Ehcache Cache Server User Guide

Version 2.9

October 2014

EHCACHE

Document ID: EHC-CSUG-29-20141015

Table of Contents

1 About Cache Server

What is Cache Server?

Ehcache comes with a Cache Server, available as a WAR for most web containers, or as a standalone server.

Note: If using with Terracotta BigMemory, the Cache Server works only with versions prior to 4.0. The Cache Server does not work with BigMemory 4.0 and higher.

The Cache Server has two APIs: RESTful resource oriented and SOAP. Both support clients in any programming language. (On terminology: Leonard Richardson and Sam Ruby have done a great job of clarifying the different Web Services architectures and distinguishing them from each other. We use their taxonomy in describing web services. See the O'Reilly catalog.)

2 Installing the Cache Server Module

Requirements

Java

Java 5 or 6.

Web Container (WAR packaged version only)

The standalone server comes with its own embedded Glassfish web container. The web container must support the Servlet 2.5 specification. The following web container configurations have been tested:

- Glassfish V2/V3
- Tomcat 6
- Jetty 6

Downloading

The server is available as follows:

SourceForge

Download from http://sourceforge.net/project/showfiles.php?group_id=93232. There are two tarball archives in tar.gz format:

- ehcache-server - this contains the WAR file which must be deployed in your own web container.
- ehcache-standalone-server - this contains a complete standalone directory structure with an embedded Glassfish V3 web container together with shell scripts for starting and stopping.

Maven

The Ehcache Server is in the central Maven repository packaged as type *war*. Use the following Maven pom snippet:

```
<dependency>
 <groupId>net.sf.ehcache</groupId>
 <artifactId>ehcache-server</artifactId>
 <version>enter_version_here</version>
 <type>war</type>
</dependency>
```

It is also available as a jar-only version, which makes it easier to embed. This version excludes all META-INF and WEB-INF configuration files, and also excludes the ehcache.xml. You need to provide these in your maven project.

```
<dependency>
   <groupId>net.sf.ehcache</groupId>
   <artifactId>ehcache-server</artifactId>
   <version>enter_version_here</version>
   <type>jar</type>
   <classifier>jaronly</classifier>
  </dependency>
```

Installing the WAR File

Use your Web container's instructions to install the WAR or include the WAR in your project with Maven's war plugin. Container-specific configuration is provided in the WAR as follows:

- sun-web.xml - Glassfish V2/V3 configuration
- jetty-web.xml - Jetty V5/V6 configuration

Tomcat V6 passes all integration tests. It does not require a specific configuration.

Configuring the Web Application

Expand the WAR. Edit the web.xml.

Disabling the RESTful Web Service

Comment out the RESTful web service section.

Disabling the SOAP Web Service

Comment out the RESTful web service section.

Configuring Caches

The ehcache.xml configuration file is located in `WEB-INF/classes/ehcache.xml`. Follow the instructions in this config file, or the core Ehcache instructions to configure.

Using the Cache Server with WebLogic

We have tested with 10.3.2, but the SOAP libraries are not compatible. Either comment out the SOAP service from web.xml or do the following:

1. Unzip `ehcache-server.war` to a folder called `ehcache`.
2. Remove the following jars from `WEB-INF/lib`:
   - jaxws-rt-2.1.4.jar
   - metro-webservices-api-1.2.jar
   - metro-webservices-rt-1.2.jar
   - metro-webservices-tools-1.2.jar
3. Deploy the folder to WebLogic.

4. Use the SoapUI GUI in WebLogic to add a project from http://<hostname>:<port>/ehcache/soap/EhcacheWebServiceEndpoint?wsdl

Installing the Standalone Server

The WAR also comes packaged with a standalone server, based on Glassfish V3 Embedded. The quick start is:

- Untar the download.
- `bin/start.sh` to start. By default it will listen on port 8080, with JMX listening on port 8081, will have both RESTful and SOAP web services enabled, and will use a sample Ehcache configuration from the WAR module.
- `bin/stop.sh` to stop.

Configuring the Standalone Server

Configuration is by editing the `war/web.xml` file as per the instructions for the WAR packaging.

Starting and Stopping the Standalone Server

The following describes ways to start and stop the server.

Using Commons Daemon jsvc

jsvc creates a daemon which returns once the service is started. jsvc works on all common Unix-based operating systems including Linux, Solaris and Mac OS X. It creates a pid file in the pid directory. This is a Unix shell script that starts the server as a daemon. To use jsvc you must install the native binary jsvc from the Apache Commons Daemon project. The source for this is distributed in the bin directory as `jsvc.tar.gz`. Untar it and follow the instructions for building it or download a binary from the Commons Daemon project. Convenience shell scripts are provided as follows:

- start - `daemon_start.sh`
- stop - `daemon_stop.sh`

jsvc is designed to integrate with Unix System 5 initialization scripts (/etc/rc.d). You can also send Unix signals to it. Meaningful ones for the Ehcache Standalone Server are:

#	Meaning	Effect
1	HUP	Restarts the server.
2	INT	Interrupts the server.

#	Meaning	Effect
9	KILL	The process is killed. The server is not given a chance to shutdown.
15	TERM	Stops the server, giving it a chance to shutdown in an orderly way.

Executable jar

The server is also packaged as an executable jar for developer convenience which will work on all operating systems. A convenience shell script is provided as follows:

- start - startup.sh

From the bin directory you can also invoke the following command directly:

```
unix    - java -jar ../lib/ehcache-standalone-server-0.7.jar 8080 ../war
windows - java -jar ..\lib\ehcache-standalone-server-0.7.jar 8080 ..\war
```

3 Monitoring the Cache Server

About Monitoring the Cache Server

The Cache Server registers Ehcache MBeans with the platform MBeanServer. Remote monitoring of the MBeanServer is the responsibility of the Web container or application server vendor. For example, some instructions for Tomcat are here. See your product documentation for how to do this for your web container.

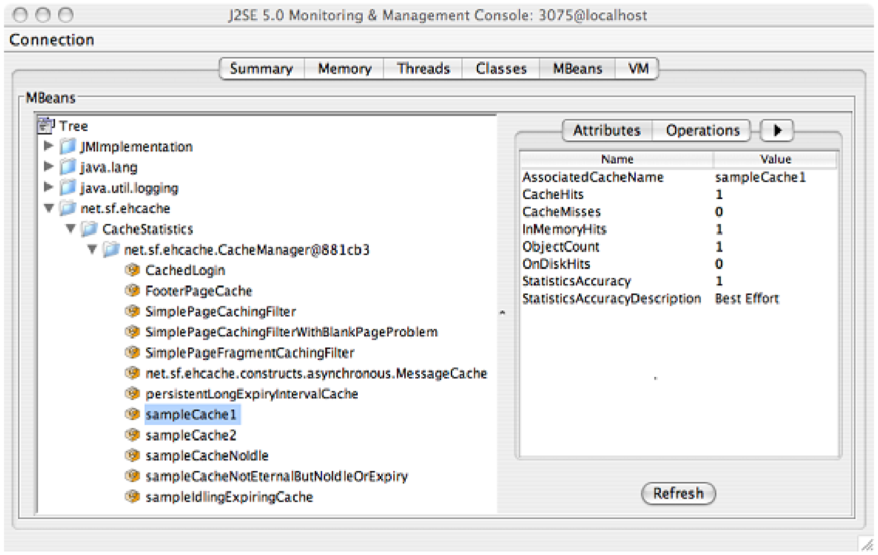
Remotely Monitoring the Standalone Server with JMX

The standalone server automatically exposes the MBeanServer on a port 1 higher than the HTTP listening port.

To connect with JConsole, simply fire up JConsole, enter the host in the **Remote** field and port. In the above example that is `192.168.1.108:8686`.

Then click **Connect**. To see the Ehcache MBeans, click on the **Mbeans** tab and expand the net.sf.ehcache tree node. You will see something like the following.

CacheStatistics MBeans in JConsole



- Tree
  - JMImplementation
  - java.lang
  - java.util.logging
  - net.sf.ehcache
    - CacheStatistics
      - net.sf.ehcache.CacheManager@881cb3
        - CachedLogin
        - FooterPageCache
        - SimplePageCachingFilter
        - SimplePageCachingFilterWithBlankPageProblem
        - SimplePageFragmentCachingFilter
        - net.sf.ehcache.constructs.asynchronous.MessageCache
        - persistentLongExpiryIntervalCache
        - sampleCache1
        - sampleCache2
        - sampleCacheNoIdle
        - sampleCacheNotEternalButNoIdleOrExpiry
        - sampleIdlingExpiringCache

Name	Value
AssociatedCacheName	sampleCache1
CacheHits	1
CacheMisses	0
InMemoryHits	1
ObjectCount	1
OnDiskHits	0
StatisticsAccuracy	1
StatisticsAccuracyDescription	Best Effort

Of course, from there you can hook the Cache Server up to your monitoring tool of choice. For more information, see “JMX Management and Monitoring” in the *Ehcache Operations Guide*.

4 Using the RESTful Services

About RESTful Web Services

Roy Fielding coined the acronym REST, denoting Representational State Transfer, in his PhD thesis. The Ehcache implementation strictly follows the RESTful resource-oriented architecture style. Specifically:

- The HTTP methods GET, HEAD, PUT/POST and DELETE are used to specify the method of the operation. The URI does not contain method information.
- The scoping information, used to identify the resource to perform the method on, is contained in the URI path.
- The RESTful Web Service is described by and exposes a Web Application Description Language (WADL) file. It contains the URIs you can call, and what data to pass and get back. Use the OPTIONS method to return the WADL.

Roy is on the JSR311 expert group. JSR311 and Jersey, the reference implementation, are used to deliver RESTful web services in Ehcache server.

The RESTful Web Services API

The Ehcache RESTful Web Services API exposes the singleton CacheManager, which typically has been configured in ehcache.xml or an Inversion of Control (IoC) container. Multiple CacheManagers are not supported. Resources are identified using a URI template. The value in parentheses should be substituted with a literal to specify a resource. Response codes and response headers strictly follow HTTP conventions.

CacheManager Resource Operations

OPTIONS /{cache}}

Retrieves the WADL for describing the available CacheManager operations.

GET} /

Lists the Caches in the CacheManager.

Cache Resource Operations

OPTIONS /{cache}}

Retrieves the WADL describing the available Cache operations.

HEAD /{cache}}

Retrieves the same metadata a GET would receive returned as HTTP headers. There is no body returned.

GET /{cache}

Gets a cache representation. This includes useful metadata such as the configuration and cache statistics.

{PUT} /{cache}

Creates a Cache using the default Cache configuration.

{DELETE} / {cache}

Deletes the Cache.

Element Resource Operations

OPTIONS /{cache}}

Retrieves the WADL describing the available Element operations.

HEAD /{cache}/{element}

Retrieves the same metadata a GET would receive returned as HTTP headers. There is no body returned.

GET /{cache}/{element}

Gets the element value.

HEAD /{cache}/{element}

Gets the element's metadata.

PUT /{cache}/{element}

Puts an element into the Cache. The time to live of new Elements defaults to that for the cache. This may be overridden by setting the HTTP request header `ehcacheTimeToLiveSeconds`. Values of 0 to 2147483647 are accepted. A value of 0 means eternal.

DELETE / {cache}/{element}

Deletes the element from the cache. The resource representation for all elements is `*`. `DELETE/\{cache\}/\*` will call `cache.removeAll()`.

Resource Representations

We deal with resource representations rather than resources themselves.

Element Resource Representations

When Elements are PUT into the cache, a MIME Type should be set in the request header. The MIME Type is preserved for later use. The new `MimeTypeByteArray` is used to store the `byte[]` and the `MimeType` in the value field of `Element`. Some common MIME Types which are expected to be used by clients are:

text/plain	Plain text
text/xml	Extensible Markup Language. Defined in RFC 3023
application/json	JavaScript Object Notation JSON. Defined in RFC 4627
application/x-java-serialized-object	A serialized Java object

Because Ehcache is a distributed Java cache, in some configurations the Cache server may contain Java objects that arrived at the Cache server via distributed replication. In this case no MIME Type will be set and the Element will be examined to determine its MIME Type. Because anything that can be PUT into the cache server must be Serializable, it can also be distributed in a cache cluster i.e. it will be Serializable.

RESTful Code Samples

These are RESTful code samples in multiple languages.

Curl Code Samples

These samples use the popular curl command line utility.

OPTIONS

This example shows how calling OPTIONS causes Ehcache server to respond with the WADL for that resource

```
curl --request OPTIONS http://localhost:8080/ehcache/rest/sampleCache2/2
```

The server responds with:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://research.sun.com/wadl/2006/10">
  <resources base="http://localhost:8080/ehcache/rest/">
```

```
    <resource path="sampleCache2/2">
      <method name="HEAD"><response><representation mediaType="
      ...
    </resource>
  </resources>
</application>
```

HEAD

```
curl --head  http://localhost:8080/ehcache/rest/sampleCache2/2
```

The server responds with:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: GlassFish/v3
Last-Modified: Sun, 27 Jul 2008 08:08:49 GMT
ETag: "1217146129490"
Content-Type: text/plain; charset=iso-8859-1
Content-Length: 157
Date: Sun, 27 Jul 2008 08:17:09 GMT
```

PUT

```
echo "Hello World" |  curl -S -T -  http://localhost:8080/ehcache/rest/sampleCache2/3
```

The server will put `Hello World` into `sampleCache2` using key `3`.

GET

```
curl http://localhost:8080/ehcache/rest/sampleCache2/2
```

The server responds with:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
```

Ruby Code Samples

GET

```
require 'rubygems'
require 'open-uri'
require 'rexml/document'
response = open('http://localhost:8080/ehcache/rest/sampleCache2/2')
xml = response.read
puts xml
```

The server responds with:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
```

```
</oldjoke>
```

Python Code Samples

GET

```
import urllib2
f = urllib2.urlopen('http://localhost:8080/ehcache/rest/sampleCache2/2')
print f.read()
```

The server responds with:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
</oldjoke>
```

Java Code Samples

Create and get a Cache and Entry

```
mport java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;
/**
* A simple example Java client which uses the built-in java.net.URLConnection.
*
* @author BryantR * @author Greg Luck
*/
public class ExampleJavaClient {
private static String TABLE_COLUMN_BASE =
       "http://localhost:8080/ehcache/rest/tableColumn";
private static String TABLE_COLUMN_ELEMENT =
       "http://localhost:8080/ehcache/rest/tableColumn/1";
/**
* Creates a new instance of EHCacheREST
*/
public ExampleJavaClient() {
}
public static void main(String[] args) {
   URL url;
   HttpURLConnection connection = null;
   InputStream is = null;
   OutputStream os = null;
   int result = 0;
   try {
       //create cache
       URL u = new URL(TABLE_COLUMN_BASE);
       HttpURLConnection urlConnection =
              (HttpURLConnection) u.openConnection();
       urlConnection.setRequestMethod("PUT");
       int status = urlConnection.getResponseCode();
       System.out.println("Status: " + status);
       urlConnection.disconnect();
        //get cache
       url = new URL(TABLE_COLUMN_BASE);
       connection = (HttpURLConnection) url.openConnection();
       connection.setRequestMethod("GET");
       connection.connect();
       is = connection.getInputStream();
       byte[] response1 = new byte[4096];
       result = is.read(response1);
       while (result != -1) {
           System.out.write(response1, 0, result);
           result = is.read(response1);
       }
       if (is != null) try {
           is.close();
       } catch (Exception ignore) {
       }        System.out.println("reading cache: " +
                   connection.getResponseCode()
               + " " + connection.getResponseMessage());
       if (connection != null) connection.disconnect();
        //create entry
       url = new URL(TABLE_COLUMN_ELEMENT);
       connection = (HttpURLConnection) url.openConnection();
       connection.setRequestProperty("Content-Type", "text/plain");
       connection.setDoOutput(true);
       connection.setRequestMethod("PUT");
       connection.connect();
       String sampleData = "Ehcache is way cool!!!";
       byte[] sampleBytes = sampleData.getBytes();
       os = connection.getOutputStream();
       os.write(sampleBytes, 0, sampleBytes.length);
       os.flush();
       System.out.println("result=" + result);
       System.out.println("creating entry: " + connection.getResponseCode()
               + " " + connection.getResponseMessage());
       if (connection != null) connection.disconnect();
        //get entry
       url = new URL(TABLE_COLUMN_ELEMENT);
       connection = (HttpURLConnection) url.openConnection();
       connection.setRequestMethod("GET");
       connection.connect();
       is = connection.getInputStream();
       byte[] response2 = new byte[4096];
       result = is.read(response2);
       while (result != -1) {
           System.out.write(response2, 0, result);
           result = is.read(response2);
       }
       if (is != null) try {
           is.close();
       } catch (Exception ignore) {
       }        System.out.println("reading entry: "
               + connection.getResponseCode()
               + " " + connection.getResponseMessage());
       if (connection != null) connection.disconnect();
   } catch (Exception e) {
       e.printStackTrace();
   } finally {
       if (os != null) try {
           os.close();
       } catch (Exception ignore) {
       }
       if (is != null) try {
           is.close();
       } catch (Exception ignore) {
       }
       if (connection != null) connection.disconnect();
   }
}
}
```

Scala Code Samples

GET

```
import java.net.URL
  import scala.io.Source.fromInputStream
object ExampleScalaGet extends Application {
val url = new URL("http://localhost:8080/ehcache/rest/sampleCache2/2")
fromInputStream(url.openStream).getLines.foreach(print)
  }
```

Run it with:

```
scala -e ExampleScalaGet
```

The program outputs:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
```

PHP Code Samples

GET

```
  ?<php
$ch = curl_init();
curl_setopt ($ch, CURLOPT_URL, "http://localhost:8080/ehcache/rest/sampleCache2/3");
  curl_setopt ($ch, CURLOPT_HEADER, 0);
curl_exec ($ch);
curl_close ($ch);
  ?>
```

The server responds with:

```
Hello Ingo
```

PUT

```
  ?<php
$url = "http://localhost:8080/ehcache/rest/sampleCache2/3";
$localfile = "localfile.txt";
$fp = fopen ($localfile, "r");
$ch = curl_init();
curl_setopt($ch, CURLOPT_VERBOSE, 1);
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_PUT, 1);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_INFILE, $fp);
curl_setopt($ch, CURLOPT_INFILESIZE, filesize($localfile));
```

```
$http_result = curl_exec($ch);
$error = curl_error($ch);
$http_code = curl_getinfo($ch ,CURLINFO_HTTP_CODE);
curl_close($ch);
fclose($fp);
print $http_code;
print "<br /><br />$http_result";
if ($error) {
  print "<br /><br />$error";
}
?>
```

The server responds with:

```
## About to connect() to localhost port 8080 (#0)
## Trying ::1... * connected
## Connected to localhost (::1) port 8080 (#0)
> PUT /ehcache/rest/sampleCache2/3 HTTP/1.1
Host: localhost:8080
Accept: */*
Content-Length: 11
Expect: 100-continue
< HTTP/1.1 100 Continue
< HTTP/1.1 201 Created
< Location: http://localhost:8080/ehcache/rest/sampleCache2/3
< Content-Length: 0
< Server: Jetty(6.1.10)
<
## Connection #0 to host localhost left intact
## Closing connection #0
```

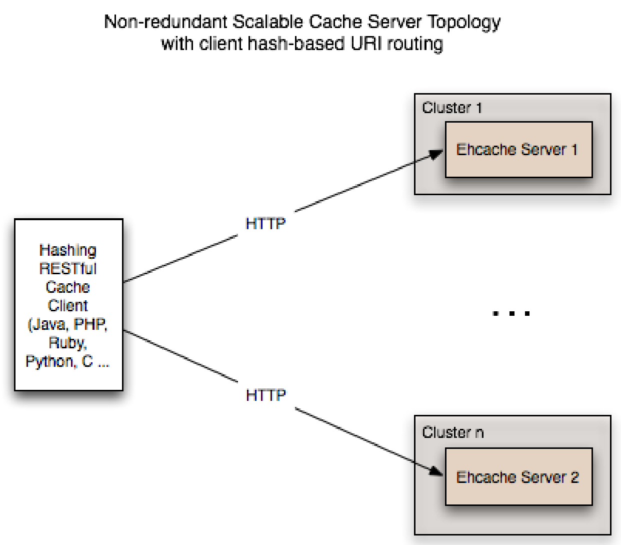
Creating Massive Caches with Load Balancers and Partitioning

The RESTful Ehcache Server is designed to achieve massive scaling using data partitioning - all from a RESTful interface. The largest Ehcache single instances run at around 20GB in memory. The largest disk stores run at 100Gb each. Add nodes together, with cache data partitioned across them, to get larger sizes. 50 nodes at 20GB gets you to 1 Terabyte. Two deployment choices need to be made:

- where is partitioning performed, and
- is redundancy required?

These choices can be mixed and matched with a number of different deployment topologies.

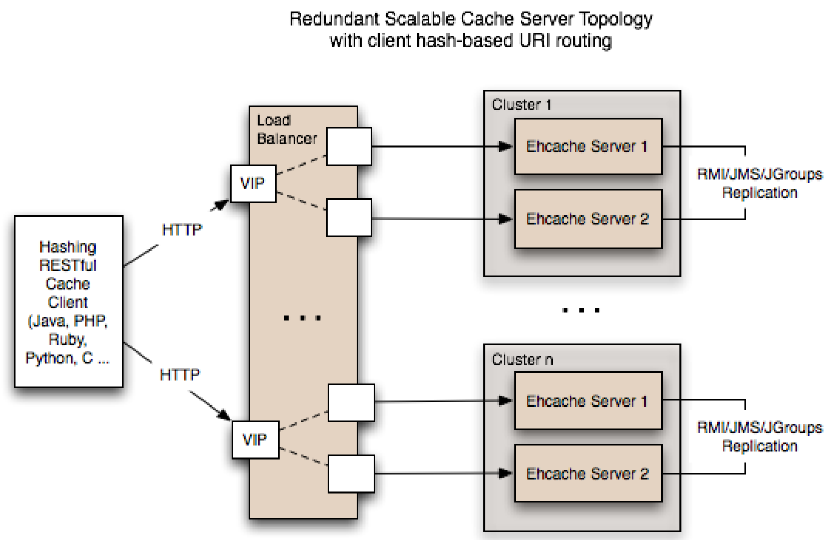
Non-redundant, Scalable with client hash-based routing


Non-redundant Scalable Cache Server Topology
with client hash-based URI routing
Cluster 1
Ehcache Server 1
HTTP
Hashing RESTful Cache Client (Java, PHP, Ruby, Python, C ...
. . .
HTTP
Cluster n
Ehcache Server 2

This topology is the simplest. It does not use a load balancer. Each node is accessed directly by the cache client using REST. No redundancy is provided. The client can be implemented in any language because it is simply a HTTP client. It must work out a partitioning scheme. Simple key hashing, as used by memcached, is sufficient. Here is a Java code sample:

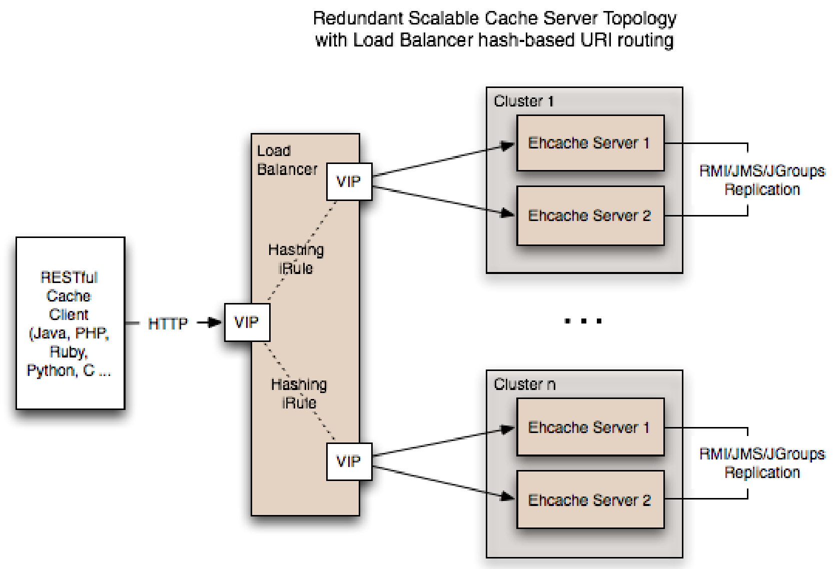
```
String[] cacheservers = new String[]{
"cacheserver0.company.com",
"cacheserver1.company.com",
"cacheserver2.company.com",
"cacheserver3.company.com",
"cacheserver4.company.com",
"cacheserver5.company.com"};
Object key = "123231";
int hash = Math.abs(key.hashCode());
int cacheserverIndex = hash % cacheservers.length;
String cacheserver = cacheservers[cacheserverIndex];
```

Redundant, Scalable with client hash-based routing


Redundant Scalable Cache Server Topology
with client hash-based URI routing
Hashing RESTful Cache Client (Java, PHP, Ruby, Python, C ...
HTTP
HTTP
Load Balancer
VIP
VIP
Cluster 1
Ehcache Server 1
Ehcache Server 2
RMI/JMS/JGroups Replication
Cluster n
Ehcache Server 1
Ehcache Server 2
RMI/JMS/JGroups Replication

Redundancy is added as shown in the above diagram by: Replacing each node with a cluster of two nodes. One of the existing distributed caching options in Ehcache is used to form the cluster. Options in Ehcache 1.5 are RMI and JGroups-based clusters. Ehcache-1.6 will add JMS as a further option. Put each Ehcache cluster behind VIPs on a load balancer.

Redundant, Scalable with load balancer hash-based routing


Redundant Scalable Cache Server Topology
with Load Balancer hash-based URI routing
RESTful Cache Client (Java, PHP, Ruby, Python, C ...
HTTP
Load Balancer
VIP
VIP
VIP
Hashing iRule
Hashing iRule
Cluster 1
Ehcache Server 1
Ehcache Server 2
RMI/JMS/JGroups Replication
Cluster n
Ehcache Server 1
Ehcache Server 2
RMI/JMS/JGroups Replication

Many content-switching load balancers support URI routing using some form of regular expressions. So, you could optionally skip the client-side hashing to achieve partitioning in the load balancer itself. For example:

```
/ehcache/rest/sampleCache1/[a-h]* => cluster1
/ehcache/rest/sampleCache1/[i-z]* => cluster2
```

Things get much more sophisticated with F5 load balancers, which let you create iRules in the TCL language. So rather than regular expression URI routing, you could implement key hashing-based URI routing. Remember in Ehcache's RESTful server, the key forms the last part of the URI. e.g. In the URI http://cacheserver.company.com/ehcache/rest/sampleCache1/3432, 3432 is the key. You hash using the last part of the URI.

5 Using the W3C (SOAP) Web Services

About W3C (SOAP) Web Services

The W3C is a standards body that defines Web Services as follows:

"The World Wide Web is more and more used for application-to-application communication. The programmatic interfaces made available are referred to as Web services."

They provide a set of recommendations for achieving this. An interoperability organization, WS-I, seeks to achieve interoperability between W3C Web Services. The W3C specifications for SOAP and WSDL are required to meet the WS-I definition. Ehcache is using Glassfish's libraries to provide it's W3C web services. The project known as Metro follows the WS-I definition.

Finally, OASIS defines a Web Services Security specification for SOAP: WS-Security. The current version is 1.1. It provides three main security mechanisms: ability to send security tokens as part of a message, message integrity, and message confidentiality. Ehcache's W3C Web Services support the stricter WS-I definition and use the SOAP and WSDL specifications. Specifically:

- The method of operation is in the entity-body of the SOAP envelope and a HTTP header. POST is always used as the HTTP method.
- The scoping information, used to identify the resource to perform the method on, is contained in the SOAP entity-body. The URI path is always the same for a given Web Service - it is the service “endpoint.”
- The Web Service is described by and exposes a {WSDL} (Web Services Description Language) file. It contains the methods, their arguments and what data types are used.
- The {WS-Security} SOAP extensions are supported.

The W3C Web Services API

The Ehcache W3C Web Services API exposes the singleton CacheManager, which typically has been configured in ehcache.xml or an Inversion of Control (IoC) container. Multiple CacheManagers are not supported. The API definition is as follows:

- WSDL - EhcacheWebServiceEndpointService.wsdl
- Types - EhcacheWebServiceEndpointService_schema1.xsd

Security

By default no security is configured. Because it is simply a Servlet 2.5 web application, it can be secured in all the usual ways by configuration in the web.xml.

In addition the cache server supports the use of XWSS 3.0 to secure the Web Service. All required libraries are packaged in the war for XWSS 3.0. A sample, commented out server_security_config.xml is provided in the WEB-INF directory. XWSS automatically looks for this configuration file. A simple example, based on an XWSS example, `net.sf.ehcache.server.soap.SecurityEnvironmentHandler`, which looks for a password in a System property for a given user name is included. This is not recommended for production use but is handy when you are getting started with XWSS. To use XWSS:

1. Add configuration in accordance with XWSS to the `server_security_config.xml` file.
2. Create a class which implements the `CallbackHandler` interface and provide its fully qualified path in the `SecurityEnvironmentHandler` element.
3. Use the integration test `EhcacheWebServiceEndpoint` to see how to use the XWSS client side.
4. On the client side, make sure configuration is provided in a file called `client_security_config.xml`, which must be in the root of the classpath.
5. To add client credentials into the SOAP request do:

```
cacheService = new EhcacheWebServiceEndpointService().getEhcacheWebServiceEndpointPort();
//add security credentials
((BindingProvider)cacheService).getRequestContext().put(BindingProvider.USERNAME_PROPERTY,
"Ron");
((BindingProvider)cacheService).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY,
"noR");
String result = cacheService.ping();
```