JCS v.1.3

Project Documentation

Apache Software Foundation **30 May 2007**

Table of Contents

1.1 Overview

News: Version 1.3 is available

The Java Caching System, version 1.3 has been released. This release is actually the first official release of JCS. You can download it from the Download page.

Java Caching System

JCS is a distributed caching system written in java. It is intended to speed up applications by providing a means to manage cached data of various dynamic natures. Like any caching system, JCS is most useful for high read, low put applications. Latency times drop sharply and bottlenecks move away from the database in an effectively cached system. Learn how to start using JCS.

The JCS goes beyond simply caching objects in memory. It provides numerous additional features:

- Memory management
- Disk overflow (and defragmentation)
- Thread pool controls
- Element grouping
- Minimal dependencies
- Quick nested categorical removal
- Data expiration (idle time and max life)
- Extensible framework
- Fully configurable runtime parameters
- Region data separation and configuration
- Fine grained element configuration options
- Remote synchronization
- Remote store recovery
- Non-blocking "zombie" (balking facade) pattern
- Lateral distribution of elements via HTTP, TCP, or UDP
- UDP Discovery of other caches
- Element event handling
- Remote server chaining (or clustering) and failover

JCS works on JDK versions 1.3 and up. It only has two dependencies: Commons Logging and Doug Lea's Util Concurrent.

JCS is a Composite Cache

The foundation of JCS is the Composite Cache, which is the pluggable controller for a cache region. Four types of caches can be plugged into the Composite Cache for any given region: (1) Memory, (2) Disk, (3) Lateral, and (4) Remote. The Composite Cache orchestrates access to the various caches configured for use in a region.

The JCS jar provides production ready implementations of each of the four types of caches. In addition to the core four, JCS also provides additional plugins of each type.

LRU Memory Cache

The LRU Memory Cache is an extremely fast, highly configurable memory cache . It uses a Least Recently Used algorithm to manage the number of items that can be stored in memory. The LRU Memory Cache uses its own LRU Map implementation that is significantly faster than both the commons LRUMap implementation and the LinkedHashMap that is provided with JDK1.4 up. This makes JCS faster than its competitors .

Indexed Disk Cache

The Indexed Disk Cache is a fast, reliable, and highly configurable swap for cached data. The indexed disk cache follows the fastest pattern for disk swapping. Cache elements are written to disk via a continuous queue-based process. The length of the item is stored in the first few bytes of the entry. The offset is stored in memory and can be reference via the key. When items are removed from the disk cache, the location and size are recorded and reused when possible. Every aspect of the disk cache is configurable, and a thread pool can be used to reduce the number of queue worker threads across the system.

TCP Lateral Cache

The TCP Lateral Cache provides an easy way to distribute cached data to multiple servers. It comes with a UDP discovery mechanism, so you can add nodes without having to reconfigure the entire farm. The TCP Lateral Cache works by establishing connections with socket server running on other nodes. Each node maintains a connection to every other. Only one server is needed for any number of regions. The client is able to re-establish connections if it looses its connection with another server. The TCP Lateral is highly configurable . You can choose to only send data, to not look for data on other servers, to send removes instead of puts, and to filter removes based on hash codes.

RMI Remote Cache

JCS also provides an RMI based Remote Cache Server . Rather than having each node connect to every other node, you can use the remote cache server as the connection point. Each node connects to the remove server, which then broadcasts events to the other nodes. To maintain consistency across a cluster without incurring the overhead of serialization, you can decide to send invalidation messages to the other locals rather than send the object over the wire. The remote cache server holds a serialized version of your objects, so it does not need to be deployed with your class libraries. The remote servers can be

chained and a list of failover servers can be configured on the client.

What JCS is not

JCS is not a tag library or a web specific application. JCS is a general purpose caching system that can be used in web applications, services, and stand alone Java applications.

JCS is not a transactional distribution mechanism. Transactional distributed caches are not scalable. JCS is a cache not a database. The distribution mechanisms provided by JCS can scale into the tens of servers. In a well-designed service oriented architecture, JCS can be used in a high demand service with numerous nodes. This would not be possible if the distribution mechanism were transactional.

JCS does not use AOP. JCS is a high performance, non-invasive cache. It does not manipulate your objects so it can just send a field or two fewer over the wire.

JCS is not a fork, an offshoot, a branch, or any other derivation of JCS. Nor is JCS named after another library. JCS is a mature project that has been under development and in use since 2001. Over the years JCS has encorporated numerous bug fixes and has added dozens of features, making it the best designed and most feature rich caching solution available.

1.2 JCS and JCACHE

JCS and JCACHE (JSR-107)

The JCS is an attempt to build a system close to JCACHE , JSR-107 , a description of the caching system used in Oracle9i. JCS grew out of my work over the past two years to build an enterprise level caching system. Though it is replete with good ideas, there are some aspects of the JCACHE architecture that could lead to inefficiency (ex, the lateral distribution and net searches) and a few programming preferences that I found cumbersome (ex, the use of exceptions to report the common place). Subsequently there are a few differences between the two systems. In some cases I have moved my original system closer to the JCACHE model where it presented a better idea. Briefly:

Element vs. Region Attributes

My original cache was regionally defined. Each entry required a very minimal wrapper. The osc4j specification is an element driven model where each element is fully configurable. This could lead to a slight performance penalty, but it is a richer model, where elements can inherit or have their own attributes. So, I converted the entire system into element centered framework.

Lateral Broadcast vs. Remote Consistency

The oracle model is a laterally distributed framework with no centralized control. The JCS model has the option for lateral broadcast (which will need to be made more efficient) and a remote store that coordinates consistency. In the JCS Local caches send data to the remote store which then notifies other local caches of changes to "regions" (caches) that are registered. In JCACHE's lateral model an update is never broadcast from the remote, rather updates come via the lateral caches. If you broadcast changes to all servers then every server must be ready for every user. The usage patterns of a user on one box can affect the whole. Also, the lateral model can make it difficult to synchronize combinations of updates and invalidations.

With a remote store the local caches are primed to take on similar patterns by talking to the remote store, but aren't flooded with the elements from another machine. This significantly cuts down on traffic. This way each local cache is a relatively separate realm with remotely configurable regions that stay in synch without overriding the user habits of any machine. It also allows for an efficient mechanism of retrieval, where searching for an element involves, at maximum, only as many steps as there are remote servers in the cluster. In the lateral model a failed net search could take an extremely long time to complete, making it necessary for the programmer to decide how long of a wait is acceptable.

Though this is by and large a poor model, the JCS will include the ability to perform full lateral searches. A more important feature is remote failover and remote server clustering. With clustering any concerns about the remote server being a single point of failure vanish and the remote server model is significantly more robust.

Put vs. Replace

The difference between put and replace is not present in the JCS by default. The overhead associated with this distinction is tremendous. However, there will be an alternate "safe-put" method to deal with special caches.

Nulls vs. Errors

I started to support `ObjectNotFoundExceptions` for failed gets but the overhead and cumbersome coding needed to surround a simple get method is ridiculous. Instead the JCS returns null.

Cache Loaders

I'm not supporting cache loaders at this time. They seem unnecessary, but may be useful in a smart portal.

Groups vs. Hierarchy

The JCS provides feature rich grouping mechanism, where groups of elements can be invalidated and whose attributes can be listed. The grouping feature is much like the session API. In addition, the JCS provides a mechanism for hierarchical removal without the overhead of keeping track of all the elements of a group across machines. Element keys with `":"` separators (a value that will be fully configurable) can be arranged in a hierarchy. A remove command ending in a `":"` will issue a removal of all child elements. I can associate search and menu drop down lists for a particular company in a multi-company system by starting each key in disparate caches with the company id followed by `":"` and then the normal key. Invalidating this data when a change is made to data affecting something falling under that company can be removed by simply calling `cacheAccess.remove(comp_id + ":")`.

1.3 Downloads

Releases

The latest release version of JCS is 1.3. Grab it here. The core JCS jar is compiled using JDK 1.3.

Binary versions

- JCS 1.3 Binary Distribution in TAR format Signature MD5
- JCS 1.3 Binary Distribution in ZIP format Signature MD5

Source versions

- JCS 1.3 Source Distribution in TAR format Signature MD5
- JCS 1.3 Source Distribution in ZIP format Signature MD5

Versioned Temp Builds

You can find versioned binary builds of JCS in the tempbuild directory of the repository.

Getting The Development Source From SVN

You can check out the latest source from the Jakarta SVN module.
JCS resides in the jakarta/jcs module.

```
svn checkout http://svn.apache.org/repos/asf/jakarta/jcs/trunk
```

or browse the source code through ViewVC .

1.4 FAQ

Frequently Asked Questions

Configuration

1. What jars are required by JCS?
2. How do I configure JCS?
3. How can I configure JCS with my own properties?
4. Can JCS use system properties during configuration?

General Questions

1. Is JCS faster than EHCache?
2. Where can I get the admin jsp?
3. Where can I get the source?
4. How do I compile the source?

Elements

1. How do I set the element attributes?
2. How do I register an element event?
3. Can I remove all items beginning with part of a key?

Indexed Disk Cache

1. How do I limit the number of threads used by the disk cache?

Remote Cache Server

1. Do I need to put my jars in the classpath of the remote server?

Configuration

Configuration

What jars are required by JCS?

As of verison 1.2.7.0, the core of JCS (the LRU memory cache, the indexed disk cache, the TCP lateral, and the RMI remote server) requires only two other jars.

concurrent

commons-logging

Versions 1.2.6.9 and below also require the following two additional jars:

commons-collections

commons-lang

All of the other dependencies listed on the project info page are for optional plugins.

How do I configure JCS?

By default JCS looks for a cache.ccf file in the classpath. You must have a configuration file on the classpath to use JCS. The documentation describes how to configure the cache.

How can I configure JCS with my own properties?

You don't have to put the cache.ccf file in the classpath; instead you can do the following:

```
CompositeCacheManager ccm =
CompositeCacheManager.getUnconfiguredInstance(); Properties props = new
Properties(); props.load(/* load properties from some location defined
by your app */); ccm.configure(props);
```

Can JCS use system properties during configuration?

Yes. JCS will look for a system property for any name inside the delimiters ${}. Also, JCS will check to see if any property key in the cache.ccf is defined in the system properties. If so, the system value will be used.

General Questions

General Questions

Is JCS faster than EHCache?

Yes. JCS is almost twice as fast as EHCache. JCS 1.2.7.0, using the default LRU Memory Cache, has proven to be nearly twice as fast as EHCache 1.2-beta4 at gets and puts. The EHCache benchmark data is unsubstantiated and very old. As such the EHCache site benchmark data is completely inaccurate. Read More

Where can I get the admin jsp?

You can download the admin jsp here .

Where can I get the source?

You can view the source here or get the source code from subversion with `svn co http://svn.apache.org/repos/asf/jakarta/jcs/trunk`. The tagged releases are available with `svn co.` ex. http://svn.apache.org/repos/asf/jakarta/jcs/tags/jcs_1_2_7_0

How do I compile the source?

You first need to install Maven 1.0.2 The download is available at http://maven.apache.org/maven-1.x/start/download.html. Maven 2.0 is not supported yet. After installing run "maven" which compiles and tests the entire package. To build a jar run "maven jar".

Elements

Elements

How do I set the element attributes?

Every element put into the cache has its own set of attributes. By default elements are given a copy of the default element attributes associated with a region. You can also specify the attributes to use for an element when you put it in the cache. See Element Attributes for more information on the attributes that are available.

How do I register an element event?

Element event handlers must be added to the element attributes. See Element Event Handling for more information on how to handle element events.

Can I remove all items beginning with part of a key?

Yes, but it is somewhat expensive, since some of the auxiliaries will have to iterate over their keysets. Although all the auxiliaries honor this, it is not part of the auxiliary API. There is no method along the lines of "removeStartingWith", but all the remove methods can do it.

By default, the hierarchical key delimiter used in JCS is a colon. You cannot add a String key that ends with a colon. If you call remove with a String key that ends in a colon, everything that has a key that starts with the argument will be removed.

If your keys are in this format

TYPE:SOURCE:OBJECT

And you put n objects int he cache with keys like this

"ABC:123:0" to "ABC:123:n"

then you could remove all the obejcts by calling

jcs.remove("ABC:123:");

Indexed Disk Cache

Indexed Disk Cache

How do I limit the number of threads used by the disk cache?

The indexed disk cache uses an event queue for each region. By default these queues are worked by their own dedicated threads. Hence, you will have one thread per disk cache region. Although the queues kill off idle threads, you may want to limit the overall number of threads used by the queues. You can do this by telling the disk cache to use a thread pool. The configuration is described on the disk cache configuration page .

Remote Cache Server

Remote Cache Server

Do I need to put my jars in the classpath of the remote server?

No. The remote server never deserializes your classes.

2.1 Overview

Getting Started

To start using JCS you need to (1) understand the core concepts, (2) download JCS, (3) get the required dependencies, (4) configure JCS, and (5) then start programming to it. The purpose of the getting started guide is to help you get up and running with JCS as quickly as possible. In depth doumentation on the various features of JCS is provided in the User's Guide.

STEP 1: Understand the Core Concepts

In order to use JCS, you must understand a few core concepts, most importantly you need to know the difference between "elements," "regions," and "auxiliaries".

JCS is an object cache. You can put objects, or "elements," into JCS and reference them via a key, much like a hashtable.

You can think of JCS as a collection of hashtables that you reference by name. Each of these hashtables is called a "region," and each region can be configured independently of the others. For instance, I may have a region called Cities where I cache City objects that change infrequently. I may also define a region called Products where I cache product data that changes more frequently. I would configure the volatile Product region to expire elements more quickly than the City region.

"Auxiliaries" are optional plugins that a region can use. The core auxiliaries are the Indexed Disk Cache, the TCP Lateral Cache, and the Remote Cache Server. The Disk Cache, for example, allows you to swap items onto disk when a memory threshold is reached. You can read more about the available auxiliaries HERE .

STEP 2: Download JCS

Download the latest version of JCS. The latest JCS builds are located HERE

If you would like to build JCS yourself, check it out from Subversion and build it as you would any other project built by Maven 1.x. The location of the repository is documented in the project info pages that are linked via the left nav.

STEP 3: Get the Required Dependencies

As of version 1.2.7.0, the core of JCS (the LRU memory cache, the indexed disk cache, the TCP lateral, and the RMI remote server) requires only two other jars.

concurrent

commons-logging

Versions 1.2.6.9 and below also require the following two additional jars:

commons-collections

commons-lang

All of the other dependencies listed on the project info page are for optional plugins.

STEP 4: Configure JCS

JCS is configured from a properties file called "cache.ccf". There are alternatives to using this file, but they are beyond the scope of the getting started guide.

The cache configuration has three parts: default, regions, and auxiliaries. You can think of the auxiliaries as log4j appenders and the regions as log4j categories. For each region (or category) you can specify and auxiliary (or appender to use). If you don't define a region in the cache.ccf, then the default settings are used. The difference between JCS and log4j is that in JCS, pre-defined regions do not inherent auxiliaries from the default region.

The following cache.ccf file defines one region called "testCache1" and uses the Indexed Disk Cache, here called "DC" by default. The LRU Memory Cache is selected as the memory manager.

```
# DEFAULT CACHE REGION
jcs.default=DC
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
jcs.default.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.default.cacheattributes.UseMemoryShrinker=false
jcs.default.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.default.cacheattributes.ShrinkerIntervalSeconds=60
jcs.default.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.default.elementattributes.IsEternal=false
jcs.default.elementattributes.MaxLifeSeconds=21600
jcs.default.elementattributes.IdleTime=1800
jcs.default.elementattributes.IsSpool=true
jcs.default.elementattributes.IsRemote=true
jcs.default.elementattributes.IsLateral=true

# PRE-DEFINED CACHE REGIONS
jcs.region.testCache1=DC
jcs.region.testCache1.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.testCache1.cacheattributes.MaxObjects=1000
jcs.region.testCache1.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.region.testCache1.cacheattributes.UseMemoryShrinker=false
jcs.region.testCache1.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.region.testCache1.cacheattributes.ShrinkerIntervalSeconds=60
jcs.region.testCache1.cacheattributes.MaxSpoolPerRun=500
jcs.region.testCache1.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.region.testCache1.elementattributes.IsEternal=false
```

```
# AVAILABLE AUXILIARY CACHES
jcs.auxiliary.DC=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
jcs.auxiliary.DC.attributes.DiskPath=${user.dir}/jcs_swap
jcs.auxiliary.DC.attributes.MaxPurgatorySize=10000000
jcs.auxiliary.DC.attributes.MaxKeySize=1000000
jcs.auxiliary.DC.attributes.MaxRecycleBinSize=5000
jcs.auxiliary.DC.attributes.OptimizeAtRemoveCount=300000
jcs.auxiliary.DC.attributes.ShutdownSpoolTimeLimit=60
```

Basic JCS configuration is described in more detail HERE

Element level configuration is described in more detail HERE

For more information on advanced configuration options and the available plugins, see the User's Guide.

STEP 5: Programming to JCS

JCS provides a convenient class that should meet all your needs. It is called, appropriately enough, `org.apache.jcs.JCS`

To get a cache region you simply ask JCS for the region by name. If you wanted to use JCS for City objects, you would do something like this:

```
import org.apache.jcs.JCS;
import org.apache.jcs.access.exception.CacheException;

. . .

    private static final String cacheRegionName = "city";

    private JCS cache = null;

. . .
                        // in your constructor you might do this
            try
            {
                setCache( JCS.getInstance( this.getCacheRegionName() ) );
            }
            catch ( CacheException e )
            {
                log.error( "Problem initializing cache for region name ["
                  + this.getCacheRegionName() + "].", e );
            }

. . .

            // to get a city out of the cache by id you might do this:
            String key = "cityId:" + String.valueOf( id );
```

```
            City city = (City) cache.get( key );

. . .

            // to put a city object in the cache, you could do this:
            try
            {
                // if it isn't null, insert it
                if ( city != null )
                {
                    cache.put( key, city );
                }
            }
            catch ( CacheException e )
            {
                 log.error( "Problem putting "
                   + city + " in the cache, for key " + key, e );
            }
```

2.2 Basic JCS Config

Basic JCS Configuration

The following document illustrates several basic JCS configurations. As you'll see, using JCS can be as simple as creating a single memory cache for you application. However, with a few configuration changes, you can quickly enable some distributed caching features that can scale your application even further.

Building a cache.ccf file

Configuring the JCS can be as simple as your needs. The most basic configuration would be a pure memory cache where every region takes the default values. The complete configuration file (cache.ccf) could look like this:

```
# DEFAULT CACHE REGION

jcs.default=
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
jcs.default.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
```

If you want to add memory shrinking then you can add these lines:

```
jcs.default.cacheattributes.UseMemoryShrinker=true
jcs.default.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.default.cacheattributes.ShrinkerIntervalSeconds=60
jcs.default.cacheattributes.MaxSpoolPerRun=500
jcs.default.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.default.elementattributes.IsEternal=false
```

Adding a disk cache is as simple as telling it what folder to use. It is recommended that you add a disk cache. If you want to add a disk cache to your default parameters, then (1) add this to the bottom of the file to create the auxiliary:

```
jcs.auxiliary.DC=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
```

```
jcs.auxiliary.DC.attributes.DiskPath=g:/dev/jcs/raf
```

and (2) change the first line to:

```
jcs.default=DC
```

If you want to predefine a specific region, say called `testCache1`, then add these lines:

```
jcs.region.testCache1=DC
jcs.region.testCache1.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.testCache1.cacheattributes.MaxObjects=1000
jcs.region.testCache1.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.region.testCache1.cacheattributes.UseMemoryShrinker=true
jcs.region.testCache1.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.region.testCache1.cacheattributes.ShrinkerIntervalSeconds=60
jcs.region.testCache1.cacheattributes.MaxSpoolPerRun=500
jcs.region.testCache1.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.region.testCache1.elementattributes.IsEternal=false
```

If you want to add a lateral cache for distribution (the TCP Lateral Auxiliary is recommended), then add these lines to the bottom of the file to define the auxiliary:

```
jcs.auxiliary.LTCP=
    org.apache.jcs.auxiliary.lateral.LateralCacheFactory
jcs.auxiliary.LTCP.attributes=
    org.apache.jcs.auxiliary.lateral.LateralCacheAttributes
jcs.auxiliary.LTCP.attributes.TransmissionTypeName=TCP
jcs.auxiliary.LTCP.attributes.TcpServers=localhost:1111
jcs.auxiliary.LTCP.attributes.TcpListenerPort=1110
jcs.auxiliary.LTCP.attributes.PutOnlyMode=false
```

See the TCP Lateral documentation for more information. If you want to set up `testCache1` to use this, then change the definition to:

```
jcs.region.testCache1=DC,LTCP
```

A few comments on configuration

Auxiliary definitions are like log4j appenders, they are defined and then associated with a region like a log4j category.

The order of configuration file is unimportant, though you should try to keep it organized for your own sake.

Configuration is being refactored and is subject to change. It should only become easier.

The complete file

The complete file from above would look like this:

```
# DEFAULT CACHE REGION

jcs.default=DC,LTCP
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
jcs.default.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache

# PRE-DEFINED CACHE REGIONS

jcs.region.testCache1=DC,LTCP
jcs.region.testCache1.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.testCache1.cacheattributes.MaxObjects=1000
jcs.region.testCache1.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.region.testCache1.cacheattributes.UseMemoryShrinker=true
jcs.region.testCache1.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.region.testCache1.cacheattributes.ShrinkerIntervalSeconds=60
jcs.region.testCache1.cacheattributes.MaxSpoolPerRun=500
jcs.region.testCache1.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.region.testCache1.elementattributes.IsEternal=false


# AVAILABLE AUXILIARY CACHES
jcs.auxiliary.DC=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
jcs.auxiliary.DC.attributes.DiskPath=g:/dev/jcs/raf
jcs.auxiliary.DC.attributes.maxKeySize=100000

jcs.auxiliary.LTCP=
    org.apache.jcs.auxiliary.lateral.LateralCacheFactory
jcs.auxiliary.LTCP.attributes=
    org.apache.jcs.auxiliary.lateral.LateralCacheAttributes
jcs.auxiliary.LTCP.attributes.TransmissionTypeName=TCP
jcs.auxiliary.LTCP.attributes.TcpServers=localhost:1111
jcs.auxiliary.LTCP.attributes.TcpListenerPort=1110
jcs.auxiliary.LTCP.attributes.PutOnlyMode=false
```

2.3 Plugin Overview

JCS Plugin Overview

JCS provides multiple auxiliaries which can be plugged into a cache region, in a manner similar to adding Log4j appenders to a logger. JCS auxiliaries are defined in the cache.ccf file. You can specify which plugins a particular cache region should use.

There are four types of auxiliaries: (1) memory, (2) disk, (3) lateral, and (4) remote. Each region is required to have one and only one memory auxiliary. No other auxiliaries are required and any possible combination of disk, lateral, and remote auxiliaries is allowed. If you do not want to store items in memory, then the maximum size for the memory caches can be set to 0 on a per region basis.

Memory Plugins

Currently, JCS provides four memory management options: (1) LRUMemoryCache, (2) LHMLRUMemoryCache, (3) MRUMemoryCache, and (4) ARCMemoryCache. All memory caches restrict the number of items that can be stored in memory per region. If a disk cache is configured for the region, the items will be spooled to disk when the memory capacity is reached. JCS enforces configurable parameters such as time to live and maximum idle time. Expired elements can be cleaned up by the ShrinkerThread, otherwise they will be removed at the next retrieval attempt or when the capacity is reached.

The LRUMemoryCache is the currently recommended plugin. Upon misconfiguration it is used as the default. The LRUMemoryCache removes the least recently used items when the cache is full.

The ARCMemoryCache is currently experimental, but will be fully tested soon. It implements an adaptive replacement caching algorithm that combines an LRU and an LFU that adapt to usage patterns.

Disk Plugins

JCS provides several disk swap options: indexed disk, HSQL, JISP, and Berkeley DB JE. The IndexedDiskCache is the recommended disk cache. It maintains the cached data on disk and the keys in memory for the fastest possible lookup times. Writing to disk is done asynchronously. Items are typically put in purgatory and queued for background disk writing. While in purgatory, the items remain available.

In addition, JCS provides a disk auxiliary that uses the Berkeley DB Java Edition for disk storage. JCS can effectively function as an expiration manager and distribution mechanism on top of a Berkeley DB JE.

Lateral Plugins

JCS provides two recommended lateral distribution options: TCP socket server distribution and JGroups

(or JavaGroups). There are also several other experimental lateral distribution auxiliaries using servlets, UDP, and xmlrpc.

Remote Plugins

JCS also provides an RMI based remote server to manage distribution of cached data.

2.4 Basic Web Example

Using JCS: Some basics for the web

The primary bottleneck in most dynamic web-based applications is the retrieval of data from the database. While it is relatively inexpensive to add more front-end servers to scale the serving of pages and images and the processing of content, it is an expensive and complex ordeal to scale the database. By taking advantage of data caching, most web applications can reduce latency times and scale farther with fewer machines.

JCS is a front-tier cache that can be configured to maintain consistency across multiple servers by using a centralized remote server or by lateral distribution of cache updates. Other caches, like the Javlin EJB data cache, are basically in-memory databases that sit between your EJB's and your database. Rather than trying to speed up your slow EJB's, you can avoid most of the network traffic and the complexity by implementing JCS front-tier caching. Centralize your EJB access or your JDBC data access into local managers and perform the caching there.

What to cache?

The data used by most web applications varies in its dynamicity, from completely static to always changing at every request. Everything that has some degree of stability can be cached. Prime candidates for caching range from the list data for stable dropdowns, user information, discrete and infrequently changing information, to stable search results that could be sorted in memory.

Since JCS is distributed and allows updates and invalidations to be broadcast to multiple listeners, frequently changing items can be easily cached and kept in sync through your data access layer. For data that must be 100% up to date, say an account balance prior to a transfer, the data should directly be retrieved from the database. If your application allows for the viewing and editing of data, the data for the view pages could be cached, but the edit pages should, in most cases, pull the data directly from the database.

How to cache discrete data

Let's say that you have an e-commerce book store. Each book has a related set of information that you must present to the user. Let's say that 70% of your hits during a particular day are for the same 1,000 popular items that you advertise on key pages of your site, but users are still actively browsing your catalog of over a million books. You cannot possibly cache your entire database, but you could dramatically decrease the load on your database by caching the 1,000 or so most popular items.

For the sake of simplicity let's ignore tie-ins and user-profile based suggestions (also good candidates for caching) and focus on the core of the book detail page.

A simple way to cache the core book information would be to create a value object for book data that

contains the necessary information to build the display page. This value object could hold data from multiple related tables or book subtype table, but lets say that you have a simple table called `BOOK` that looks something like this:

```
Table BOOK
BOOK_ID_PK
TITLE
AUTHOR
ISBN
PRICE
PUBLISH_DATE
```

We could create a value object for this table called `BookVObj` that has variables with the same names as the table columns that might look like this:

```
package com.genericbookstore.data;

import java.io.Serializable;
import java.util.Date;

public class BookVObj implements Serializable
{
    public int bookId = 0;
    public String title;
    public String author;
    public String ISBN;
    public String price;
    public Date publishDate;

    public BookVObj()
    {
    }
}
```

Then we can create a manager called `BookVObjManager` to store and retrieve `BookVObj`'s. All access to core book data should go through this class, including inserts and updates, to keep the caching simple. Let's make `BookVObjManager` a singleton that gets a JCS access object in initialization. The start of the class might look like:

```
package com.genericbookstore.data;

import org.apache.jcs.JCS;
// in case we want to set some special behavior
import org.apache.jcs.engine.behavior.IElementAttributes;

public class BookVObjManager
{
    private static BookVObjManager instance;
    private static int checkedOut = 0;
    private static JCS bookCache;
```

```
private BookVObjManager()
{
    try
    {
        bookCache = JCS.getInstance("bookCache");
    }
    catch (Exception e)
    {
        // Handle cache region initialization failure
    }

    // Do other initialization that may be necessary, such as getting
    // references to any data access classes we may need to populate
    // value objects later
}

/**
 * Singleton access point to the manager.
 */
public static BookVObjManager getInstance()
{
    synchronized (BookVObjManager.class)
    {
        if (instance == null)
        {
            instance = new BookVObjManager();
        }
    }

    synchronized (instance)
    {
        instance.checkedOut++;
    }

    return instance;
}
```

To get a `BookVObj` we will need some access methods in the manager. We should be able to get a non-cached version if necessary, say before allowing an administrator to edit the book data. The methods might look like:

```
/**
 * Retrieves a BookVObj.  Default to look in the cache.
 */
public BookVObj getBookVObj(int id)
{
    return getBookVObj(id, true);
}

/**
 * Retrieves a BookVObj. Second argument decides whether to look
 * in the cache. Returns a new value object if one can't be
 * loaded from the database. Database cache synchronization is
 * handled by removing cache elements upon modification.
 */
public BookVObj getBookVObj(int id, boolean fromCache)
```

```
{
    BookVObj vObj = null;

    // First, if requested, attempt to load from cache

    if (fromCache)
    {
        vObj = (BookVObj) bookCache.get("BookVObj" + id);
    }

    // Either fromCache was false or the object was not found, so
    // call loadBookVObj to create it

    if (vObj == null)
    {
        vObj = loadvObj(id);
    }

    return  vObj;
}

/**
 * Creates a BookVObj based on the id of the BOOK table.  Data
 * access could be direct JDBC, some or mapping tool, or an EJB.
 */
public BookVObj loadBookVObj(int id)
{
    BookVObj vObj = new BookVObj();

    vObj.bookID = id;

    try
    {
        boolean found = false;

        // load the data and set the rest of the fields
        // set found to true if it was found

        found = true;

        // cache the value object if found

        if (found)
        {
            // could use the defaults like this
            // bookCache.put( "BookVObj" + id, vObj );
            // or specify special characteristics

            // put to cache

            bookCache.put("BookVObj" + id, vObj);
        }

    }
    catch (Exception e)
    {
        // Handle failure putting object to cache
    }

    return vObj;
}
```

We will also need a method to insert and update book data. To keep the caching in one place, this should be the primary way core book data is created. The method might look like:

```
    /**
     * Stores BookVObj's in database.  Clears old items and caches
     * new.
     */
    public int storeBookVObj(BookVObj vObj)
    {
        try
        {
            // since any cached data is no longer valid, we should
            // remove the item from the cache if it an update.

            if (vObj.bookID != 0)
            {
                bookCache.remove("BookVObj" + vObj.bookID);
            }

            // put the new object in the cache

            bookCache.put("BookVObj" + id, vObj);
        }
        catch (Exception e)
        {
            // Handle failure removing object or putting object to cache.
        }
    }
}
```

As elements are placed in the cache via `put`, it is possible to specify custom attributes for those elements such as its maximum lifetime in the cache, whether or not it can be spooled to disk, etc. It is also possible (and easier) to define these attributes in the configuration file as demonstrated later. We now have the basic infrastructure for caching the book data.

Selecting the appropriate auxiliary caches

The first step in creating a cache region is to determine the makeup of the memory cache. For the book store example, I would create a region that could store a bit over the minimum number I want to have in memory, so the core items always readily available. I would set the maximum memory size to `1200`. In addition, I might want to have all objects in this cache region expire after `7200` seconds. This can be configured in the element attributes on a default or per-region basis as illustrated in the configuration file below.

For most cache regions you will want to use a disk cache if the data takes over about .5 milliseconds to create. The indexed disk cache is the most efficient disk caching auxiliary, and for normal usage it is recommended.

The next step will be to select an appropriate distribution layer. If you have a back-end server running an appserver or scripts or are running multiple webserver VMs on one machine, you might want to use the centralized remote cache . The lateral cache would be fine, but since the lateral cache binds to a port,

you'd have to configure each VM's lateral cache to listen to a different port on that machine.

If your environment is very flat, say a few load-balanced webservers and a database machine or one webserver with multiple VMs and a database machine, then the lateral cache will probably make more sense. The TCP lateral cache is recommended.

For the book store configuration I will set up a region for the `bookCache` that uses the LRU memory cache, the indexed disk auxiliary cache, and the remote cache. The configuration file might look like this:

```
# DEFAULT CACHE REGION

# sets the default aux value for any non configured caches
jcs.default=DC,RFailover
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
jcs.default.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.default.elementattributes.IsEternal=false
jcs.default.elementattributes.MaxLifeSeconds=3600
jcs.default.elementattributes.IdleTime=1800
jcs.default.elementattributes.IsSpool=true
jcs.default.elementattributes.IsRemote=true
jcs.default.elementattributes.IsLateral=true


# CACHE REGIONS AVAILABLE

# Regions preconfigured for caching
jcs.region.bookCache=DC,RFailover
jcs.region.bookCache.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.bookCache.cacheattributes.MaxObjects=1200
jcs.region.bookCache.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.region.bookCache.elementattributes.IsEternal=false
jcs.region.bookCache.elementattributes.MaxLifeSeconds=7200
jcs.region.bookCache.elementattributes.IdleTime=1800
jcs.region.bookCache.elementattributes.IsSpool=true
jcs.region.bookCache.elementattributes.IsRemote=true
jcs.region.bookCache.elementattributes.IsLateral=true

# AUXILIARY CACHES AVAILABLE

# Primary Disk Cache -- faster than the rest because of memory key storage
jcs.auxiliary.DC=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
jcs.auxiliary.DC.attributes.DiskPath=/usr/opt/bookstore/raf
jcs.auxiliary.DC.attributes.MaxPurgatorySize=10000
jcs.auxiliary.DC.attributes.MaxKeySize=10000
jcs.auxiliary.DC.attributes.OptimizeAtRemoveCount=300000
jcs.auxiliary.DC.attributes.MaxRecycleBinSize=7500

# Remote RMI Cache set up to failover
jcs.auxiliary.RFailover=
    org.apache.jcs.auxiliary.remote.RemoteCacheFactory
jcs.auxiliary.RFailover.attributes=
```

```
    org.apache.jcs.auxiliary.remote.RemoteCacheAttributes
jcs.auxiliary.RFailover.attributes.RemoteTypeName=LOCAL
jcs.auxiliary.RFailover.attributes.FailoverServers=scriptserver:1102
jcs.auxiliary.RFailover.attributes.GetOnly=false
```

I've set up the default cache settings in the above file to approximate the `bookCache` settings. Other non-preconfigured cache regions will use the default settings. You only have to configure the auxiliary caches once. For most caches you will not need to pre-configure your regions unless the size of the elements varies radically. We could easily put several hundred thousand `BookVObj`'s in memory. The `1200` limit was very conservative and would be more appropriate for a large data structure.

To get running with the book store example, I will also need to start up the remote cache server on the scriptserver machine. The remote cache documentation describes the configuration.

I now have a basic caching system implemented for my book data. Performance should improve immediately.

3.1 Core

Configuring the Local Cache

This document is intended to provide various answers to questions regarding the configuration of a local cache. The document is presented in a question / answer format.

Where is the configuration information?

Configuration of local caches involves editing the cache configuration file, named `cache.ccf`. The classpath should include the directory where this file is located or the file should be placed at the root of the classpath, since it is discovered automatically.

What is in the cache.ccf file?

The `cache.ccf` file contains default configuration information for cache regions and specific configuration information for regions that you predefine. Regions not using default behaviors should generally be configured via the `cache.ccf` file. If you can put configuration information in a class, you can edit a props file just as easily. This makes modification of the regional setting more efficient and allows for startup error checking.

There are three main sections of the `cache.ccf` file:

- the default and system settings
- the region specific settings
- the auxiliary cache definitions

How do I set up default values for regions?

You can establish default values that any non-preconfigured region will inherit. The non-predefined region will be created when you call `CacheAccess.getAccess("cacheName")`. The default setting look like this:

```
# DEFAULT CACHE REGION

# sets the default aux value for any non configured caches
jcs.default=DC,RFailover
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
```

The most important line is `jcs.default=DC,Rfailover`. This tells the cache what auxiliary caches should be used. Auxiliary caches are configured in the third section of the `cache.ccf` and are referenced in a comma separated list. You can add as many auxiliary caches as you want, but the behavior of remote and lateral auxiliaries may conflict. This allows you to define different configurations for auxiliary caches and to use these different configurations for different regions.

How do I define a region?

Defining a region involves specifying which auxiliary caches it will use and how many objects it will store in memory. A typical region definition looks like:

```
jcs.region.testCache=DC,RFailover
jcs.region.testCache.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.testCache.cacheattributes.MaxObjects=1000
```

The region name is `testCache`. It will have a 1000 item memory limit and will use the DC and RFailover auxiliary caches. If a typical element for this region was very large, you might want to lower the number of items stored in memory. The size of the memory storage is dependent on the priority of the cache, the size of its elements, and the amount of RAM on the machine.

How do I configure an auxiliary cache?

Each auxiliary cache is created through a factory that passes an attribute object to the constructor. The attributes are set via reflection and should be fairly simple to understand. Each auxiliary cache will be fully documented. Plugging in your own auxiliary cache become a simple matter given the reflexive manner of initialization.

The most important settings for common usage are the disk path and the remote cache location. It is recommended that only disk and remote auxiliaries be used. The lateral caches are functional but not as efficient.

The default configuration code above specifies that non-preconfigured caches use the auxiliary cache by the name DC. This cache is defined in the third section of the file:

```
jcs.auxiliary.DC=
    org.apache.jcs.auxiliary.disk.DiskCacheFactory
jcs.auxiliary.DC.attributes=
    org.apache.jcs.auxiliary.disk.DiskCacheAttributes
jcs.auxiliary.DC.attributes.DiskPath=c:/dev/cache/raf
```

The only thing that needs to be set here is the `DiskPath` value. Change it to wherever you want the cache to persist unused items.

The default region is also set to use an auxiliary called `RFailover`. This is a remote cache that is

designed to failover to other remote servers in a cluster:

```
jcs.auxiliary.RFailover=
    org.apache.jcs.auxiliary.remote.RemoteCacheFactory
jcs.auxiliary.RFailover.attributes=
    org.apache.jcs.auxiliary.remote.RemoteCacheAttributes
jcs.auxiliary.RFailover.attributes.RemoteTypeName=LOCAL
jcs.auxiliary.RFailover.attributes.FailoverServers=
    localhost:1102,localhost:1101
```

If you don't have more than one remote server running, just specify it by itself in the `FailoverServers` attribute.

3.1.1 Basic JCS Config

Basic JCS Configuration

The following document illustrates several basic JCS configurations. As you'll see, using JCS can be as simple as creating a single memory cache for you application. However, with a few configuration changes, you can quickly enable some distributed caching features that can scale your application even further.

Building a cache.ccf file

Configuring the JCS can be as simple as your needs. The most basic configuration would be a pure memory cache where every region takes the default values. The complete configuration file (cache.ccf) could look like this:

```
# DEFAULT CACHE REGION

jcs.default=
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
jcs.default.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
```

If you want to add memory shrinking then you can add these lines:

```
jcs.default.cacheattributes.UseMemoryShrinker=true
jcs.default.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.default.cacheattributes.ShrinkerIntervalSeconds=60
jcs.default.cacheattributes.MaxSpoolPerRun=500
jcs.default.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.default.elementattributes.IsEternal=false
```

Adding a disk cache is as simple as telling it what folder to use. It is recommended that you add a disk cache. If you want to add a disk cache to your default parameters, then (1) add this to the bottom of the file to create the auxiliary:

```
jcs.auxiliary.DC=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
```

```
jcs.auxiliary.DC.attributes.DiskPath=g:/dev/jcs/raf
```

and (2) change the first line to:

```
jcs.default=DC
```

If you want to predefine a specific region, say called `testCache1`, then add these lines:

```
jcs.region.testCache1=DC
jcs.region.testCache1.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.testCache1.cacheattributes.MaxObjects=1000
jcs.region.testCache1.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.region.testCache1.cacheattributes.UseMemoryShrinker=true
jcs.region.testCache1.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.region.testCache1.cacheattributes.ShrinkerIntervalSeconds=60
jcs.region.testCache1.cacheattributes.MaxSpoolPerRun=500
jcs.region.testCache1.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.region.testCache1.elementattributes.IsEternal=false
```

If you want to add a lateral cache for distribution (the TCP Lateral Auxiliary is recommended), then add these lines to the bottom of the file to define the auxiliary:

```
jcs.auxiliary.LTCP=
    org.apache.jcs.auxiliary.lateral.LateralCacheFactory
jcs.auxiliary.LTCP.attributes=
    org.apache.jcs.auxiliary.lateral.LateralCacheAttributes
jcs.auxiliary.LTCP.attributes.TransmissionTypeName=TCP
jcs.auxiliary.LTCP.attributes.TcpServers=localhost:1111
jcs.auxiliary.LTCP.attributes.TcpListenerPort=1110
jcs.auxiliary.LTCP.attributes.PutOnlyMode=false
```

See the TCP Lateral documentation for more information. If you want to set up `testCache1` to use this, then change the definition to:

```
jcs.region.testCache1=DC,LTCP
```

A few comments on configuration

Auxiliary definitions are like log4j appenders, they are defined and then associated with a region like a log4j category.

The order of configuration file is unimportant, though you should try to keep it organized for your own sake.

Configuration is being refactored and is subject to change. It should only become easier.

The complete file

The complete file from above would look like this:

```
# DEFAULT CACHE REGION

jcs.default=DC,LTCP
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
jcs.default.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache

# PRE-DEFINED CACHE REGIONS

jcs.region.testCache1=DC,LTCP
jcs.region.testCache1.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.testCache1.cacheattributes.MaxObjects=1000
jcs.region.testCache1.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.region.testCache1.cacheattributes.UseMemoryShrinker=true
jcs.region.testCache1.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.region.testCache1.cacheattributes.ShrinkerIntervalSeconds=60
jcs.region.testCache1.cacheattributes.MaxSpoolPerRun=500
jcs.region.testCache1.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.region.testCache1.elementattributes.IsEternal=false


# AVAILABLE AUXILIARY CACHES
jcs.auxiliary.DC=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
jcs.auxiliary.DC.attributes.DiskPath=g:/dev/jcs/raf
jcs.auxiliary.DC.attributes.maxKeySize=100000

jcs.auxiliary.LTCP=
    org.apache.jcs.auxiliary.lateral.LateralCacheFactory
jcs.auxiliary.LTCP.attributes=
    org.apache.jcs.auxiliary.lateral.LateralCacheAttributes
jcs.auxiliary.LTCP.attributes.TransmissionTypeName=TCP
jcs.auxiliary.LTCP.attributes.TcpServers=localhost:1111
jcs.auxiliary.LTCP.attributes.TcpListenerPort=1110
jcs.auxiliary.LTCP.attributes.PutOnlyMode=false
```

3.1.2 Element Config

Element Attribute Configuration

The following document describes the various configuration options available for cache elements. Each element put into the cache can be configured independently. You can define element behavior in three ways: as a default setting, as a region setting, or at the element level.

Setting the defaults

The configuration below can be put in the cache.ccf configuration file. It establishes the default behavior for all regions. A region can override these defaults and an individual element can override these defaults and the region settings.

```
# DEFAULT CACHE REGION

jcs.default=DC
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
jcs.default.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.default.cacheattributes.UseMemoryShrinker=true
jcs.default.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.default.cacheattributes.ShrinkerIntervalSeconds=60
jcs.default.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.default.elementattributes.IsEternal=false
jcs.default.elementattributes.MaxLifeSeconds=700
jcs.default.elementattributes.IdleTime=1800
jcs.default.elementattributes.IsSpool=true
jcs.default.elementattributes.IsRemote=true
jcs.default.elementattributes.IsLateral=true
```

The default and region configuration settings have three components. They define what auxiliaries are available, how the cache should control the memory, and how the elements should behave. This configuration tells all regions to use an auxiliary called DC by default. It also establishes several settings for memory management (see Basic JCS Configuration for more information on the cacheattribute settings). In addition, by default all regions will take these element configuration settings.

These settings specify that elements are not eternal, i.e. they can expire. By default elements are considered eternal.

You can define the maximum life of an item by setting the `MaxLifeSeconds` parameter. If an item has

been in the cache for longer than the set number of seconds it will not be retrieved on a get request. If you use the memory shrinker the item will be actively removed from memory. Currently there is no background disk shrinker, but the disk cache does allow for a maximum number of keys (see Indexed Disk Cache for more information on the disk cache settings).

You can define the maximum time an item can live without being accessed by setting the `IdleTime` parameter. This is different than the `MaxMemoryIdleTimeSeconds` parameter, which just specifies how long an object can be in memory before it is subjected to removal or being spooled to a disk cache if it is available. Note: the `IdleTime` parameter may not function properly for items retrieved from disk, if you have a memory size of 0.

`IsSpool` determines whether or not the element can go to disk, if a disk cache is configured for the region.

`IsRemote` determines whether or not the element can be sent to a remote server, if one is configured for the region.

`IsLateral` determines whether or not the element can be laterally distributed, if a lateral auxiliary is configured for the region.

Programmatic Configuration

Every element put into the cache has its own set of attributes. By default elements are given a copy of the default element attributes associated with a region. You can also specify the attributes to use for an element when you put it in the cache.

```
JCS jcs = JCS.getInstance( "myregion" );

. . .

// jcs.getDefaultElementAttributes returns a copy not a reference
IElementAttributes attributes = jcs.getDefaultElementAttributes();

// set some special value
attributes.setIsEternal( true );
jcs.put( "key", "data", attributes );
```

You can also programmatically modify the default element attributes.

```
JCS jcs = JCS.getInstance( "myregion" );

. . .

// jcs.getDefaultElementAttributes returns a copy not a reference
IElementAttributes attributes = jcs.getDefaultElementAttributes();

// set some special value
attributes.setIsEternal( true );
```

```
jcs.setDefaultElementAttributes( attributes );
```

3.1.3 Element Event Handling

Element Event Handling

JCS allows you to attach event handlers to elements in the local memory cache.

There are several events that you can listen for. All of the events are local memory related events. Element event handlers are not transmitted to other caches via lateral or remote auxiliaries, nor are they spooled to disk.

You can register multiple handlers for a single item. Although the handlers are associated with particular items, you can also setup default handlers for any region. Each item put into the region, that will take the default element attributes, will be assigned the event default event handlers.

The various events that you can handle have all been assigned integer codes. The codes are defined in the org.apache.jcs.engine.control.event.behavior.IElementEventConstants interface. The events are named descriptively and include:

Name	Description
ELEMENT_EVENT_EXCEEDED_MAXLIFE_BACKGROUND	The element exceeded its max life. This was detected in a background cleanup.
ELEMENT_EVENT_EXCEEDED_MAXLIFE_ONREQUEST	The element exceeded its max life. This was detected on request.
ELEMENT_EVENT_EXCEEDED_IDLETIME_BACKGROUND	The element exceeded its max idle. This was detected in a background cleanup.
ELEMENT_EVENT_EXCEEDED_IDLETIME_ONREQUEST	The element exceeded its max idle time. This was detected on request.
ELEMENT_EVENT_SPOOLED_DISK_AVAILABLE	The element was pushed out of the memory store, there is a disk store available for the region, and the element is marked as spoolable.
ELEMENT_EVENT_SPOOLED_DISK_NOT_AVAILABLE	The element was pushed out of the memory store, and there is not a disk store available for the region.
ELEMENT_EVENT_SPOOLED_NOT_ALLOWED	The element was pushed out of the memory store, there is a disk store available for the region, but the element is marked as not spoolable.

To create an event handler you must implement the org.apache.jcs.engine.control.event.behavior.IElementEventHandler interface. This interface contains only one method:

```
public void handleElementEvent( IElementEvent event );
```

The IElementEvent object contains both the event code and the source. The source is the element for which the event occurred. The code is the type of event. If you have an event handler registered, it will be called whenever any event occurs. It is up to the handler to decide what it would like to do for the particular event. Since there are not that many events, this does not create too much activity. Also, the event handling is done asynchronously. Events are added to an event queue and processed by background threads.

Once you have an IElementEventHandler implementation, you can attach it to an element via the Element Attributes. You can either add it to the element attributes when you put an item into the cache, add it to the attributes of an item that exist in the cache (which just results in a re-put), or add the event handler to the default element attributes for a region. If you add it to the default attributes, then all elements subsequently added to the region that do not define their own element attributes will be assigned the default event handlers.

```
JCS jcs = JCS.getInstance( "myregion" );

. . .

MyEventHandler meh = new MyEventHandler();

// jcs.getDefaultElementAttributes returns a copy not a reference
IElementAttributes attributes = jcs.getDefaultElementAttributes();
attributes.addElementEventHandler( meh );
jcs.put( "key", "data", attributes );
```

Here is how to setup an event handler as a default setting for a region:

```
JCS jcs = JCS.getInstance( "myregion" );

. . .

MyEventHandler meh = new MyEventHandler();

// this should add the event handler to all items as
//they are created.
// jcs.getDefaultElementAttributes returns a copy not a reference
IElementAttributes attributes = jcs.getDefaultElementAttributes();
attributes.addElementEventHandler( meh );
jcs.setDefaultElementAttributes( attributes );
```

3.1.4 Region Properties

Cache Region Configuration

The following properties apply to any cache region. They can be specified as default values and specified on a region by region basis. There are three types of settings: auxiliary, cache, and element. The cache settings define the memory management for the region. The element settings define default element behavior within the region.

Region (Auxiliary) Properties

Property	Description	Required	Default Value
	You can specify the list of auxiliaries that regions can use. This has no attribute name. The list can be empty, otherwise it should be comma delimited.	Y	n/a

Region (Cache) Properties

Property	Description	Required	Default Value
MaxObjects	The maximum number of items allowed in memory. Eviction of elements in excess of this number is determined by the memory cache. By default JCS uses the LRU memory cache.	Y	n/a
MemoryCacheName	This property allows you to specify what memory manager you would like to use. You can create your own memory manager by implementing the org.apache.jcs.engine.memory.MemoryCache interface. Alternatively, you can extend the org.apache.jcs.engine.memory.AbstractMemoryCache class. Several different memory caches are available: two LRU implementations, an LFU, and an adaptive replacement algorithm.	N	org.apache.jcs.engine.memory.lru.LRUMemoryCache
UseMemoryShrinker	By default, the memory shrinker is shared by all regions that use the LRU memory cache. The memory shrinker iterates through the items in memory, looking for items that have expired or that have exceeded their max memory idle time.	N	false

Property	Description	Required	Default Value
MaxMemoryIdleTimeSeconds	This is only used if you are using the memory shrinker. If this value is set above -1, then if an item has not been accessed in this number of seconds, it will be spooled to disk if the disk is available. You can register an event handler on this event.	N	-1
ShrinkerIntervalSeconds	This specifies how often the shrinker should run, if it has been activated. If you set UseMemoryShrinker to false, then this setting has no effect.	N	60
DiskUsagePattern	SWAP is the default. Under the swap pattern, data is only put to disk when the max memory size is reached. Since items puled from disk are put into memory, if the memory cache is full and you get an item off disk, the lest recently used item will be spooled to disk. If you have a low memory hit ration, you end up thrashing. The UPDATE usage pattern allows items to go to disk on an update. It disables the swap. This allows you to persist all items to disk. If you are using the JDBC disk cache for instance, you can put all the items on disk while using the memory cache for performance, and not worry about lossing data from a system crash or improper shutdown. Also, since all items are on disk, there is no need to swap to disk. This prevents the possibility of thrashing.	N	SWAP

Region (Element) Properties

Property	Description	Required	Default Value
IsEternal	If an element is specified as eternal, then it will never be subject to removal for exceeding its max life.	N	true
MaxLifeSeconds	If you specify that elements within a region are not eternal, then you can set the max life seconds. If this is exceeded the elmenets will be removed passively when a client tries to retrieve them. If you are using a memory shrinker, then the items can be removed actively.	N	-1
IsSpool	By default, can elements in this region be sent to a disk cache if one is available.	N	true
IsRemote	By default, can elements in this region be sent to a lateral cache if one is available.	N	true

Property	Description	Required	Default Value
IsLateral	By default, can elements in this region be sent to a remote cache if one is available.	N	true

Example Configuration

```
jcs.default=
jcs.default.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=200001
jcs.default.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.default.cacheattributes.UseMemoryShrinker=true
jcs.default.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.default.cacheattributes.ShrinkerIntervalSeconds=60
jcs.default.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.default.elementattributes.IsEternal=false
jcs.default.elementattributes.MaxLifeSeconds=700
jcs.default.elementattributes.IsSpool=true
jcs.default.elementattributes.IsRemote=true
jcs.default.elementattributes.IsLateral=true

# optional region "testCache1" specific configuration settings
jcs.region.testCache1=
jcs.region.testCache1.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.testCache1.cacheattributes.MaxObjects=123456
jcs.region.testCache1.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.region.testCache1.cacheattributes.UseMemoryShrinker=true
jcs.region.testCache1.cacheattributes.ShrinkerIntervalSeconds=30
jcs.region.testCache1.cacheattributes.MaxMemoryIdleTimeSeconds=300
jcs.region.testCache1.cacheattributes.MaxSpoolPerRun=100
jcs.region.testCache1.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.region.testCache1.elementattributes.IsEternal=false
jcs.region.testCache1.elementattributes.MaxLifeSeconds=60000
jcs.region.testCache1.elementattributes.IsSpool=true
jcs.region.testCache1.elementattributes.IsLateral=true
jcs.region.testCache1.elementattributes.IsRemote=true
```

3.1.5 Basic Web Example

Using JCS: Some basics for the web

The primary bottleneck in most dynamic web-based applications is the retrieval of data from the database. While it is relatively inexpensive to add more front-end servers to scale the serving of pages and images and the processing of content, it is an expensive and complex ordeal to scale the database. By taking advantage of data caching, most web applications can reduce latency times and scale farther with fewer machines.

JCS is a front-tier cache that can be configured to maintain consistency across multiple servers by using a centralized remote server or by lateral distribution of cache updates. Other caches, like the Javlin EJB data cache, are basically in-memory databases that sit between your EJB's and your database. Rather than trying to speed up your slow EJB's, you can avoid most of the network traffic and the complexity by implementing JCS front-tier caching. Centralize your EJB access or your JDBC data access into local managers and perform the caching there.

What to cache?

The data used by most web applications varies in its dynamicity, from completely static to always changing at every request. Everything that has some degree of stability can be cached. Prime candidates for caching range from the list data for stable dropdowns, user information, discrete and infrequently changing information, to stable search results that could be sorted in memory.

Since JCS is distributed and allows updates and invalidations to be broadcast to multiple listeners, frequently changing items can be easily cached and kept in sync through your data access layer. For data that must be 100% up to date, say an account balance prior to a transfer, the data should directly be retrieved from the database. If your application allows for the viewing and editing of data, the data for the view pages could be cached, but the edit pages should, in most cases, pull the data directly from the database.

How to cache discrete data

Let's say that you have an e-commerce book store. Each book has a related set of information that you must present to the user. Let's say that 70% of your hits during a particular day are for the same 1,000 popular items that you advertise on key pages of your site, but users are still actively browsing your catalog of over a million books. You cannot possibly cache your entire database, but you could dramatically decrease the load on your database by caching the 1,000 or so most popular items.

For the sake of simplicity let's ignore tie-ins and user-profile based suggestions (also good candidates for caching) and focus on the core of the book detail page.

A simple way to cache the core book information would be to create a value object for book data that

contains the necessary information to build the display page. This value object could hold data from multiple related tables or book subtype table, but lets say that you have a simple table called `BOOK` that looks something like this:

```
Table BOOK
BOOK_ID_PK
TITLE
AUTHOR
ISBN
PRICE
PUBLISH_DATE
```

We could create a value object for this table called `BookVObj` that has variables with the same names as the table columns that might look like this:

```
package com.genericbookstore.data;

import java.io.Serializable;
import java.util.Date;

public class BookVObj implements Serializable
{
    public int bookId = 0;
    public String title;
    public String author;
    public String ISBN;
    public String price;
    public Date publishDate;

    public BookVObj()
    {
    }
}
```

Then we can create a manager called `BookVObjManager` to store and retrieve `BookVObj`'s. All access to core book data should go through this class, including inserts and updates, to keep the caching simple. Let's make `BookVObjManager` a singleton that gets a JCS access object in initialization. The start of the class might look like:

```
package com.genericbookstore.data;

import org.apache.jcs.JCS;
// in case we want to set some special behavior
import org.apache.jcs.engine.behavior.IElementAttributes;

public class BookVObjManager
{
    private static BookVObjManager instance;
    private static int checkedOut = 0;
    private static JCS bookCache;
```

```
private BookVObjManager()
{
    try
    {
        bookCache = JCS.getInstance("bookCache");
    }
    catch (Exception e)
    {
        // Handle cache region initialization failure
    }

    // Do other initialization that may be necessary, such as getting
    // references to any data access classes we may need to populate
    // value objects later
}

/**
 * Singleton access point to the manager.
 */
public static BookVObjManager getInstance()
{
    synchronized (BookVObjManager.class)
    {
        if (instance == null)
        {
            instance = new BookVObjManager();
        }
    }

    synchronized (instance)
    {
        instance.checkedOut++;
    }

    return instance;
}
```

To get a `BookVObj` we will need some access methods in the manager. We should be able to get a non-cached version if necessary, say before allowing an administrator to edit the book data. The methods might look like:

```
/**
 * Retrieves a BookVObj.  Default to look in the cache.
 */
public BookVObj getBookVObj(int id)
{
    return getBookVObj(id, true);
}

/**
 * Retrieves a BookVObj. Second argument decides whether to look
 * in the cache. Returns a new value object if one can't be
 * loaded from the database. Database cache synchronization is
 * handled by removing cache elements upon modification.
 */
public BookVObj getBookVObj(int id, boolean fromCache)
```

```
{
    BookVObj vObj = null;

    // First, if requested, attempt to load from cache

    if (fromCache)
    {
        vObj = (BookVObj) bookCache.get("BookVObj" + id);
    }

    // Either fromCache was false or the object was not found, so
    // call loadBookVObj to create it

    if (vObj == null)
    {
        vObj = loadvObj(id);
    }

    return  vObj;
}

/**
 * Creates a BookVObj based on the id of the BOOK table.  Data
 * access could be direct JDBC, some or mapping tool, or an EJB.
 */
public BookVObj loadBookVObj(int id)
{
    BookVObj vObj = new BookVObj();

    vObj.bookID = id;

    try
    {
        boolean found = false;

        // load the data and set the rest of the fields
        // set found to true if it was found

        found = true;

        // cache the value object if found

        if (found)
        {
            // could use the defaults like this
            // bookCache.put( "BookVObj" + id, vObj );
            // or specify special characteristics

            // put to cache

            bookCache.put("BookVObj" + id, vObj);
        }

    }
    catch (Exception e)
    {
        // Handle failure putting object to cache
    }

    return vObj;
}
```

We will also need a method to insert and update book data. To keep the caching in one place, this should be the primary way core book data is created. The method might look like:

```
    /**
     * Stores BookVObj's in database.  Clears old items and caches
     * new.
     */
    public int storeBookVObj(BookVObj vObj)
    {
        try
        {
            // since any cached data is no longer valid, we should
            // remove the item from the cache if it an update.

            if (vObj.bookID != 0)
            {
                bookCache.remove("BookVObj" + vObj.bookID);
            }

            // put the new object in the cache

            bookCache.put("BookVObj" + id, vObj);
        }
        catch (Exception e)
        {
            // Handle failure removing object or putting object to cache.
        }
    }
}
```

As elements are placed in the cache via `put`, it is possible to specify custom attributes for those elements such as its maximum lifetime in the cache, whether or not it can be spooled to disk, etc. It is also possible (and easier) to define these attributes in the configuration file as demonstrated later. We now have the basic infrastructure for caching the book data.

Selecting the appropriate auxiliary caches

The first step in creating a cache region is to determine the makeup of the memory cache. For the book store example, I would create a region that could store a bit over the minimum number I want to have in memory, so the core items always readily available. I would set the maximum memory size to `1200`. In addition, I might want to have all objects in this cache region expire after `7200` seconds. This can be configured in the element attributes on a default or per-region basis as illustrated in the configuration file below.

For most cache regions you will want to use a disk cache if the data takes over about .5 milliseconds to create. The indexed disk cache is the most efficient disk caching auxiliary, and for normal usage it is recommended.

The next step will be to select an appropriate distribution layer. If you have a back-end server running an appserver or scripts or are running multiple webserver VMs on one machine, you might want to use the centralized remote cache . The lateral cache would be fine, but since the lateral cache binds to a port,

you'd have to configure each VM's lateral cache to listen to a different port on that machine.

If your environment is very flat, say a few load-balanced webservers and a database machine or one webserver with multiple VMs and a database machine, then the lateral cache will probably make more sense. The TCP lateral cache is recommended.

For the book store configuration I will set up a region for the `bookCache` that uses the LRU memory cache, the indexed disk auxiliary cache, and the remote cache. The configuration file might look like this:

```
# DEFAULT CACHE REGION

# sets the default aux value for any non configured caches
jcs.default=DC,RFailover
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
jcs.default.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.default.elementattributes.IsEternal=false
jcs.default.elementattributes.MaxLifeSeconds=3600
jcs.default.elementattributes.IdleTime=1800
jcs.default.elementattributes.IsSpool=true
jcs.default.elementattributes.IsRemote=true
jcs.default.elementattributes.IsLateral=true


# CACHE REGIONS AVAILABLE

# Regions preconfigured for caching
jcs.region.bookCache=DC,RFailover
jcs.region.bookCache.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.bookCache.cacheattributes.MaxObjects=1200
jcs.region.bookCache.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.region.bookCache.elementattributes.IsEternal=false
jcs.region.bookCache.elementattributes.MaxLifeSeconds=7200
jcs.region.bookCache.elementattributes.IdleTime=1800
jcs.region.bookCache.elementattributes.IsSpool=true
jcs.region.bookCache.elementattributes.IsRemote=true
jcs.region.bookCache.elementattributes.IsLateral=true

# AUXILIARY CACHES AVAILABLE

# Primary Disk Cache -- faster than the rest because of memory key storage
jcs.auxiliary.DC=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
jcs.auxiliary.DC.attributes.DiskPath=/usr/opt/bookstore/raf
jcs.auxiliary.DC.attributes.MaxPurgatorySize=10000
jcs.auxiliary.DC.attributes.MaxKeySize=10000
jcs.auxiliary.DC.attributes.OptimizeAtRemoveCount=300000
jcs.auxiliary.DC.attributes.MaxRecycleBinSize=7500

# Remote RMI Cache set up to failover
jcs.auxiliary.RFailover=
    org.apache.jcs.auxiliary.remote.RemoteCacheFactory
jcs.auxiliary.RFailover.attributes=
```

```
    org.apache.jcs.auxiliary.remote.RemoteCacheAttributes
jcs.auxiliary.RFailover.attributes.RemoteTypeName=LOCAL
jcs.auxiliary.RFailover.attributes.FailoverServers=scriptserver:1102
jcs.auxiliary.RFailover.attributes.GetOnly=false
```

I've set up the default cache settings in the above file to approximate the `bookCache` settings. Other non-preconfigured cache regions will use the default settings. You only have to configure the auxiliary caches once. For most caches you will not need to pre-configure your regions unless the size of the elements varies radically. We could easily put several hundred thousand `BookVObj`'s in memory. The `1200` limit was very conservative and would be more appropriate for a large data structure.

To get running with the book store example, I will also need to start up the remote cache server on the scriptserver machine. The remote cache documentation describes the configuration.

I now have a basic caching system implemented for my book data. Performance should improve immediately.

3.2 Auxiliary

JCS Plugin Overview

JCS provides multiple auxiliaries which can be plugged into a cache region, in a manner similar to adding Log4j appenders to a logger. JCS auxiliaries are defined in the cache.ccf file. You can specify which plugins a particular cache region should use.

There are four types of auxiliaries: (1) memory, (2) disk, (3) lateral, and (4) remote. Each region is required to have one and only one memory auxiliary. No other auxiliaries are required and any possible combination of disk, lateral, and remote auxiliaries is allowed. If you do not want to store items in memory, then the maximum size for the memory caches can be set to 0 on a per region basis.

Memory Plugins

Currently, JCS provides four memory management options: (1) LRUMemoryCache, (2) LHMLRUMemoryCache, (3) MRUMemoryCache, and (4) ARCMemoryCache. All memory caches restrict the number of items that can be stored in memory per region. If a disk cache is configured for the region, the items will be spooled to disk when the memory capacity is reached. JCS enforces configurable parameters such as time to live and maximum idle time. Expired elements can be cleaned up by the ShrinkerThread, otherwise they will be removed at the next retrieval attempt or when the capacity is reached.

The LRUMemoryCache is the currently recommended plugin. Upon misconfiguration it is used as the default. The LRUMemoryCache removes the least recently used items when the cache is full.

The ARCMemoryCache is currently experimental, but will be fully tested soon. It implements an adaptive replacement caching algorithm that combines an LRU and an LFU that adapt to usage patterns.

Disk Plugins

JCS provides several disk swap options: indexed disk, HSQL, JISP, and Berkeley DB JE. The IndexedDiskCache is the recommended disk cache. It maintains the cached data on disk and the keys in memory for the fastest possible lookup times. Writing to disk is done asynchronously. Items are typically put in purgatory and queued for background disk writing. While in purgatory, the items remain available.

In addition, JCS provides a disk auxiliary that uses the Berkeley DB Java Edition for disk storage. JCS can effectively function as an expiration manager and distribution mechanism on top of a Berkeley DB JE.

Lateral Plugins

JCS provides two recommended lateral distribution options: TCP socket server distribution and JGroups

(or JavaGroups). There are also several other experimental lateral distribution auxiliaries using servlets, UDP, and xmlrpc.

Remote Plugins

JCS also provides an RMI based remote server to manage distribution of cached data.

3.2.1 Indexed Disk Cache

Indexed Disk Auxiliary Cache

The Indexed Disk Auxiliary Cache is an optional plugin for the JCS. It is primarily intended to provide a secondary store to ease the memory burden of the cache. When the memory cache exceeds its maximum size it tells the cache hub that the item to be removed from memory should be spooled to disk. The cache checks to see if any auxiliaries of type "disk" have been configured for the region. If the "Indexed Disk Auxiliary Cache" is used, the item will be spooled to disk.

Disk Indexing

The Indexed Disk Auxiliary Cache follows the fastest pattern of disk caching. Items are stored at the end of a file dedicated to the cache region. The first byte of each disk entry specifies the length of the entry. The start position in the file is saved in memory, referenced by the item's key. Though this still requires memory, it is insignificant given the performance trade off. Depending on the key size, 500,000 disk entries will probably only require about 3 MB of memory. Locating the position of an item is as fast as a map lookup and the retrieval of the item only requires 2 disk accesses.

When items are removed from the disk cache, the location of the available block on the storage file is recorded in a sorted preferential array of a size not to exceed the maximum number of keys allowed in memory. This allows the disk cache to reuse empty spots, thereby keeping the file size to a minimum.

Purgatory

Writing to the disk cache is asynchronous and made efficient by using a memory staging area called purgatory. Retrievals check purgatory then disk for an item. When items are sent to purgatory they are simultaneously queued to be put to disk. If an item is retrieved from purgatory it will no longer be written to disk, since the cache hub will move it back to memory. Using purgatory insures that there is no wait for disk writes, unecessary disk writes are avoided for borderline items, and the items are always available.

Persistence

When the disk cache is properly shutdown, the memory index is written to disk and the value file is defragmented. When the cache starts up, the disk cache can be configured to read or delete the index file. This provides an unreliable persistence mechanism.

Configuration

Configuration is simple and is done in the auxiliary cache section of the `cache.ccf` configuration file. In the example below, I created an Indexed Disk Auxiliary Cache referenced by `DC` . It uses files located in the "DiskPath" directory.

The Disk indexes are equipped with an LRU storage limit. The maximum number of keys is configured by the maxKeySize parameter. If the maximum key size is less than 0, no limit will be placed on the number of keys. By default, the max key size is 5000.

```
jcs.auxiliary.DC=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=
    org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
jcs.auxiliary.DC.attributes.DiskPath=g:\dev\jakarta-turbine-stratum\raf
jcs.auxiliary.DC.attributes.MaxKeySize=100000
```

Additional Configuration Options

The indexed disk cache provides some additional configuration options.

The purgatory size of the Disk cache is equipped with an LRU storage limit. The maximum number of elements allowed in purgatory is configured by the MaxPurgatorySize parameter. By default, the max purgatory size is 5000.

Initial testing indicates that the disk cache performs better when the key and purgatory sizes are limited.

```
jcs.auxiliary.DC.attributes.MaxPurgatorySize=10000
```

Slots in the data file become empty when items are removed from the disk cache. The indexed disk cache keeps track of empty slots in the data file, so they can be reused. The slot locations are stored in a sorted preferential array -- the recycle bin. The smallest items are removed from the recycle bin when it reaches the specified limit. The MaxRecycleBinSize cannot be larger than the MaxKeySize. If the MaxKeySize is less than 0, the recycle bin will default to 5000.

If all the items put on disk are the same size, then the recycle bin will always return perfect matches. However, if the items are of various sizes, the disk cache will use the free spot closest in size but not smaller than the item being written to disk. Since some recycled spots will be larger than the items written to disk, unusable gaps will result. Optimization is intended to remove these gaps.

```
jcs.auxiliary.DC.attributes.MaxRecycleBinSize=10000
```

The Disk cache can be configured to defragment the data file at runtime. Since defragmentation is only

necessary if items have been removed, the deframentation interval is determined by the number of removes. Currently there is no way to schedule defragmentation to run at a set time. If you set the OptimizeAtRemoveCount to -1, no optimizations of the data file will occur until shutdown. By default the value is -1.

In version 1.2.7.9 of JCS, the optimization routine was significantly improved. It now occurs in place, without the aid of a temporary file.

```
jcs.auxiliary.DC.attributes.OptimizeAtRemoveCount=30000
```

A Complete Configuration Example

In this sample cache.ccf file, I configured the cache to use a disk cache, called DC, by default. Also, I explicitly set a cache region called myRegion1 to use DC. I specified custom settings for all of the Indexed Disk Cache configuration parameters.

```
#################################################################
##### Default Region Configuration
jcs.default=DC
jcs.default.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=100
jcs.default.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory.lru.LRUMemoryCache

#################################################################
##### CACHE REGIONS
jcs.region.myRegion1=DC
jcs.region.myRegion1.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.myRegion1.cacheattributes.MaxObjects=1000
jcs.region.myRegion1.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory.lru.LRUMemoryCache

#################################################################
##### AUXILIARY CACHES
# Indexed Disk Cache
jcs.auxiliary.DC=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
jcs.auxiliary.DC.attributes.DiskPath=target/test-sandbox/indexed-disk-cache
jcs.auxiliary.DC.attributes.MaxPurgatorySize=10000
jcs.auxiliary.DC.attributes.MaxKeySize=10000
jcs.auxiliary.DC.attributes.OptimizeAtRemoveCount=300000
jcs.auxiliary.DC.attributes.OptimizeOnShutdown=true
jcs.auxiliary.DC.attributes.MaxRecycleBinSize=7500
```

Using Thread Pools to Reduce Threads

The Indexed Disk Cache allows you to use fewer threads than active regions. By default the disk cache will use the standard cache event queue which has a dedicated thread. Although the standard queue kills its worker thread after a minute of inactivity, you may want to restrict the total number of threads. You can accomplish this by using a pooled event queue.

The configuration file below defines a disk cache called DC2. It uses an event queue of type POOLED. The queue is named disk_cache_event_queue. The disk_cache_event_queue is defined in the bottom of the file.

```
################################################################
################## DEFAULT CACHE REGION  ####################
# sets the default aux value for any non configured caches
jcs.default=DC2
jcs.default.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=200001
jcs.default.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.default.cacheattributes.UseMemoryShrinker=false
jcs.default.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.default.cacheattributes.ShrinkerIntervalSeconds=60
jcs.default.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.default.elementattributes.IsEternal=false
jcs.default.elementattributes.MaxLifeSeconds=700
jcs.default.elementattributes.IdleTime=1800
jcs.default.elementattributes.IsSpool=true
jcs.default.elementattributes.IsRemote=true
jcs.default.elementattributes.IsLateral=true

################################################################
################## AUXILIARY CACHES AVAILABLE ################

# Disk Cache Using a Pooled Event Queue -- this allows you
# to control the maximum number of threads it will use.
# Each region uses 1 thread by default in the SINGLE model.
# adding more threads than regions does not help performance.
# If you want to use a separate pool for each disk cache, either use
# the single model or define a different auxiliary for each region and use the
Pooled type.
# SINGLE is generally best unless you ahve a huge # of regions.
jcs.auxiliary.DC2=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC2.attributes=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
jcs.auxiliary.DC2.attributes.DiskPath=target/test-sandbox/raf
jcs.auxiliary.DC2.attributes.MaxPurgatorySize=10000
jcs.auxiliary.DC2.attributes.MaxKeySize=10000
jcs.auxiliary.DC2.attributes.MaxRecycleBinSize=5000
jcs.auxiliary.DC2.attributes.OptimizeAtRemoveCount=300000
jcs.auxiliary.DC.attributes.OptimizeOnShutdown=true
jcs.auxiliary.DC2.attributes.EventQueueType=POOLED
jcs.auxiliary.DC2.attributes.EventQueuePoolName=disk_cache_event_queue

################################################################
################## OPTIONAL THREAD POOL CONFIGURATION ########

# Disk Cache Event Queue Pool
thread_pool.disk_cache_event_queue.useBoundary=false
thread_pool.remote_cache_client.maximumPoolSize=15
thread_pool.disk_cache_event_queue.minimumPoolSize=1
thread_pool.disk_cache_event_queue.keepAliveTime=3500
thread_pool.disk_cache_event_queue.startUpSize=1
```

3.2.2 Indexed Disk Properties

Indexed Disk Auxiliary Cache Configuration

The following properties apply to the Indexed Disk Cache plugin.

Indexed Disk Configuration Properties

Property	Description	Required	Default Value
DiskPath	The directory where the disk cache should write its files.	Y	n/a
MaxPurgatorySize	The maximum number of items allowed in the queue of items to be written to disk.	N	5000
MaxKeySize	The maximum number of keys that the indexed disk cache can have. Since the keys are stored in memory, you may want to limit this number to something reasonable. The default is a bit small.	N	5000
OptimizeAtRemoveCount	At how many removes should the cache try to defragment the data file. Since we recycle empty spots, defragmentation is usually not needed. To prevent the cache from defragmenting the data file, you can set this to -1. This is the default value.	N	-1
OptimizeOnShutdown	By default the Indexed Disk Cache will optimize on shutdown if the free data size is greater than 0. If you want to prevent this behavior, you can set this parameter to false.	N	true
MaxRecycleBinSize	The maximum number of empty spots the cache will keep track of. The smallest are removed when the maximum size is reached. Keeping track of empty spots on disk allows us to reuse spots, thereby keeping the file from growing unncessarily.	N	5000

Example Configuration

```
jcs.auxiliary.DC=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
```

```
jcs.auxiliary.DC.attributes.DiskPath=target/test-sandbox/indexed-disk-cache
jcs.auxiliary.DC.attributes.MaxPurgatorySize=10000
jcs.auxiliary.DC.attributes.MaxKeySize=10000
jcs.auxiliary.DC.attributes.OptimizeAtRemoveCount=300000
jcs.auxiliary.DC.attributes.OptimizeOnShutdown=true
jcs.auxiliary.DC.attributes.MaxRecycleBinSize=7500
```

Indexed Disk Event Queue Configuration

Property	Description	Required	Default Value
EventQueueType	This should be either SINGLE or POOLED. By default the single style pool is used. The single style pool uses a single thread per event queue. That thread is killed whenever the queue is inactive for 30 seconds. Since the disk cache uses an event queue for every region, if you have many regions and they are all active, you will be using many threads. To limit the number of threads, you can configure the disk cache to use the pooled event queue. Using more threads than regions will not add any benefit for the indexed disk cache, since only one thread can read or write at a time for a single region.	N	SINGLE
EventQueuePoolName	This is the name of the pool to use. It is required if you choose the POOLED event queue type, otherwise it is ignored.	Y	n/a

Example Configuration Using Thread Pool

```
jcs.auxiliary.DC=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttributes
jcs.auxiliary.DC.attributes.DiskPath=target/test-sandbox/indexed-disk-cache
jcs.auxiliary.DC.attributes.MaxPurgatorySize=10000
jcs.auxiliary.DC.attributes.MaxKeySize=10000
jcs.auxiliary.DC.attributes.OptimizeAtRemoveCount=300000
jcs.auxiliary.DC.attributes.OptimizeOnShutdown=true
jcs.auxiliary.DC.attributes.MaxRecycleBinSize=7500
jcs.auxiliary.DC.attributes.EventQueueType=POOLED
jcs.auxiliary.DC.attributes.EventQueuePoolName=disk_cache_event_queue

# Disk Cache pool
thread_pool.disk_cache_event_queue.boundarySize=50
thread_pool.disk_cache_event_queue.useBoundary=true
thread_pool.disk_cache_event_queue.maximumPoolSize=15
thread_pool.disk_cache_event_queue.minimumPoolSize=1
thread_pool.disk_cache_event_queue.keepAliveTime=3500
```

```
thread_pool.disk_cache_event_queue.startUpSize=1
```

3.2.3 Block Disk Cache

Block Disk Auxiliary Cache

The Block Disk Cache stores cached values on disk. Like the Indexed Disk Cache, the Block Disk Cache keeps the keys in memory. The Block Disk Cache stores the values in a group of fixed size blocks, whereas the Indexed Disk Cache writes items to disk in one chunk.

The Block Disk Cache has advantages over the normal indexed model for regions where the size of the items varies. Since all the blocks are the same size, the recycle bin is very simple. It is just a list of block numbers. Also, the Block Disk Cache will never need to be optimized. Once the maximum number of keys is reached, blocks will be reused.

Example cache.ccf

```
################################################################
##### DEFAULT REGION  ##########################################

jcs.default=blockDiskCache
jcs.default.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=0
jcs.default.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory.lru.LRUMemoryCache

################################################################
##### AUXILIARY CACHES  ########################################

# Block Disk Cache
jcs.auxiliary.blockDiskCache=org.apache.jcs.auxiliary.disk.block.BlockDiskCacheFactory
jcs.auxiliary.blockDiskCache.attributes=org.apache.jcs.auxiliary.disk.block.BlockDiskCacheAttributes
jcs.auxiliary.blockDiskCache.attributes.DiskPath=target/test-sandbox/block-disk-cache-huge
jcs.auxiliary.blockDiskCache.attributes.MaxPurgatorySize=300000
jcs.auxiliary.blockDiskCache.attributes.MaxKeySize=1000000
jcs.auxiliary.blockDiskCache.attributes.blockSizeBytes=500
jcs.auxiliary.blockDiskCache.attributes.EventQueueType=SINGLE
#jcs.auxiliary.blockDiskCache.attributes.EventQueuePoolName=disk_cache_event_queue

################################################################
################## THREAD POOL CONFIGURATION ###################

# Default thread pool config
thread_pool.default.boundarySize=2000
thread_pool.default.maximumPoolSize=150
thread_pool.default.minimumPoolSize=4
thread_pool.default.keepAliveTime=350000
#RUN ABORT WAIT BLOCK DISCARDOLDEST
thread_pool.default.whenBlockedPolicy=RUN
thread_pool.default.startUpSize=4
```

```
# Disk Cache pool
thread_pool.disk_cache_event_queue.useBoundary=false
thread_pool.disk_cache_event_queue.minimumPoolSize=2
thread_pool.disk_cache_event_queue.keepAliveTime=3500
thread_pool.disk_cache_event_queue.startUpSize=10
```

3.2.4 JDBC Disk Cache

JDBC Disk Auxiliary Cache

The JDBC disk cache uses a relational database such as MySQL as a persistent store. It works with Oracle, MySQL and HSQL. The cache elements are serialized and written into a BLOB. Multiple regions can share a single table. You can define multiple, differently configured JDBC disk caches in one JCS instance. This allows you to use different tables for different cache regions.

Example cache.ccf (MySQL)

```
#####################################################################
################## DEFAULT CACHE REGION  ####################
# sets the default aux value for any non configured caches
jcs.default=MYSQL,RCluster
jcs.default.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=5000
jcs.default.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory.lru.LRUMemoryCache
jcs.default.cacheattributes.UseMemoryShrinker=true
jcs.default.cacheattributes.MaxMemoryIdleTimeSeconds=7200
jcs.default.cacheattributes.ShrinkerIntervalSeconds=60
jcs.default.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.default.elementattributes.IsEternal=false
jcs.default.elementattributes.MaxLifeSeconds=14400
jcs.default.elementattributes.IdleTime=14400
jcs.default.elementattributes.IsSpool=true
jcs.default.elementattributes.IsRemote=true
jcs.default.elementattributes.IsLateral=true

#####################################################################
################## CACHE REGIONS AVAILABLE ###################

#####################################################################
################## AUXILIARY CACHES AVAILABLE ################
# MYSQL disk cache used for flight options
jcs.auxiliary.MYSQL=org.apache.jcs.auxiliary.disk.jdbc.JDBCDiskCacheFactory
jcs.auxiliary.MYSQL.attributes=org.apache.jcs.auxiliary.disk.jdbc.JDBCDiskCacheAttributes
jcs.auxiliary.MYSQL.attributes.userName=myUsername
jcs.auxiliary.MYSQL.attributes.password=myPassword
jcs.auxiliary.MYSQL.attributes.url=${MYSQL}
jcs.auxiliary.MYSQL.attributes.driverClassName=org.gjt.mm.mysql.Driver
jcs.auxiliary.MYSQL.attributes.tableName=JCS_STORE
jcs.auxiliary.MYSQL.attributes.testBeforeInsert=false
jcs.auxiliary.MYSQL.attributes.maxActive=100
jcs.auxiliary.MYSQL.attributes.MaxPurgatorySize=10000000
jcs.auxiliary.MYSQL.attributes.UseDiskShrinker=true
jcs.auxiliary.MYSQL.attributes.ShrinkerIntervalSeconds=1800
jcs.auxiliary.MYSQL.attributes.allowRemoveAll=false
jcs.auxiliary.MYSQL.attributes.EventQueueType=POOLED
```

```
jcs.auxiliary.MYSQL.attributes.EventQueuePoolName=disk_cache_event_queue

#############################################################
################## OPTIONAL THREAD POOL CONFIGURATION #########
# Disk Cache pool
thread_pool.disk_cache_event_queue.useBoundary=true
thread_pool.disk_cache_event_queue.boundarySize=1000
thread_pool.disk_cache_event_queue.maximumPoolSize=50
thread_pool.disk_cache_event_queue.minimumPoolSize=10
thread_pool.disk_cache_event_queue.keepAliveTime=3500
thread_pool.disk_cache_event_queue.whenBlockedPolicy=RUN
thread_pool.disk_cache_event_queue.startUpSize=10
```

Table Creation Script (MySQL)

```
drop TABLE JCS_STORE;

CREATE TABLE JCS_STORE
(
  CACHE_KEY                   VARCHAR(250)          NOT NULL,
  REGION                      VARCHAR(250)          NOT NULL,
  ELEMENT                     BLOB,
  CREATE_TIME                 DATETIME,
  CREATE_TIME_SECONDS         BIGINT,
  MAX_LIFE_SECONDS            BIGINT,
  SYSTEM_EXPIRE_TIME_SECONDS  BIGINT,
  IS_ETERNAL                  CHAR(1),
  PRIMARY KEY (CACHE_KEY, REGION)
);

alter table JCS_STORE MAX_ROWS = 10000000;

alter table JCS_STORE AVG_ROW_LENGTH = 2100;

create index JCS_STORE_DELETE_IDX on JCS_STORE
(SYSTEM_EXPIRE_TIME_SECONDS,IS_ETERNAL,REGION);
```

3.2.5 JDBC Disk Properties

JDBC Disk Auxiliary Cache Configuration

The following properties apply to the JDBC Disk Cache plugin.

JDBC Disk Configuration Properties

Property	Description	Required	Default Value
MaxPurgatorySize	The maximum number of items allowed in the queue of items to be written to disk.	N	5000
url	The database url. The database name will be added to this value to create the full database url.	Y	
database	This is appended to the url.	Y	
driverClassName	The class name of the driver to talk to your database.	Y	
tableName	The name of the table.	N	JCS_STORE
testBeforeInsert	Should the disk cache do a select before trying to insert new element on update, or should it try to insert and handle the error.	N	true
maxActive	This sets the maximum number of connections allowed.	Y	
allowRemoveAll	Should the disk cache honor remove all (i.e. clear) requests. You might set this to false to prevent someone from accidentally clearing out an entire database.	N	true
UseDiskShrinker	Should the disk cache try to delete expired items from the database.	N	true
ShrinkerIntervalSeconds	How often should the disk shrinker run.	N	300

Example Configuration

```
#################################################################
################## AUXILIARY CACHES AVAILABLE ###################
# JDBC disk cache
```

```
jcs.auxiliary.JDBC=org.apache.jcs.auxiliary.disk.jdbc.JDBCDiskCacheFactory
jcs.auxiliary.JDBC.attributes=org.apache.jcs.auxiliary.disk.jdbc.JDBCDiskCacheAttributes
jcs.auxiliary.JDBC.attributes.userName=sa
jcs.auxiliary.JDBC.attributes.password=
jcs.auxiliary.JDBC.attributes.url=jdbc:hsqldb:
jcs.auxiliary.JDBC.attributes.database=target/cache_hsql_db
jcs.auxiliary.JDBC.attributes.driverClassName=org.hsqldb.jdbcDriver
jcs.auxiliary.JDBC.attributes.tableName=JCS_STORE2
jcs.auxiliary.JDBC.attributes.testBeforeInsert=false
jcs.auxiliary.JDBC.attributes.maxActive=15
jcs.auxiliary.JDBC.attributes.allowRemoveAll=true
jcs.auxiliary.JDBC.attributes.MaxPurgatorySize=10000000
jcs.auxiliary.JDBC.attributes.UseDiskShrinker=true
jcs.auxiliary.JDBC.attributes.ShrinkerIntervalSeconds=300
```

JDBC Disk Event Queue Configuration

Property	Description	Required	Default Value
EventQueueType	This should be either SINGLE or POOLED. By default the single style pool is used. The single style pool uses a single thread per event queue. That thread is killed whenever the queue is inactive for 30 seconds. Since the disk cache uses an event queue for every region, if you have many regions and they are all active, you will be using many threads. To limit the number of threads, you can configure the disk cache to use the pooled event queue. Using more threads than regions will not add any benefit for the indexed disk cache, since only one thread can read or write at a time for a single region.	N	SINGLE
EventQueuePoolName	This is the name of the pool to use. It is required if you choose the POOLED event queue type, otherwise it is ignored.	Y	n/a

Example Configuration Using Thread Pool

```
#################################################################
################## AUXILIARY CACHES AVAILABLE ###################
# JDBC disk cache
jcs.auxiliary.JDBC=org.apache.jcs.auxiliary.disk.jdbc.JDBCDiskCacheFactory
jcs.auxiliary.JDBC.attributes=org.apache.jcs.auxiliary.disk.jdbc.JDBCDiskCacheAttributes
jcs.auxiliary.JDBC.attributes.userName=sa
jcs.auxiliary.JDBC.attributes.password=
jcs.auxiliary.JDBC.attributes.url=jdbc:hsqldb:
jcs.auxiliary.JDBC.attributes.database=target/cache_hsql_db
```

```
jcs.auxiliary.JDBC.attributes.driverClassName=org.hsqldb.jdbcDriver
jcs.auxiliary.JDBC.attributes.tableName=JCS_STORE2
jcs.auxiliary.JDBC.attributes.testBeforeInsert=false
jcs.auxiliary.JDBC.attributes.maxActive=15
jcs.auxiliary.JDBC.attributes.allowRemoveAll=true
jcs.auxiliary.JDBC.attributes.MaxPurgatorySize=10000000
jcs.auxiliary.JDBC.attributes.UseDiskShrinker=true
jcs.auxiliary.JDBC.attributes.ShrinkerIntervalSeconds=300
jcs.auxiliary.JDBC.attributes.EventQueueType=POOLED
jcs.auxiliary.JDBC.attributes.EventQueuePoolName=disk_cache_event_queueue

##########################################################################
################## OPTIONAL THREAD POOL CONFIGURATION #########
# Disk Cache pool
thread_pool.disk_cache_event_queue.useBoundary=false
thread_pool.disk_cache_event_queue.boundarySize=500
thread_pool.disk_cache_event_queue.maximumPoolSize=15
thread_pool.disk_cache_event_queue.minimumPoolSize=10
thread_pool.disk_cache_event_queue.keepAliveTime=3500
thread_pool.disk_cache_event_queue.whenBlockedPolicy=RUN
thread_pool.disk_cache_event_queue.startUpSize=10
```

3.2.6 MySQL Disk Properties

MySQL Disk Auxiliary Cache Configuration

The MySQL Disk Cache uses all of the JDBC Disk Cache properties. It adds a few of its own. The following properties only apply to the MySQL Disk Cache plugin.

MySQL Disk Configuration Properties

Property	Description	Required	Default Value
optimizationSchedule	For now this is a simple comma delimited list of HH:MM:SS times to optimize the table. If none is supplied, then no optimizations will be performed. In the future we can add a cron like scheduling system. This was created to meet a pressing need to optimize fragmented MyISAM tables. When the table becomes fragmented, it starts to take a long time to run the shrinker that deletes expired elements. Setting the value to "03:01,15:00" will cause the optimizer to run at 3 am and at 3 pm.	N	null
balkDuringOptimization	If this is true, then when JCS is optimizing the table it will return null from get requests and do nothing for put requests. If you are using the remote cache and have a failover server configured in a remote cache cluster, and you allow clustered gets, the primary server will act as a proxy to the failover. This way, optimization should have no impact for clients of the remote cache.	N	true

Example Configuration

```
################################################################
################## AUXILIARY CACHES AVAILABLE ###################
# MYSQL disk cache
jcs.auxiliary.MYSQL=org.apache.jcs.auxiliary.disk.jdbc.mysql.MySQLDiskCacheFactory
jcs.auxiliary.MYSQL.attributes=org.apache.jcs.auxiliary.disk.jdbc.mysql.MySQLDiskCacheAttributes
jcs.auxiliary.MYSQL.attributes.userName=sa
jcs.auxiliary.MYSQL.attributes.password=
jcs.auxiliary.MYSQL.attributes.url=jdbc:hsqldb:target/cache_hsql_db
jcs.auxiliary.MYSQL.attributes.driverClassName=org.hsqldb.jdbcDriver
```

```
jcs.auxiliary.MYSQL.attributes.tableName=JCS_STORE_MYSQL
jcs.auxiliary.MYSQL.attributes.testBeforeInsert=false
jcs.auxiliary.MYSQL.attributes.maxActive=15
jcs.auxiliary.MYSQL.attributes.allowRemoveAll=true
jcs.auxiliary.MYSQL.attributes.MaxPurgatorySize=10000000
jcs.auxiliary.MYSQL.attributes.optimizationSchedule=12:34:56,02:34:54
jcs.auxiliary.MYSQL.attributes.balkDuringOptimization=true
```

MySQL Disk Event Queue Configuration

Property	Description	Required	Default Value
EventQueueType	This should be either SINGLE or POOLED. By default the single style pool is used. The single style pool uses a single thread per event queue. That thread is killed whenever the queue is inactive for 30 seconds. Since the disk cache uses an event queue for every region, if you have many regions and they are all active, you will be using many threads. To limit the number of threads, you can configure the disk cache to use the pooled event queue. Using more threads than regions will not add any benefit for the indexed disk cache, since only one thread can read or write at a time for a single region.	N	SINGLE
EventQueuePoolName	This is the name of the pool to use. It is required if you choose the POOLED event queue type, otherwise it is ignored.	Y	n/a

Example Configuration Using Thread Pool

```
#####################################################################
################### AUXILIARY CACHES AVAILABLE ###################
# MYSQL disk cache
jcs.auxiliary.MYSQL=org.apache.jcs.auxiliary.disk.jdbc.mysql.MySQLDiskCacheFactory
jcs.auxiliary.MYSQL.attributes=org.apache.jcs.auxiliary.disk.jdbc.mysql.MySQLDiskCacheAttributes
jcs.auxiliary.MYSQL.attributes.userName=sa
jcs.auxiliary.MYSQL.attributes.password=
jcs.auxiliary.MYSQL.attributes.url=jdbc:hsqldb:target/cache_hsql_db
jcs.auxiliary.MYSQL.attributes.driverClassName=org.hsqldb.jdbcDriver
jcs.auxiliary.MYSQL.attributes.tableName=JCS_STORE_MYSQL
jcs.auxiliary.MYSQL.attributes.testBeforeInsert=false
jcs.auxiliary.MYSQL.attributes.maxActive=15
jcs.auxiliary.MYSQL.attributes.allowRemoveAll=true
jcs.auxiliary.MYSQL.attributes.MaxPurgatorySize=10000000
jcs.auxiliary.MYSQL.attributes.optimizationSchedule=12:34:56,02:34:54
jcs.auxiliary.MYSQL.attributes.balkDuringOptimization=true
```

```
jcs.auxiliary.MYSQL.attributes.EventQueueType=POOLED
jcs.auxiliary.MYSQL.attributes.EventQueuePoolName=disk_cache_event_queue

####################################################################
################## OPTIONAL THREAD POOL CONFIGURATION #########
# Disk Cache pool
thread_pool.disk_cache_event_queue.useBoundary=false
thread_pool.disk_cache_event_queue.boundarySize=500
thread_pool.disk_cache_event_queue.maximumPoolSize=15
thread_pool.disk_cache_event_queue.minimumPoolSize=10
thread_pool.disk_cache_event_queue.keepAliveTime=3500
thread_pool.disk_cache_event_queue.whenBlockedPolicy=RUN
thread_pool.disk_cache_event_queue.startUpSize=10
```

3.2.7 Remote Cache

Remote Auxiliary Cache Client / Server

The Remote Auxiliary Cache is an optional plug in for JCS. It is intended for use in multi-tiered systems to maintain cache consistency. It uses a highly reliable RMI client server framework that currently allows for any number of clients. Using a listener id allows multiple clients running on the same machine to connect to the remote cache server. All cache regions on one client share a listener per auxiliary, but register separately. This minimizes the number of connections necessary and still avoids unnecessary updates for regions that are not configured to use the remote cache.

Local remote cache clients connect to the remote cache on a configurable port and register a listener to receive cache update callbacks at a configurable port.

If there is an error connecting to the remote server or if an error occurs in transmission, the client will retry for a configurable number of tries before moving into a failover-recovery mode. If failover servers are configured the remote cache clients will try to register with other failover servers in a sequential order. If a connection is made, the client will broadcast all relevant cache updates to the failover server while trying periodically to reconnect with the primary server. If there are no failovers configured the client will move into a zombie mode while it tries to re-establish the connection. By default, the cache clients run in an optimistic mode and the failure of the communication channel is detected by an attempted update to the server. A pessimistic mode is configurable so that the clients will engage in active status checks.

The remote cache server broadcasts updates to listeners other than the originating source. If the remote cache fails to propagate an update to a client, it will retry for a configurable number of tries before de-registering the client.

The cache hub communicates with a facade that implements a zombie pattern (balking facade) to prevent blocking. Puts and removals are queued and occur asynchronously in the background. Get requests are synchronous and can potentially block if there is a communication problem.

By default client updates are light weight. The client listeners are configured to remove elements form the local cache when there is a put order from the remote. This allows the client memory store to control the memory size algorithm from local usage, rather than having the usage patterns dictated by the usage patterns in the system at large.

When using a remote cache the local cache hub will propagate elements in regions configured for the remote cache if the element attributes specify that the item to be cached can be sent remotely. By default there are no remote restrictions on elements and the region will dictate the behavior. The order of auxiliary requests is dictated by the order in the configuration file. The examples are configured to look in memory, then disk, then remote caches. Most elements will only be retrieved from the remote cache once, when they are not in memory or disk and are first requested, or after they have been invalidated.

Client Configuration

The configuration is fairly straightforward and is done in the auxiliary cache section of the `cache.ccf` configuration file. In the example below, I created a Remote Auxiliary Cache Client referenced by `RFailover` .

This auxiliary cache will use `localhost:1102` as its primary remote cache server and will attempt to failover to `localhost:1103` if the primary is down.

Setting `RemoveUponRemotePut` to `false` would cause remote puts to be translated into put requests to the client region. By default it is `true` , causing remote put requests to be issued as removes at the client level. For groups the put request functions slightly differently: the item will be removed, since it is no longer valid in its current form, but the list of group elements will be updated. This way the client can maintain the complete list of group elements without the burden of storing all of the referenced elements. Session distribution works in this half-lazy replication mode.

Setting `GetOnly` to `true` would cause the remote cache client to stop propagating updates to the remote server, while continuing to get items from the remote store.

```
# Remote RMI Cache set up to failover
jcs.auxiliary.RFailover=
    org.apache.jcs.auxiliary.remote.RemoteCacheFactory
jcs.auxiliary.RFailover.attributes=
    org.apache.jcs.auxiliary.remote.RemoteCacheAttributes
jcs.auxiliary.RFailover.attributes.FailoverServers=
    localhost:1102,localhost:1103
jcs.auxiliary.RC.attributes.RemoveUponRemotePut=true
jcs.auxiliary.RFailover.attributes.GetOnly=false
```

This cache region is setup to use a disk cache and the remote cache configured above:

```
#Regions preconfirgured for caching
jcs.region.testCache1=DC,RFailover
jcs.region.testCache1.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.testCache1.cacheattributes.MaxObjects=1000
jcs.region.testCache1.cacheattributes.MemoryCacheName=
    org.apache.jcs.engine.memory.lru.LRUMemoryCache
```

Server Configuration

The remote cache configuration is growing. For now, the configuration is done at the top of the `remote.cache.ccf` file. The `startRemoteCache` script passes the configuration file name to the

server when it starts up. The configuration parameters below will create a remote cache server that listens to port `1102` and performs call backs on the `remote.cache.service.port` , also specified as port `1102` .

```
# Registry used to register and provide the
# IRemoteCacheService service.
registry.host=localhost
registry.port=1102
# call back port to local caches.
remote.cache.service.port=1102
# cluster setting
remote.cluster.LocalClusterConsistency=true
remote.cluster.AllowClusterGet=true
```

Remote servers can be chained (or clustered). This allows gets from local caches to be distributed between multiple remote servers. Since gets are the most common operation for caches, remote server chaining can help scale a caching solution.

The `LocalClusterConsistency` setting tells the remote cache server if it should broadcast updates received from other cluster servers to registered local caches.

The `AllowClusterGet` setting tells the remote cache server whether it should allow the cache to look in non-local auxiliaries for items if they are not present. Basically, if the get request is not from a cluster server, the cache will treat it as if it originated locally. If the get request originated from a cluster client, then the get will be restricted to local (i.e. memory and disk) auxiliaries. Hence, cluster gets can only go one server deep. They cannot be chained. By default this setting is true.

To use remote server clustering, the remote cache will have to be told what regions to cluster. The configuration below will cluster all non-preconfigured regions with `RCluster1` .

```
# sets the default aux value for any non configured caches
jcs.default=DC,RCluster1
jcs.default.cacheattributes=
    org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000

jcs.auxiliary.RCluster1=
    org.apache.jcs.auxiliary.remote.RemoteCacheFactory
jcs.auxiliary.RCluster1.attributes=
    org.apache.jcs.auxiliary.remote.RemoteCacheAttributes
jcs.auxiliary.RCluster1.attributes.RemoteTypeName=CLUSTER
jcs.auxiliary.RCluster1.attributes.RemoveUponRemotePut=false
jcs.auxiliary.RCluster1.attributes.ClusterServers=localhost:1103
jcs.auxiliary.RCluster1.attributes.GetOnly=false
```

RCluster1 is configured to talk to a remote server at `localhost:1103` . Additional servers can be

added in a comma separated list.

If we startup another remote server listening to port 1103, (ServerB) then we can have that server talk to the server we have been configuring, listening at 1102 (ServerA). This would allow us to set some local caches to talk to ServerA and some to talk to ServerB. The two remote servers will broadcast all puts and removes between themselves, and the get requests from local caches could be divided. The local caches do not need to know anything about the server chaining configuration, unless you want to use a standby, or failover server.

We could also use ServerB as a hot standby. This can be done in two ways. You could have all local caches point to ServerA as a primary and ServerB as a secondary. Alternatively, you can set ServerA as the primary for some local caches and ServerB for the primary for some others.

The local cache configuration below uses ServerA as a primary and ServerB as a backup. More than one backup can be defined, but only one will be used at a time. If the cache is connected to any server except the primary, it will try to restore the primary connection indefinitely, at 20 second intervals.

```
# Remote RMI Cache set up to failover
jcs.auxiliary.RFailover=
    org.apache.jcs.auxiliary.remote.RemoteCacheFactory
jcs.auxiliary.RFailover.attributes=
    org.apache.jcs.auxiliary.remote.RemoteCacheAttributes
jcs.auxiliary.RFailover.attributes.FailoverServers=
    localhost:1102,localhost:1103
jcs.auxiliary.RC.attributes.RemoveUponRemotePut=true
jcs.auxiliary.RFailover.attributes.GetOnly=false
```

Server Startup / Shutdown

It is highly recommended that you embed the Remote Cache Server in a Servlet container such as Tomcat. Running inside Tomcat allows you to use the JCSAdmin.jsp page. It also takes care of the complexity of creating working startup and shutdown scripts.

JCS provides a convenient startup servlet for this purpose. It will start the registry and bind the JCS server to the registry. To use the startup servlet, add the following to the web.xml file and make sure you have the remote.cache.ccf file in the WEB-INF/classes directly of your war file.

```
<servlet>
    <servlet-name>JCSRemoteCacheStartupServlet</servlet-name>
    <servlet-class>
         org.apache.jcs.auxiliary.remote.server.RemoteCacheStartupServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>


<servlet-mapping>
```

```
        <servlet-name>JCSRemoteCacheStartupServlet</servlet-name>
        <url-pattern>/jcs</url-pattern>
    </servlet-mapping>
```

3.2.8 Remote Cache Properties

Remote Auxiliary Cache Configuration

The following properties apply to the Remote Cache plugin.

Remote Client Configuration Properties

Property	Description	Required	Default Value
FailoverServers	This is a comma separated list of remote servers to use. They should be specified in the host:port format. The first server in the list will be used as the primary server. If the connection is lost with the primary, the cache will try to connect to the next server in the list. If a connection is successfully established with a failover server, then the cache will attempt to restore the conenction with the primary server.	Y	n/a
LocalPort	This is the port on which the client will receive callbacks from the remote server. If it is not specified, then some port in the default range used by RMI will be the callback port.	N	default RMI port range
RemoveUponRemotePut	If you configure the cache to remove upon a remote put, this means that the client will translate updates into removes. The client will remove any local copy it has of the object rather than storing the new version. If you have sticky load balancing across your client servers, then it would make sense to set RemoveUponRemotePut to true if the data is mostly client specific. If the data is re-usable, the you should most likely set this option to false, which is the default.	N	false
RmiSocketFactoryTimeoutMillis	If this is greater than 0, then a custom socket factory will be installed in the VM. It will then use this timeout for all RMI communication.	N	5000
GetOnly	GetOnly is somewhat misnamed. If it is set to true, then the client will not send updates or removes to the remote server. It can still receive updates and removes.	N	false

Property	Description	Required	Default Value
Receive	By default Receive is set to true. This means that the remote client will receive updates and removes from the remote server. If you set Receive to false, the remote client will not register a listener with the remote server. This means that the client can send update and remove requests to the server, and it can get from the server, but it will never receive notifications from the server. You might configure Receive to false if you just want to use the remote server as a data store. For instance, you may back the Remote Cache Server with the JDBC disk cache and set Receive=false when you have a high put and low read region.	N	true
ZombieQueueMaxSize	The number of elements the zombie queue will hold. This queue is used to store events if we lose our connection with the server.	N	1000

Example Configuration

```
# This remote client does not receive
jcs.auxiliary.RC=org.apache.jcs.auxiliary.remote.RemoteCacheFactory
jcs.auxiliary.RC.attributes=org.apache.jcs.auxiliary.remote.RemoteCacheAttributes
jcs.auxiliary.RC.attributes.FailoverServers=localhost:1101,localhost:1102
jcs.auxiliary.RC.attributes.LocalPort=1201
jcs.auxiliary.RC.attributes.RemoveUponRemotePut=false
jcs.auxiliary.RC.attributes.RmiSocketFactoryTimeoutMillis=5000
jcs.auxiliary.RC.attributes.GetOnly=false
jcs.auxiliary.RC.attributes.Receive=false
```

3.2.9 Lateral TCP Cache

Lateral TCP Auxiliary Cache

The TCP Lateral Auxiliary Cache is an optional plug in for the JCS. It is primarily intended to broadcast puts and removals to other local caches, though it can also get cached objects. It functions by opening up a `SocketServer` that listens to a configurable port and by creating `Socket` connections with other local cache `SocketServers`. It can be configured to connect to any number of servers.

If there is an error connecting to another server or if an error occurs in transmission, it will move into a recovery mode. In recovery mode the TCP Lateral Auxiliary Cache will continue to communicate with healthy servers while it tries to restore the connection with the server that is in error.

The cache hub communicates with a facade that implements a zombie pattern (balking facade) to prevent blocking. Puts and removals are queued and occur synchronously in the background. Get requests are synchronous and can potentially block for a configurable interval if there is a communication problem.

Non-UDP Discovery Configuration

The configuration is fairly straightforward and is done in the auxiliary cache section of the `cache.ccf` configuration file. In the example below, I created a TCP Lateral Auxiliary Cache referenced by `LTCP`. It connects to two servers defined in a comma separated list in the `TcpServers` attribute. It listens to port `1110` and does `AllowGet`. Setting `AllowGet` equal to `false` would cause the auxiliary cache to return `null` from any get request. In most cases this attribute should be set to `false`, since if the lateral caches were properly configured, the elements in one would be present in all.

```
jcs.auxiliary.LTCP=org.apache.jcs.auxiliary.lateral.socket.tcp.LateralTCPCacheFactory
jcs.auxiliary.LTCP.attributes=org.apache.jcs.auxiliary.lateral.socket.tcp.TCPLateralCacheAttributes
jcs.auxiliary.LTCP.attributes.TcpServers=localhost:1111,localhost:1112
jcs.auxiliary.LTCP.attributes.TcpListenerPort=1110
jcs.auxiliary.LTCP.attributes.AllowGet=true
```

A mostly configurationless mode is available for the TCP lateral cache if you use the UDP Discovery mechanism.

Send Only Configuration

You can configure the TCP lateral cache to operate in send only mode by setting the `Receive` attribute to false. By default the receive attribute is true. When it is set to false, the lateral cache will not establish a socket server.

Setting receive to false allows you to broadcast puts and removes, but not receive any. This is useful for nodes of an application that produce data, but are not involved in data retrieval.

The configuration below is the same as above, except the `Receive` attribute is set to false. It also uses UDP discovery to find the servers, rather than listing them in the servers attribute.

```
jcs.auxiliary.LTCP=org.apache.jcs.auxiliary.lateral.socket.tcp.LateralTCPCacheFactory
jcs.auxiliary.LTCP.attributes=org.apache.jcs.auxiliary.lateral.socket.tcp.TCPLateralCacheAttributes
#jcs.auxiliary.LTCP.attributes.TcpServers=
jcs.auxiliary.LTCP.attributes.TcpListenerPort=1118
jcs.auxiliary.LTCP.attributes.UdpDiscoveryAddr=228.5.6.8
jcs.auxiliary.LTCP.attributes.UdpDiscoveryPort=6780
jcs.auxiliary.LTCP.attributes.UdpDiscoveryEnabled=true
jcs.auxiliary.LTCP.attributes.Receive=true
jcs.auxiliary.LTCP.attributes.AllowGet=false
jcs.auxiliary.LTCP.attributes.IssueRemoveOnPut=false
jcs.auxiliary.LTCP.attributes.FilterRemoveByHashCode=false
```

Potential Issues

The TCP Lateral Auxiliary Cache can provide a high level of consistency but it does not guarantee consistency between caches. A put for the same object could be issued in two different local caches. Since the transmission is queued, a situation could occur where the item put last in one cache is overridden by a put request from another local cache. The two local caches could potentially have different versions of the same item. Like most caches, this is intended for high get and low put utilization, and this occurrence would hint at improper usage. The RMI Remote cache makes this situation a bit less likely to occur, since the default behavior is to remove local copies on put operations. If either local cache needed the item put in the above situation, it would have to go remote to retrieve it. Both local copies would have been expired and would end up using the same version, though it is possible that the version stored remotely would not be the last version created. The OCS4J tries to implement a locking system to prevent this from occurring, but the locking system itself could suffer from similar problems (when granting locks from two roughly simultaneous lock requests) and it would create a significant burden on all the caches involved. Since this situation would be extremely rare and is nearly impossible to solve practically, for now JCS will not offer any type of locking.

Recent

I added a `IssueRemoveOnPut` attribute that causes the lateral cache to remove an element from the cache rather than inserting it when a put. This allows the local caches to dictate their own memory usage pattern.

3.2.10 Lateral TCP Properties

Lateral TCP Auxiliary Cache Configuration

The following properties apply to the TCP Lateral Cache plugin.

TCP Configuration Properties

Property	Description	Required	Default Value
TcpServers	This is the list of servers this cache should try to connect to. With UDP discovery this is not necessary.	N	none
TcpListenerPort	This is the port this cache should listen on.	Y	n/a
AllowGet	Should this cache be allowed to get from other laterals. False means that it can only put, i.e. send updates and remove requests to other laterals. Lateral gets are not recommended for performance reasons. This used to be controlled by the attribute PutOnlyMode.	N	true
Receive	Should this cache receive or only send to other laterals. You may want to set receive to false if you just need to broadcast to other caches. If you have a feed data parser, that doesn't need to receive updates, but you do want it to send invalidation messages, then you would set receive to false. If receive is false, the discovery service, if enabled, will only listen.	N	true
IssueRemoveOnPut	If this is set to true, then the lateral client will send a remove command rather than a put command to any registered listeners.	N	false
FilterRemoveByHashCode	If this is true, and IssueRemoveOnPut is true, the client will include the hashCode of the element to remove. If it is also true on the receiving end, the receiver will check to see if the element exists. If the element exists, and the hashCodes are the same, the item will not be removed.	N	false

Property	Description	Required	Default Value
UdpDiscoveryAddr	The address the UDP discovery process should broadcast messages to.	N	228.5.6.7
UdpDiscoveryPort	The port the UDP discovery process should send messages to.	N	6789
UdpDiscoveryEnabled	Whether or not the UDP discovery service should be used to locate other lateral caches.	N	true

Example Configuration

```
jcs.auxiliary.LTCP=org.apache.jcs.auxiliary.lateral.socket.tcp.LateralTCPCacheFactory
jcs.auxiliary.LTCP.attributes=org.apache.jcs.auxiliary.lateral.socket.tcp.TCPLateralCacheAttributes
#jcs.auxiliary.LTCP.attributes.TcpServers=
jcs.auxiliary.LTCP.attributes.TcpListenerPort=1118
jcs.auxiliary.LTCP.attributes.UdpDiscoveryAddr=228.5.6.8
jcs.auxiliary.LTCP.attributes.UdpDiscoveryPort=6780
jcs.auxiliary.LTCP.attributes.UdpDiscoveryEnabled=true
jcs.auxiliary.LTCP.attributes.Receive=true
jcs.auxiliary.LTCP.attributes.AllowGet=false
jcs.auxiliary.LTCP.attributes.IssueRemoveOnPut=false
jcs.auxiliary.LTCP.attributes.FilterRemoveByHashCode=false
```

3.2.11 Lateral UDP Discovery

Lateral UDP Discovery

Rather than list all the other lateral servers in the configuration file, you can configure the TCP lateral to use UDP discovery. In discovery mode, lateral TCP caches will broadcast to a multicast address and port, letting all listeners know where they are.

On startup each lateral will issue a special message requesting a broadcast from the other caches. Normal broadcasts occur every 30 seconds. (This is to be made configurable.) Regions that don't receive, are running in send only mode, don't broadcast anything but requests.

When a lateral receives a discovery message it will try to add the lateral to the nowait facade for the region. If it already exists nothing happens. If a region is not configured to send laterally, nothing happens, since it doesn't have a no wait.

This allows you to have the same configuration on every machine.

Configuration

The configuration is fairly straightforward and is done in the auxiliary cache section of the `cache.ccf` configuration file. In the example below, I created a TCP Lateral Auxiliary Cache referenced by `LTCP`. It uses UDP Discovery to locate other servers. It broadcasts to multicast address `228.5.6.8` and port `6780`. It listens to port `1110`.

```
jcs.auxiliary.LTCP=org.apache.jcs.auxiliary.lateral.socket.tcp.LateralTCPCacheFactory
jcs.auxiliary.LTCP.attributes=org.apache.jcs.auxiliary.lateral.socket.tcp.TCPLateralCacheAttributes
jcs.auxiliary.LTCP.attributes.TcpListenerPort=1110
jcs.auxiliary.LTCP.attributes.PutOnlyMode=true
jcs.auxiliary.LTCP.attributes.UdpDiscoveryAddr=228.5.6.8
jcs.auxiliary.LTCP.attributes.UdpDiscoveryPort=6780
jcs.auxiliary.LTCP.attributes.UdpDiscoveryEnabled=true
```

3.2.12 Lateral JGroups Cache

Lateral JGroups Auxiliary Cache

The Lateral JGroups Auxiliary Cache is an optional plug in for JCS. It is primarily intended to broadcast puts and removals to other local caches, though it can also get cached objects. It uses JGroups for distribution.

The Lateral Lateral JGroups Auxiliary Cache is far slower than that Lateral TCP Auxiliary Cache. Since the Lateral TCP Auxiliary is faster and has UDP discovery built in, the TCP auxiliary is the recommended form of lateral distribution. However, the JGroups Auxiliary requires fewer socket connections than the TCP lateral.

A functional configuration example is below:

```
# Lateral JavaGroups Distribution
jcs.auxiliary.LJG=org.apache.jcs.auxiliary.lateral.LateralCacheFactory
jcs.auxiliary.LJG.attributes=org.apache.jcs.auxiliary.lateral.LateralCacheAttributes
jcs.auxiliary.LJG.attributes.TransmissionTypeName=JAVAGROUPS
jcs.auxiliary.LJG.attributes.PutOnlyMode=true
jcs.auxiliary.LJG.attributes.JGChannelProperties=UDP(mcast_addr=224.0.0.100;mcast_port=7501):PING:FD:STAB
```