POJO Cache

User Documentation

Ben Wang

Jason Greene

ISBN:

Publication date: March 2008

POJO Cache: User Documentation

Ben Wang
Jason Greene

Preface

POJO Cache is an in-memory, transactional, and clustered cache system that allows users to operate on a POJO (Plain Old Java Object) transparently and without active user management of either replication or persistence aspects. JBoss Cache, which includes POJO Cache, is a 100% Java based library that can be run either as a standalone program or inside an application server.

This document is meant to be a user and reference guide to explain the architecture, api, configuration, and examples for POJO Cache. We assume the readers are familiar with both JGroups and the core JBoss Cache usages.

If you have questions, use the user *forum* [http://www.jboss.com/index.html?module=bb&op=viewforum&f=157] linked on the JBoss Cache website. We also provide tracking links for tracking bug reports and feature requests on *JBoss Jira web site* [http://jira.jboss.com] . If you are interested in the development of POJO Cache, post a message on the forum. If you are interested in translating this documentation into your language, contact us on the developer mailing list.

JBoss Cache is an open source product, using the business and OEM-friendly OSI-approved LGPL license. Commercial development support, production support and training for JBoss Cache is available through *JBoss, a division of Red Hat Inc.* [http://www.jboss.com]

In some of the example listings, what is meant to be displayed on one line does not fit inside the available page width. These lines have been broken up. A '\' at the end of a line means that a break has been introduced to fit in the page, with the following lines indented. So:

```
Let's pretend to have an extremely \
long line that \
does not fit
This one is short
```

Is really:

```
Let's pretend to have an extremely long line that does not fit
This one is short
```

Terminology

1. Overview

The section lists some basic terminology that will be used throughout this guide.

Aop
: Aspect-Oriented Programming (AOP) is a new paradigm that allows you to organize and layer your software applications in ways that are impossible with traditional object-oriented approaches. Aspects allow you to transparently glue functionality together so that you can have a more layered design. AOP allows you to intercept any event in a Java program and trigger functionality based on those events.

JBoss Aop
: JBoss Aop is an open-source Aop framework library developed by JBoss. It is 100% Java based and can be run either as a standalone or inside an application server environment. More details can be found at www.jboss.com. PojoCache uses JBoss Aop library in two ways. It uses JBoss Aop firstly for its own interceptor-based architecture and secondly to realize the fine-grained replication aspects.

Dynamic Aop
: Dynamic Aop is a feature of JBoss Aop that provides a hook so that a caller can insert event interception on the POJO at runtime. PojoCache currently uses this feature to perform field level interception.

JGroups
: JGroups is a reliable Java group messaging library that is open-source and LGPL. In addition to reliable messaging transport, it also performs group membership management. It has been a de facto replication layer used by numerous open-source projects for clustering purposes. It is also used by JBossCache for replication layer.

Core Cache
: Core Cache is a tree-structured, clustered, transactional cache. Simple and Serializable java types are stored as key/value pairs on nodes within the tree using a collection-like API. It also provides a number of configurable aspects such as node locking strategies, data isolation, eviction, and so on. POJO Cache leverages Core Cache as the underlying data-store in order to provide the same capabilities.

POJO
: Plain old Java object.

Annotation
: Annotation is a new feature in JDK5.0. It introduces metadata along side the Java code that can be accessed at runtime. PojoCache currently uses JDK50 annotation to support POJO instrumentation (JDK1.4 annotation has been deprecated since release 2.0).

Prepare

Prepare is a keyword in JBoss Aop pointcut language used to specify which POJO needs to be instrumented. It appears in a `pojocache-aop.xml` file. However, if you can use annotation to specify the POJO instrumentation, there is no need for a `pojocache-aop.xml` listing. Note that When a POJO is declared properly either through the xml or annotation, we consider it "aspectized".

Instrumentation

Instrumentation is an Aop process that basically pre-processes (e.g., performing byte-code weaving) on the POJO. There are two modes: compile- or load-time. Compile-time weaving can be done with an Aop precompiler (`aopc`) while load-time is done to specify a special classloader in the run script. This step is necessary for an Aop system to intercept events that are interesting to users.

Introduction

1. Overview

JBoss Cache consists of two components, Core Cache, and POJO Cache. Core Cache provides efficient memory storage, transactions, replication, eviction, persistent storage, and many other "core" features you would expect from a distributed cache. The Core Cache API is tree based. Data is arranged on the tree using nodes that each offer a map of attributes. This map-like API is intuitive and easy to use for caching data, but just like the Java Collection API, it operates only off of simple and serializable types. Therefore, it has the following constraints:

- If replication or persistence is needed, the object will then need to implement the `Serializable` interface. E.g.,

```
public Class Foo implements Serializable
```

- If the object is mutable, any field change will require a successive put operation on the cache:

```
value = new Foo();
cache.put(fqn, key, value);
value.update(); // update value
cache.put(fqn, key, value); // Need to repeat this step again to ask cache to persist or replicate
 the changes
```

- Java serialization always writes the entire object, even if only one field was changed. Therefore, large objects can have significant overhead, especially if they are updated frequently:

```
thousand = new ThousandFieldObject();
cache.put(fqn, key, thousand);
thousand.setField1("blah"); // Only one field was modified
cache.put(fqn, key, thousand); // Replicates 1000 fields
```

- The object structure can not have a graph relationship. That is, the object can not have references to objects that are shared (multiple referenced) or to itself (cyclic). Otherwise, the relationship will be broken upon serialization (e.g., when replicate each parent object separately). For example, Figure 1 illustrates this problem during replication. If we have two `Person` instances that share the same `Address` , upon replication, it will be split into two separate

`Address` instances (instead of just one). The following is the code snippet using Cache that illustrates this problem:

```
joe = new Person("joe");
mary = new Person("mary");
addr = new Address("Taipei");
joe.setAddress(addr);
mary.setAddress(addr);
cache.put("/joe", "person", joe);
cache.put("/mary", "person", mary);
```

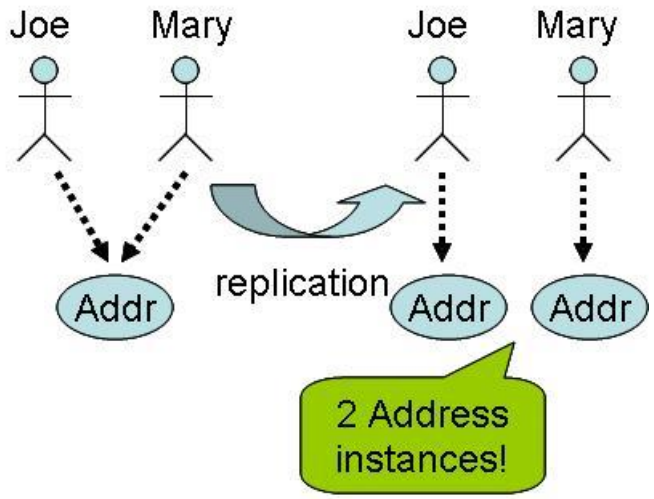

Figure 2.1. Illustration of shared objects problem during replication

POJO Cache attempts to address these issues by building a layer on top of Core Cache which transparently maps normal Java object model operations to individual Node operations on the cache. This offers the following improvements:

- Objects do not need to implement `Serializable` interface. Instead they are instrumented, allowing POJO Cache to intercept individual operations.
- Replication is fine-grained. Only modified fields are replicated, and they can be optionally batched in a transaction.
- Object identity is preserved, so graphs and cyclical references are allowed.
- Once attached to the cache, all subsequent object operationis will trigger a cache operation (like replication) automatically:

```
POJO pojo = new POJO();
pojoCache.attach("id", pojo);
pojo.setName("some pojo"); // This will trigger replication automatically.
```

In POJO Cache, these are the typical development and programming steps:

- Annotate your object with `@Replicable`
- Use `attach()` to put your POJO under cache management.
- Operate on the object directly. The cache will then manage the replication or persistence automatically and transparently.

More details on these steps will be given in later chapters.

Since POJO Cache is a layer on-top of Core Cache, all features available in Core Cache are also available in POJO Cache. Furthermore, you can obtain an instance to the underlying Core Cache by calling `PojoCache.getCache()`. This is useful for resusing the same cache instance to store custom data, along with the POJO model.

2. Features

Here are the current features and benefits of PojoCache:

- Fine-grained replication. The replication modes supported are the same as that of Core Cache: `LOCAL`, `REPL_SYNC`, `REPL_ASYNC`, `INVALIDATION_SYNC`, and `INVALIDATION_ASYNC` (see the main JBoss Cache reference documentation for details). The replication level is fine-grained and is performed automatically once the POJO is mapped into the internal cache store. When a POJO field is updated, a replication request will be sent out only to the key corresponding to that modified attribute (instead of the whole object). This can have a potential performance boost during the replication process; e.g., updating a single key in a big HashMap will only replicate the single field instead of the whole map!
- Transactions. All attached objects participate in a user transaction context. If a rollback occurs, the previous internal field state of the object will be restored:

```
POJO p = new POJO();
p.setName("old value");
pojoCache.attach("id", p);
tx.begin(); // start a user transaction
p.setName("some pojo");
tx.rollback(); // this will cause the rollback
p.getName(); // is "old value"
```

In addition, operations under a transaction is batched. That is, the update is not performed until the `commit` phase. Further, if replication is enabled, other nodes will not see the changes until the transaction has completed successfully.

- Passivation. POJO Cache supports the same passivation provided by Core Cache. When a node mapped by POJO Cache has reached a configured threshold, it is evicted from memory and stored using a cache loader. When the node is accessed again, it will be retrieved from the cache loader and put into memory. The configuration parameters are the same as those of the Cache counterpart. To configure the passivation, you will need to configure both the eviction policy and cache loader.
- Object cache by reachability, i.e., recursive object mapping into the cache store. On attach, `POJO Cache` will attach all referenced objects as well. This feature is explained in more detail later.
- Natural Object Relationships. Java references are preserved as they were written. That is, a user does not need to declare any object relationship (e.g., one-to-one, or one-to-many) to use the cache.
- Object Identity. Object identity is preserved. Not only can a cached object be compared using `equals()`, but the comparison operator, `==`, can be used as well. For example, an object such as `Address` may be multiple referenced by two `Person`s (e.g., `joe` and `mary`). The objects retrieved from `joe.getAddress()` and `mary.getAddress()` should be identicali, when when retrieved from a different node in the cluster then that which attached them.
- Inheritance. POJO Cache preserves the inheritance hierarchy of any object in the cache. For example, if a `Student` class inherits from a Person class, once a `Student` object is mapped to POJO Cache (e.g., `attach` call), the fields in the base class `Person` are mapped as well.
- Collections. Java Collection types (e.g. List, Set, and Map) are transparently mapped using Java proxies. Details are described later.
- Annotation based. Starting from release 2.0, JDK 5 annotations are used to indicate that an object should be instrumented for use under POJO Cache (once attached).
- Transparent. Once a POJO is attached to the cache, subsequent object model changes are transparently handled. No further API calls are required.

3. Usage

To use POJO Cache, you obtain the instance from the PojoCacheFactory by supplying a config file that is used by the delegating Cache implementation. Once the PojoCache instance is obtained, you can call the cache life cycle method to start the cache. Below is a code snippet that creates and starts the cache:

```
String configFile = "replSync-service.xml";
```

```
boolean toStart = false;
PojoCache pcache = PojoCacheFactory.createCache(configFiel, toStart);
pcache.start(); // if toStart above is true, it will starts the cache automatically.
pcache.attach(id, pojo);
// ...
pcache.stop(); // stop the cache. This will take PojoCache out of the clustering group, if any, e.g.
```

4. Requirements

`POJO Cache` is currently supported on JDK 5 (since release 2.0). It requires the following libraries (in addition to jboss-cache.jar and the required libraries for Core Cache) to start up:

- Library:
  - pojocache.jar. Main POJO Cache library.
  - jboss-aop-jdk50.jar. Main JBoss Aop library.
  - javassist.jar. Java byte code manipulation library.
  - trove.jar. High performance collections for Java.

Architecture

POJO Cache internally uses the JBoss Aop framework to both intercept object field access, and to provide an internal interceptor stack for centralizing common behavior (e.g. locking, transactions).

The following figure is a simple overview of the POJO Cache architecture. From the top, it can be can seen that when a call comes in (e.g., `attach` or `detach`), it will go through the POJO Cache interceptor stack first. After that, it will store the object's fields into the underlying Core Cache, which will be replicated (if enabled) using JGroups.

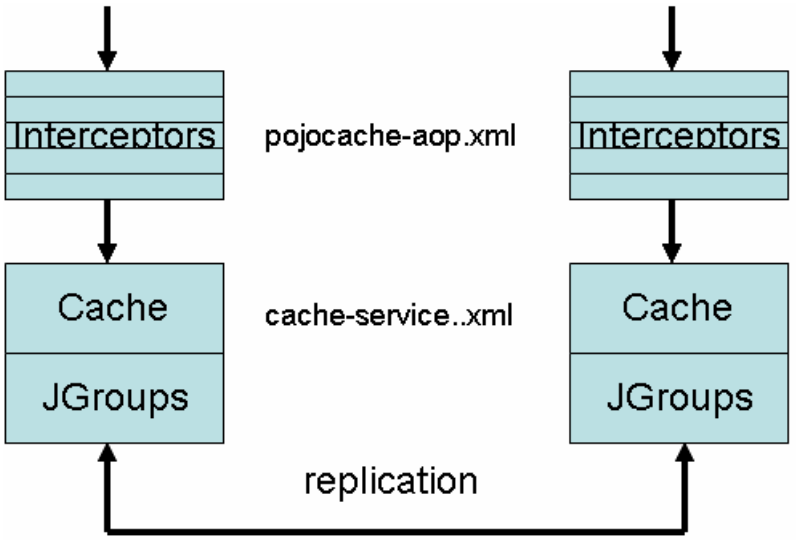

Figure 3.1. POJO Cache architecture overview

1. POJO Cache interceptor stack

As mentioned, the JBoss Aop framework is used to provide a configurable interceptor stack. In the current implementation, the main POJO Cache methods have their own independant stack. These are specified in `META-INF/pojocache-aop.xml` In most cases, this file should be left alone, although advanced users may wish to add their own interceptors. The Following is the default configuration:

```
<!-- Check id range validity -->
<interceptor name="CheckId" class="org.jboss.cache.pojo.interceptors.CheckIdInterceptor"
    scope="PER_INSTANCE"/>

<!-- Track Tx undo operation -->
<interceptor name="Undo" class="org.jboss.cache.pojo.interceptors.PojoTxUndoInterceptor"
    scope="PER_INSTANCE"/>
```

```
<!-- Begining of interceptor chain -->
<interceptor name="Start" class="org.jboss.cache.pojo.interceptors.PojoBeginInterceptor"
    scope="PER_INSTANCE"/>

<!-- Check if we need a local tx for batch processing -->
<interceptor name="Tx" class="org.jboss.cache.pojo.interceptors.PojoTxInterceptor"
    scope="PER_INSTANCE"/>

<!--
                    Mockup failed tx for testing. You will need to set
PojoFailedTxMockupInterceptor.setRollback(true)
  to activate it.
-->
                                        <interceptor                name="MockupTx"
class="org.jboss.cache.pojo.interceptors.PojoFailedTxMockupInterceptor"
    scope="PER_INSTANCE"/>

<!-- Perform parent level node locking -->
<interceptor name="TxLock" class="org.jboss.cache.pojo.interceptors.PojoTxLockInterceptor"
    scope="PER_INSTANCE"/>

<!-- Interceptor to perform Pojo level rollback -->
                                          <interceptor                name="TxUndo"
class="org.jboss.cache.pojo.interceptors.PojoTxUndoSynchronizationInterceptor"
        scope="PER_INSTANCE"/>

<!-- Interceptor to used to check recursive field interception. -->
                                        <interceptor                name="Reentrant"
class="org.jboss.cache.pojo.interceptors.MethodReentrancyStopperInterceptor"
        scope="PER_INSTANCE"/>

<!-- Whether to allow non-serializable pojo. Default is false. -->
                                    <interceptor              name="MarshallNonSerializable"
class="org.jboss.cache.pojo.interceptors.CheckNonSerializableInterceptor"
        scope="PER_INSTANCE">
    <attribute name="marshallNonSerializable">false</attribute>
        </interceptor>

<stack name="Attach">
  <interceptor-ref name="Start"/>
  <interceptor-ref name="CheckId"/>
  <interceptor-ref name="Tx"/>
  <interceptor-ref name="TxLock"/>
  <interceptor-ref name="TxUndo"/>
```

```
  </stack>

  <stack name="Detach">
    <interceptor-ref name="Start"/>
    <interceptor-ref name="CheckId"/>
    <interceptor-ref name="Tx"/>
    <interceptor-ref name="TxLock"/>
    <interceptor-ref name="TxUndo"/>
  </stack>

  <stack name="Find">
    <interceptor-ref name="Start"/>
    <interceptor-ref name="CheckId"/>
  </stack>
```

The stack should be self-explanatory. For example, for the `Attach` stack, we currently have `Start, CheckId, Tx, TxLock`, and `TxUndo` interceptors. The stack always starts with a `Start` interceptor such that initialization can be done properly. `CheckId` is to ensure the validity of the Id (e.g., it didn't use any internal Id string). Finally, `Tx, TxLock`, and `TxUndo` are handling the the proper transaction locking and rollback behavior (if needed).

2. Field interception

POJO Cache currently uses JBoss AOP to intercept field operations. If a class has been properly instrumented (by either using the `@Replicable` annotation, or if the object has already been advised by JBoss AOP), then a cache interceptor is added during an `attach()` call. Afterward, any field modification will invoke the corresponding `CacheFieldInterceptor` instance. Below is a schematic illustration of this process.

Only fields, and not methods are intercepted, since this is the most efficient and accurate way to gaurantee the same data is visible on all nodes in the cluster. Further, this allows for objects that do not conform to the JavaBean specficiation to be replicable. There are two important aspects of field interception:

- All access qualifiers are intercepted. In other words, all `private`, all `protected`, all default, and all `public` fields will be intercepted.
- Any field with `final`, `static`, and/or `transient` qualifiers, **will be skipped**. Therefore, they will not be replicated, passivated, or manipulated in any way by POJO Cache.

The figure below illustrates both field read and write operations. Once an POJO is managed by POJO Cache (i.e., after an `attach()` method has been called), JBoss Aop will invoke the `CacheFieldInterceptor` every time a class operates on a field. The cache is always consulted, since it is in control of the mapped data (i.e. it gaurantess the state changes made by other nodes

in the cluster are visible). Afterwords, the in-memmory copy is updated. This is mainly to allow transaction rollbacks to restore the previous state of the object.

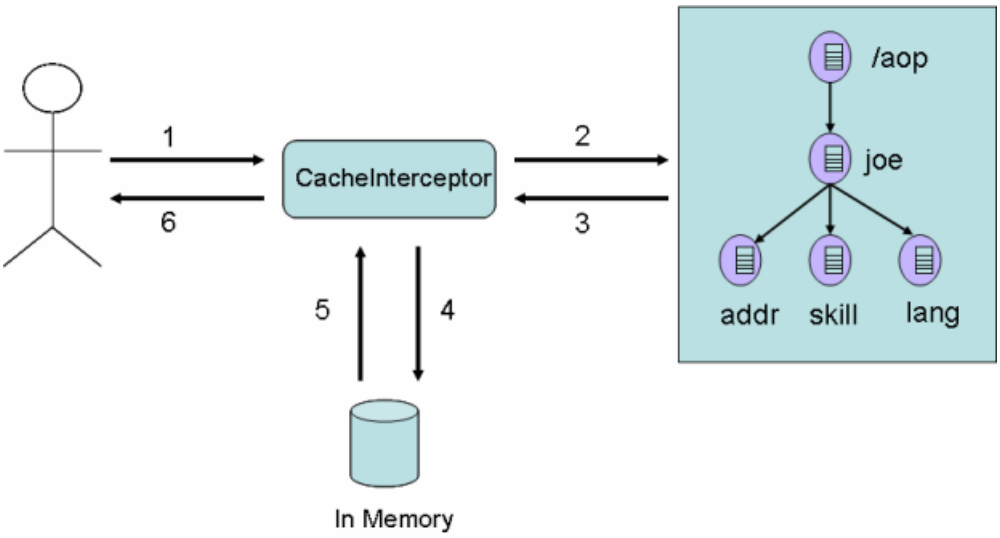

Figure 3.2. POJO Cache field interception

3. Object relationship management

As previously mentioned, unlike a traditional cache system, POJO Cache preserves object identity. This allows for any type of object relationship available in the Java language to be transparently handled.

During the mapping process, all object references are checked to see if they are already stored in the cache. If already stored, instead of duplicating the data, a reference to the original object is written in the cache. All referenced objects are reference counted, so they will be removed once they are no longer referenced.

To look at one example, let's say that multiple `Persons` ("joe" and "mary") objects can own the same `Address` (e.g., a household). The following diagram is a graphical representation of the pysical cache data. As can be seen, the "San Jose" address is only stored once.

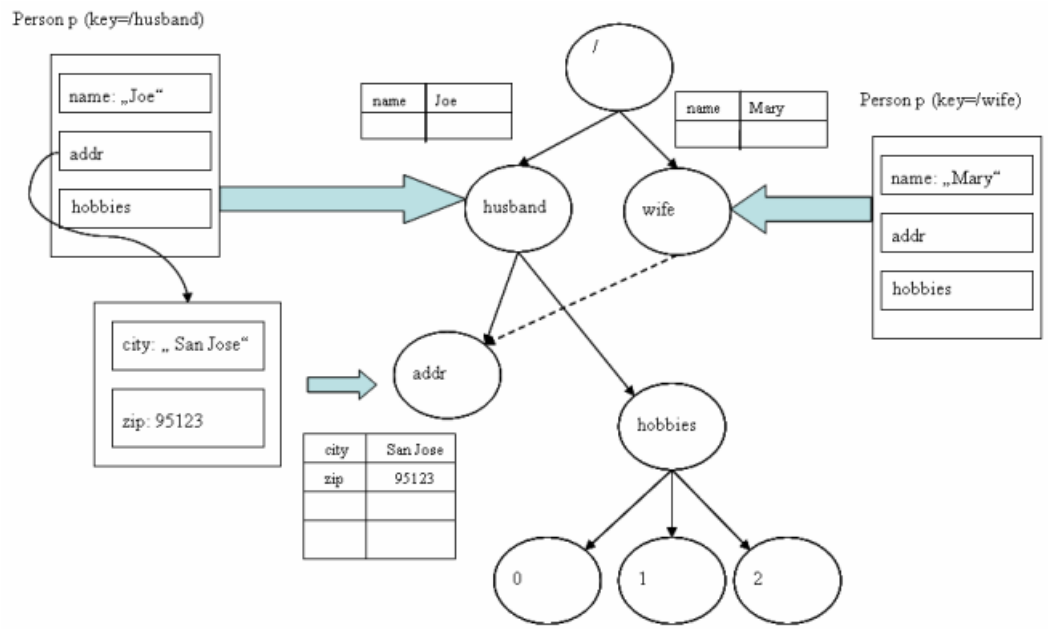

Figure 3.3. Schematic illustration of object relationship mapping

In the following code snippet, we show programmatically the object sharing example.

```
import org.jboss.cache.pojo.PojoCache;
import org.jboss.cache.pojo.PojoCacheFactory;
import org.jboss.test.cache.test.standAloneAop.Person;
import org.jboss.test.cache.test.standAloneAop.Address;

String configFile = "META-INF/replSync-service.xml";
PojoCache cache = PojoCacheFactory.createCache(configFile); // This will start PojoCache automatically

Person joe = new Person(); // instantiate a Person object named joe
joe.setName("Joe Black");
joe.setAge(41);

Person mary = new Person(); // instantiate a Person object named mary
mary.setName("Mary White");
mary.setAge(30);

Address addr = new Address(); // instantiate a Address object named addr
addr.setCity("Sunnyvale");
addr.setStreet("123 Albert Ave");
addr.setZip(94086);
```

```
joe.setAddress(addr); // set the address reference
mary.setAddress(addr); // set the address reference

cache.attach("pojo/joe", joe); // add aop sanctioned object (and sub-objects) into cache.
cache.attach("pojo/mary", mary); // add aop sanctioned object (and sub-objects) into cache.

Address joeAddr = joe.getAddress();
Address maryAddr = mary.getAddress(); // joeAddr and maryAddr should be the same instance

cache.detach("pojo/joe");
maryAddr = mary.getAddress(); // Should still have the address.
```

If `joe` is removed from the cache, `mary` should still have reference the same `Address` object in the cache store.

To further illustrate this relationship management, let's examine the Java code under a replicated environment. Imagine two separate cache instances in the cluster now (`cache1` and `cache2`). On the first cache instance, both `joe` and `mary` are attached as above. Then, the application fails over to `cache2`. Here is the code snippet for `cache2` (assume the objects were already attached):

```
/**
 * Code snippet on cache2 during fail-over
 */
import org.jboss.cache.PropertyConfigurator;
import org.jboss.cache.pojo.PojoCache;
import org.jboss.test.cache.test.standAloneAop.Person;
import org.jboss.test.cache.test.standAloneAop.Address;

String configFile = "META-INF/replSync-service.xml";
PojoCache cache2 = PojoCacheFactory.createCache(configFile); // This will start PojoCache
 automatically

Person joe = cache2.find("pojo/joe"); // retrieve the POJO reference.
Person mary = cache2.find("pojo/mary"); // retrieve the POJO reference.

Address joeAddr = joe.getAddress();
Address maryAddr = mary.getAddress(); // joeAddr and maryAddr should be the same instance!!!

maryAddr = mary.getAddress().setZip(95123);
int zip = joeAddr.getAddress().getZip(); // Should be 95123 as well instead of 94086!
```

4. Object Inheritance

POJO Cache preserves the inheritance hierarchy of all attached objects. For example, if a `Student` extends `Person` with an additional field `year`, then once `Student` is put into the cache, all the class attributes of `Person` are mapped to the cache as well.

Following is a code snippet that illustrates how the inheritance behavior of a POJO is maintained. Again, no special configuration is needed.

```
import org.jboss.test.cache.test.standAloneAop.Student;

Student joe = new Student(); // Student extends Person class
joe.setName("Joe Black"); // This is base class attributes
joe.setAge(22); // This is also base class attributes
joe.setYear("Senior"); // This is Student class attribute

cache.attach("pojo/student/joe", joe);

//...

joe = (Student)cache.attach("pojo/student/joe");
Person person = (Person)joe; // it will be correct here
joe.setYear("Junior"); // will be intercepted by the cache
joe.setName("Joe Black II"); // also intercepted by the cache
```

5. Physical object cache mapping model

The previous sections describe the logical object mapping model. In this section, we will explain the physical mapping model, that is, how do we map the POJO into Core Cache for transactional state replication. However, it should be noted that the physical structure of the cache is purely an internal implementation detail, it should not be treated as an API as it may change in future releases. This information is provided solely to aid in better understanding the mapping process in POJO Cache.

When an object is first attached in POJO Cache, the Core Cache node representation is created in a special internal area. The `Id` fqn that is passed to `attach()` is used to create an empty node that references the internal node. Future references to the same object will point to the same internal node location, and that node will remain until all such references have been removed (detached).

The example below demonstrates the mapping of the `Person` object under id "pojo/joe" and "pojo/mary" as metioned in previous sections. It is created from a two node replication group where one node is a Beanshell window and the other node is a Swing Gui window (shown here). For clarity, multiple snapshots were taken to highlight the mapping process.

The first figure illustrates the first step of the mapping approach. From the bottom of the figure, it can be seen that the `PojoReference` field under `pojo/joe` is pointing to an internal location, `/__JBossInternal__/5c4o12-lpaf5g-esl49n5e-1-esl49n5o-2`. That is, under the user-specified Id string, we store only an indirect reference to the internal area. Please note that `Mary` has a similar reference.

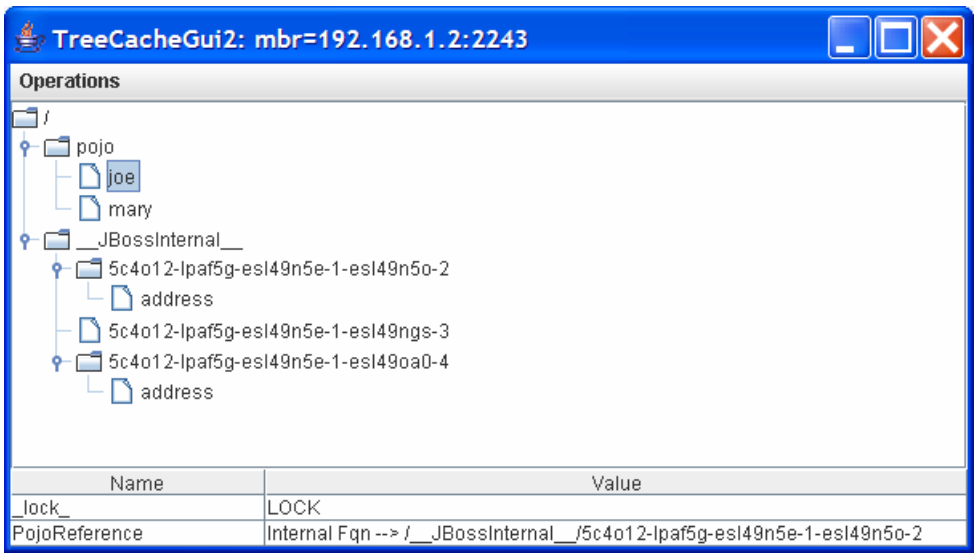

Figure 3.4. Object cache mapping for `Joe`

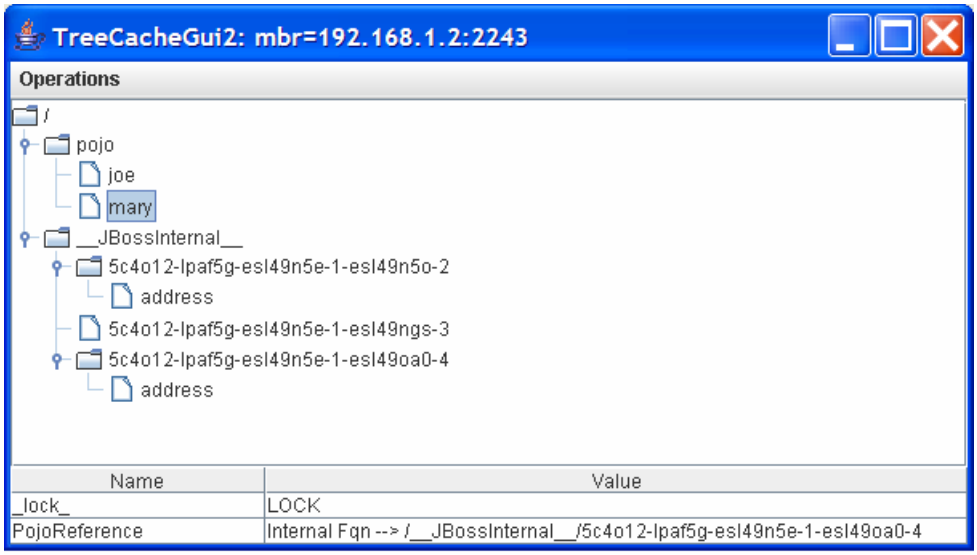

Figure 3.5. Object cache mapping for `Mary`

Then by clicking on the referenced internal node (from the following figure), it can seen that the primitive fields for `Joe` are stored there. E.g., `Age` is `41` and `Name` is `Joe Black`. And similarly for `Mary` as well.

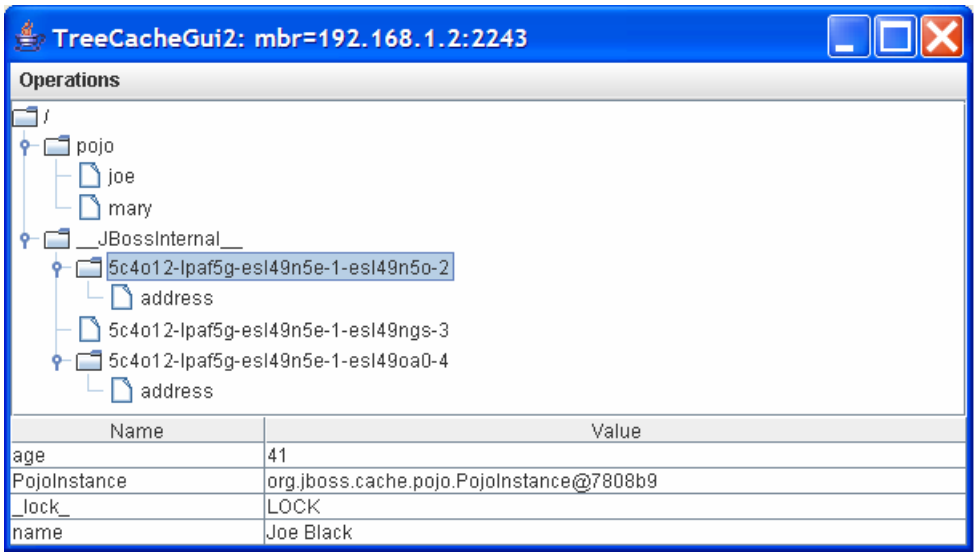

Figure 3.6. Object cache mapping for internal node `Joe`

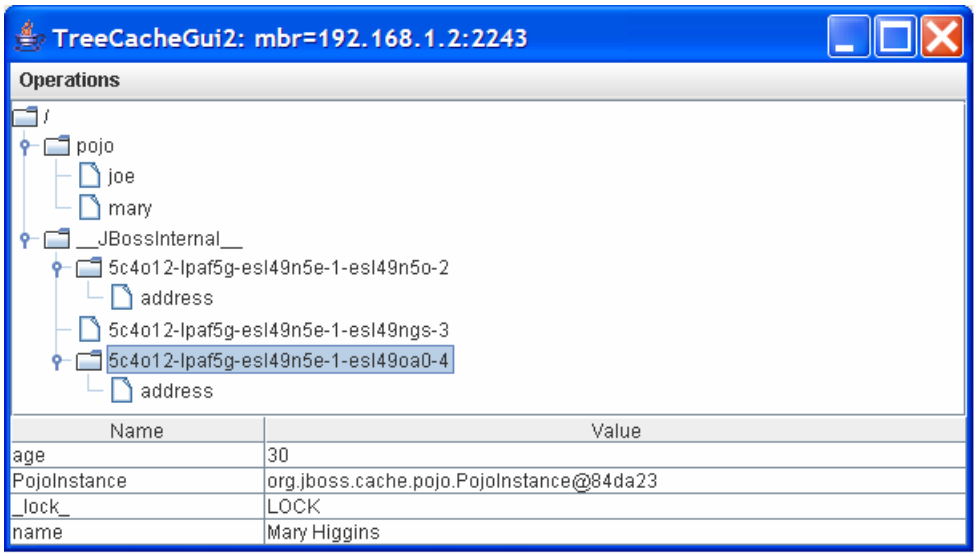

Figure 3.7. Object cache mapping for internal node `Mary`

Under the `/__JBossInternal__/5c4o12-lpaf5g-esl49n5e-1-esl49n5o-2`, it can be seen that there is an `Address` node. Clicking on the `Address` node shows that it references another internal location: `/__JBossInternal__/5c4o12-lpaf5g-esl49n5e-1-esl49ngs-3` as shown in the following figure. Then by the same token, the `Address` node under `/__JBossInternal__/5c4o12-lpaf5g-esl49n5e-1-esl49na0-4` points to the same address reference. That is, both `Joe` and `Mary` share the same `Address` reference.

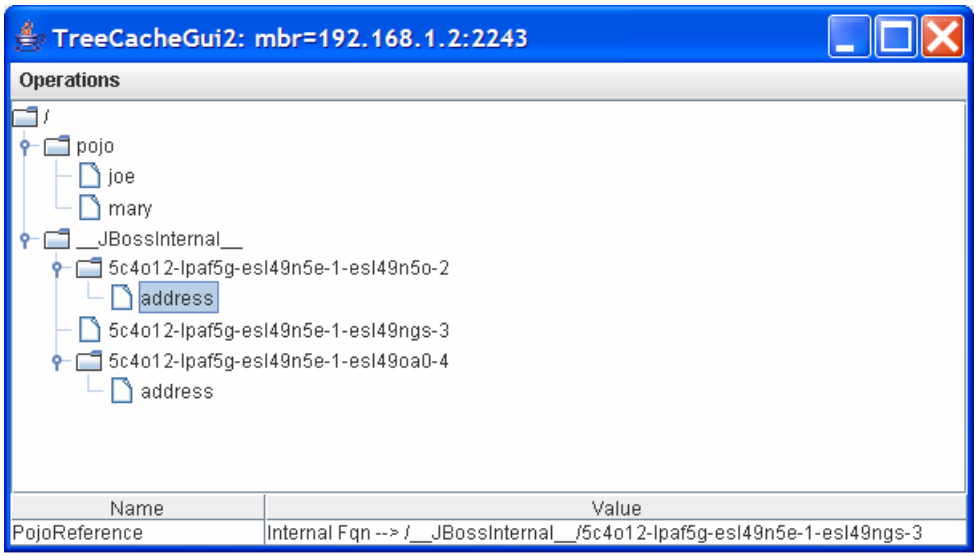

Figure 3.8. Object cache mapping: Joe's internal address

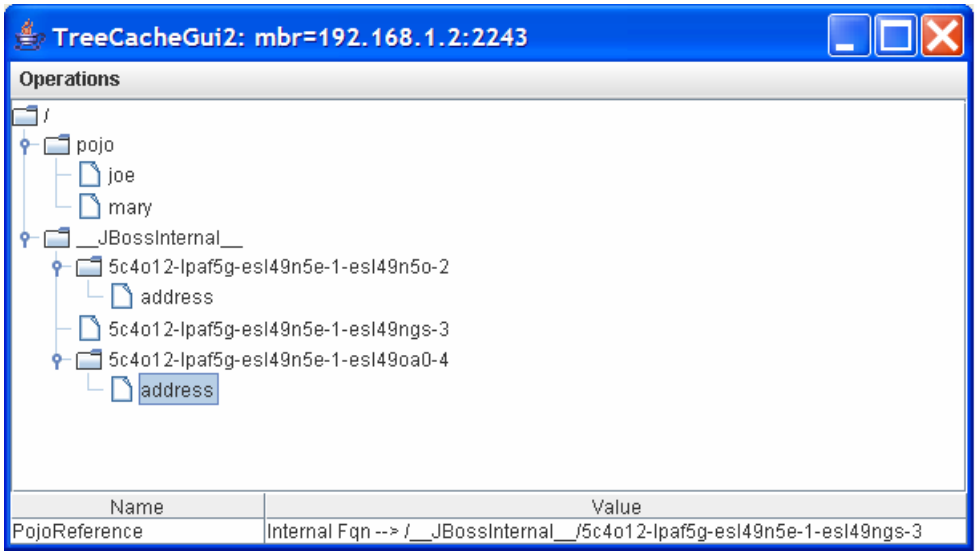

Figure 3.9. Object cache mapping: Mary's internal address

Finally, the `/__JBossInternal__/5c4o12-lpaf5g-esl49n5e-1-esl49ngs-3` node contains the various various primitive fields of `Address`, e.g., `Street`, `Zip`, and `City`. This is illustrated in the following figure.

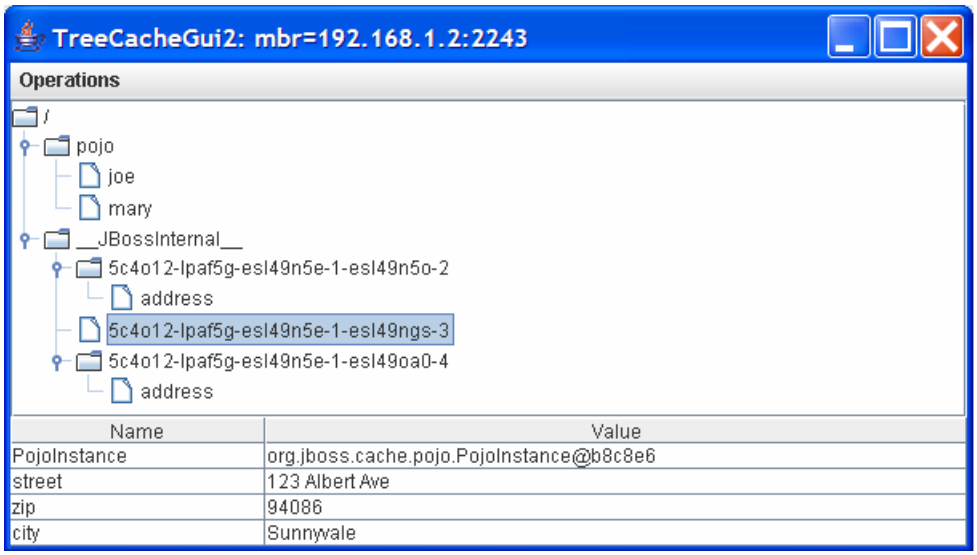

Figure 3.10. Object cache mapping: Address fields

6. Collection Mapping

Due to current Java limitations, Collection classes that implement `Set`, `List`, and `Map` are substituted with a Java proxy. That is, whenever POJO Cache encounters any Collection instance, it will:

- Create a Collection proxy instance and place it in the cache (instead of the original reference). The mapping of the Collection elements will still be carried out recursively as expected.
- If the Collection instance is referenced from another object, POJO Cache will swap out the original reference with the new proxy, so that operations performed by the refering object will be picked up by the cache.

The drawback to this approach is that the calling application must re-get any collection references that were attached. Otherwise, the cache will not be aware of future changes. If the collection is referenced from another object, then the calling app can obtain the proxy by using the publishing mechanism provided by the object (e.g. Person.getHobbies()). If, however, the collection is directly attached to the cache, then a subsequent `find()` call will need to be made to retrieve the proxy.

The following code snippet illustrates obtaining a direct Collection proxy reference:

```
List list = new ArrayList();
list.add("ONE");
list.add("TWO");

cache.attach("pojo/list", list);
list.add("THREE"); // This won't be intercepted by the cache!
```

```
List proxyList = cache.find("pojo/list"; // Note that list is a proxy reference
proxyList.add("FOUR"); // This will be intercepted by the cache
```

This snippet illustrates obtaining the proxy reference from a refering object:

```
Person joe = new Person();
joe.setName("Joe Black"); // This is base class attributes
List lang = new ArrayList();
lang.add("English");
lang.add("Mandarin");
joe.setLanguages(lang);

// This will map the languages List automatically and swap it out with the proxy reference.
cache.attach("pojo/student/joe", joe);
lang = joe.getLanguages(); // Note that lang is now a proxy reference
lang.add("French"); // This will be intercepted by the cache
```

Finally, when a Collection is removed from the cache (e.g., via `detach`), you still can use the proxy reference. POJO Cache will just redirect the call back to the in-memory copy. See below:

```
List list = new ArrayList();
list.add("ONE");
list.add("TWO");

cache.attach("pojo/list", list);
List proxyList = cache.find("pojo/list"); // Note that list is a proxy reference
proxyList.add("THREE"); // This will be intercepted by the cache

cache.detach("pojo/list"); // detach from the cache
proxyList.add("FOUR"); // proxyList has 4 elements still.
```

6.1. Limitations

The current implementation has the following limitations with collections:

- Only List, Set and Map are supported. Also it should be noted that the Java Collection API does not fully describe the behavior of implementations, so the cache versions may differ slightly from the common Java implementations (e.g. handling of NULL)

- As of PojoCache 2.0, HashMap keys must be serializable. Prior to PojoCache 2.0, HashMap keys were converted to String. This was fixed as you couldn't get the key back in its original form. See issue JBCACHE-399 for more details.

API Overview

This section provides a brief overview of the POJO Cache APIs. Please consult the javadoc for the full API.

1. PojoCacheFactory Class

PojoCacheFactory provides a couple of static methods to instantiate and obtain a PojoCache instance.

```
/**
 * Create a PojoCache instance. Note that this will start the cache life cycle automatically.
 * @param config A configuration string that represents the file name that is used to
 * configure the underlying Cache instance.
 * @return PojoCache
 */
public static PojoCache createInstance(String config);

/**
 * Create a PojoCache instance.
 * @param config A configuration string that represents the file name that is used to
 * configure the underlying Cache instance.
 * @param start If true, it will start the cache life cycle.
 * @return PojoCache
 */
public static PojoCache createInstance(String config, boolean start);

/**
 * Create a PojoCache instance.
 * @param config A configuration object that is used to configure the underlying Cache instance.
 * @param start If true, it will start the cache life cycle.
 * @return PojoCache
 */
public static PojoCache createInstance(Configuration config, boolean start);
```

For example, to obtain a PojoCache instance and start the cache lifestyle automatically, we can do:

```
String configFile = "META-INF/replSync-service.xml";
PojoCache cache = PojoCacheFactory.createInstance(configFile);
```

2. PojoCache Interface

`PojoCache` is the main interface for POJO Cache operations. Since most of the cache interaction is performed against the application domain model, there are only a few methods on this interface.

2.1. Attachment

```
/**
   * Attach a POJO into PojoCache. It will also recursively put any sub-POJO into
   * the cache system. A POJO can be the following and have the consequences when attached:
   *
   * It is PojoCacheable, that is, it has been annotated with
   * {@see org.jboss.cache.aop.annotation.PojoCacheable} annotation (or via XML), and has
   * been "instrumented" either compile- or load-time. The POJO will be mapped recursively to
   * the system and fine-grained replication will be performed.
   *
   * It is Serializable. The POJO will still be stored in the cache system. However, it is
   * treated as an "opaque" object per se. That is, the POJO will neither be intercepted
   * (for fine-grained operation) or object relationship will be maintained.
   *
   * Neither of above. In this case, a user can specify whether it wants this POJO to be
   * stored (e.g., replicated or persistent). If not, a PojoCacheException will be thrown.
   *
   * @param id   An id String to identify the object in the cache. To promote concurrency, we
  *          recommend the use of hierarchical String separating by a designated separator. Default
   *           is "/" but it can be set differently via a System property, jbosscache.separator
   *           in the future release. E.g., "/ben", or "/student/joe", etc.
   * @param pojo object to be inserted into the cache. If null, it will nullify the fqn node.
   * @return Existing POJO or null if there is none.
   * @throws PojoCacheException Throws if there is an error related to the cache operation.
   */
  Object attach(String id, Object pojo) throws PojoCacheException;
```

As described in the above javadoc, this method "attaches" the passed object to the cache at the specified location (`id`). The passed in object (`pojo`) must have been instrumented (using the `@Replicable` annotation) or implement the `Serializable` interface.

If the object is not instrumented, but serializable, POJO Cache will simply treat it as an opaque "primitive" type. That is, it will simply store it without mapping the object's fields into the cache. Replication is done on the object wide level and therefore it will not be fine-grained.

If the object has references to other objects, this call will issue `attach()` calls recursively until the entire object graph is traversed. In addition, object identity and object references are preserved. So both circular and multiply referenced objects are mapped as expected.

The return value after the call is the previous object under `id` , if any. As a result, a successful call i will replace that old value with the new instance. Note that a user will only need to issue this call once for each top-level object. Further calls can be made directly on the graph, and they will be mapped as expected.

2.2. Detachment

```
/**
   * Remove POJO object from the cache.
   *
   * @param id Is string that associates with this node.
   * @return Original value object from this node.
   * @throws PojoCacheException Throws if there is an error related to the cache operation.
   */
  Object detach(String id) throws PojoCacheException;
```

This call will detach the POJO from the cache by removing the contents under `id` and return the POJO instance stored there (null if it doesn't exist). If successful, further operations against this object will not affect the cache. Note this call will also remove everything stored under `id` even if you have put other plain cache data there.

2.3. Query

```
/**
   * Retrieve POJO from the cache system. Return null if object does not exist in the cache.
   * Note that this operation is fast if there is already a POJO instance attached to the cache.
   *
   * @param id that associates with this node.
   * @return Current content value. Null if does not exist.
   * @throws PojoCacheException Throws if there is an error related to the cache operation.
   */
  Object find(String id) throws PojoCacheException;
```

This call will return the current object content located under `id` . This method call is useful when you don't have the exact POJO reference. For example, when you fail over to the replicated node, you want to get the object reference from the replicated cache instance. In this case, PojoCache will create a new Java object if it does not exist and then add the cache interceptor such that every future access will be in sync with the underlying cache store.

```
/**
   * Query all managed POJO objects under the id recursively. Note that this will not return
```

```
   * the sub-object POJOs, e.g., if Person has a sub-object of Address, it
   * won't return Address pojo. Also note also that this operation is not thread-safe
   * now. In addition, it assumes that once a POJO is found with a id, no more POJO is stored
   * under the children of the id. That is, we don't mix the id with different POJOs.
   *
   * @param id The starting place to find all POJOs.
   * @return Map of all POJOs found with (id, POJO) pair. Return size of 0, if not found.
   * @throws PojoCacheException Throws if there is an error related to the cache operation.
   */
  Map findAll(String id) throws PojoCacheException;
```

This call will return all the managed POJOs under cache with a base Fqn name. It is recursive, meaning that it will traverse all the sub-trees to find the POJOs under that base. For example, if you specify the fqn to be root, e.g., `"/"` , then it will return all the managed POJOs under the cache.

3. POJO Annotations

There are currently three annotations that can be used on attached objects in POJO Cache: `@Replicable` , `@Transient` , and `@Serializable` . All of which affect how the annotated object is replicated.

3.1. @Replicable annotation

The `@org.jboss.cache.annotation.Replicable` annotation is used to mark classes for instrumentation. Once instrumented, individual field changes can be replicated separately. See the instrumentation chapter for more information on this process.

The following example will replicate changes to resource and name separately:

```
@Replicable
public class Gadget
{
  private Resource resource;
  private String name;
}
```

3.2. @Transient annotation

The `@org.jboss.cache.annotation.Transient` annotation is used to indicate that a field should not be replicable. This allows non-transient fields to be included in normal Java serialization output, but not when replicated by POJO Cache.

The following object will not replicate changes to resource.

```
@Replicable
public class Gadget
{
  // resource won't be replicated
  @Transient
  private Resource resource;
}
```

3.3. @Serializable annotation

The `@org.jboss.cache.annotation.Serializable` annotation indicates a field should be treated as a serialized attribute. This, of course, requires that the type of the field implement `Serializable` . If the marked field type has an @Replicable annotation, it will be ignored.

```
@Replicable
public class Gadget
{
  // resource won't be replicated
  @Transient
  private Resource resource;

  // serialized even if SpecialAddress is @Replicable
  @Serializable
  private SpecialAddress specialAddress;

  // other state variables
}
```

Configuration and Deployment

Since POJO Cache uses Core Cache for the underlying node replication, transaction, locking, and passivation behavior, the configuration is mostly the same.

1. Cache configuration xml file

When a PojoCache instance is obtained from a PojoCacheFactory, it is required that the either a `org.jboss.cache.config.Configuration` object is passed, or more typically a String indicating the location on the classpath or filesystem of an xml configuration file is provided. In the latter case, PojoCacheFactory will parse the xml to create a `Configuration`. PojoCache will simply pass the resulting `Configuration` to the underlying Core Cache implementation. For details on the configuration please see the "Configuration" chapter in the the JBoss Cache User Guide.

2. Passivation

A common use-case is to configure the underlying Core Cache to enable passivation. Passivation is a feature used to reduce cache memory usage by evicting stale data that can later be reloaded. In JBoss Cache, it is done via a combination of an eviction policy and a cache loader. That is, when a node is evicted from the Cache's in-memory store, it will be stored in a persistent store by the cache loader. When the node is requested again, it will be loaded from the persistent store and stored into memory.

There is a restriction, however. Since POJO Cache maps object data into an internal area, there are two places that have object information. One is under the regular String ID that the user specifies, and the other is located under `/__JBossInternal__`. Therefore, to maintain consistentency, when you specify the eviction region, you can only specify one global (i.e., `/_default_`) region. This way, when the nodes associated with a POJO are passivated, they will do so across the whole region.

Below is a snippet from a cache configuration xml illustrating how the eviction policy along with cache loader can be configured. Please note that this is simply an aspect of the underlying Cache. That is, PojoCache layer is agnostic to this behavior.

```
<attribute name="EvictionPolicyConfig">
  <config>
    <attribute name="wakeUpIntervalSeconds">5</attribute>
    <attribute name="policyClass">org.jboss.cache.eviction.LRUPolicy</attribute>
    <!-- Cache wide default -->
    <region name="/_default_">
      <attribute name="maxNodes">5000</attribute>
      <attribute name="timeToLiveSeconds">3</attribute>
    </region>
```

```
  </config>
</attribute>

<attribute name="CacheLoaderConfiguration">
  <config>
    <passivation>true</passivation>
    <preload>/</preload>
    <shared>false</shared>

    <!-- we can now have multiple cache loaders, which get chained -->
    <cacheloader>
      <class>org.jboss.cache.loader.FileCacheLoader</class>
      <!-- whether the cache loader writes are asynchronous -->
      <async>false</async>
      <!-- only one cache loader in the chain may set fetchPersistentState to true.
          An exception is thrown if more than one cache loader sets this to true. -->
      <fetchPersistentState>true</fetchPersistentState>
      <!-- determines whether this cache loader ignores writes - defaults to false. -->
      <ignoreModifications>false</ignoreModifications>
    </cacheloader>
  </config>
</attribute>
```

Another way to support multiple regions in eviction is to use region-based marshalling. See the "Architecture" chapter in the JBoss Cache User Guide for more information on region-based marshalling. When the Cache uses region-based marshalling, POJO Cache will store internal node data on the region that is specified. This allows for a more flexible eviction policy.

3. AOP Configuration

POJO Cache supplies a `pojocache-aop.xml` that is required to be set via a system property: `jboss.aop.path` during compile- or load-time, or placed in the user's classpath. The file now consists of the interceptor stack specification, as well as annotations for POJO instrumentation. It is listed fully in the Appendix section. Note that the file should not normally need to be modified. Only an advanced use-case would require changes.

4. Deployment Options

There are a number of ways to deploy POJO Cache:

4.1. Programatic Deployment

Simply instantiate a PojoCacheFactory and invoke one of the overloaded `createCache` methods shown in the *API Overview*.

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

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

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

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

Note in the *example* the value of the `mbean` element's `code` attribute: `org.jboss.cache.pojo.jmx.PojoCacheJmxWrapper`. This is the class JBoss Cache uses to handle JMX integration; the PojoCache itself does not expose an MBean interface. See the *JBoss Cache MBeans section* for more on the PojoCacheJmxWrapper.

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

```
MBeanServer server = MBeanServerLocator.locateJBoss();
ObjectName on = new ObjectName("jboss.cache:service=PojoCache");
PojoCacheJmxWrapperMBean cacheWrapper =
   (PojoCacheJmxWrapperMBean) MBeanServerInvocationHandler.newProxyInstance(server,
on,
                                PojoCacheJmxWrapperMBean.class, false);
PojoCache cache = cacheWrapper.getPojoCache();
```

The MBeanServerLocator class is a helper to find the (only) JBoss MBean server inside the current JVM. The `javax.management.MBeanServerInvocationHandler` class' `newProxyInstance`

method creates a dynamic proxy implementing the given interface and uses JMX to dynamically dispatch methods invoked against the generated interface to the MBean. The name used to look up the MBean is the same as defined in the cache's configuration file.

Once the proxy to the `PojoCacheJmxWrapper` is obtained, the `getPojoCache()` will return a reference to the PojoCache itself.

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

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

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

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

Following is an abbreviated example `-beans.xml` file. The details of building up the Configuration are omitted; see the "Deploying JBoss Cache" chapter in the JBoss Cache User Guide for a more complete example. If you look in the `server/all/deploy` directory of an AS 5 installation, you can find several more examples.

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

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

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

    ... details omitted

  </bean>

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

    <constructor factoryClass="org.jboss.cache.pojo.PojoCacheFactory
```

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

   </bean>

</deployment>
```

An interesting thing to note in the above example is the difference between POJO Cache and a plain Cache in the use of a factory to create the cache. (See the "Deploying JBoss Cache" chapter in the JBoss Cache User Guide for the comparable plain Cache example.) The PojoCacheFactory exposes static methods for creating a PojoCache; as a result there is no need to add a separate `bean` element for the factory. Core Cache's `DefaultCacheFactory` creates caches from a singleton instance, requiring a bit more boilerplate in the config file.

5. POJO Cache MBeans

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

- Get a reference to the underlying PojoCache.
- Invoke create/start/stop/destroy lifecycle operations on the underlying PojoCache.
- See numerous details about the cache's configuration, and change those configuration items that can be changed when the cache has already been started.

See the `PojoCacheJmxWrapperMBean` javadoc for more details.

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

If a `PojoCacheJmxWrapper` is registered, the wrapper also registers MBeans for the underlying plain Cache and for each interceptor configured in the cache's interceptor stack. These MBeans are used to capture and expose statistics related to cache operations; see the JBoss Cache User Guide for more. They are hierarchically associated with

the `PojoCacheJmxWrapper` MBean and have service names that reflect this relationship. For example, a plain Cache associated with a `jboss.cache:service=PojoCache` will be accessible through an mbean named `jboss.cache:service=PojoCache,cacheType=Cache`. The replication interceptor MBean for that cache will be accessible through the mbean named `jboss.cache:service=PojoCache,cacheType=Cache,cache-interceptor=ReplicationInterceptor`.

6. Registering the PojoCacheJmxWrapper

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

6.1. Programatic Registration

Simplest way to do this is to create your `PojoCache` and pass it to the `PojoCacheJmxWrapper` constructor.

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

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

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

... use the cache

... on application shutdown

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

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

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

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

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

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

// ... use the cache

// ... on application shutdown

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

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

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

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

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

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

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

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

    ... build up the Configuration

  </bean>

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

    <constructor factoryClass="org.jboss.cache.pojo.PojoCacheFactory
                 factoryMethod="createCache">
       <parameter><inject bean="ExampleCacheConfig"/></parameter>
       <parameter>false</false>
    </constructor>

  </bean>

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

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

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

  </bean>

</deployment>
```

As discussed in the *Programatic Registration* section, `PojoCacheJmxWrapper` can do the work of building, creating and starting the PojoCache if it is provided with a `Configuration`:

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

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

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

    ... build up the Configuration

  </bean>

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

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

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

  </bean>

</deployment>
```

7. Runtime Statistics and JMX Notifications

As mentioned above, the cache exposes a variety of statistical information through its MBeans. It also emits JMX notifications when events occur in the cache. See the JBoss Cache User Guide for more on the statistics and notifications that are available.

The only PojoCache addition to the plain JBoss Cache behavior described in the User Guide is that you can register with the PojoCacheJmxWrapper to get the notifications. There is no requirement to figure out the ObjectName of the underlying cache's CacheJmxWrapper and register with that.

Instrumentation

In order to store an object in POJO Cache, it must be either instrumented or made serializable. Instrumentation is the most optimal approach since it preserves object identity and provides field granular replication. POJO Cache currently uses the JBoss AOP project to provide instrumentation, so the same processes described in the AOP documentation are used with POJO Cache.

The primary input to JBoss AOP is the AOP binding file, which is responsible for specifying which classes should be instrumented. POJO Cache provides a binding file, `pojocache-aop.xml` , which matches all classes that have been annotated with the `@Replicable` annotation. Advanced users may choose to alter this definition to instrument classes in other various interesting ways. However, it is recommended to just stick with the default annotation binding.

The instrumentation process can be executed at either load-time, or compile-time. Load-time instrumentation uses a Java agent to intercept and modify classes as they are loaded; whereas compile-time instrumentation requires running `aopc` as part of the compilation process.

Note

Load-time is the recommended approach, since compile-time instrumentation adds hard dependencies to the weaved bytecode which ties the output to a particular version of JBoss AOP.

1. Load-time instrumentation

Load-time instrumentation uses a Java agent to intercept all classes loaded by the JVM. As they are loaded JBoss AOP instruments them, allowing POJO Cache to monitor field changes. To enable load time instrumentation the JVM must be started with the following specified:

1. The `jboss.aop.path` system property set to the location of `pojocache-aop.xml`
2. A javaagent argument which includes *jboss-aop-jdk50.jar*

These requirements lead to the following example ant task:

```
<java classname="Foo" fork="yes">
  <jvmarg value="-javaagent:lib/jboss-aop.jar"/>
  <jvmarg value="-Djboss.aop.path=etc/META-INF/pojocache-aop.xml"/>
  <classpath refid="test.classpath"/>
</java>
```

There is also a *pojo-run* command line script in the POJO Cache distribution that passes the proper arguments to the JVM for you.

```
$ pojo-run -classpath myclasses org.foo.Bar
```

Once the JVM is executed in this manner, any class with the `@Replicable` annotation will be instrumented when it is loaded.

2. Compile-time instrumentation

While load-time is the preferred approach, it is also possible to instrument classes at compile-time. To do this, the aopc tool is used, with the following requirements:

1. The `aoppath` option must point to the location of pojocache-aop.xml
2. The `src` option must point to the location of your class files that are to be instrumented. This is typically the output folder of a `javac` run.

The following is an example ant task which performs compile-time instrumentation:

```
<taskdef name="aopc" classname="org.jboss.aop.ant.AopC" classpathref="aop.classpath"/>
<target name="aopc" depends="compile" description="Precompile aop class">
   <aopc compilerclasspathref="aop.classpath" verbose="true">
      <src path="${build}"/>
      <include name="org/jboss/cache/aop/test/**/*.class"/>
      <aoppath path="${output}/resources/pojocache-aop.xml"/>
      <classpath path="${build}"/>
      <classpath refid="lib.classpath"/>
   </aopc>
</target>
```

In this example, once the aopc target is executed the classes in the build directory are modified. They can then be packaged in a jar and loaded using the normal Java mechanisms.

3. Understanding the provided AOP descriptor

The advanced user might decide to alter the provided AOP descriptor. In order to do this, it is important to understand the reasons behind what is provided, and what is required by POJO Cache. Previous sections have mentioned that any class with the `@Replicable` annotation will be instrumented. This happens, because the provided AOP descriptor, `pojocache-aop.xml` , has a prepare statement which matches any class (or subclass) using the annotation. This is shown in the following snippet:

```
<prepare expr="field(* $instanceof{@org.jboss.cache.pojo.annotation.Replicable}->*)"/>
```

More specifically, any code which accesses a field on a class which has been annotated with `@Replicable` , will be instrumented: The "field" pointcut in the expression matches both read and write operations. The wildcard "*" indicates that all java protection modes are intercepted (private, package, protected, public). The `$instanceof` expression refers to any annotation that subclasses `@Replicable` . Finally, the ending wildcard allows the matched field to have any name.

TroubleShooting

We have maintained a *PojoCache wiki troubleshooting page* [http://wiki.jboss.org/wiki/Wiki.jsp?page=PojoCacheTroubleshooting]. Please refer it first. We will keep adding troubleshooting tips there.

All the current outstanding issues are documented in *JBossCache Jira page* [http://jira.jboss.com/jira/secure/BrowseProject.jspa?id=10051] . Please check it for details. If you have discovered additional issues, please report it there as well.

Appendix

1. Example POJO

The example POJO classes used for are: `Person`, `Student`, and `Address`. Below are their definition (note that neither class implements `Serializable`) along with the annotation.

```
@org.jboss.cache.pojo.annotation.Replicable
public class Person {
   String name=null;
   int age=0;
   Map hobbies=null;
   Address address=null;
   Set skills;
   List languages;

   public String getName() { return name; }
   public void setName(String name) { this.name=name; }

   public int getAge() { return age; }
   public void setAge(int age) { this.age = age; }

   public Map getHobbies() { return hobbies; }
   public void setHobbies(Map hobbies) { this.hobbies = hobbies; }

   public Address getAddress() { return address; }
   public void setAddress(Address address) { this.address = address; }

   public Set getSkills() { return skills; }
   public void setSkills(Set skills) { this.skills = skills; }

   public List getLanguages() { return languages; }
   public void setLanguages(List languages) { this.languages = languages; }
}
```

```
public class Student extends Person {
   String year=null;

   public String getYear() { return year; }
```

```
  public void setYear(String year) { this.year=year; }
}
```

```
  @org.jboss.cache.pojo.annotation.Replicable
  public class Address {
  String street=null;
  String city=null;
  int zip=0;

  public String getStreet() { return street; }
  public void setStreet(String street) { this.street=street; }
  // ...
}
```

2. Sample Cache configuration xml

Below is a sample xml configuration for Cache that you can use for PojoCache creation.

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

<server>
  <mbean code="org.jboss.cache.pojo.jmx.PojoCacheJmxWrapper"
       name="jboss.cache:service=PojoCache">

    <depends>jboss:service=TransactionManager</depends>

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

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

```
<!-- Valid modes are LOCAL, REPL_ASYNC and REPL_SYNC -->
<attribute name="CacheMode">REPL_SYNC</attribute>

<!-- Name of cluster. Needs to be the same for all caches,
    in order for them to find each other
-->
<attribute name="ClusterName">PojoCacheCluster</attribute>

<!-- JGroups protocol stack properties. -->
<attribute name="ClusterConfig">
  <config>
    <!-- UDP: if you have a multihomed machine, set the bind_addr
        attribute to the appropriate NIC IP address -->
    <!-- UDP: On Windows machines, because of the media sense feature
        being broken with multicast (even after disabling media sense)
        set the loopback attribute to true -->
    <UDP mcast_addr="228.1.2.3" mcast_port="48866"
        ip_ttl="64" ip_mcast="true"
        mcast_send_buf_size="150000" mcast_recv_buf_size="80000"
        ucast_send_buf_size="150000" ucast_recv_buf_size="80000"
        loopback="false"/>
    <PING timeout="2000" num_initial_members="3"/>
    <MERGE2 min_interval="10000" max_interval="20000"/>
    <FD shun="true"/>
    <FD_SOCK/>
    <VERIFY_SUSPECT timeout="1500"/>
    <pbcast.NAKACK gc_lag="50" retransmit_timeout="600,1200,2400,4800"
                max_xmit_size="8192"/>
    <UNICAST timeout="600,1200,2400",4800/>
    <pbcast.STABLE desired_avg_gossip="400000"/>
    <FC max_credits="2000000" min_threshold="0.10"/>
    <FRAG2 frag_size="8192"/>
    <pbcast.GMS join_timeout="5000" join_retry_timeout="2000"
            shun="true" print_local_addr="true"/>
    <pbcast.STATE_TRANSFER/>
  </config>
</attribute>

<!-- Whether or not to fetch state on joining a cluster -->
<attribute name="FetchInMemoryState">true</attribute>

<!-- The max amount of time (in milliseconds) we wait until the
    initial state (ie. the contents of the cache) are retrieved from
    existing members in a clustered environment
```

```
    -->
    <attribute name="InitialStateRetrievalTimeout">15000</attribute>

    <!-- Number of milliseconds to wait until all responses for a
         synchronous call have been received.
    -->
    <attribute name="SyncReplTimeout">15000</attribute>

    <!--  Max number of milliseconds to wait for a lock acquisition -->
    <attribute name="LockAcquisitionTimeout">10000</attribute>

  </mbean>
</server>
```

3. PojoCache configuration xml

Attached is a full listing for `pojocache-aop.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    This is the PojoCache configuration file that specifies:
      1. Interceptor stack for API
      2. Annotation binding for POJO (via "prepare" element)

    Basically, this is a variant of jboss-aop.xml. Note that
    except for the customization of interceptor stack, you should
    not need to modify this file.

    To run PojoCache, you will need to define a system property:
    jboss.aop.path that contains the path to this file such that JBoss Aop
    can locate it.
-->
<aop>

    <!--
    This defines the PojoCache 2.0 interceptor stack. Unless necessary, don't modify the stack
here!
    -->

    <!-- Check id range validity -->
    <interceptor name="CheckId" class="org.jboss.cache.pojo.interceptors.CheckIdInterceptor"
                 scope="PER_INSTANCE"/>
```

```
    <!-- Track Tx undo operation -->
   <interceptor name="Undo" class="org.jboss.cache.pojo.interceptors.PojoTxUndoInterceptor"
            scope="PER_INSTANCE"/>

    <!-- Begining of interceptor chain -->
    <interceptor name="Start" class="org.jboss.cache.pojo.interceptors.PojoBeginInterceptor"
            scope="PER_INSTANCE"/>

    <!-- Check if we need a local tx for batch processing -->
    <interceptor name="Tx" class="org.jboss.cache.pojo.interceptors.PojoTxInterceptor"
            scope="PER_INSTANCE"/>

    <!--
                            Mockup failed tx for testing. You will need to set
PojoFailedTxMockupInterceptor.setRollback(true)
       to activate it.
     -->
                                                <interceptor name="MockupTx"
class="org.jboss.cache.pojo.interceptors.PojoFailedTxMockupInterceptor"
            scope="PER_INSTANCE"/>

    <!-- Perform parent level node locking -->
                                                  <interceptor name="TxLock"
class="org.jboss.cache.pojo.interceptors.PojoTxLockInterceptor"
            scope="PER_INSTANCE"/>

    <!-- Interceptor to perform Pojo level rollback -->
                                                  <interceptor name="TxUndo"
class="org.jboss.cache.pojo.interceptors.PojoTxUndoSynchronizationInterceptor"
            scope="PER_INSTANCE"/>

    <!-- Interceptor to used to check recursive field interception. -->
                                                  <interceptor name="Reentrant"
class="org.jboss.cache.pojo.interceptors.MethodReentrancyStopperInterceptor"
            scope="PER_INSTANCE"/>

    <!-- Whether to allow non-serializable pojo. Default is false. -->
    <interceptor name="MarshallNonSerializable"
            class="org.jboss.cache.pojo.interceptors.CheckNonSerializableInterceptor"
            scope="PER_INSTANCE">
        <attribute name="marshallNonSerializable">false</attribute>
            </interceptor>
```

```
    <!-- This defines the stack macro -->
    <stack name="Attach">
      <interceptor-ref name="Start"/>
      <interceptor-ref name="CheckId"/>
      <interceptor-ref name="MarshallNonSerializable"/>
      <interceptor-ref name="Tx"/>
      <!-- NOTE: You can comment this out during production although leaving it here is OK. -->
      <interceptor-ref name="MockupTx"/>
      <interceptor-ref name="TxLock"/>
      <interceptor-ref name="TxUndo"/>
    </stack>

    <stack name="Detach">
      <interceptor-ref name="Start"/>
      <interceptor-ref name="CheckId"/>
      <interceptor-ref name="Tx"/>
      <!-- NOTE: You can comment this out during production although leaving it here is OK. -->
      <interceptor-ref name="MockupTx"/>
      <interceptor-ref name="TxLock"/>
      <interceptor-ref name="TxUndo"/>
    </stack>

    <stack name="Find">
      <interceptor-ref name="Start"/>
      <interceptor-ref name="CheckId"/>
    </stack>

    <!--
     The following section should be READ-ONLY!! It defines the annotation binding to the stack.
    -->

    <!-- This binds the jointpoint to specific in-memory operations. Currently in PojoUtil. -->
    <bind pointcut="execution(*
      @org.jboss.cache.pojo.annotation.Reentrant->toString())">
      <interceptor-ref name="Reentrant"/>
    </bind>

    <bind pointcut="execution(*
      org.jboss.cache.pojo.PojoUtil->@org.jboss.cache.pojo.annotation.TxUndo(..))">
      <interceptor-ref name="Undo"/>
    </bind>

                    <bind    pointcut="execution(*    org.jboss.cache.pojo.impl.PojoCacheImpl-
>@org.jboss.cache.pojo.annotation.Attach(..))">
```

```
      <stack-ref name="Attach"/>
    </bind>

                    <bind    pointcut="execution(*    org.jboss.cache.pojo.impl.PojoCacheImpl-
>@org.jboss.cache.pojo.annotation.Detach(..))">
      <stack-ref name="Detach"/>
    </bind>

                    <bind    pointcut="execution(*    org.jboss.cache.pojo.impl.PojoCacheImpl-
>@org.jboss.cache.pojo.annotation.Find(..))">
      <stack-ref name="Find"/>
    </bind>


    <!--
      Following is declaration for JDK50 annotation. You use the specific annotation on your
      POJO such that it can be instrumented. Idea is user will then need only to annotate like:
  @org.jboss.cache.pojo.annotation.Replicable
  in his POJO. There will be no need of jboss-aop.xml from user's side.
    -->

    <!-- If a POJO has PojoCachable annotation, it will be asepctized. -->
    <prepare expr="field(* $instanceof{@org.jboss.cache.pojo.annotation.Replicable}->*)" />

    <!-- Observer and Observable to monitor field modification -->
    <bind pointcut="
      set(* $instanceof{@org.jboss.cache.pojo.annotation.Replicable}->*)
      ">
        <interceptor class="org.jboss.cache.pojo.observable.SubjectInterceptor"/>
    </bind>

    <introduction class="$instanceof{@org.jboss.cache.pojo.annotation.Replicable}">
      <mixin>
        <interfaces>org.jboss.cache.pojo.observable.Subject</interfaces>
        <class>org.jboss.cache.pojo.observable.SubjectImpl</class>
        <construction>new org.jboss.cache.pojo.observable.SubjectImpl(this)</construction>
      </mixin>
    </introduction>
  </aop>
```