able to clean u
     * dispose.
     * <p/>
     * Cache operations are illegal when this method is called. The cache itself is par
     * disposed when this method is called.
     */
```

```
    void dispose() throws CacheException;
    /**
     * Write the specified value under the specified key to the underlying store.
     * This method is intended to support both key/
value creation and value update for a
     * specific key.
     *
     * @param element the element to be written
     */
    void write(Element element) throws CacheException;
    /**
     * Write the specified Elements to the underlying store. This method is intended to
     * support both insert and update.
     * If this operation fails (by throwing an exception) after a partial success,
     * the convention is that entries which have been written successfully are to be re
     * from the specified mapEntries, indicating that the write operation for the entri
     * in the map has failed or has not been attempted.
     *
     * @param elements the Elements to be written
     */
    void writeAll(Collection<Element> elements) throws CacheException;
    /**
     * Delete the cache entry from the store
     *
     * @param entry the cache entry that is used for the delete operation
     */
    void delete(CacheEntry entry) throws CacheException;
    /**
     * Remove data and keys from the underlying store for the given collection of keys,
     * present. If this operation fails * (by throwing an exception) after a partial su
     * the convention is that keys which have been erased successfully are to be remove
     * the specified keys, indicating that the erase operation for the keys left in the
     * has failed or has not been attempted.
     *
     * @param entries the entries that have been removed from the cache
     */
    void deleteAll(Collection<CacheEntry> entries) throws CacheException;
}
```

36.1.10 FAQ

36.1.10.1 Is there a way to monitor the write-behind queue size?

Use the method `net.sf.ehcache.statistics.LiveCacheStatistics#getWriterQueueLength()`. This method returns the number of elements on the local queue (in all local buckets) that are waiting to be processed, or -1 if no write-behind queue exists. Note that elements or a batch currently being processed (and coalesced elements) are not included in the returned value.

36.1.10.2 What happens if an exception occurs when the writer is called?

In the clustered async implementation inside the Terracotta Toolkit this is implemented as such:

```
    try {
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
        processItems();
      } catch (final Throwable e) {
        errorHandler.onError(ProcessingBucket.this, e);
        continue;
      }
```

This works since there's a concept of error handlers that isn't present in the non-clustered write behind implementation in Ehcache core. The default error handler simply logs the exceptions that occurred.

In standalone Ehcache, users should be careful to catch Exceptions. One solution is to put the item back on the queue with a call to `cache.write()`.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

37 Cache Server with SOAP and RESTful Web Services

37.1 Cache Server

37.1.1 Introduction

Ehcache now comes with a Cache Server, available as a WAR for most web containers, or as a standalone server. The Cache Server has two APIs: RESTful resource oriented, and SOAP. Both support clients in any programming language.

(A Note on terminology: Leonard Richardson and Sam Ruby have done a great job of clarifying the different Web Services architectures and distinguishing them from each other. We use their taxonomy in describing web services. See http://www.oreilly.com/catalog/9780596529260/.)

37.1.2 RESTful Web Services

Roy Fielding coined the acronym REST, denoting Representational State Transfer, in his PhD thesis.

The Ehcache implementation strictly follows the RESTful resource-oriented architecture style.

Specifically:

- The HTTP methods GET, HEAD, PUT/POST and DELETE are used to specify the method of the operation. The URI does not contain method information.
- The scoping information, used to identify the resource to perform the method on, is contained in the URI path.
- The RESTful Web Service is described by and exposes a WADL (Web Application Description Language) file. It contains the URIs you can call, and what data to pass and get back. Use the OPTIONS method to return the WADL.

  Roy is on the JSR311 expert group. JSR311 and Jersey, the reference implementation, are used to deliver RESTful web services in Ehcache server.

37.1.2.1 RESTFul Web Services API

The Ehcache RESTFul Web Services API exposes the singleton CacheManager, which typically has been configured in ehcache.xml or an IoC container. Multiple CacheManagers are not supported.

Resources are identified using a URI template. The value in parentheses should be substituted with a literal to specify a resource.

Response codes and response headers strictly follow HTTP conventions.

37.1.2.2 CacheManager Resource Operations

37.OPTIONS /{cache}}

Retrieves the WADL for describing the available CacheManager operations.

37. GET /

Lists the Caches in the CacheManager.

37.1.2.3 Cache Resource Operations

37.OPTIONS /{cache}}

Retrieves the WADL describing the available Cache operations.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

37.HEAD /{cache}}

Retrieves the same metadata a GET would receive returned as HTTP headers. There is no body returned.

37.GET /{cache}

Gets a cache representation. This includes useful metadata such as the configuration and cache statistics.

37. PUT /{cache}

Creates a Cache using the defaultCache configuration.

37. DELETE / {cache}

Deletes the Cache.

37.1.2.4 Element Resource Operations

37.OPTIONS /{cache}}

Retrieves the WADL describing the available Element operations.

37.HEAD /{cache}/{element}

Retrieves the same metadata a GET would receive returned as HTTP headers. There is no body returned.

37.GET /{cache}/{element}

Gets the element value.

37.HEAD /{cache}/{element}

Gets the element's metadata.

37.PUT /{cache}/{element}

Puts an element into the Cache.

The time to live of new Elements defaults to that for the cache. This may be overridden by setting the HTTP request header `ehcacheTimeToLiveSeconds`. Values of 0 to 2147483647 are accepted. A value of 0 means eternal.

37.DELETE / {cache}/{element}

Deletes the element from the cache.

The resource representation for all elements is `*. DELETE/{cache}/* will call <<<cache.removeAll().`

37.1.2.5 Resource Representations

We deal with resource representations rather than resources themselves.

37.Element Resource Representations

When Elements are PUT into the cache, a MIME Type should be set in the request header. The MIME Type is preserved for later use.

The new `MimeTypeByteArray` is used to store the `byte[]` and the `MimeType` in the value field of `Element`.

Some common MIME Types which are expected to be used by clients are:

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

text/plain	Plain text
text/xml	Extensible Markup Language. Defined in RFC 3023
application/json	JavaScript Object Notation JSON. Defined in RFC 4627
application/x-java-serialized-object	A serialized Java object

Because Ehcache is a distributed Java cache, in some configurations the Cache server may contain Java objects that arrived at the Cache server via distributed replication. In this case no MIME Type will be set and the Element will be examined to determine its MIME Type.

Because anything that can be PUT into the cache server must be Serializable, it can also be distributed in a cache cluster i.e. it will be Serializable.

37.1.2.6 RESTful Code Samples

These are RESTful code samples in multiple languages.

37.Curl Code Samples

These samples use the popular curl command line utility.

37.OPTIONS

This example shows how calling OPTIONS causes Ehcache server to respond with the WADL for that resource

```
curl --request OPTIONS http://localhost:8080/ehcache/rest/sampleCache2/2
```

The server responds with:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://research.sun.com/wadl/2006/10">
<resources base="http://localhost:8080/ehcache/rest/">
<resource path="sampleCache2/2">
<method name="HEAD"><response><representation mediaType="
    ...
</resource>
</resources>
</application>
```

37.HEAD

```
curl --head  http://localhost:8080/ehcache/rest/sampleCache2/2
```

The server responds with:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: GlassFish/v3
Last-Modified: Sun, 27 Jul 2008 08:08:49 GMT
ETag: "1217146129490"
Content-Type: text/plain; charset=iso-8859-1
Content-Length: 157
Date: Sun, 27 Jul 2008 08:17:09 GMT
```

37.PUT

```
echo "Hello World" |  curl -S -T -  http://localhost:8080/ehcache/rest/
sampleCache2/3
```

The server will put `Hello World` into `sampleCache2` using key `3`.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

37.GET

```
curl http://localhost:8080/ehcache/rest/sampleCache2/2
```

The server responds with:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
```

37.Ruby Code Samples

37.GET

```
require 'rubygems'
require 'open-uri'
require 'rexml/document'
response = open('http://localhost:8080/ehcache/rest/sampleCache2/2')
xml = response.read
puts xml
```

The server responds with:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
</oldjoke>
```

37.Python Code Samples

37.GET

```
import urllib2
f = urllib2.urlopen('http://localhost:8080/ehcache/rest/sampleCache2/2')
print f.read()
```

The server responds with:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
</oldjoke>
```

37.Java Code Samples

37.Create and Get a Cache and Entry

```
package samples;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
import java.net.URL;
/**
 * A simple example Java client which uses the built-
in java.net.URLConnection.
 *
 * @author BryantR
 * @author Greg Luck
 */
public class ExampleJavaClient {
    private static String TABLE_COLUMN_BASE =
            "http://localhost:8080/ehcache/rest/tableColumn";
    private static String TABLE_COLUMN_ELEMENT =
            "http://localhost:8080/ehcache/rest/tableColumn/1";
    /**
     * Creates a new instance of EHCacheREST
     */
    public ExampleJavaClient() {
    }
    public static void main(String[] args) {
        URL url;
        HttpURLConnection connection = null;
        InputStream is = null;
        OutputStream os = null;
        int result = 0;
        try {
            //create cache
            URL u = new URL(TABLE_COLUMN_BASE);
            HttpURLConnection urlConnection = (HttpURLConnection) u.openConnection();
            urlConnection.setRequestMethod("PUT");
            int status = urlConnection.getResponseCode();
            System.out.println("Status: " + status);
            urlConnection.disconnect();
            //get cache
            url = new URL(TABLE_COLUMN_BASE);
            connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("GET");
            connection.connect();
            is = connection.getInputStream();
            byte[] response1 = new byte[4096];
            result = is.read(response1);
            while (result != -1) {
                System.out.write(response1, 0, result);
                result = is.read(response1);
            }
            if (is != null) try {
                is.close();
            } catch (Exception ignore) {
            }
            System.out.println("reading cache: " + connection.getResponseCode()
                    + " " + connection.getResponseMessage());
            if (connection != null) connection.disconnect();
            //create entry
            url = new URL(TABLE_COLUMN_ELEMENT);
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
            connection = (HttpURLConnection) url.openConnection();
            connection.setRequestProperty("Content-Type", "text/plain");
            connection.setDoOutput(true);
            connection.setRequestMethod("PUT");
            connection.connect();
            String sampleData = "Ehcache is way cool!!!";
            byte[] sampleBytes = sampleData.getBytes();
            os = connection.getOutputStream();
            os.write(sampleBytes, 0, sampleBytes.length);
            os.flush();
            System.out.println("result=" + result);
            System.out.println("creating entry: " + connection.getResponseCode()
                    + " " + connection.getResponseMessage());
            if (connection != null) connection.disconnect();
            //get entry
            url = new URL(TABLE_COLUMN_ELEMENT);
            connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("GET");
            connection.connect();
            is = connection.getInputStream();
            byte[] response2 = new byte[4096];
            result = is.read(response2);
            while (result != -1) {
                System.out.write(response2, 0, result);
                result = is.read(response2);
            }
            if (is != null) try {
                is.close();
            } catch (Exception ignore) {
            }
            System.out.println("reading entry: " + connection.getResponseCode()
                    + " " + connection.getResponseMessage());
            if (connection != null) connection.disconnect();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (os != null) try {
                os.close();
            } catch (Exception ignore) {
            }
            if (is != null) try {
                is.close();
            } catch (Exception ignore) {
            }
            if (connection != null) connection.disconnect();
        }
    }
}
```

37.Scala Code Samples

37.GET

```
import java.net.URL
import scala.io.Source.fromInputStream
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
object ExampleScalaGet extends Application {
  val url = new URL("http://localhost:8080/ehcache/rest/sampleCache2/2")
  fromInputStream(url.openStream).getLines.foreach(print)
}
```

Run it with:

```
scala -e ExampleScalaGet
```

The program outputs:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
```

37.PHP Code Samples

37.GET

```
<?php
$ch = curl_init();
curl_setopt ($ch, CURLOPT_URL, "http://localhost:8080/ehcache/rest/
sampleCache2/3");
curl_setopt ($ch, CURLOPT_HEADER, 0);
curl_exec ($ch);
curl_close ($ch);
?>
```

The server responds with:

```
Hello Ingo
```

37.PUT

```
<?php
$url = "http://localhost:8080/ehcache/rest/sampleCache2/3";
$localfile = "localfile.txt";
$fp = fopen ($localfile, "r");
$ch = curl_init();
curl_setopt($ch, CURLOPT_VERBOSE, 1);
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_PUT, 1);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_INFILE, $fp);
curl_setopt($ch, CURLOPT_INFILESIZE, filesize($localfile));
$http_result = curl_exec($ch);
$error = curl_error($ch);
$http_code = curl_getinfo($ch ,CURLINFO_HTTP_CODE);
curl_close($ch);
fclose($fp);
print $http_code;
print "<br /><br />$http_result";
if ($error) {
   print "<br /><br />$error";
}
?>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

The server responds with:

```
* About to connect() to localhost port 8080 (#0)
*   Trying ::1... * connected
* Connected to localhost (::1) port 8080 (#0)
> PUT /ehcache/rest/sampleCache2/3 HTTP/1.1
Host: localhost:8080
Accept: */*
Content-Length: 11
Expect: 100-continue
< HTTP/1.1 100 Continue
< HTTP/1.1 201 Created
< Location: http://localhost:8080/ehcache/rest/sampleCache2/3
< Content-Length: 0
< Server: Jetty(6.1.10)
<
* Connection #0 to host localhost left intact
* Closing connection #0
```

37.1.3 Creating Massive Caches with Load Balancers and Partitioning

The RESTful Ehcache Server is designed to achieve massive scaling using data partitioning - all from a RESTful interface. The largest Ehcache single instances run at around 20GB in memory. The largest disk stores run at 100Gb each. Add nodes together, with cache data partitioned across them, to get larger sizes. 50 nodes at 20GB gets you to 1 Terabyte.

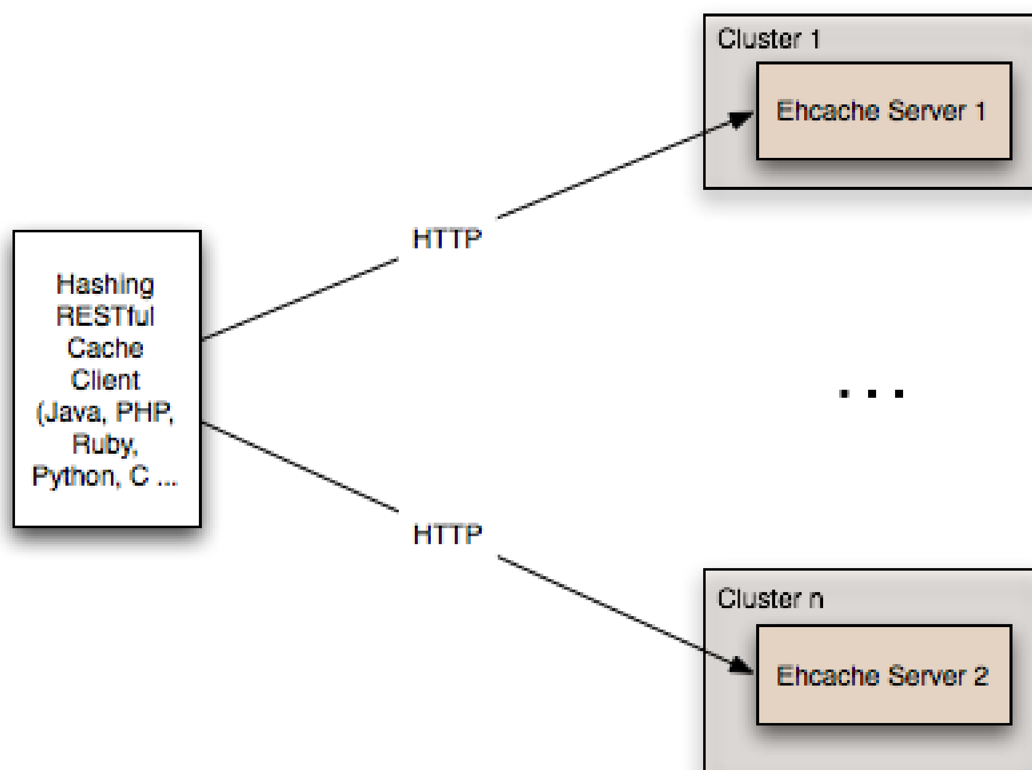
Two deployment choices need to be made:

- where is partitoning performed, and
- is redundancy required?

These choices can be mixed and matched with a number of different deployment topologies.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

37.1.3.1 Non-redundant, Scalable with client hash-based routing



This topology is the simplest. It does not use a load balancer. Each node is accessed directly by the cache client using REST. No redundancy is provided.

The client can be implemented in any language because it is simply a HTTP client.

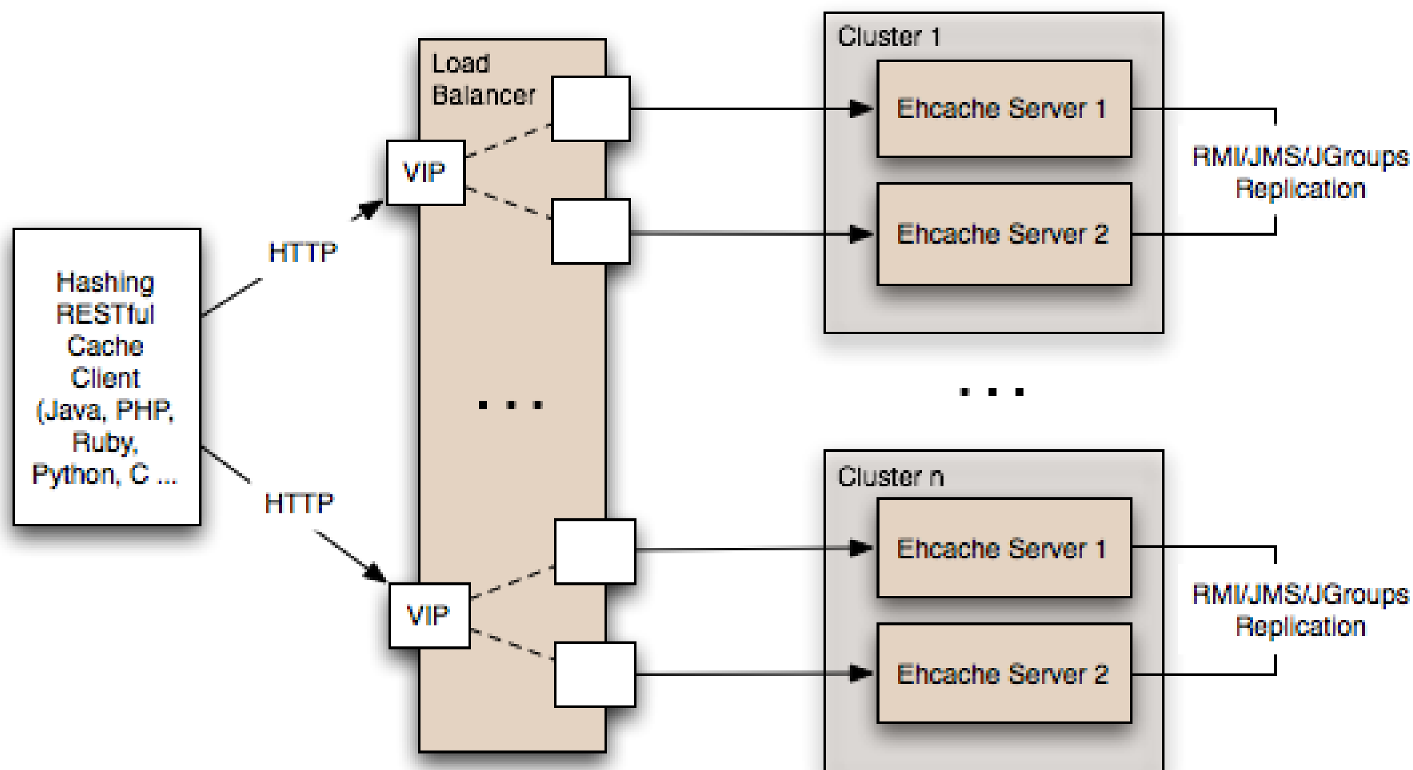
It must work out a partitioning scheme. Simple key hashing, as used by memcached, is sufficient.

Here is a Java code sample:

```
String[] cacheservers = new String[]
{"cacheserver0.company.com", "cacheserver1.company.com",
    "cacheserver2.company.com", "cacheserver3.company.com", "cacheserver4.company.com",
    "cacheserver5.company.com"};
Object key = "123231";
int hash = Math.abs(key.hashCode());
int cacheserverIndex = hash % cacheservers.length;
String cacheserver = cacheservers[cacheserverIndex];
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

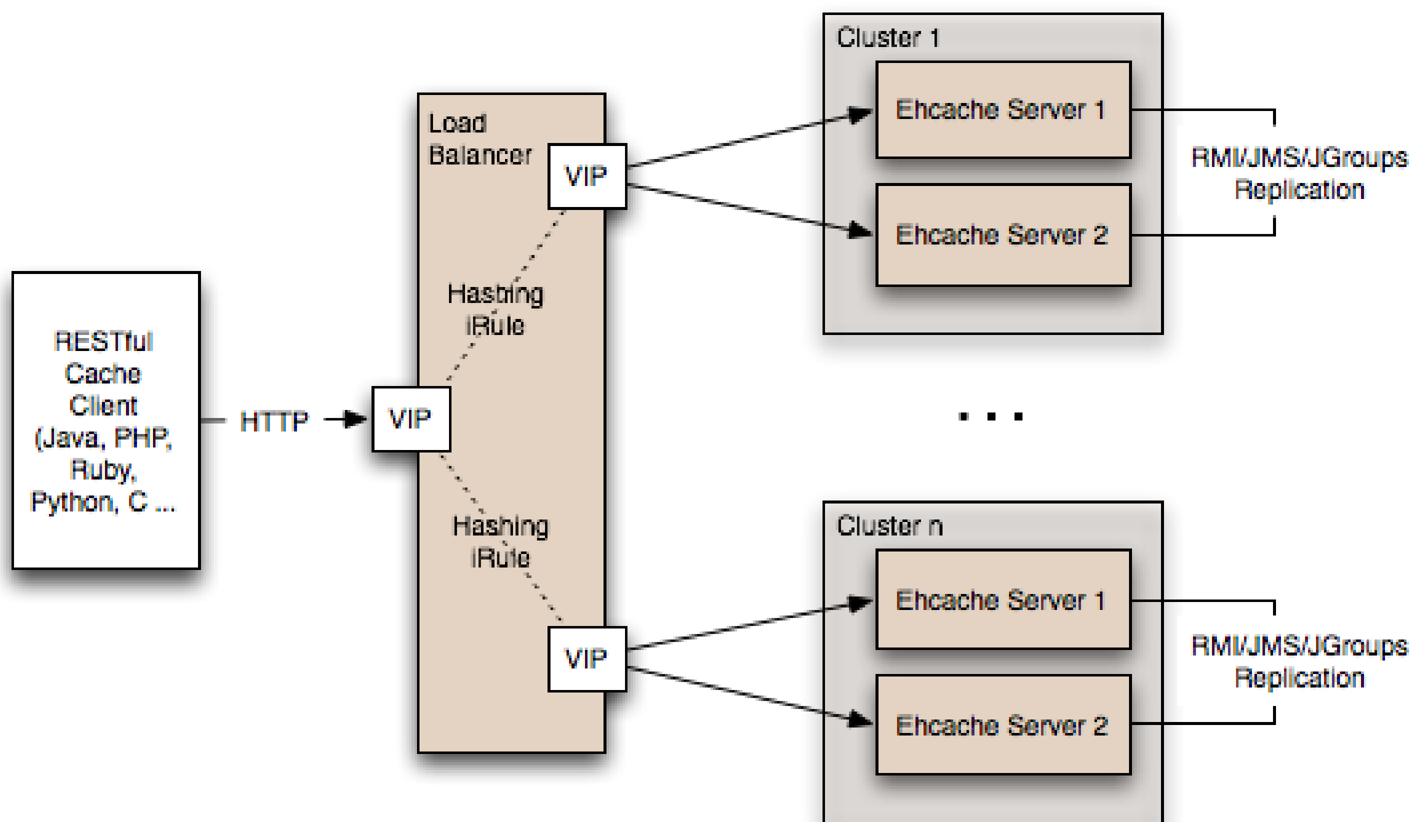
37.1.3.2 Redundant, Scalable with client hash-based routing



Redundancy is added as shown in the above diagram by: Replacing each node with a cluster of two nodes. One of the existing distributed caching options in Ehcache is used to form the cluster. Options in Ehcache 1.5 are RMI and JGroups-based clusters. Ehcache-1.6 will add JMS as a further option. Put each Ehcache cluster behind VIPs on a load balancer.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

37.1.3.3 Redundant, Scalable with load balancer hash-based routing



Many content-switching load balancers support URI routing using some form of regular expressions.

So, you could optionally skip the client-side hashing to achieve partitioning in the load balancer itself.

For example:

```
/ehcache/rest/sampleCache1/[a-h]* => cluster1
/ehcache/rest/sampleCache1/[i-z]* => cluster2
```

Things get much more sophisticated with F5 load balancers, which let you create iRules in the TCL language. So rather than regular expression URI routing, you could implement key hashing-based URI routing. Remember in Ehcache's RESTful server, the key forms the last part of the URI. e.g. In the URI http://cacheserver.company.com/ehcache/rest/sampleCache1/3432 , 3432 is the key.

You hash using the last part of the URI.

See http://devcentral.f5.com/Default.aspx?tabid=63&PageID=153&ArticleID=135&articleType=ArticleView for how to implment a URI hashing iRule on F5 load balancers.

37.1.4 W3C (SOAP) Web Services

The W3C (http://www.w3.org/ is a standards body that defines Web Services as

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

`The World Wide Web is more and more used for application to application communication. The programmatic interfaces made available are referred to as Web services.`
They provide a set of recommendations for achieving this. See http://www.w3.org/2002/ws/.

An interoperability organisation, WS-I http://www.ws-i.org/, seeks to achieve interoperabilty between W3C Web Services. The W3C specifications for SOAP and WSDL are required to meet the WS-I definition.

Ehcache is using Glassfish's libraries to provide it's W3C web services. The project known as Metro follows the WS-I definition.

Finally, OASIS (http://oasis-open.org), defines a Web Services Security specification for SOAP: WS-Security. The current version is 1.1. It provides three main security mechanisms: ability to send security tokens as part of a message, message integrity, and message confidentiality.

Ehcache's W3C Web Services support the stricter WS-I definition and use the SOAP and WSDL specfications.

Specifically:

- The method of operation is in the entity-body of the SOAP envelope and a HTTP header. POST is always used as the HTTP method.
- The scoping information, used to identify the resource to perform the method on, is contained in the SOAP entity-body. The URI path is always the same for a given Web Service - it is the service "endpoint".
- The Web Service is described by and exposes a WSDL (Web Services Description Language) file. It contains the methods, their arguments and what data types are used.
- The WS-Security SOAP extensions are supported

37.1.4.1 W3C Web Services API

The Ehcache RESTFul Web Services API exposes the singleton CacheManager, which typically has been configured in ehcache.xml or an IoC container. Multiple CacheManagers are not supported.

The API definition is as follows:

- WSDL - EhcacheWebServiceEndpointService.wsdl
- Types - EhcacheWebServiceEndpointService_schema1.xsd

37.1.4.2 Security

By default no security is configured. Because it is simply a Servlet 2.5 web application, it can be secured in all the usual ways by configuration in the web.xml.

In addition the cache server supports the use of XWSS 3.0 to secure the Web Service. See https://xwss.dev.java.net/. All required libraries are packaged in the war for XWSS 3.0.

A sample, commented out server_security_config.xml is provided in the WEB-INF directory. XWSS automatically looks for this configuration file.

A simple example, based on an XWSS example, `net.sf.ehcache.server.soap.SecurityEnvironmentHandler`, which looks for a password in a System property for a given username is included. This is not recommended for production use but is handy when you are getting started with XWSS.

To use XWSS:

Add configuration in accordance with XWSS to the server_security_config.xml file. Create a class which implements the `CallbackHandler` interface and provide its fully qualified path in the `SecurityEnvironmentHandler` element.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

The integration test `EhcacheWebServiceEndpoint` test shows how to use the XWSS client side. On the client side, configuration must be provided in a file called `client_security_config.xml` must be in the root of the classpath.

To add client credentials into the SOAP request do:

```
cacheService = new EhcacheWebServiceEndpointService().getEhcacheWebServiceEndpointPort(
//add security credentials
((BindingProvider)cacheService).getRequestContext().put(BindingProvider.USERNAME_PROPER
 "Ron");
((BindingProvider)cacheService).getRequestContext().put(BindingProvider.PASSWORD_PROPER
 "noR");
String result = cacheService.ping();
```

37.1.5 Requirements

37.1.5.1 Java

Java 5 or 6

37.1.5.2 Web Container (WAR packaged version only)

The standalone server comes with its own embedded Glassfish web container.

The web container must support the Servlet 2.5 specification.

The following web container configuration have been tested:

- Glassfish V2/V3
- Tomcat 6
- Jetty 6

37.1.6 Downloading

The server is available as follows:

37.1.6.1 Sourceforge

Download here.

There are two tarball archives in tar.gz format:

- ehcache-server - this contains the WAR file which must be deployed in your own web container.
- ehcache-standalone-server - this contains a complete standalone directory structure with an embedded Glassfish V3 web container together with shell scripts for starting and stopping.

37.1.6.2 Maven

The Ehcache Server is in the central Maven repository packaged as type *war*. Use the following Maven pom snippet:

```
<dependency>
     <groupId>net.sf.ehcache</groupId>
     <artifactId>ehcache-server</artifactId>
     <version>enter_version_here</version>
     <type>war</type>
</dependency>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

It is also available as a jaronly version, which makes it easier to embed. This version excludes all META-INF and WEB-INF configuration files, and also excludes the ehcache.xml. You need to provide these in your maven project.

```
 <dependency>
     <groupId>net.sf.ehcache</groupId>
     <artifactId>ehcache-server</artifactId>
     <version>enter_version_here</version>
     <type>jar</type>
     <classifier>jaronly</classifier>
</dependency>
```

37.1.7 Installation

37.1.7.1 Installing the WAR

Use your Web Container's instructions to install the WAR or include the WAR in your project with Maven's war plugin.

Web Container specific configuration is provided in the WAR as follows:

- sun-web.xml - Glassfish V2/V3 configuration
- jetty-web.xml - Jetty V5/V6 configuration

  Tomcat V6 passes all integration tests. It does not require a specific configuration.

37.1.7.2 Configuring the Web Application

Expand the WAR.

Edit the web.xml.

37.Disabling the RESTful Web Service

Comment out the RESTful web service section.

37.Disabling the SOAP Web Service

Comment out the RESTful web service section.

37.Configuring Caches

The ehcache.xml configuration file is located in WEB-INF/classes/ehcache.xml.

Follow the instructions in this config file, or the core Ehcache instructions to configure.

37.SOAP Web Service Security

37.1.8 Installing the Standalone Server

The WAR also comes packaged with a standalone server, based on Glassfish V3 Embedded.

The quick start is:

- Untar the download
- bin/start.sh to start. By default it will listen on port 8080, with JMX listening on port 8081, will have both RESTful and SOAP web services enabled, and will use a sample Ehcache configuration from the WAR module.
- bin/stop.sh to stop

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

37.1.8.1 Configuring the Standalone Server

Configuration is by editing the war/web.xml file as per the instructions for the WAR packaging.

37.1.8.2 Starting and Stopping the Standalone Server

37.Using Commons Daemon jsvc

jsvc creates a daemon which returns once the service is started. jsvc works on all common Unix-based operating systems including Linux, Solaris and Mac OS X.

It creates a pid file in the pid directory.

This is a Unix shell script that starts the server as a daemon.

To use jsvc you must install the native binary jsvc from the Apache Commons Daemon project. The source for this is distributed in the bin directory as jsvc.tar.gz. Untar it and follow the instructions for building it or download a binary from the Commons Daemon project.

Convenience shell scripts are provided as follows:

start - `daemon_start.sh`

stop - `daemon_stop.sh`

jsvc is designed to integrate with Unix System 5 initialization scripts. (/etc/rc.d)

You can also send Unix signals to it. Meaningful ones for the Ehcache Standalone Server are:

No	Meaning
1	HUP
2	INT
9	KILL
15	TERM

37.Executable jar

The server is also packaged as an executable jar for developer convenience which will work on all operating systems.

A convenience shell script is provided as follows:

start - startup.sh

From the bin directory you can also invoke the following command directly:

```
    unix    - java -jar ../lib/ehcache-standalone-server-0.7.jar 8080 ../
war
    windows - java -jar ..\lib\ehcache-standalone-server-0.7.jar 8080 ..
\war
```

37.1.9 Monitoring

The CacheServer registers Ehcache MBeans with the platform MBeanServer.

Remote monitoring of the MBeanServer is the responsibility of the Web Container or Application Server vendor.

For example, some instructions for Tomcat are here: https://wiki.internet2.edu/confluence/display/CPD/Monitoring+Tomcat+with+JMX

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

See your Web Container documentation for how to do this for your web container.

37.1.9.1 Remotely Monitoring the Standalone Server with JMX

The standalone server automatically exposes the MBeanServer on a port 1 higher than the HTTP listening port.

To connect with `JConsole` simply fire up JConsole, enter the host in the Remote field and portIn the above example that is

```
192.168.1.108:8686
```

Then click `Connect`.

To see the Ehcache MBeans, click on the `Mbeans` tab and expand the `net.sf.ehcache` tree node.

You will see something like the following.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

CacheStatistics MBeans in JConsole

Of course, from there you can hook the Cache Server up to your monitoring tool of choice. See the chapter on JMX Management and Monitoring for more information.

37.1.10 Download

Download the ehcache-standalone-server from http://sourceforge.net/projects/ehcache/files/ehcache-server.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

37.1.11 FAQ

37.1.11.1 Does Cache Server work with WebLogic?

Yes (we have tested 10.3.2), but the SOAP libraries are not compatible. Either comment out the SOAP service from web.xml or do the following:

1. Unzip ehcache-server.war to a folder called ehcache
2. Remove the following jars from WEB-INF/lib: jaxws-rt-2.1.4.jar metro-webservices-api-1.2.jar metro-webservices-rt-1.2.jar metro-webservices-tools-1.2.jar
3. Deploy the folder to WebLogic.
4. Use the soapUI GUI in WebLogic to add a project from: http:// *hostname*: *port*/ehcache/soap/EhcacheWebServiceEndpoint?wsdl

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

38 Explicit Locking API

38.1 Explicit Locking

This package contains an implementation of an Ehcache which provides for explicit locking, using Read and Write locks.

It is possible to get more control over Ehcache's locking behaviour to allow business logic to apply an atomic change with guaranteed ordering across one or more keys in one or more caches. It can theefore be used as a custom alternative to `XA` Transactions or `Local` transactions.

With that power comes a caution. It is possible to create deadlocks in your own business logic using this API.

Note that prior to Ehcache 2.4, this API was implemented as a CacheDecorator and was available in the `ehcache-explicitlocking` module. It is now built into the `core` module.

38.1.1 The API

The following methods are available on `Cache` and `Ehcache`.

```
/**
 * Acquires the proper read lock for a given cache key
 *
 * @param key - The key that retrieves a value that you want to protect via locking
 */
public void acquireReadLockOnKey(Object key) {
    this.acquireLockOnKey(key, LockType.READ);
}
/**
 * Acquires the proper write lock for a given cache key
 *
 * @param key - The key that retrieves a value that you want to protect via locking
 */
public void acquireWriteLockOnKey(Object key) {
    this.acquireLockOnKey(key, LockType.WRITE);
}
/**
 * Try to get a read lock on a given key. If can't get it in timeout millis then
 * return a boolean telling that it didn't get the lock
 *
 * @param key - The key that retrieves a value that you want to protect via locking
 * @param timeout - millis until giveup on getting the lock
 * @return whether the lock was awarded
 * @throws InterruptedException
 */
public boolean tryReadLockOnKey(Object key, long timeout) throws InterruptedExcepti
    Sync s = getLockForKey(key);
    return s.tryLock(LockType.READ, timeout);
}
/**
 * Try to get a write lock on a given key. If can't get it in timeout millis then
 * return a boolean telling that it didn't get the lock
 *
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
     * @param key - The key that retrieves a value that you want to protect via locking
     * @param timeout - millis until giveup on getting the lock
     * @return whether the lock was awarded
     * @throws InterruptedException
     */
    public boolean tryWriteLockOnKey(Object key, long timeout) throws InterruptedExcept
        Sync s = getLockForKey(key);
        return s.tryLock(LockType.WRITE, timeout);
    }
    /**
     * Release a held read lock for the passed in key
     *
     * @param key - The key that retrieves a value that you want to protect via locking
     */
    public void releaseReadLockOnKey(Object key) {
        releaseLockOnKey(key, LockType.READ);
    }
    /**
     * Release a held write lock for the passed in key
     *
     * @param key - The key that retrieves a value that you want to protect via locking
     */
    public void releaseWriteLockOnKey(Object key) {
        releaseLockOnKey(key, LockType.WRITE);
    }
```

38.1.2 Example

Here is a brief example:

```
String key = "123";
Foo val = new Foo();
cache.acquireWriteLockOnKey(key);
try {
        cache.put(new Element(key, val));
} finally {
        cache.releaseWriteLockOnKey(key);
}
...sometime later
String key = "123";
cache.acquireWriteLockOnKey(key);
try {
        Object cachedVal = cache.get(key).getValue();
        cachedVal.setSomething("abc");
        cache.put(new Element(key, cachedVal));
 } finally {
   cache.releaseWriteLockOnKey(key);
 }
```

38.1.3 Supported Topologies

Explicit Locking is supported in Ehcache standalone and also in Distributed Ehcache when the cache is configured with `consistency=strong`.

It is not supported in Replicated Ehcache.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

38.1.4 How it works

A READ lock does not prevent other READers from also acquiring a READ lock and reading. A READ lock cannot be obtained if there is an outstanding WRITE lock - it will queue.

A WRITE lock cannot be obtained while there are outstanding READ locks - it will queue.

In each case the lock should be released after use to avoid locking problems. The lock release should be in a `finally` block.

If before each read you acquire a READ lock and then before each write you acquire a WRITE lock, then an isolation level akin to READ_COMMITTED is achieved.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

39 BlockingCache and SelfPopulatingCache

39.1 BlockingCache and SelfPopulatingCache

The `net.sf.ehcache.constructs` package contains some applied caching classes which use the core classes to solve everyday caching problems.

39.1.1 Blocking Cache

Imagine you have a very busy web site with thousands of concurrent users. Rather than being evenly distributed in what they do, they tend to gravitate to popular pages. These pages are not static, they have dynamic data which goes stale in a few minutes. Or imagine you have collections of data which go stale in a few minutes. In each case the data is extremely expensive to calculate.

Let's say each request thread asks for the same thing. That is a lot of work. Now, add a cache. Get each thread to check the cache; if the data is not there, go and get it and put it in the cache. Now, imagine that there are so many users contending for the same data that in the time it takes the first user to request the data and put it in the cache, 10 other users have done the same thing. The upstream system, whether a JSP or velocity page, or interactions with a service layer or database are doing 10 times more work than they need to.

Enter the BlockingCache.

It is blocking because all threads requesting the same key wait for the first thread to complete. Once the first thread has completed the other threads simply obtain the cache entry and return.

The BlockingCache can scale up to very busy systems. Each thread can either wait indefinitely, or you can specify a timeout using the `timeoutMillis` constructor argument.

39.1.2 SelfPopulatingCache

You want to use the BlockingCache, but the requirement to always release the lock creates gnarly code. You also want to think about what you are doing without thinking about the caching.

Enter the SelfPopulatingCache. The name SelfPopulatingCache is synonymous with Pull-through cache, which is a common caching term. SelfPopulatingCache though always is in addition to a BlockingCache.

SelfPopulatingCache uses a `CacheEntryFactory`, that given a key, knows how to populate the entry.

Note: JCache inspired getWithLoader and getAllWithLoader directly in `Ehcache` which work with a `CacheLoader` may be used as an alternative to SelfPopulatingCache.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

40 OpenJPA Caching

40.1 OpenJPA Caching Provider

Ehcache easily integrates with the OpenJPA persistence framework.

40.1.1 Installing

To use it, add a Maven dependency for ehcache-openjpa.

```
<groupId>net.sf.ehcache</groupId>
<artifactId>ehcache-openjpa</artifactId>
<version>0.1</version>
```

or download from downloads.

40.1.2 Configuration

Set OpenJPA#s `openjpa.QueryCache` to `ehcache` and `openjpa.DataCacheManager` to `ehcache`. That#s it!

See http://openjpa.apache.org/builds/1.0.2/apache-openjpa-1.0.2/docs/manual/ref_guide_caching.html for more on caching in OpenJPA.

40.1.3 Default Cache

As with Hibernate, Ehcache's OpenJPA module (from 0.2) uses the `defaultCache` configured in ehcache.xml to create caches.

For production, we recommend configuring a cache configuration in ehcache.xml for each cache, so that it may be correctly tuned.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

41 Grails Caching

41.1 Using Grails and Ehcache

41.1.1 Using Ehcache as a Second Level Caching Provider for Hibernate within Grails

Grails 1.2RC1 and higher use Ehcache as the default Hibernate second level cache. However earlier versions of Grails ship with the Ehcache library and are very simple to enable.

The following steps show how to configure Grails to use Ehcache. For 1.2RC1 and higher some of these steps are already done for you.

41.1.2 Configuring Ehcache as the second level Hibernate cache

Edit `DataSource.groovy` as follows:

```
    hibernate {
    cache.use_second_level_cache=true
    cache.use_query_cache=true
    cache.provider_class='org.hibernate.cache.EhCacheProvider'
}
```

41.1.3 Overriding defaults by specifying cache configurations

As is usual with Hibernate, it will use the `defaultCache` configuration as a template to create new caches as required. For production use you often want to customise the cache configuration. To do so, add an ehcache.xml configuration file to the `conf` directory (the same directory that contains `DataSource.groovy`).

A sample ehcache.xml which works with the Book demo app and is good as a starter configuration for Grails is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd" >
    <diskStore path="java.io.tmpdir"/>
    <cacheManagerEventListenerFactory class="" properties=""/>
    <defaultCache
            maxElementsInMemory="10000"
            eternal="false"
            timeToLiveSeconds="120"
            overflowToDisk="false"
            diskPersistent="false"
            />
    <cache name="Book"
           maxElementsInMemory="10000"
           timeToIdleSeconds="300"
            />
    <cache name="org.hibernate.cache.UpdateTimestampsCache"
           maxElementsInMemory="10000"
           timeToIdleSeconds="300"
            />
    <cache name="org.hibernate.cache.StandardQueryCache"
           maxElementsInMemory="10000"
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
            timeToIdleSeconds="300"
             />
</ehcache>
```

41.1.4 Springcache Plugin

The Springcache plugin allows you to easily add the following functionality to your Grails project:

- Caching of Spring bean methods (typically Grails service methods).
- Caching of page fragments generated by Grails controllers.
- Cache flushing when Spring bean methods or controller actions are invoked.

  The plugin depends on the EhCache and EhCache-Web libraries.

  See Springcache Plugin}, part of the Grails project for more information.

41.1.5 Clustering Web Sessions

This is not handled by Ehcache, but by a sister product from Terracotta, web sessions.

See http://gquick.blogspot.com/2010/03/clustering-grails-app-with-terracotta.html for a great intro on getting this going with Grails and Tomcat.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

42 JRuby Caching

42.1 Rails and JRuby Caching

jruby-ehcache is a JRuby Ehcache library which makes a commonly used subset of Ehcache's API available to JRuby. All of the strength of Ehcache is there, including BigMemory and the ability to cluster with Terracotta.

It can be used directly via its own API, or as a Rails caching provider.

42.1.1 Installation

Ehcache JRuby integration is provided by the jruby-ehcache gem. To install it simply execute (note: you may need to use "sudo" to install gems on your system):

```
jgem install jruby-ehcache
```

If you also want Rails caching support, also install the correct gem for your Rails version:

```
jgem install jruby-ehcache-rails2 # for Rails 2
jgem install jruby-ehcache-rails3 # for Rails 3
```

42.1.2 Configuring Ehcache

Configuring Ehcache for JRuby is done using the same ehcache.xml file as used for native Java Ehcache. The ehcache.xml file can be placed either in your CLASSPATH or, alternatively, can be placed in the same directory as the Ruby file in which you create the CacheManager object from your Ruby code. In a Rails application, the ehcache.xml file should reside in the config directory of the Rails application.

42.1.3 Dependencies

- JRuby 1.5 and higher
- Rails 2 for the jruby-ehcache-rails2
- Rails 3 for the jruby-ehcache-rails3
- Ehcache 2.4.2 is the declared dependency, although any version of Ehcache will work

As usual these should all be installed with jgem

42.1.4 Using the jruby-ehcache API directly

42.1.4.1 To make Ehcache available to JRuby

```
require 'ehcache'
```

Note that, because jruby-ehcache is provided as a Ruby Gem, you must make your Ruby interpreter aware of Ruby Gems in order to load it. You can do this by either including -rubygems on your jruby command line, or you can make Ruby Gems available to JRuby globally by setting the RUBYOPT environment variable as follows:

```
export RUBYOPT=rubygems
```

42.1.4.2 Creating a CacheManager

To create a CacheManager, which you do once when the application starts:

```
manager = Ehcache::CacheManager.new
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

42.1.4.3 Accessing an existing Cache

To access a cache called "sampleCache1":

```
cache = manager.cache("sampleCache1")
```

42.1.4.4 Creating a Cache

To create a new cache from the defaultCache

```
cache = manager.cache
```

42.1.4.5 Putting in a cache

```
cache.put("key", "value", {:ttl => 120})
```

42.1.4.6 Getting from a cache

```
cache.get("key")  # Returns the Ehcache Element object
cache["key"]      # Returns the value of the element directly
```

42.1.4.7 Shutting down the CacheManager

This is only when you shut your application down.

It is only necessary to call this if the cache is `diskPersistent` or is clustered with Terracotta, but it is always a good idea to do it.

```
manager.shutdown
```

42.1.5 Complete Example

```
class SimpleEhcache
  #Code here
  require 'ehcache'
  manager = Ehcache::CacheManager.new
  cache = manager.cache
  cache.put("answer", "42", {:ttl => 120})
  answer = cache.get("answer")
  puts "Answer: #{answer.value}"
  question = cache["question"] || 'unknown'
  puts "Question: #{question}"
  manager.shutdown
end
```

As you can see from the example, you create a cache using CacheManager.new, and you can control the "time to live" value of a cache entry using the :ttl option in cache.put. Note that not all of the Ehcache API is currently exposed in the JRuby API, but most of what you need is available and we plan to add a more complete API wrapper in the future.

42.1.6 Using ehcache from within Rails

42.1.6.1 The ehcache.xml file

Configuration of Ehcache is still done with the ehcache.xml file, but for Rails applications you must place this file in the config directory of your Rails app.

Also note that you must use JRuby to execute your Rails application, as these gems utilize JRuby's Java integration to call the Ehcache API.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

With this configuration out of the way, you can now use the Ehcache API directly from your Rails controllers and/or models. You could of course create a new Cache object everywhere you want to use it, but it is better to create a single instance and make it globally accessible by creating the Cache object in your Rails environment.rb file.

For example, you could add the following lines to config/environment.rb:

```
require 'ehcache'
EHCACHE = Ehcache::CacheManager.new.cache
```

By doing so, you make the EHCACHE constant available to all Rails-managed objects in your application. Using the Ehcache API is now just like the above JRuby example.

If you are using Rails 3 then you have a better option at your disposal: the built-in Rails 3 caching API. This API provides an abstraction layer for caching underneath which you can plug in any one of a number of caching providers. The most common provider to date has been the memcached provider, but now you can also use the Ehcache provider.

Switching to the Ehcache provider requires only one line of code in your Rails environment file (e.g. development.rb or production.rb):

```
config.cache_store = :ehcache_store, {
                            :cache_name => 'rails_cache',
                            :ehcache_config => 'ehcache.xml'
                        }
```

This configuration will cause the Rails.cache API to use Ehcache as its cache store. The :cache_name and :ehcache_config are both optional parameters, the default values for which are shown in the above example. The value of the :ehcache_config parameter can be either an absolute path or a relative path, in which case it is interpreted relative to the Rails app's config directory.

A very simple example of the Rails caching API is as follows:

```
Rails.cache.write("answer", "42")
Rails.cache.read("answer")  # => '42'
```

Using this API, your code can be agnostic about the underlying provider, or even switch providers based on the current environment (e.g. memcached in development mode, Ehcache in production).

The write method also supports options in the form of a Hash passed as the final parameter. The following options are supported:

- unlessExist, ifAbsent (boolean) - If true, use the putIfAbsent method
- elementEvictionData (ElementEvictionData)
- eternal (boolean)
- timeToIdle, tti (int)
- timeToLive, ttl, expiresIn (int)
- version (long)

These options are passed to the write method as Hash options using either camelCase or underscore notation, as in the following example:

```
Rails.cache.write('key', 'value', :time_to_idle => 60.seconds, :timeToLive => 600.secon
```

42.1.6.2 Turn on caching in your controllers

You can also configure Rails to use Ehcache for its automatic action caching and fragment caching, which is the most common method for caching at the controller level. To enable this, you must configure Rails to perform controller caching, and then set Ehcache as the provider in the same way as for the Rails cache API:

```
config.action_controller.perform_caching = true
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
config.action_controller.cache_store = :ehcache_store
```

42.1.7 Sample Rails application

The easiest way to get started is to play with a simple sample app. We provide a simple Rails application which stores an integer value in a cache along with increment and decrement operations.

The sample app shows you how to use Ehcache as a caching plugin and how to use it directly from the Rails caching API. It is a simple demo application demonstrating the use of Ehcache in a Rails 3 environment. This demo requires JRuby 1.5.0 or later.

42.1.7.1 Checking it out

```
svn checkout http://svn.terracotta.org/svn/forge/projects/ehcache-rails-demo/trunk ehcache-rails-demo
```

42.1.7.2 Dependencies

To start the demo, make sure you are using JRuby 1.5.0 or later.

The demo uses sqlite3 which needs to be installed on your OS (it is by default on Mac OS X).

There is a Gemfile which will pull down all of the required Ruby dependencies using Bundler.

From the ehcache-rails-demo directory:

```
jgem install bundler
jruby -S bundle install
```

42.1.7.3 Starting the demo

You can start the demo application with the following command:

```
jruby -S rails server -e production
```

42.1.7.4 Exploring the demo

To use the demo application, open a web browser to the following URL:

http://localhost:3000/cache/index

This will display a simple screen allowing you to manipulate cached values either through the Ehcache API directly, or through the Rails.cache API backed by Ehcache.

42.1.8 Leveraging the power of Ehcache

Once you have the Ruby/Rails caching modules up and running with Ehcache you can then go on to leverage the power of Ehcache through for example creating a distributed cache backed by Terracotta.

There are no limits on what you can do. Please see the rest of this documentation.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

43 Glassfish HowTo

43.1 Glassfish How To & FAQ

The maintainer uses Ehcache in production with Glassfish. This chapter provides a Glassfish HOWTO.

43.1.1 Versions

Ehcache has been tested with and is used in production with Glassfish V1, V2 and V3.

In particular:

- Ehcache 1.4 - 1.7 has been tested with Glassfish 1 and 2.
- Ehcache 2.0 has been tested with Glassfish 3.

43.1.2 Usage and Troubleshooting

43.1.2.1 How To Package A Sample Application Using Ehcache and Deploy to Glassfish

Ehcache comes with a sample web application which is used to test the page caching. The page caching is the only area that is sensitive to the Application Server. For Hibernate and general caching, it is only dependent on your Java version.

From a checkout of Ehcache run the following from the `core` directory:

You need:

- a Glassfish installation.
- a `GLASSFISH_HOME` environment variable defined.
- `$GLASSFISH_HOME/bin` added to your `PATH`

  Do the following:

```
# To package and deploy to domain1:
ant deploy-default-web-app-glassfish
# Start domain1:
asadmin start-domain domain1
# Stop domain1:
asadmin stop-domain domain1
# Overwrite the config with our own which changes the port to 9080:
ant glassfish-configuration
# Start domain1:
asadmin start-domain domain1
```

  You can then run the web tests in the web package or point your browser at `http://localhost:9080`.

  See for a quickstart to Glassfish.

43.1.2.2 How to get around the EJB Container restrictions on thread creation

When Ehcache is running in the EJB Container, for example for Hibernate caching, it is in technical breach of the EJB rules. Some app servers let you override this restriction.

I am not exactly sure how this in done in Glassfish. For a number of reasons we run Glassfish without the Security Manager, and we do not have any issues.

In domain.xml ensure that the following is not included.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
<jvm-options>-Djava.security.manager</jvm-options>
```

43.1.3 Glassfish FAQ

43.1.3.1 Ehcache page caching versions below Ehcache 1.3 get an IllegalStateException in Glassfish.

This issue was fixed in Ehcache 1.3.

43.1.3.2 I get a `Could not ungzip. Heartbeat will not be working. Not in GZIP format` reported from PayloadUtil exception when using Ehcache with my Glassfish cluster. Why?

Ehcache and Glassfish clustering have nothing to do with each other. The error is caused because Ehcache has received a multicast message from the Glassfish cluster. Ensure that Ehcache clustering has its own unique multicast address different to Glassfish.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

44 Google App Engine Caching

44.1 Google App Engine Caching

44.1.1 Using Ehcache on Google App Engine (GAE)

The ehcache-googleappengine module combines the speed of Ehcache with the scale of Google's memcache and provide the best of both worlds:

- Speed - Ehcache cache operations take a few microseconds, versus around 60ms for Google's provided client-server cache, memcacheg.
- Cost - Because it uses way less resources, it is also cheaper.
- Object Storage - Ehcache in-process cache works with Objects that are not Serializable.

44.1.2 Compatibility

Ehcache is compatible and works with Google App Engine.

Google App Engine provides a constrained runtime which restricts networking, threading and file system access.

44.1.3 Limitations

All features of Ehcache can be used except for the DiskStore and replication. Having said that, there are workarounds for these limitations. See the Recipes section below.

As of June 2009, Google App Engine appears to be limited to a heap size of 100MB. (See http://gregluck.com/blog/?s=limitations for the evidence of this).

44.1.4 Dependencies

Version 2.3 and higher of Ehcache are compatible with Google App Engine.

Older versions will not work.

44.1.5 Configuring ehcache.xml

Make sure the following elements are commented out:

- *diskStore path="java.io.tmpdir"/*
- *cacheManagerPeerProviderFactory class= ../*
- *cacheManagerPeerListenerFactory class= ../*

  Within each cache element, ensure that:
- overFlowToDisk=false or overFlowToDisk is omitted
- diskPersistent=false or diskPersistent is omitted
- no replicators are added
- there is no bootstrapCacheLoaderFactory
- there is no Terracotta configuration

  Copy and past this one to get started.

```
<?xml version="1.0" encoding="UTF-8"?>
<Ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd" >
    <cacheManagerEventListenerFactory class="" properties=""/>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
        <defaultCache
                maxElementsInMemory="10000"
                eternal="false"
                timeToIdleSeconds="120"
                timeToLiveSeconds="120"
                overflowToDisk="false"
                diskPersistent="false"
                memoryStoreEvictionPolicy="LRU"
                />
        <!--Example sample cache-->
        <cache name="sampleCache1"
               maxElementsInMemory="10000"
               maxElementsOnDisk="1000"
               eternal="false"
               timeToIdleSeconds="300"
               timeToLiveSeconds="600"
               memoryStoreEvictionPolicy="LFU"
                />
    </ehcache>
```

44.1.6 Recipes

44.1.6.1 Setting up Ehcache as a local cache in front of memcacheg

The idea here is that your caches are set up in a cache hierarchy. Ehcache sits in front and memcacheg behind. Combining the two lets you elegantly work around limitations imposed by Googe App Engine. You get the benefits of the #s speed of Ehcache together with the umlimited size of memcached.

Ehcache contains the hooks to easily do this.

To update memcached, use a `CacheEventListener`.

To search against memcacheg on a local cache miss, use `cache.getWithLoader()` together with a `CacheLoader` for memcacheg.

44.1.6.2 Using memcacheg in place of a `DiskStore`

In the `CacheEventListener`, ensure that when `notifyElementEvicted()` is called, which it will be when a put exceeds the MemoryStore's capacity, that the key and value are put into memcacheg.

44.1.6.3 Distributed Caching

Configure all notifications in `CacheEventListener` to proxy throught to memcacheg.

Any work done by one node can then be shared by all others, with the benefit of local caching of frequently used data.

44.1.6.4 Dynamic Web Content Caching

Google App Engine provides acceleration for files declared static in appengine-web.xml.

e.g.

```
<static-files>
    <include path="/**.png" />
    <exclude path="/data/**.png" />
</static-files>
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

You can get acceleration for dynamic files using Ehcache's caching filters as you usually would.

See the Web Caching chapter.

44.1.7 Google App Engine FAQ

44.1.7.1 I get an error `java.lang.NoClassDefFoundError: java.rmi.server.UID is a restricted class`

You are using a version of Ehcache prior to 1.6.

44.1.8 Sample application

The easiest way to get started is to play with a simple sample app. We provide a simple Rails application which stores an integer value in a cache along with increment and decrement operations.

The sample app shows you how to use ehcache as a caching plugin and how to use it directly from the Rails caching API.

Checkout http://svn.terracotta.org/svn/forge/projects/ehcache-rails-demo/ terracotta_community_login a Maven-based performance comparisons between different store configurations.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

45 Tomcat Issues and Best Practices

45.1 Tomcat Issues and Best Practices

Ehcache is probably used most commonly with Tomcat. This chapter documents some known issues with Tomcat and recommended practices.

Ehcache's own caching and gzip filter integration tests run against Tomcat 5.5 and Tomcat 6. Tomcat will continue to be tested against ehcache. Accordingly Tomcat is tier one for ehcache.

45.1.1 Tomcat Known Issues

Because Tomcat is so widely used, over time a list of known issues has been compiled. These issues and their solutions are listed below.

45.1.1.1 Problem rejoining a cluster after a reload

If I restart/reload a web application in Tomcat that has a CacheManager that is part of a cluster, the CacheManager is unable to rejoin the cluster. If I set logging for net.sf.ehcache.distribution to FINE I see the following exception: "FINE: Unable to lookup remote cache peer for .... Removing from peer list. Cause was: error unmarshalling return; nested exception is: java.io.EOFException.

The Tomcat and RMI class loaders do not get along that well. Move ehcache.jar to $TOMCAT_HOME/common/lib. This fixes the problem. This issue happens with anything that uses RMI, not just ehcache.

45.1.1.2 In development, there appear to be class loader memory leak as I continually redeploy my web application.

There are lots of causes of memory leaks on redeploy. Moving Ehcache out of the WAR and into $TOMCAT/common/lib fixes this leak.

45.1.1.3 net.sf.ehcache.CacheException: Problem starting listener for RMICachePeer ...

I get net.sf.ehcache.CacheException: Problem starting listener for RMICachePeer ... java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is: java.net.MalformedURLException: no protocol: Files/Apache. What is going on?

This issue occurs to any RMI listener started on Tomcat whenever Tomcat has spaces in its installation path.

It is is a JDK bug which can be worked around in Tomcat.

See http://archives.java.sun.com/cgi-bin/wa?A2=ind0205&L=rmi-users&P=797 and http://www.ontotext.com/kim/doc/sys-doc/faq-howto-bugs/known-bugs.html.

The workaround is to remove the spaces in your tomcat installation path.

45.1.1.4 Multiple Host Entries in Tomcat's server.xml stops replication from occurring

The presence of multiple *Host* entries in Tomcat's server.xml prevents replication from occuring. The issue is with adding multiple hosts on a single Tomcat connector. If one of the hosts is localhost and another starts with v, then the caching between machines when hitting localhost stops working correctly.

The workaround is to use a single *Host* entry or to make sure they don't start with "v".

Why this issue occurs is presently unknown, but is Tomcat specific.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

46 JSR107 (JCACHE) Support

46.1 JSR107 (JCACHE) Support

46.1.1 JSR107 Implementation

Ehcache provides a preview implementation of JSR107 via the `net.sf.cache.jcache` package.

WARNING: JSR107 is still being drafted with the Ehcache maintainer as Co Spec Lead. This package will continue to change until JSR107 is finalised. No attempt will be made to maintain backward compatiblity between versions of the package. It is therefore recommended to use Ehcache's proprietary API directly.

46.1.2 Using JCACHE

46.1.2.1 Creating JCaches

JCaches can be created in two ways:

- as an Ehcache decorator
- from JCache's CacheManager

46.Creating a JCache using an Ehcache decorator

`manager` in the following sample is an net.sf.ehcache.CacheManager

```
net.sf.jsr107cache.Cache cache = new JCache(manager.getCache("sampleCacheNoIdle"), null
```

46.Creating a JCache from an existing Cache in Ehcache's CacheManager

This is the recommended way of using JCache. Caches can be configured in ehcache.xml and wrapped as JCaches with the getJCache method of CacheManager.

`manager` in the following sample is an net.sf.ehcache.CacheManager

```
        net.sf.jsr107cache.Cache cache = manager.getJCache("sampleCacheNoIdle");
```

46.Adding a JCache to Ehcache's CacheManager

`manager` in the following sample is an net.sf.ehcache.CacheManager

```
        Ehcache Ehcache = new net.sf.ehcache.Cache(...);
        net.sf.jsr107cache.Cache cache = new JCache(ehcache);
        manager.addJCache(cache);
```

46.Creating a JCache using the JCache CacheManager

Warning: The JCache CacheManager is unworkable and will very likely be dropped in the final JCache as a Class. It will likely be replaced with a CacheManager interface.

The JCache CacheManager only works as a singleton. You obtain it with `getInstance`

The CacheManager uses a CacheFactory to create Caches. The CacheFactory is specified using the Service Provider Interface mechanism introduced in JDK1.3.

The factory is specified in the `META-INF/services/net.sf.jsr107cache.CacheFactory` resource file. This would normally be packaged in a jar. The default value for the Ehcache implementation is `net.sf.ehcache.jcache.JCacheFactory`

The configuration for a cache is assembled as a map of properties. Valid properties can be found in the JavaDoc for the JCacheFactory.createCache() method.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

See the following full example.

```
CacheManager singletonManager = CacheManager.getInstance();
CacheFactory cacheFactory = singletonManager.getCacheFactory();
assertNotNull(cacheFactory);
Map config = new HashMap();
config.put("name", "test");
config.put("maxElementsInMemory", "10");
config.put("memoryStoreEvictionPolicy", "LFU");
config.put("overflowToDisk", "true");
config.put("eternal", "false");
config.put("timeToLiveSeconds", "5");
config.put("timeToIdleSeconds", "5");
config.put("diskPersistent", "false");
config.put("diskExpiryThreadIntervalSeconds", "120");
Cache cache = cacheFactory.createCache(config);
singletonManager.registerCache("test", cache);
```

46.1.2.2 Getting a JCache

Once a cache is registered in CacheManager, you get it from there.

The following example shows how to get a Cache.

```
manager = CacheManager.getInstance();
Ehcache Ehcache = new net.sf.ehcache.Cache("UseCache", 10,
MemoryStoreEvictionPolicy.LFU,
 false, null, false, 10, 10, false, 60, null);
manager.registerCache("test", new JCache(ehcache, null));
Cache cache = manager.getCache("test");
```

46.1.2.3 Using a JCache

The JavaDoc is the best place to learn how to use a JCache.

The main point to remember is that JCache implements Map and that is the best way to think about it.

JCache also has some interesting asynchronous methods such as `load` and `loadAll` which can be used to preload the JCache.

46.1.3 Problems and Limitations in the early draft of JSR107

If you are used to the richer API that Ehcache provides, you need to be aware of some problems and limitations in the draft specification.

You can generally work around these by getting the `Ehcache` backing cache. You can then access the extra features available in ehcache.

Of course the biggest limitation is that JSR107 (as of Augut 2007) is a long way from final.

```
/**
 * Gets the backing Ehcache
 */
public Ehcache getBackingCache() {
    return cache;
}
```

The following is both a critique of JCache and notes on the Ehcache implementation. As a member of the JSR107 Expert Group these notes are also intended to be used to improve the specification.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

46.1.3.1 net.sf.jsr107cache.CacheManager

CacheManager does not have the following features:

- shutdown the CacheManager - there is no way to free resources or persist. Implementations may utilise a shutdown hook, but that does not work for application server redeployments, where a shutdown listener must be used.
- List caches in the CacheManager. There is no way to iterate over, or get a list of caches.
- remove caches from the CacheManager - once its there it is there until JVM shutdown. This does not work well for dynamic creation, destruction and recreation of caches.
- CacheManager does not provide a standard way to configure caches. A Map can be populated with properties and passed to the factory, but there is no way a configuration file can be configured. This should be standardised so that declarative cache configuration, rather than programmatic, can be achieved.

46.1.3.2 net.sf.jsr107cache.CacheFactory

A property is specified in the resource services/net.sf.jsr107cache.CacheFactory for a CacheFactory.

The factory then resolves the CacheManager which must be a singleton.

A singleton CacheManager works in simple scenarios. But there are many where you want multiple CacheManagers in an application. Ehcache supports both singleton creation semantics and instances and defines the way both can coexist.

The singleton CacheManager is a limitation of the specification.

(Alternatives: Some form of annotation and injection scheme)

Pending a final JSR107 implementation, the Ehcache configuration mechanism is used to create JCaches from ehcache.xml config.

46.1.3.3 net.sf.jsr107cache.Cache

- The spec is silent on whether a Cache can be used in the absence of a CacheManager. Requiring a CacheManager makes a central place where concerns affecting all caches can be managed, not just a way of looking them up. For example, configuration for persistence and distribution.
- Cache does not have a lifecycle. There is no startup and no shutdown. There is no way, other than a shutdown hook, to free resources or perform persistence operations. Once again this will not work for redeployment of applications in an app server.
- There is no mechanism for creating a new cache from a default configuration such as a `public void registerCache(String cacheName)` on CacheManager. This feature is considered indispensable by frameworks such as Hibernate.
- Cache does not have a `getName()` method. A cache has a name; that is how it is retrieved from the CacheManager. But it does not know its own name. This forces API users to keep track of the name themselves for reporting exceptions and log messages.
- Cache does not support setting a TTL override on a put. e.g. `put(Object key, Object value, long timeToLive)`. This is a useful feature.
- The spec is silent on whether the cache accepts null keys and elements. Ehcache allows all implementations. i.e.

```
cache.put(null, null);
assertNull(cache.get(null));
cache.put(null, "value");
assertEquals("value", cache.get(null));
cache.put("key", null);
assertEquals(null, cache.get("key"));
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

`null` is effectively a valid key. However because `null` id not an instance of `Serializable` null-keyed entries will be limited to in-process memory.

- The `load(Object key)`, `loadAll(Collection keys)` and `getAll(Collection collection)` methods specify in the javadoc that they should be asynchronous. Now, most load methods work off a database or some other relatively slow resource (otherwise there would be no need to have a cache in the first place).

  To avoid running out of threads, these load requests need to be queued and use a finite number of threads. The Ehcache implementation does that. However, due to the lack of lifecycle management, there is no immediate way to free resources such as thread pools.
- The `load` method ignores a request if the element is already loaded in for that key.
- `get` and `getAll` are inconsistent. `getAll` throws CacheException, but `get` does not. They both should.

```
    /**
     * Returns a collection view of the values contained in this map.  The
     * collection is backed by the map, so changes to the map are reflected in
     * the collection, and vice-versa.  If the map is modified while an
     * iteration over the collection is in progress (except through the
     * iterator's own <tt>remove</tt> operation), the results of the
     * iteration are undefined.  The collection supports element removal,
     * which removes the corresponding mapping from the map, via the
     * <tt>Iterator.remove</tt>, <tt>Collection.remove</tt>,
     * <tt>removeAll</tt>, <tt>retainAll</tt> and <tt>clear</
tt> operations.
     * It does not support the add or <tt>addAll</tt> operations.
     * <p/>
     *
     * @return a collection view of the values contained in this map.
     */
    public Collection values() {
```

  It is not practical or desirable to support this contract. Ehcache has multiple maps for storage of elements so there is no single backing map. Allowing changes to propagate from a change in the collection maps would break the public interface of the cache and introduce subtle threading issues.

  The Ehcache implementation returns a new collection which is not connected to internal structures in ehcache.

46.1.3.4 net.sf.jsr107cache.CacheEntry

- getHits() returns int. It should return long because production cache systems have entries hit more than Integer.MAX_VALUE times.

  Once you get to Integer.MAX_VALUE the counter rolls over. See the following test:

```
    @Test public void testIntOverflow() {
        long value = Integer.MAX_VALUE;
        value += Integer.MAX_VALUE;
        value += 5;
        LOG.info("" + value);
        int valueAsInt = (int) value;
        LOG.info("" + valueAsInt);
        assertEquals(3, valueAsInt);
    }
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- `getCost()` requirs the CacheEntry to know where it is. If it is in a DiskStore then its cost of retrieval could be higher than if it is in heap memory. Ehcache elements do not have this concept, and it is not implemented. i.e. getCost always returns 0. Also, if it is in the DiskStore, when you retrieve it is in then in the MemoryStore and its retrieval cost is a lot lower. I do not see the point of this method.
- `getLastUpdateTime()` is the time the last "update was made". JCACHE does not support updates, only puts

46.1.3.5 net.sf.jsr107cache.CacheStatistics

- getObjectCount() is a strange name. How about getSize()? If a cache entry is an object graph each entry will have more than one "object" in it. But the cache size is what is really meant, so why not call it that?
- Once again `getCacheHits` and `getCacheMisses` should be longs.

```
public interface CacheStatistics {
    public static final int STATISTICS_ACCURACY_NONE = 0;
    public static final int STATISTICS_ACCURACY_BEST_EFFORT = 1;
    public static final int STATISTICS_ACCURACY_GUARANTEED = 2;
    public int getStatisticsAccuracy();
    public int getObjectCount();
    public int getCacheHits();
    public int getCacheMisses();
    public void clearStatistics();
```

- There is a `getStatisticsAccuracy()` method but not a corresponding setStatisticsAccuracy method on Cache, so that you can alter the accuracy of the Statistics returned.

  Ehcache supports this behaviour.
- There is no method to estimate memory use of a cache. Ehcache serializes each Element to a byte[] one at a time and adds the serialized sizes up. Not perfect but better than nothing and works on older JDKs.
- CacheStatistics is obtained using `cache.getCacheStatistics()` It then has getters for values. In this way it feels like a value object. The Ehcache implementation is Serializable so that it can act as a DTO. However it also has a clearStatistics() method. This method clear counters on the Cache. Clearly CacheStatistics must hold a reference to Cache to enable this to happen.

  But what if you are really using it as a value object and have serialized it? The Ehcache implementation marks the Cache reference as `transient`. If clearStatistics() is called when the cache reference is no longer there, an IllegalStateException is thrown.

  A much better solution would be to move clearStatistics() to Cache.

46.1.3.6 net.sf.jsr107cache.CacheListener

```
/**
 *  Interface describing various events that can happen as elements are added to
 *  or removed from a cache
 */
public interface CacheListener {
    /
** Triggered when a cache mapping is created due to the cache loader being consulted */
    public void onLoad(Object key);
    /
** Triggered when a cache mapping is created due to calling Cache.put() */
    public void onPut(Object key);
    /** Triggered when a cache mapping is removed due to eviction */
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
    public void onEvict(Object key);
    /
** Triggered when a cache mapping is removed due to calling Cache.remove() */
    public void onRemove(Object key);
    public void onClear();
}
```

- Listeners often need not just the key, but the cache Entry itself. This listener interface is extremely limiting.
- There is no onUpdate notification method. These are mapped to JCACHE's onPut notification.
- There is no onExpired notification method. These are mapped to JCACHE's onEvict notification.

46.1.3.7 net.sf.jsr107cache.CacheLoader

- JCache can store null values against a key. In this case, on JCache# `get` or `getAll` should an implementation attempt to load these values again? They might have been null in the system the CacheLoader loads from, but now aren't. The Ehcache implementation will still return nulls, which is probably the correct behaviour. This point should be clarified.

46.1.4 Other Areas

46.1.4.1 JMX

JSR107 is silent on JMX which has been included in the JDK since 1.5.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

47 Building From Source

47.1 Building from Source

These instructions work for each of the modules, except for JMS Replication, which requires installation of a message queue. See that module for details.

47.1.1 Building an Ehcache distribution from source

To build Ehcache from source:

1. Check the source out from the subversion repository.
2. Ensure you have a valid JDK and Maven 2 installation.
3. From within the ehcache/core directory, type `mvn -Dmaven.test.skip=true install`

47.1.2 Running Tests for Ehcache

To run the test suite for Ehcache:

1. Check the source out from the subversion repository.
2. Ensure you have a valid JDK and Maven 2 installation.
3. From within the ehcache/core directory, type `mvn test`
4. If some performance tests fail, add a `-D net.sf.ehcache.speedAdjustmentFactor=x` System property to your command line, where x is how many times your machine is slower than the reference machine. Try setting it to 5 for a start.

47.1.3 Deploying Maven Artifacts

Ehcache has a repository and snapshot repository at oss.sonatype.org.

The repository is synced with the Maven Central Repository.

To deploy:

```
mvn deploy
```

This will fail because SourceForge has disabled ssh exec. You need to create missing directories manually using sftp access `sftp gregluck,ehcache@web.sourceforge.net`

47.1.4 Building the Site

(These instructions are for project maintainers)

To build the site use:

```
mvn -Dmaven.test.skip=true package site
```

The site needs to be deployed from the target/site directory using:

rsync -v -r * ehcache-stage.terracotta.lan:/export1/ehcache.org

sudo -u maven -H /usr/local/bin/syncEHcache.sh

47.1.5 Deploying a release

47.1.5.1 Maven Release

```
mvn deploy
```

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

47.1.5.2 Sourceforge Release

```
mvn assembly:assembly
```

then manually upload to SourceForge

```
sftp gregluck@frs.sourceforge.net
```

and complete the file release process

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

48 FAQ

48.1 Ehcache - Frequently Asked Questions

48.1.1 Does Ehcache run on JDK1.3/ JDK1.4?

Older versions run on 1.3. Ehcache 1.5 runs on 1.4. Ehcache 1.6 required JDK 1.5.

48.1.2 Can you use more than one instance of Ehcache in a single VM?

As of ehcache-1.2, yes. Create your CacheManager using new CacheManager(...) and keep hold of the reference. The singleton approach accessible with the getInstance(...) method is still available too. Remember that Ehcache can support hundreds of caches within one CacheManager. You would use separate CacheManagers where you want different configurations.

The Hibernate EhCacheProvider has also been updated to support this behaviour.

48.1.3 Can you use Ehcache with Hibernate and outside of Hibernate at the same time?

Yes. You use 1 instance of Ehcache and 1 ehcache.xml. You configure your caches with Hibernate names for use by Hibernate. You can have other caches which you interact with directly outside of Hibernate.

That is how I use Ehcache in the original project it was developed in. For Hibernate we have about 80 Domain Object caches, 10 StandardQueryCaches, 15 Domain Object Collection caches.

We have around 5 general caches we interact with directly using BlockingCacheManager. We have 15 general caches we interact with directly using SelfPopulatingCacheManager. You can use one of those or you can just use CacheManager directly.

See the tests for example code on using the caches directly. Look at CacheManagerTest, CacheTest and SelfPopulatingCacheTest.

48.1.4 What happens when maxElementsInMemory is reached? Are the oldest items expired when new ones come in?

When the maximum number of elements in memory is reached, the least recently used ("LRU") element is removed. Used in this case means inserted with a put or accessed with a get.

If the overflowToDisk cache attribute is false, the LRU Element is evicted. If true, it is flushed asynchronously to the DiskStore.

48.1.5 Is it thread-safe to modify Element values after retrieval from a Cache?

Remember that a value in a cache element is globally accessible from multiple threads. It is inherently not thread safe to modify the value. It is safer to retrieve a value, delete the cache element and then reinsert the value.

The UpdatingCacheEntryFactory does work by modifying the contents of values in place in the cache. This is outside of the core of Ehcache and is targeted at high-performance CacheEntryFactories for SelfPopulatingCaches.

48.1.6 Can non-Serializable objects be stored in a cache?

As of ehcache-1.2, they can be stored in caches with MemoryStores.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

If an attempt is made to replicate or overflow a non-serializable element to disk, the element is removed and a warning logged.

48.1.7 Why is there an expiry thread for the DiskStore but not for the MemoryStore?

Because the memory store has a fixed maximum number of elements, it will have a maximum memory use equal to the number of elements * the average size. When an element is added beyond the maximum size, the LRU element gets pushed into the DiskStore.

While we could have an expiry thread to expire elements periodically, it is far more efficient to only check when we need to. The tradeoff is higher average memory use.

The expiry thread keeps the disk store clean. There is hopefully less contention for the DiskStore's locks because commonly used values are in the MemoryStore. We mount our DiskStore on Linux using RAMFS so it is using OS memory. While we have more of this than the 2GB 32 bit process size limit it is still an expensive resource. The DiskStore thread keeps it under control.

If you are concerned about cpu utilisation and locking in the DiskStore, you can set the diskExpiryThreadIntervalSeconds to a high number - say 1 day. Or you can effectively turn it off by setting the diskExpiryThreadIntervalSeconds to a very large value.

48.1.8 What elements are mandatory in ehcache.xml?

The documentation has been updated with comprehensive coverage of the schema for Ehcache and all elements and attributes, including whether they are mandatory. See the Declarative Configuration chapter.

48.1.9 Can I use Ehcache as a memory cache only?

Yes. Just set the overflowToDisk attribute of cache to false.

48.1.10 Can I use Ehcache as a disk cache only?

As of Ehcache 2.0 this is not possible. You can set the maxElementsInMemory to 1, but setting the max size to 0 now gives an infinite capacity.

48.1.11 Where is the source code?

The source code is distributed in the root directory of the download. It is called ehcache-x.x.zip. It is also available from SourceForge online or through SVN.

48.1.12 How do you get an Element without affecting statistics?

Use the Cache.getQuiet() method. It returns an Element without updating statistics.

48.1.13 How do you get WebSphere to work with ehcache?

It has been reported that IBM Websphere 5.1 running on IBM JDK1.4 requires commons-collection.jar in its classpath even though Ehcache will not use it for JDK1.4 and JDK5. (This is for versions of Ehcache lower than 1.6)

48.1.14 Do you need to call CacheManager.getInstance().shutdown() when you finish with ehcache?

Yes, it is recommended. If the JVM keeps running after you stop using ehcache, you should call CacheManager.getInstance().shutdown() so that the threads are stopped and cache memory released

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

back to the JVM. Calling shutdown also insures that your persistent disk stores get written to disk in a consistent state and will be usable the next time they are used.

If the CacheManager does not get shutdown it should not be a problem. There is a shutdown hook which calls the shutdown on JVM exit. This is explained in the documentation here.

48.1.15 Can you use Ehcache after a CacheManager.shutdown()?

Yes. When you call CacheManager.shutdown() is sets the singleton in CacheManager to null. If you try an use a cache after this you will get a CacheException.

You need to call CacheManager.create(). It will create a brand new one good to go. Internally the CacheManager singleton gets set to the new one. So you can create and shutdown as many times as you like.

There is a test which expliciyly confirms this behaviour. See CacheManagerTest#testCreateShutdownCreate()

48.1.16 I have created a new cache and its status is STATUS_UNINITIALISED. How do I initialise it?

You need to add a newly created cache to a CacheManager before it gets intialised. Use code like the following:

```
CacheManager manager = CacheManager.create();
Cache myCache = new Cache("testDiskOnly", 0, true, false, 5, 2);
manager.addCache(myCache);
```

48.1.17 Is there a simple way to disable Ehcache when testing?

Yes. There is a System Property based method of disabling ehcache. If disabled no elements will be added to a cache. Set the property "net.sf.ehcache.disabled=true" to disable ehcache.

This can easily be done using `-Dnet.sf.ehcache.disabled=true`> in the command line.

48.1.18 How do I dynamically change Cache attributes at runtime?

You can't but you can achieve the same result as follows:

```
Cache cache = new Cache("test2", 1, true, true, 0, 0, true, 120, ...);
cacheManager.addCache(cache);
```

See the JavaDoc for the full parameters, also reproduced here:

Having created the new cache, get a list of keys using cache.getKeys, then get each one and put it in the new cache. None of this will use much memory because the new cache element have values that reference the same data as the original cache. Then use cacheManager.removeCache("oldcachename") to remove the original cache.

48.1.19 I get net.sf.ehcache.distribution.RemoteCacheException: Error doing put to remote peer. Message was: Error unmarshaling return header; nested exception is: java.net.SocketTimeoutException: Read timed out. What does this mean.

It typically means you need to increase your socketTimeoutMillis. This is the amount of time a sender should wait for the call to the remote peer to complete. How long it takes depends on the network and the size of the Elements being replicated.

The configuration that controls this is the socketTimeoutMillis setting in cacheManagerPeerListenerFactory. 120000 seems to work well for most scenarios.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

```
<cacheManagerPeerListenerFactory
        class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"
        properties="hostName=fully_qualified_hostname_or_ip,
                    port=40001,
                    socketTimeoutMillis=120000"/>
```

48.1.20 Should I use this directive when doing distributed caching? *cacheManagerEventListenerFactory class="" properties=""/*

No. It is unrelated. It is for listening to changes in your local CacheManager.

48.1.21 What is the minimum config to get distributed caching going?

The minimum configuration you need to get distributed caching going is:

```
<cacheManagerPeerProviderFactory
        class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
        properties="peerDiscovery=automatic,
                    multicastGroupAddress=230.0.0.1,
                    multicastGroupPort=4446"/>
<cacheManagerPeerListenerFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"/>
```

and then at least one cache declaration with

```
<cacheEventListenerFactory
    class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>>>>
```

in it. An example cache is:

```
<cache name="sampleDistributedCache1"
       maxElementsInMemory="10"
       eternal="false"
       timeToIdleSeconds="100"
       timeToLiveSeconds="100"
       overflowToDisk="false">
    <cacheEventListenerFactory
        class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>
</cache>
```

Each server in the cluster can have the same config.

48.1.22 How can I see if distributed caching is working?

You should see the listener port open on each server.

You can use the distributed debug tool to see what is going on. (See http://ehcache.org/documentation/remotedebugger.html).

48.1.23 Why can't I run multiple applications using Ehcache on one machine?

Because of an RMI bug, in JDKs before JDK1.5 such as JDK1.4.2, Ehcache is limited to one CacheManager operating in distributed mode per virtual machine. (The bug limits the number of RMI registries to one per virtual machine). Because this is the expected deployment configuration, however, there should be no practical effect. The tell tail error is `java.rmi.server.ExportException: internal error: ObjID already in use`

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

On JDK1.5 and higher it is possible to have multiple CacheManagers per VM each participating in the same or different clusters. Indeed the replication tests do this with 5 CacheManagers on the same VM all run from JUnit.

48.1.24 How many threads does Ehcache use, and how much memory does that consume?

The amount of memory consumed per thread is determined by the Stack Size. This is set using -Xss. The amount varies by OS. It is 512KB for Linux. I tend to override the default and set it to 100kb.

The threads are created per cache as follows:

- DiskStore expiry thread - if DiskStore is used
- DiskStore spool thread - if DiskStore is used
- Replication thread - if asynchronous replication is configured.
  If you are not doing any of the above, no extra threads are created

48.1.25 I am using Tomcat 5, 5.5 or 6 and I am having a problem. What can I do?

Tomcat is such a common deployment option for applications using Ehcache that there is a chapter on known issues and recommended practices.

See the Using Ehcache with Tomcat chapter. (http://ehcache.org/documentation/tomcat.html)

48.1.26 I am using Java 6 and getting a java.lang.VerifyError on the Backport Concurrent classes. Why?

The backport-concurrent library is used in Ehcache to provide java.util.concurrency facilities for Java 4 - Java 6. Use either the Java 4 version which is compatible with Java 4-6 or use the version for your JDK.

48.1.27 How do I get a memory only-cache to persist to disk between VM restarts?

While disk persistence between restarts is a feature of the DiskStore only, you can get the same behaviour for a memory only cache by setting up a cache with maxElementsInMemory set to Integer.MAX_VALUE, 2147483647 and diskPersistent set to true.

You can manually call flush() to flush to disk. It is a good idea to set `clearOnFlush` to false so that the MemoryStore is not cleared each time. You can then call flush() to persist whenever you wish.

48.1.28 I get a javax.servlet.ServletException: Could not initialise servlet filter when using SimplePageCachingFilter. Why?

If you use this default implementation, the cache name is called "SimplePageCachingFilter". You need to define a cache with that name in ehcache.xml. If you override CachingFilter you are required to set your own cache name.

48.1.29 Why is there a warning in my application's log that a new CacheManager is using a resource already in use by another CacheManager.

```
WARN  CacheManager ...  Creating a new instance of CacheManager using the diskStorePath
 "C:\temp\tempcache" which is already used by an existing CacheManager.
```

This means, that for some reason, your application is trying to create a second or more instance of Ehcache's CacheManager with the same configuration. Ehcache is automatically resolving the Disk path conflict, which works fine.

To eliminate the warning:

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- Use a separate configuration per instance
- If you only want one instance use the singleton creation methods i.e `CacheManager.getInstance()`. In Hibernate there is a special provider for this called `net.sf.ehcache.hibernate.SingletonEhCacheProvider`.

  See the Hibernate page for details.

48.1.30 How do I add a CacheReplicator to a cache that already exists? The cache event listening works but it does not get plumbed into the peering mechanism.

The current API does not have a CacheManager event for cache configuration change. You can however make it work by calling the notifyCacheAdded event.

```
getCache().getCacheManager().getCacheManagerEventListenerRegistry()
    .notifyCacheAdded("cacheName");
```

48.1.31 I am using the RemoteDebugger to monitor cluster messages but all I see is "Cache size: 0"

If you see nothing happening, but cache operations should be going through, enable trace (LOG4J) or finest (JDK) level logging on *code*net.sf.ehcache.distribution */code* in the logging configuration being used by the debugger. A large volume of log messages will appear. The normal problem is that the CacheManager has not joined the cluster. Look for the list of cache peers.

Finally, the debugger in ehcache-1.5 has been improved to provide far more information on the caches that are replicated and events which are occurring.

48.1.32 With distributed replication on Ubuntu or Debian, I see the following warning,

```
WARN [Replication Thread] RMIAsynchronousCacheReplicator.flushReplicationQueue(324)
| Unable to send message to remote peer.
Message was: Connection refused to host: 127.0.0.1; nested exception is:
java.net.ConnectException: Connection refused
java.rmi.ConnectException: Connection refused to host: 127.0.0.1; nested exception is:
java.net.ConnectException: Connection refused
```

This is caused by a 2008 change to the Ubuntu/Debian linux default network configuration.

Essentially, this java call: `InetAddress.getLocalHost();` always returns the loopback address, which is 127.0.0.1. Why? Because in these recent distros, a system call of $ hostname always returns an address mapped onto the loopback device. Which causes ehcache's RMI Peer creation logic to always assign the loopback address, which causes the error you are seeing.

All you need to do is crack open the network config and make sure that the hostname of the machine returns a valid network address accessible by other peers on the network.

48.1.33 I see log messages about SoftReferences. What are these about and how do I stop getting the messages?

Ehcache uses SoftReferences with asynchronous RMI based replication, so that replicating caches do not run out of memory if the network is interrupted. Elements scheduled for replication will be collected instead. If this is happening, you will see warning messages from the replicator. It is also possible that a SoftReference can be reclaimed during the sending in which case you will see a debug level message in the receiving CachePeer.

Some things you can do to fix them:

- Set -Xms equal to -Xms. SoftReferences are also reclaimed in preference to increasing the heap size, which is a problem when an application is warming up.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

- Set the -Xmx to a high enough value so that SoftReferences do not get reclaimed.

  Having done the above, SoftReferences will then only be reclaimed if there is some interruption to replication and the message queue gets dangerously high.

48.1.34 My Hibernate Query caches entries are replicating but the other caches in the cluster are not using them.

This is a Hibernate 3 bug. See http://opensource.atlassian.com/projects/hibernate/browse/HHH-3392 for tracking. It is fixed in 3.3.0.CR2 which was released in July 2008.

48.1.35 Active MQ Temporary Destinatons

ActiveMQ seems to have a bug in at least ActiveMQ 5.1 where it does not cleanup temporary queues, even though they have been deleted. That bug appears to be long standing but was though to have been fixed.

See:

- http://www.nabble.com/Memory-Leak-Using-Temporary-Queues-td11218217.html#a11218217
- http://issues.apache.org/activemq/browse/AMQ-1255

  The JMSCacheLoader uses temporary reply queues when loading. The Active MQ issue is readily reproduced in Ehcache integration testing. Accordingly, use of the JMSCacheLoader with ActiveMQ is not recommended. Open MQ tests fine.

48.1.36 Is Ehcache compatible with Google App Engine?

Version 1.6 is compatible. See Google App Engine Caching.

48.1.37 Can my app server use JMS Replication?

Some App Servers do not permit the creation of message listeners. This issue has been reported on Websphere 5. Websphere 4 did allow it. Tomcat allows it. Glassfish Allows it. Jetty allows it.

Usually there is a way to turn off strict EJB compliance checks in your app server. See your vendor documentation.

48.1.38 Why does Ehcache 1.6 use more memory than 1.5?

ConcurrentHashMap does not provide an eviction mechanism. We add that ourselves. For caches larger than 5000 elements, we create an extra ArrayList equal to the size of the cache which holds keys. This can be an issue with larger keys. An optimisation which cache clients can use is:

```
    http://www.codeinstructions.com/2008/09/instance-pools-with-
weakhashmap.html
    To reduce the number of key instances in memory to just one per logical
    key, all puts to the underlying ConcurrentHashMap could be replaced by
    map.put(pool.replace(key), value), as well as keyArray.set(index,
    pool.replace(key))
    You can take this approach when producing the keys before handing them over to EhCa
```

Even with this approach there is still some added overhead consumed by a reference consumed by each ArrayList element.

Update: Ehcache 2.0 will introduce a new implementation for MemoryStore based on a custom ConcurrentHashMap. This version provides fast iteration and does away with the need for the keyArray thus bringing memory use back down to pre 1.6 levels. And with other memory

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.

optimisations made to Element in 1.7, memory use will actually be considerably lower than pre 1.6 levels.

48.1.39 What does this mean? "Caches cannot be added by name when default cache config is not specified in the config. Please add a default cache config in the configuration."

From Ehcache 2.4, we have made the `defaultCache` optional. When you try to add a cache by name, `CacheManager.add(String name)`, a default cache is expected to exist in the CacheManager configuration.

©2011, Terracotta, Inc. • ALL RIGHTS RESERVED.