Preview Release—send your feedback to peter.veentjer@hazelcast.com

THE BOOK OF Hazelcast

by PETER VEENTJER

Preface

Writing concurrent system has been a long passion of mine and it is a very logical step to go from concurrency control within a single JVM to concurrency control over multiple JVM's. There is a big overlap in functionality; a lot of the knowledge that is applicable to concurrency control in a single JVM also applies to concurrency over multiple JVM's; but there also is a whole new dimension of problems that make distributed systems even more interesting to deal with.

What is Hazelcast

When you write applications for the JVM for your profession, it is likely that you are going to write server-side applications. Although Java has support for writing desktop applications, the server-side is really where Java shines.

Today, especially with the introduction of cloud computing, it becomes more and more important that server-side systems are:

1. Scalable: just add and remove machines to match required capacity
2. Highly available: if one or more machines in a system fail, the system should continue as if nothing happened.
3. High performing: the performance per machine should be good enough to make it cost efficient.

Hazelcast is an Open Source clustering and highly scalable data distribution platform for the JVM. It is:

1. Dynamically scalable: This is done by making certain Hazelcast data-structures, like Hazelcast Map, partitioned. So that partitions can be spread evenly among the members. When members join or leave the cluster, Hazelcast will automatically move partitions from one member to another.
2. Highly available: It does not lose data after a JVM crash. This is done by automatically replicating partition data on other cluster members. In case

of a member going down, the system will automatically failover by restoring the backup. Another important design feature of Hazelcast is that there is no master member that can form a single point of failure; each member has equal responsibilities.
3. Lightning-fast: Each Hazelcast member can do thousands of operations per second.

Hazelcast will not automatically spawn additional JVM's to become members in the cluster when the load exceeds a certain upper threshold because this is very environment specific so it will not fit you in a once size fits all solution. For the same reason it will not shutdown JVM's when the load exceeds a certain lower threshold.

One of the things I like most about Hazelcast is that it isn't very intrusive; as a developer/architect you are in control how much Hazelcast you get in your system. You are not forced to mutilate objects so they can be distributed, forced to use specific (application) servers or complex api's or the need to install software; just add the Hazelcast jar to your classpath and you are done.

This freedom combined with very well thought out API's, in a lot of cases you can just use interfaces like the java Executor, BlockingQueue or Map, makes Hazelcast really a joy to work with. So it helps you with implementing highly available, scalable and high-performing systems, written in little time and based on very simple and elegant code.

Who should read this book

This books aims at developers/architects that build applications on top of the JVM and want to get a better understanding of how to write distributed applications using Hazelcast. It doesn't matter if you are using Java or any other the other JVM based languages like Scala, Groovy, Clojure. It is even possible to call Hazelcast from .NET or C++ using the new Hazelcast 3 Portable and client functionality.

If you are a developer that has no prior experience with Hazelcast, then you will learn the basics to get up and running. If you already have some experience, it might be that you learn some new tricks since the book contains a lot of information that is not (yet) part of the Hazelcast manual.

What is in this book

This book shows you how to make use of Hazelcast by going through most important features. It also includes the newest Hazelcast 3 improvements. Some of these improvements are minor changes, but can have a huge impact on a system. Others are very big like the SPI which makes it possible to write your own distributed data-structures if you are not happy with the ones provided by Hazelcast.

In 'Chapter 1: Learning the Basics', you will learn how to download and set up Hazelcast and to create a basic project. You will also learn about some of the general Hazelcast concepts.

In 'Chapter 2: Distributed Primitives', you will learn how to use basic concurrency primitives like ILock, IAtomicLong, IdGenerator, ISemaphore and ICountDownLatch and about their advanced settings.

In 'Chapter 3: Distributed Collections', you will learn how to make use of distributed collections like the IQueue, IList and ISet.

In 'Chapter 4: Distributed Map', you will learn about the IMap functionality. Since its functionality is very extensive, there is a whole topic about dealing with its configuration options like high availability, scalability etc. You will also learn how to use Hazelcast as a cache and persist its values.

In 'Chapter 5: Distributed Executor', you will learn about executing tasks using the distributed Executor. By using the executor you turn Hazelcast into a computing grid.

In 'Chapter 6: Distributed Topic', you will learn about creating a publish/subscribe solution using the Distributed Topic functionality.

In 'Chapter 7: Hazelcast clients', you will learn about connecting to a Hazelcast cluster as a client. This topic not only deals with creating a client but also with more complex features like load-balancing and failover.

In 'Chapter 8: Serialization', you will learn more about the different serialization technologies that are supported by Hazelcast. Not only Java Serializable and Externalizable will be explained, but also the native Hazelcast serialization techniques like DataSerializable and the new Portable functionality.

In 'Chapter 9: Transactions', you will learn about Hazelcast's transaction support to prevent transactional data-structures from being left in inconsistent state.

After that in 'Chapter 10: Network Configuration', you will learn about Hazelcast's network configuration. You will learn about different member discovery mechanism like multicast, Amazon EC2 and security.

Finally in 'Chapter 12: Performance', you will learn more about performance tuning. Out of the box a lot data-structures like the map have all kinds of default settings that are perhaps good for certain situations, like high availability, but could be impacting your performance.

Online resources

There is a website for this book that contains a link to an interactive discussion forum and you can submit your errata to the book here as well. Also the Java source code and the configuration files can be found here.

The Hazelcast website and various other useful sites can be found here:

1. Hazelcast Website: http://hazelcast.com/
2. Hazelcast Documentation: http://hazelcast.com/docs.jsp
3. Hazelcast Usergroup: http://groups.google.com/group/hazelcast
4. Hazelcast on Github: https://github.com/hazelcast/hazelcast/

Building distributed systems on Hazelcast is really joy to do and I hope I can make you as enthusiastic about it as I am. So lets get started with building distributed applications you can be proud of.

Contents

Chapter 1

Getting started

In this chapter we'll learn the basic steps to getting started; like downloading Hazelcast, configuring Hazelcast in a Maven project and checking out the Hazelcast sources to be able to build yourself.

1.1 Installing Hazelcast

Hazelcast relies on Java 6 or higher. So if you want to compile the examples, make sure Java 6 or higher is installed. If not installed, it can be downloaded from the Oracle site: http://java.com/en/download/index.jsp.

For this book we rely on the community edition of Hazelcast 3.2-SNAPSHOT which can be downloaded from http://www.hazelcast.com/downloads.jsp. If your project uses Maven, there is no need to install Hazelcast at all, see [link to Hazelcast and Maven]. Otherwise make sure that the Hazelcast jar is added to the classpath. Apart from this jar, there is no need to install Hazelcast. The lack of an installation process for Hazelcast is something I really like because it saves quite a lot of time, time that can be used to solve real problems instead of environmental ones.

1.2 Hazelcast and Maven

Hazelcast is very easy to include in your Maven 3 project without needing to go through a complex installation process. Hazelcast can be found in the standard Maven repositories, so you do not need to add additional repositories to the pom. To include Hazelcast in your project, just add the following to your pom:

```
<dependencies>
   <dependency>
      <groupId>com.hazelcast</groupId>
```

```
      <artifactId>hazelcast</artifactId>
      <version>3.2-SNAPSHOT</version>
   </dependency>
</dependencies>
```

That is it. Make sure that you check the Hazelcast website to make use of the most recent version. After this dependency is added, Maven will automatically download the dependencies needed.

If you want to make use the latest snapshot, you need to add the snapshot repository to your pom

```
<repositories>
   <repository>
      <id>snapshot-repository</id>
      <name>Maven2 Snapshot Repository</name>
      <url>https://oss.sonatype.org/content/repositories/snapshots
         </url>
   </repository>
</repositories>
```

The latest snapshot is updated as soon as a change is merged in the Git repository. Using a snapshot can be useful if you need to work with the latest and greatest, but it could be that a snapshot version contains bugs.

1.3 Download examples

The examples used in the book can be accessed from this website: https://github.com/hazelcast/haze book-examples.

And if you want to clone the Git repository, just type:

```
git clone https://github.com/hazelcast/hazelcast.git
```

The examples are very useful to get started and see how something works. The examples are modules within a Maven project and can be build using:

```
mvn clean install
```

Each example has one or more bash scripts to run it. But personally I prefer to run them from my IDE.

1.4 Building Hazelcast

If you want to build Hazelcast yourself, e.g. because you want to provide a bugfix, debug, see how things work, add new features etc, you can clone the Git repository (download the sources) using git.

```
git clone https://github.com/hazelcast/hazelcast.git
```

The master branch contains the latest 3.x code.

To build the Hazelcast project, execute the following command:

```
mvn clean install -Pparallel-test
```

This will build all the jars and run all the tests; which can take some time. If you don't want to execute all the tests, use the following command:

```
mvn clean install -DskipTests
```

If you want to create patches for Hazelcast, you want to fork the Hazelcast Git repository and add the official Hazelcast repository as an upstream repository. If you have a change you want to offer to the Hazelcast team, you commit and push the changes to your own forked repository and create a pull request that will be reviewed by the Hazelcast team. Once your pull request is verified it will be merged and a new snapshot will automatically appear in the Hazelcast snapshot repository.

1.5 What is next

Now that we have checked out the sources and have installed the right tools, we can get started with building amazing Hazelcast applications.

Chapter 2

Learning the basics

In this chapter we'll learn the basic steps to getting started like starting Hazelcast instances, how to load an configure DistributedObjects, configure logging etc.

2.1 Configuring Hazelcast

Hazelcast can be configured in different ways:

1. Programmatic configuration
2. XML configuration
3. Spring configuration

The programmatic configuration is the most essential one, the other mechanisms are build on top of it. In this book we'll use the xml configuration file since that is most often used in production.

When you are running a Maven project; just create a resources directory under src/main/ and create a file 'hazelcast.xml'. The following shows an empty configuration:

```
<hazelcast
   xsi:schemaLocation="http://www.hazelcast.com/schema/config
      http://www.hazelcast.com/schema/config/hazelcast-config-3.0.
         xsd"
   xmlns="http://www.hazelcast.com/schema/config"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
</hazelcast>
```

This example imports an XML schema (XSD) for validation. If you are using an IDE, you probably get code completion. To reduce the size of the examples in the book, only the elements inside the ¡hazelcast¿ tags are listed. In the example code for the book you can find the full xml configuration. Another thing you might run into is the strange formatting of the java code; this is also done to reduce the size.

In most of our examples we will rely on multicast for member discovery so that the members will join the cluster:

```
<network>
   <join><multicast enabled="true"/></join>
</network>
```

See [chapter Cluster Configuration: Multicast] if multicast doesn't work or you want to know more about it. If you are using the programmatic configuration, then multicast is enabled by default.

In this book, the following approach is used to create a new Hazelcast instance:

```
public class Main {

   public static void main(String[] args){
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      ...
   }
}
```

Behind the scenes the following alternatives are used to resolve the configuration:

1. First checks if the 'hazelcast.config' system property is set; if it is, then the value is used as path. This is useful if you want to choose the application which hazelcast configuration file should be used on startup. The config option can be set by adding the following to the java command: '-Dhazelcast.config=¡path to the hazelcast.xml¿'. The value can be a normal file path, but can also be a classpath reference if it is prefixed with 'classpath:'.
2. Else it checks if there is a 'hazelcast.xml' in the working directory.
3. After that it check if there is a 'hazelcast.xml' on the classpath.
4. Finally loads the default hazelcast configuration 'hazelcast-default.xml' that is part of the Hazelcast jar

Also be careful to check the Hazelcast logging when you are relying on an hazelcast.xml file. If it contains errors, Hazelcast will not abort the startup but will default to the 'hazelcast-default.xml'. When this happens, the system could behave completely different than you would expect.

If you need more flexibility to load a Hazelcast config object from XML, you should have a look at:

1. ClasspathXmlConfig: a Config loaded from a classpath resource containing the XML configuration

   ```
   Config config = new ClasspathXmlConfig("hazelcast.xml");
   HazelcastInstance hz = Hazelcast.newHazelcastInstance(config);
   ```

2. FileSystemXmlConfig: a Config loaded from a file

```
Config config = new FileSystemXmlConfig("hazelcast.xml");
HazelcastInstance hz = Hazelcast.newHazelcastInstance(config);
```

3. InMemoryXmlConfig: a Config loaded from an in memory string containing the XML configuration.

```
String s = "<hazelcast>....</hazelcast>"
Config config = new InMemoryXmlConfig(s);
HazelcastInstance hz = Hazelcast.newHazelcastInstance(config);
```

4. UrlXmlConfig: a Config loaded from an url pointing to a XML file.

```
Config config = new UrlXmlConfig("http://foo/hazelcast.xml");
HazelcastInstance hz = Hazelcast.newHazelcastInstance(config);
```

All these Config subclasses rely on the XmlConfigBuilder:

```
Config config = new XmlConfigBuilder().build();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(config);
```

Another option to load a HazelcastInstance, is to make use of programmatic configuration, e.g:

```
public class Main {

   public static void main(String[] args){
      ExecutorConfig executorConfig = new ExecutorConfig()
        .setName("someExecutor")
        .setPoolSize(10);
      Config config = new Config().addExecutorConfig(
         executorConfig);
      HazelcastInstance hz = Hazelcast.newHazelcastInstance(config
         );
      ...
   }
}
```

The Hazelcast Config object has a fluent interface; meaning that the Config instance is returned when a config method on this instance is called. This makes chaining method calls very easy. The programmatic configuration is not only very useful for testing, but it also a solution for the static nature of the XML configuration. The content of the programmatic configuration can easily be created on the fly, e.g. based on database content. You could even decide to move the 'static' configuration to the hazelcast.xml, load it and then modify the dynamic parts, e.g. the network configuration.

In Hazelcast releases prior to 3.0, there was functionality for a static default HazelcastInstance, so you could say: 'Queue q = Hazelcast.getQueue("foo")'. This functionality has been removed because it lead to confusion when explicitly created HazelcastInstances are combined with calls to the implicit default HazelcastInstance. So you probably want to keep a handle to the Hazelcast instance somewhere for future usage.

Same configuration: Hazelcast will not copy configuration from one member to another. Therefore it is very important that the configuration on all members in the cluster is exactly the same: it doesn't matter if you use the XML based configuration or the programmatic configuration. If there are differences between configurations it could lead to problems. Either Hazelcast will detect it or the Distributed Objects are running with a different configuration depending on how the member was configured.

2.2 Multiple Hazelcast instances

In most cases you will have a single Hazelcast Instance per JVM. But Multiple Hazelcast Instances can run in a single JVM. This is not only useful for testing, but also needed for more complex setups e.g. application servers running multiple independent applications using Hazelcast. Multiple Hazelcast instances can be started like this:

```
public class MultipleMembers {

    public static void main(String[] args){
        HazelcastInstance hz1 = Hazelcast.newHazelcastInstance();
        HazelcastInstance hz2 = Hazelcast.newHazelcastInstance();
    }
}
```

When you start this MultipleMembers, you see something like this in the output for one member

```
Members [2] {
    Member [192.168.1.100]:5701 this
    Member [192.168.1.100]:5702
}
```

And something like this for the other member:

```
Members [2] {
    Member [192.168.1.100]:5701
    Member [192.168.1.100]:5702 this
}
```

As you can see the created cluster has 2 members.

2.3 Loading a DistributedObject

In the previous sections we saw how a HazelcastInstance can be created, but in most cases you want to load a DistributedObject, e.g. a queue, from this HazelcastInstance. So lets define a queue in the hazelcast.xml:

```
<queue name="q"/>
```

And the queue can be loaded like this:

```
public class Member {

   public static void main(String[] args) throws Exception{
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      IQueue<String> q = hz.getQueue("q");
   }
}
```

For most of the DistributedObjects you can find a get method on the HazelcastInstance. In case you are writing custom distributed DistributedObjects using the SPI [reference to SPI chapter], you can use the HazelcastInstance.getDistributedObject. One thing worth mentioning is that most of the DistributedObjects defined in the configuration are created lazily; so they are only created on the first operation that accesses them.

If there is no explicit configuration available for a DistributedObject, Hazelcast will use the default settings from the file 'hazelcast-default.xml'. This means that you can safely load a DistributedObject from the HazelcastInstance without it being explicitly configured.

To learn more about the queue and its configuration check [reference to Distributed Collections: Queue].

2.4 Unique names for Distributed Objects

Some of the DistributedObjects will be static; they will be created and used through the application and the id's of these objects will be known up front. Other DistributedObjects are created on the fly. One of the problems is finding unique names when new DataStructures need to be created. One of the solutions to this problem is to make use of the IdGenerator which will generate cluster wide unique id's.

```
HazelcastInstance hz = Hazelcast.newHazelcastInstance();
```

```
IdGenerator idGenerator = hz.getIdGenerator("idGenerator");
IMap someMap = hz.getMap("somemap-"+idGenerator.newId());
```

This technique can be used with wildcard configuration to create similar objects using a single definition [reference to wildcard configuration].

A distributed object created with a unique name often needs to be shared between member. This can be done by passing the id to that other members and using one of the HazelcastInstance.get methods, the DistributedObject can be retrieved. For more information see [reference to serialization chapter:Serialize DistributedObject]

In Hazelcast the name and the type of the DistributedObject uniquely identify that object:

```
IAtomicLong atomicLong = hz.getAtomicLong("a");
IMap map = hz.getMap("a");
```

In this example 2 different DistributedObject are created with the same name, but different types. In normal applications you want to prevent that different types of DistributedObjects share the same name. I often add the type to the name e.g. 'personMap' or 'failureCounter' to make the names self explanatory.

2.5 Reloading a DistributedObject

In most cases once you have loaded the DistributedObject, you probably keep a reference to it and inject it all places where it is needed. But you can safely reload the same DistributedObject from the HazelcastInstance without additional instances being created if you only have the name. In some cases, like deserialization, when you need to get reference to a Hazelcast DistributedObject, this is the only solution. If you have a Spring background, you could consider the configuration to be singleton bean definition.

2.6 Destroying a DistributedObject

DistributedObject can be destroyed using the DistributedObject.destroy() method which clears and releases all resources for this object within the cluster. But it should be used with a lot of care because of the following reason: once the destroy method is called and the resources are released, a subsequent load with the same

id from the HazelcastInstance will result in a new data-structure and will not lead to an exception.

A similar issues happens to references: if a reference to a DistributedObject is used after the DistributedObject is destroyed, new resources will be created. In the following case we create a cluster with 2 members, and each member gets a reference to the queue q. First we place an item in the queue. When the queue is destroyed by the first member (q1) and q2 is accessed, a new queue will be created.

```
public class Member {

   public static void main(String[] args) throws Exception {
      HazelcastInstance hz1 = Hazelcast.newHazelcastInstance();
      HazelcastInstance hz2 = Hazelcast.newHazelcastInstance();
      IQueue<String> q1 = hz1.getQueue("q");
      IQueue<String> q2 = hz2.getQueue("q");
      q1.add("foo");
      System.out.println("q1.size: "+q1.size()+
          " q2.size:"+q2.size());
      q1.destroy();
      System.out.println("q1.size: "+q1.size()+
          " q2.size:"+q2.size());
    }
}
```

When we start this Member, the output will show the following:

```
q1.size: 1 q2.size:1
q1.size: 0 q2.size:0
```

So there are no errors, the system will behave as if nothing happened, the only difference is that the new queue resources have been created. Therefore a lot of care needs to be taken into consideration when destroying DistributedObjects.

2.7 Wildcard configuration

The Hazelcast xml configuration can contain configuration elements for all kinds of distributed data-structures: sets, executors, maps etc. For example:

```
<map name="testmap">
   <time-to-live-seconds>10</time-to-live-seconds>
</map>
```

But what if we want to create multiple map instances using the same configuration? Do we need to configure them individually? This is impossible to do if you have a dynamic number of distributed data-structures and you don't know up front how many need to be created. The solution to this problem is wildcard configuration,

which is available for all data-structures. This makes it possible to use the same configuration for multiple instances. For example, we could configure the previous 'testmap' example with a 10 'time-to-live-seconds' using a wildcard configuration like this:

```
<map name="testmap*">
   <time-to-live-seconds>10</time-to-live-seconds>
</map>
```

Using a single asterisk (*) character any place in the name, the same configuration can be shared by different data-structures. The wildcard configuration can be used like this:

```
   Map map1 = hz.getMap("testmap1");
   Map map2 = hz.getMap("testmap2");
```

The maps 'testmap1' and 'testmap2' both match 'testmap*' so they will use the same configuration. If you have a Spring background, you could consider the wildcard configuration to be a prototype bean definition although the difference is that in Hazelcast multiple gets of a data-structure with the same id will still result in the same instance and with prototype beans new instances are returned.

An important thing you need to watch out for are ambiguous configurations, e.g.:

```
<map name="m*">
   <time-to-live-seconds>10</time-to-live-seconds>
</map>
<map name="ma*">
   <time-to-live-seconds>10</time-to-live-seconds>
</map>
```

If a map is loading using 'hz.getMap("map")' then Hazelcast will not throw an error or log a warning but instead selects one of them. The selection doesn't depend on the definition order in the configuration file and also isn't based on the best fitting match. So make sure that your wildcard configurations are very specific. One of the ways to do it is to include the package name:

```
<map name="com.foo.testmap*">
    <time-to-live-seconds>10</time-to-live-seconds>
</map>
```

A map can be loaded by calling 'Map map = hi.getMap("com.foo.testmap1")'.

2.8 Controlled partitioning

There are 2 types of DistributedObjects in Hazelcast. One type is the truly partitioned data-structure, like the IMap, where each partition will store a section of the Map. The other type is a non partitioned data-structure, like the IAtomicLong or the ISemaphore, where only a single partition is responsible for storing the main instance. For this type, you sometimes want to control that partition.

Normally Hazelcast will not only use the name of a DistributedObject for identification, but also uses it to determine the partition. The problem is that you sometimes want to control the partition without depending on the name. Imagine that you have the following 2 semaphores:

```
ISemaphore s1 = hz.getSemaphore("s1");
ISemaphore s2 = hz.getSemaphore("s2");
```

They would end up in different partitions, because they have different names. Luckily Hazelcast provides a solution for that using the '@' symbol, e.g.:

```
ISemaphore s1 = hz.getSemaphore("s1@foo");
ISemaphore s2 = hz.getSemaphore("s2@foo");
```

Now s1 and s2 will end up in the same partition because they share the same partition key: 'foo'. This partition-key can not only be used to control the partition of DistributedObjects, but can also be used to send a Runnable to the correct member using IExecutor.executeOnKeyOwner method [see Executor: executeOnKeyOwner] or to control in which partition a map entry is stored [see Map: Partition Control].

If a DistributedObject name includes a partition-key, then Hazelcast will use the base-name, so without the partition-key, to match with the configuration. For example semaphore s1 could be configured using:

```
<semaphore name="s1">
   <initial-permits>3</initial-permits>
</semaphore>
```

This means that you can safely combine explicit partition keys with normal configuration. It is important to understand that the name of the DistributedObject will contain the @partition-key section. So the following 2 semaphores are different:

```
ISemaphore s1 = hz.getSemaphore("s1@foo");
ISemaphore s2 = hz.getSemaphore("s1");
```

To access the partition-key of a DistributedObject, the 'DistributedObject.getPartitionKey'

method can be called, e.g.:

```
String parKey = s1.getPartitionKey();
ISemaphore s3 = hz.getSemaphore("s3@"+parKey);
```

This is useful if you need to create a DistributedObject in the same partition of an existing DistributedObject, but you don't have the partition-key available. If you only have the name of the partition-key available, you can have a look at the PartitionKeys class which exposes methods to retrieve the base-name or the partition-key.

In the previous examples the 'foo' partition-key was used. In a lot of cases you don't care what the partition-key is, as long as the same partition-key is shared between structures. Hazelcast provides an easy solution to obtain a random partition key:

```
String parKey = hz.getPartitionService().randomPartitionKey();
ISemaphore s1 = hz.getSemaphore("s1@"+parKey);
ISemaphore s2 = hz.getSemaphore("s2@"+parKey);
```

You are completely free to come up with a partition-key yourself. You can have a look at the UUID, although due to its length it will cause some overhead. Another options is to look at the Random class. The only thing you need to watch out for is that the partition-keys are evenly distributed among the partitions.

If the @ is used in the name of a partitioned DistributedObject like the IMap or the IExecutorService, then Hazelcast keeps using the full String as the name of the DistributedObject, but ignores the partition-key since for these types a partition-key doesn't have any meaning.

For more information about why you want to control partitioning, see [chapter performance: partitioning schema]

2.9 Properties

Hazelcast provides an option to configure certain properties which are not part of an explicit configuration section like the Map. This can be done using the properties section:

```
<properties>
   <property name="hazelcast.icmp.enabled">true</property>
</properties>
```

For a full listing of available properties see the 'Advanced configuration' chapter of the Hazelcast reference manual or have a look at the GroupProperties class.

Apart from properties in the hazelcast.xml, they can also be passed on the commandline 'java -Dproperty-name=property-value'. One thing to watch out for; you can't override properties in the hazelcast.xml/programmatic-configuration from the command line because the latter has a lower priority.

Properties are not shared between members; so you can't put properties in one member and read them from the next. You need to use a distributed map for that.

2.10 Variables

One of the new features of Hazelcast 3 is the ability to specify variables in the Hazelcast xml configuration file. This makes it a lot easier to share the same hazelcast configuration between different environment and .. it also makes it easier to tune settings.

Variables can be used like this:

```
<executor-service name="exec">
   <pool-size>${pool.size}</pool-size>
</executor-service>
```

In this example the pool-size is configurable using the 'pool.size' variable. In a production environment you might want to set it so a high value since you have beefier machines there and in a development environment you might want to use a low value.

By default Hazelcast will use the System properties to replace variables by their actual value. To pass this system property, you could add the following on the commandline: -Dpool.size=1. If a variable is not found, a log warning will be displayed but the value will not be replaced.

It also is possible to use a different mechanism than the System properties, e.g. a property file or a database. This can be done by explicitly setting the Properties object on the XmlConfigBuilder:

```
Properties properties = new Properties();
properties.setProperty("pool.size","10");
Config config = new XmlConfigBuilder()
        .setProperties(properties)
        .build();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(config);
```

Also the Config subclasses like FileSystemXmlConfig accept Properties in their

constructors.

Although variables will give you a lot more flexibility, it has its limitations since you can only parametrize but you can't add new xml sections. If your needs go beyond what the variables provide, it might be an idea to use some kind of template engine like Velocity to generate your hazelcast.xml file. Another option is use programmatic configuration either by creating a completely new Config instance or loading a 'template' from XML and enhance where needed.

2.11 Logging

Hazelcast supports various logging mechanisms; 'jdk', 'log4', 'sl4j' or 'none' if you don't want to have any logging. The default is 'jdk': the logging library that is part of the JRE, so no additional dependencies are needed. Logging can be set by adding a property in the hazelcast.xml:

```
<properties>
   <property name="hazelcast.logging.type">log4j</property>
</properties>
```

Or if you are using the programmatic configuration:

```
Config cfg = new Config() ;
cfg.setProperty("hazelcast.logging.type", "log4j");
```

It can also be configured from the command line using: 'java -Dhazelcast.logging.type=log4j'. If you are going to use 'log4j' or 'slf4j', make sure that the correct dependencies are included in the classpath. See the example sources for more information.

If you are not satisfied with the provided logging implementations, you can always implement your own logging implementation by implementing the LogListener interface. See the Hazelcast reference manual for more information.

Warning: if you are not making use of configuring logging from the command-line, be very careful of touching Hazelcast classes. It could be that they default to the 'jdk' logging before the actual configured logging is read. Once the logging mechanism is selected, it will not change. Personally I prefer to make use of the command-line version and don't use the properties section for logging because it causes confusion.

If you are making use of 'jdk' logging and you are annoyed by having your log entry spread over 2 lines, have a look at the SimpleLogFormatter, e.g.

```
java.util.logging.SimpleFormatter.format='%4$s: %5$s%6$s%n'
```

2.12 Good to know

Hazelcast config not updatable: once a HazelcastInstance is created, the Config that was used to create that HazelcastInstance should not be updated. A lot of the internal configuration objects are not thread-safe and there is no guarantee that a property is going to be read after it has been read for the first time.

HazelcastInstance.shutdown(): if you are not using your HazelcastInstance anymore, make sure to shut it down by calling the 'shutdown()' method on the HazelcastInstance. This will release all its resources and end network communication.

Hazelcast.shutdownAll(): This method is very practical for testing purposes if don't have control on the creation of the HazelcastInstances, but want to make sure that all the instances are being destroyed.

What happened to the Hazelcast.getDefaultInstance: If you have been using Hazelcast 2.x, you might wonder what happened to static methods like the Hazelcast.getDefaultInstance and Hazelcast.getSomeStructure. These methods have been dropped because they rely on a singleton HazelcastInstance and if this is combined with explicit HazelcastInstances, it caused confusion. In Hazelcast 3 it is only possible to work with an explicit HazelcastInstance.

2.13 What is next?

In this chapter you saw how a HazelcastInstance instance is created, how it can be configured and how DistributedObject are created. In the following chapters you learn about all the different DistributedObjects like the ILock, IMap etc and the configuration details.

Chapter 3

Distributed Primitives

If you have programmed applications in Java you have probably worked with concurrency primitives like the synchronized statement (the intrinsic lock) or perhaps even the concurrency library that was introduced in Java 5 under java.util.concurrent like the Executor, Lock and AtomicReference.

This concurrency functionality is useful if you want to write a Java application that use multiple threads, but the focus is to provide synchronization in a single JVM and not distributed synchronization over multiple JVM's. Luckily Hazelcast provides support for various distributed synchronization primitives like the ILock, IAtomicLong etc. Apart from making synchronization between different JVM's possible, they also support high availably; so if one machine fails, the primitive remains usable for other JVM's.

3.1 IAtomicLong

The IAtomicLong, formally known as the AtomicNumber, is the distributed version of the java.util.concurrent.atomic.AtomicLong, so if you have used that before, working with the IAtomicLong should feel very similar. The IAtomicLong exposes most of the operations the AtomicLong provides like get, set, getAndSet, compareAndSet and incrementAndGet, but there is of course a big difference in performance since remote calls are involved.

I'll demonstrate the IAtomicLong by creating an instance and incrementing it one million times:

```
public class Member {
   public static void main(String[] args) {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      IAtomicLong counter = hz.getAtomicLong("counter");
      for (int k = 0; k < 1000 * 1000; k++) {
```

```
            if (k % 500000 == 0){
                System.out.println("At: "+k);
            }
            counter.incrementAndGet();
        }
        System.out.printf("Count is %s\n", counter.get());
        System.exit(0);
    }
}
```

If you start this Member, you will see this:

```
At: 0
At: 500000
Count is 1000000
```

If you run multiple instances of this member, then the total count should be equal to one million times the number of members you have started.

If the IAtomicLong becomes a contention point in your system, there are a few ways of dealing with it, depending on your requirements. One of the options it to create a stripe (essentially an array) of IAtomicLong instances to reduce pressure. Another options is to keep changes local and only publish them to the IAtomicLong once a while. There are a few downsides here; you could loose information if a member goes down and the newest value is not always immediately visible to the outside world.

3.1.1 Functions

Since Hazelcast 3.2 it is possible to send a function to an IAtomicLong. The Function class is a single method interface; which is part of the Hazelcast codebase since we can't yet have a dependency on Java 8. An example of a function implementation is the following function which doubles the original value:

```
private static class Add2Function implements Function<Long,Long> {
    @Override
    public Long apply(Long input) {
        return input+2;
    }
}
```

The function can be executed on an IAtomicLong using one of the following 4 methods:

1. apply: applies the function to the value in the IAtomicLong without changing the actual value and returning the result.

2. alterAndGet: alters the value stored in the IAtomicLong by applying the function, storing the result in the IAtomicLong and returning the result.
3. getAndAlter: alters the value stored in the IAtomicLong by applying the function and returning the original value.
4. alter: alters the value stored in the IAtomicLong by applying the function. This method will not send back a result.

In the following code example you can see these methods in action:

```
public class Member {
    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IAtomicLong atomicLong = hz.getAtomicLong("counter");

        atomicLong.set(1);
        long result = atomicLong.apply(new Add2Function());
        System.out.println("apply.result:" + result);
        System.out.println("apply.value:" + atomicLong.get());

        atomicLong.set(1);
        atomicLong.alter(new Add2Function());
        System.out.println("alter.value:"+atomicLong.get());

        atomicLong.set(1);
        result = atomicLong.alterAndGet(new Add2Function());
        System.out.println("alterAndGet.result:" + result);
        System.out.println("alterAndGet.value:"+atomicLong.get());

        atomicLong.set(1);
        result = atomicLong.getAndAlter(new Add2Function());
        System.out.println("getAndAlter.result:"+result);
        System.out.println("getAndAlter.value:"+atomicLong.get());

        System.exit(0);
    }
}
```

When we execute this program, we'll see the following output:

```
apply.result:3
apply.value:1
alter.value:3
alterAndGet.result:3
alterAndGet.value:3
getAndAlter.result:1
getAndAlter.value:3
```

You might ask yourself the question why not do the following approach to double an IAtomicLong:

```
atomicLong.set(atomicLong.get()+2));
```

This requires a lot less code. The biggest problem here is that this code has a raceproblem; the read and the write of the IAtomicLong are not atomic. So it

could be that they are interleaved with other operations. If you have experience with the AtomicLong from Java, then you probably have some experience with the compareAndSet method where you can create an atomic read and write:

```
for(;;){
    long oldValue = atomicLong.get();
    long newValue = oldValue+2;
    if(atomicLong.compareAndSet(oldValue,newValue)){
        break;
    }
}
```

The problem here is that the atomiclong, could be on a remote machine and therefor in get and compareAndSet are remote operations. With the function approach, you send the code to the data instead of pulling the data to the code and therefor is a lot more scalable.

3.1.2 Good to know

1. replication: the IAtomicLong has 1 synchronous backup and zero asyncronous backups and is not configurable.

3.2 IAtomicReference

3.3 IdGenerator

In previous section the IAtomicLong was introduced and one of the things it can be used for is to generated unique id's within a cluster. Although it will work, it probably isn't the most scalable solution since all member will content on incrementing the value. If you are only interested in unique id's you can have a look at the com.hazelcast.core.IdGenerator.

The way the IdGenerator works is that each member claims a segment of 1 million id's to generate. This is done behind the scenes by using an IAtomicLong and claiming a segment is done by incrementing that IAtomicLong by a million. After claiming the segment, the IdGenerator can increment a local counter. Once all id's in the segment are used, it will claim a new segment. The consequence of this approach is that only 1 in a million times network traffic is needed; so 999.999 out of 1.000.000 the id generation can be done in memory and therefor is extremely fast. Another consequence is that this approach scales a lot better than an IAtomicLong

because there is a lot less contention: 1 out of 1.000.000 instead of 1 out of 1.

So lets see the IdGenerator in action:

```
public class IdGeneratorMember {
   public static void main(String[] args) throws Exception{
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      IdGenerator idGenerator = hz.getIdGenerator("id");
      for (int k = 0; k < 1000; k++){
         Thread.sleep(1000);
         System.out.printf("Id : %s\n", idGenerator.newId());
      }
   }
}
```

If you start this multiple times, you will see in the console that there will not be any duplicate id's. If you do see duplicates, it could be that the IdGeneratorMembers didn't form a cluster [see Network configuration:Multicast]

Of course there are some issues you need to be aware of:

1. id's generated by different members will be out of order
2. if a member goes down without fully using its segment, there might be gaps.

For id generation, in most cases, this isn't relevant. There are alternative solutions for creating cluster wide unique id's like the java.util.UUID. Although it will take up more space than a long, it doesn't rely on access to a Hazelcast cluster.

Another important issue you need to know is that if the cluster restarts, then the IdGenerator is reset and starts from 0 because the IdGenerator doesn't persist its state using e.g. a database. If you need this, you could create your own IdGenerator based on the same implementation mechanism the IdGenerator uses, but you persist the updates to the IAtomicLong.

By default, the id generation will start at 0, but in some case you want to start with a higher value. This can be changed using the IdGenerator.init(long value) method. It returns true if the initialization was a success; so if no other thread called the init method, no id's have been generated and the desired starting value is bigger than 0.

3.3.1 Good to know

1. replication: the IdGenerator has 1 synchronous backup and zero asyncronous backups and is not configurable.

3.4 ILock

A lock is a synchronization primitive that makes it possible that only a single thread is able to access to a critical section of code; if multiple threads at the same moment would access that critical section concurrently, you would get race problems.

Hazelcast provides a distributed lock implementation and makes it possible to create a critical section within a cluster of JVM's; so only a single thread from one of the JVM's in the cluster is allowed to acquire that lock. Other threads that want to acquire the lock, no matter if they are on the same JVM's or not, will not be able to acquire it and depending on the locking method they called, they either block or fail. The com.hazelcast.core.ILock extends the java.util.concurrent.locks.Lock interface, so using the lock is quite simple.

The following example shows how a lock can be used to solve a race problem:

```
public class RaceFreeMember {
   public static void main(String[] args) throws Exception {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      IAtomicLong number1 = hz.getAtomicLong("number1");
      IAtomicLong number2 = hz.getAtomicLong("number2");
      ILock lock = hi.getLock("lock");
      System.out.println("Started");
      for (int k = 0; k < 10000; k++) {
         if (k % 100 == 0)
            System.out.println("at: " + k);
         lock.lock();
         try {
            if (k % 2 == 0) {
               long n1 = number1.get();
               Thread.sleep(10);
               long n2 = number2.get();
               if (n1 - n2 != 0)
                  System.out.println("Datarace detected!");
               } else {
                     number1.incrementAndGet();
                     number2.incrementAndGet();
               }
         } finally {
           lock.unlock();
         }
      }
      System.out.println("Finished");
   }
}
```

When this code is executed; you will not see "Datarace detected!". This is because

the lock provides a critical section around writing and reading of the numbers. In the example code you will also find the version with a data race.

The following idiom is recommended when using a Lock (doesn't matter if it is a Hazelcast lock or lock provided by the JRE):

```
lock.lock();
try{
    ...do your stuff.
}finally{
    lock.unlock();
}
```

It is important that the Lock is acquired before the try/finally block is entered. So the following example is not good:

```
try{
    lock.lock();
    ...do your stuff.
}finally{
    lock.unlock();
}
```

In case of Hazelcast it can happen that the lock is not granted because the lock method has a timeout of 5 minutes. If this happens, an exception is thrown and the finally block is executed and the lock.unlock is called. Hazelcast will see that the lock is not acquired and an IllegalMonitorStateException with the message "Current thread is not owner of the lock!" is thrown. In case of a tryLock with a timeout, the following idiom is recommended:

```
if(!lock.tryLock(timeout, timeunit)){
   throw new RuntimeException();
}
try{
    ...do your stuff.
}finally{
    lock.unlock();
}
```

As you can see the tryLock is acquired outside of the try/finally block. In this case an exception is thrown if the lock can't be acquired within the given timeout, but another flow that prevents entering the try/finally block also is valid.

A few things worth knowing about the Hazelcast lock and locking in general:

1. It is reentrant, so you can acquire it multiple times in a single thread without causing a deadlock, of course you need to release it as many times as you have acquired it, to make it available to other threads.
2. Just like with the other Lock implementations, it should always be acquired

outside of a try/finally block. Else it can happen that the lock acquire failed, but an unlock is still executed.

3. Keep locks as short as possible. If locks are kept too long, it can lead to performance problems or worse: deadlock.
4. With locks it is easy to run into deadlocks if you don't know what you are doing; so make sure that you do. Having code you don't control running inside your locks is asking for problems. Make sure you understand exactly the scope of the lock.
5. To reduce the chance of a deadlock, the Lock.tryLock methods can be used to control the waiting period. The lock.lock() method will not block indefinitely, but will timeout with a OperationTimeoutException after 300 seconds.
6. Locks are automatically released when a member has acquired a lock and this member goes down. This prevents threads that are waiting for a lock to wait indefinitely and needed for failover to work in a distributed system. The downside however is that if a member goes down that acquired the lock and started making changes, other members could start to see partial changes. In these cases either the system could do some self repair or else a transaction might solve the problem.
7. A lock must always be released by the same thread that acquired it, otherwise look at the ISemaphore.
8. Locks are fair, so they will be granted in the order they are requested.
9. There are no configuration options available for the lock
10. A lock cab be checked if it is locked using the ILock.isLocked method, although the value could be stale as soon as it is returned.
11. A lock can be forced to unlock using the ILock.forceUnlock() method. It should be used with extreme care since it could break a critical section.
12. The Hazelcast.getLock doesn't work on name of type String, but can be key of any type. This key will be serialized and the byte array content determines the actual lock to acquire. So if you are passing in an object as key, it isn't the monitor lock of that object that is being acquired.
13. replication: the ILock has 1 synchronous backup and zero asyncronous backups and is not configurable.

3.5 ICondition

With a Condition it is possible to wait for certain conditions to happen, e.g. to wait for an item to be placed on a queue. Each lock can have mulitple conditions e.g.

item available in queue and room available in queue. In Hazelcast 3 the ICondition, which extends the java.util.concurrent.locks.Condition, has been added.

There is 1 difference, with the normal Java version you create a condition using the Lock.newCondition() method. Unfortunately this doesn't work in a distributed environment since Hazelcast has no way of knowing if Conditions created on different members are the same Condition or not. Since you don't want to rely on the order of their creation. That is why in Hazelcast a Condition needs to be created using the ILock.newCondition(String name) method.

In the following example we are going to create one member that waits for a counter to have a certain value. And another member will set the value on that counter. So lets get started with the waiting member:

```
public class WaitingMember {
   public static void main(String[] args) throws
       InterruptedException {
       HazelcastInstance hz = Hazelcast.newHazelcastInstance();
       IAtomicLong counter = hz.getAtomicLong("counter");
       ILock lock = hz.getLock("lock");
       ICondition isOneCondition = lock.newCondition("one");
       lock.lock();
       try {
          while (counter.get() != 1) {
             System.out.println("Waiting");
             isOneCondition.await();
          }
       } finally {
          lock.unlock();
       }
       System.out.println("Wait finished, counter: "+counter.get())
          ;
   }
}
```

As you can see the lock is acquired first and then in a loop the is counter is checked. As long as the counter is not 1, the waiter will wait on the isOneCondition. Once the isOneCondition.await() method is called, Hazelcast will automatically release the lock so that a different thread can acquire it and the calling thread will block. Once the isOneCondition is signaled, the thread will unblock and it will automatically reacquire the lock. This is exactly the same behavior as the ReentrantLock/Condition or with a normal intrinsic lock and waitset. If the WaitingMember is started, it will output:

```
Waiting
```

The next part will be the NotifyMember where the Lock is acquired, the value set

to 1 and the isOneCondition will be signaled:

```
public class NotifyMember {
   public static void main(String[] args) throws
       InterruptedException {
       HazelcastInstance hz = Hazelcast.newHazelcastInstance();
       IAtomicLong counter = hz.getAtomicLong("counter");
       ILock lock = hz.getLock("lock");
       ICondition isOneCondition = lock.newCondition("isOne");
       lock.lock();
       try {
          counter.set(1);
          isOneCondition.signalAll();
       } finally {
          lock.unlock();
       }
   }
}
```

After the NotifyMember is started, the WaitingMember will display:

```
Waiting
Wait finished, counter: 1
```

A few things worth knowing about the ICondition:

1. just as with the normal Condition, the ICondition can suffer from spurious wakeups. That is why the condition always needs to be checked inside a loop, instead of an if statement.
2. you can choose to signal only a single thread instead of all thread by calling the ICondition.signal() method instead of the ICondition.signalAll() method.
3. in the example the waiting thread waits indefinitely because it calls 'await()'. In practice this can be undesirable since a member that is supposed to signal the condition, can fail. When this happens, the threads that are waiting for the signal, wait indefinitely. That is why it often is a good practice to wait with a timeout using the 'await(long time, TimeUnit unit)' or 'awaitNanos(long nanosTimeout)' method.
4. waiting threads or signaled in FIFO order.
5. replication: the ICondition has 1 synchronous backup and zero asyncronous backups and is not configurable.

3.6 ISemaphore

The semaphore is a classic synchronization aid that can be used to control the number of threads doing a certain activity concurrently, e.g. using a resource. Conceptually each semaphore has a number of permits, where each permit repre-

sents a single thread allowed to execute that activity concurrently. As soon as a thread want to start with the activity, it takes a permit (or waits until one becomes available) and once finished with the activity, the permit is returned.

If you initialize the semaphore with a single permit, it looks a lot like a lock. One of the big difference is that the Semaphore has no concept of ownership. So with a lock; the thread that acquired the lock must release it, but with a semaphore any thread can release an acquired permit. Another difference is that an exclusive lock only has 1 permit and a semaphore can have more than 1.

Hazelcast provides a distributed version of the java.util.concurrent.Semaphore named the com.hazelcast.core.ISemaphore. When a permit is acquired on the ISemaphore, the following can happen:

1. A permit is available. The number of permits in the semaphore is decreased by one and the calling thread can continue.
2. No permit is available. The calling thread will block until a permit comes available, a timeout happens, the thread is interrupted or when the semaphore is destroyed an InstanceDestroyedException is thrown.

I'll explain the semaphore with an example. To simulate a shared resource we have an IAtomicLong initialized with the value 0. This resource is going to used 1000 times, When a thread starts to use that resource it will be incremented it and once completed it will be decremented.

```
public class SemaphoreMember {
   public static void main(String[] args)throws Exception{
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      ISemaphore semaphore = hz.getSemaphore("semaphore");
      IAtomicLong resource = hz.getAtomicLong("resource");
      for(int k=0;k<1000;k++){
         System.out.println("At iteration: "+k +
            ", Active Threads: " + resource.get());
         semaphore.acquire();
         try{
            resource.incrementAndGet();
            Thread.sleep(1000);
            resource.decrementAndGet();
         }finally{
            semaphore.release();
         }
      }
      System.out.println("Finished");
   }
}
```

We want to limit the concurrent access to the resource by allowing for at most 3 thread. This can be done by configuring the initial-permits for the semaphore in

the Hazelcast config file:

```
<semaphore name="semaphore">
   <initial-permits>3</initial-permits>
</semaphore>
```

When you start the SemaphoreMember 5 times you will see output like this:

```
At iteration: 0, Active Threads: 1
At iteration: 1, Active Threads: 2
At iteration: 2, Active Threads: 3
At iteration: 3, Active Threads: 3
At iteration: 4, Active Threads: 3
```

As you can see the maximum number of concurrent threads using that resource always is equal or smaller than 3. As an experiment you can remove the semaphore acquire/release statements and see for yourself that there is no longer control on the number of concurrent usages of the resources.

3.6.1 Replication

Hazelcast provides replication support for the ISemaphore. This means that if a member that goes and replication is enabled (by default it is), that another member take over the semaphore without permit information getting lost. This can be done by synchronous and asynchronous replication which can be configured using the backup-count and async-backup-count properties:

1. backup-count: the number of synchronous replica's and defaults to 1.
2. async-backup-count: the number of asynchronous replica's and defaults to 0.

I high performance is more important than permit information getting lost, you might consider setting backup-count to 0.

A few things worth knowing about the ISemaphore:

1. fairness: the Semaphore acquire methods are fair and this is not configurable. So under contention, the longest waiting thread for a permit will acquire it before all other threads. This is done to prevent starvation, at the expense of reduced throughput.
2. automatic permit release: one of the features of the ISemaphore to make it more reliable in a distributed environment, is the automatic release of a permit when the member fails (similar as with the Hazelcast Lock). If the permit would not be released the system could run in a deadlock.
3. the acquire() method doesn't timeout, unlike the Hazelcast Lock.lock() method. To prevent running into a deadlock, using one of timed acquire methods like ISemaphore.tryAcquire(int permits, long timeout, TimeUnit unit) is a good

solution.

4. the initial-permits is allowed to be negative, meaning that there is a shortage of permits when the semaphore is created.

3.7 ICountDownLatch

The java.util.concurrent.CountDownLatch was introduced in Java 1.5 and is a synchronization aid that makes it possible for threads to wait until a set of operations, being performed by one or more threads to complete. Very simplistically; a CountDownLatch could be seen as a gate containing a counter. Behind this gate, threads can wait till the counter reaches 0. In my experience CountDownLatches often are used when you have some kind of processing operation, and one or more threads need to wait till this operation completes so they can execute their logic. Hazelcast also contains a CountDownLatch; the org.hazelcast.core.ICountDownLatch.

To explain the ICountDownLatch, imagine that there is a leader process that is executing some action and eventually completes. Also imagine that there are one or more follower processes that need to do something after the leader has completed. We can implement the behavior of the Leader:

```
public class Leader{
   public static void main(String[] args)throws Exception{
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      ICountDownLatch latch = hz.getCountDownLatch("latch");
      System.out.println("Starting");
      latch.trySetCount(1);
      Thread.sleep(5000);
      latch.countDown();
      System.out.println("Leader finished");
      latch.destroy();
   }
}
```

The Leader retrieves the CountDownLatch, calls ICountDownLatch.trySetCount on it which makes him owner of that latch, does some waiting and then calls countdown; which notifies the listeners for that latch. Currently we ignore the boolean return value of trySetCount since there will be only a single Leader, but in practice you probably want deal with the return value. Although there will only be a single owner of the Latch, the countDown method can be called by other threads/processes.

The next part is the Follower:

```
public class Follower {
```

```
    public static void main(String[] args) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        ICountDownLatch latch = hz.getCountDownLatch("latch");
        System.out.println("Waiting");
        boolean success = latch.await(10, TimeUnit.SECONDS);
        System.out.println("Complete:"+success);
    }
}
```

As you can see we first retrieve the ICountDownLatch and then call await on it so the thread listens for the ICountDownLatch to reach 0. In practice it can happen than a process that should have decremented the counter by calling the ICountDownLatch.countDown method, fails and therefore the ICountDownLatch will never reach 0. To force you to deal with this situation, there is no await method without a timeout to prevent waiting indefinitely.

If we first start a leader and then one or more followers, the followers will wait till the leader completes. It is important that the leader is started first, else the followers will immediately complete since the latch already is 0. The example show a ICountDownLatch with only a single step. But if a process has n steps, initialize the ICountdownLatch with n instead of 1 and for each completed step call the countDown method.

One thing to watch out for is that an ICountDownLatch waiter can be notified prematurely. In a distributed environment the leader could go down before it has reached zero and this would result in the waiters to wait till the end of time. Because this behavior is undesirable, Hazelcast will automatically notify all listeners if the owner gets disconnected. So it can be that listeners are notified before all steps of a certain process are completed. To deal with this situation the current state of the process needs to be verified and appropriate actions need to be taken e.g. restart all operations, continue with the first failed operation, or throw an exception.

Although the ICountDownLatch is a very useful synchronization aid, it probably isn't one you will use on a daily basis. Unlike Java's implementation, Hazelcast's ICountDownLatch count can be re-set after a countdown has finished but not during an active count.

3.7.1 Good to know

1. replication: the ICountDownLatch has 1 synchronous backup and zero asyncronous backups and is not configurable.

3.8 Good to know

Cluster Singleton Service: In some cases you need a thread that will only run on a limited number of members. Often it is only a single thread that is needed. But if the member running this thread fails, another machine needs to take over. Hazelcast doesn't have direct support for this, but it is very easily to implement using an ILock (for a single thread) or using an ISemaphore (for multiple threads).

On each cluster member you start this service thread, but the first thing this service needs to do is to acquire the lock or a license and on success the thread can starts with its logic. All other threads will block till the lock is released or a license is returned.

The nice thing about the ILock and the ISemaphore is when a member exists the cluster (crash, network disconnect etc), that automatically the lock is released and the license returned. So other cluster members that are waiting to acquire the lock/license, can now have their turn.

3.9 What is next?

In this chapter we looked at various synchronization primitives that are supported by Hazelcast. If for whatever reason you need a different one you can try to build it on top of existing ones, or create a custom one using the Hazelcast SPI. One of the things I would like to see added the ability to control the partition the primitive is living on since this would improve locality of reference.

Chapter 4

Distributed Collections

Hazelcast provides a set of collections that implement interfaces from the Java collection framework and therefor make it easy to integrate distributed collections in your system without making too many code changes. A distributed collection can not only be called concurrently from the same JVM, it also can be called concurrently by different JVM's. Another advantage is that the distributed collections provide high availability, so if a member hosting the collection fails, another member will take over.

4.1 IQueue

A BlockingQueue is one of the work horses for concurrent system because it allows producers and consumers of messages, which can be POJO's, to work at different speeds. The Hazelcast com.hazelcast.core.IQueue, which extends the java.util.concurrent.BlockingQueue, not only allows threads from the same JVM to interact with that queue, but since the queue is distributed, it also allows different JVM's to interact with it. So you can add items in one JVM and remove them in another.

As an example we'll create a producer/consumer implementation that is connected by a distributed queue. The producer is going to put a total of 100 Integers on the queue with a rate of 1 message/second.

```
public class ProducerMember {
    public static void main(String[] args) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IQueue<Integer> queue = hz.getQueue("queue");
        for (int k = 1; k < 100; k++) {
            queue.put(k);
            System.out.println("Producing: " + k);
            Thread.sleep(1000);
        }
```

```
        queue.put(-1);
        System.out.println("Producer Finished!");
    }
}
```

To make sure that the consumers are going to terminate when the producer is finished, the producer will put a -1 on the queue to indicate that it is finished.

The consumer will take the message from the queue, print it and waits 5 seconds before consuming the next message and stops when it receives the -1, also called a poison pill:

```
public class ConsumerMember {
    public static void main(String[] args) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IQueue<Integer> queue = hz.getQueue("queue");
        while (true){
            int item = queue.take();
            System.out.println("Consumed: " + item);
            if(item == -1){
                queue.put(-1);
                break;
            }
            Thread.sleep(5000);
        }
        System.out.println("Consumer Finished!");
    }
}
```

If you take a closer look at the consumer, you see that when the consumer receives the poison pill, it puts the poison pill back on the queue before it ends the loop. This is done to make sure that all consumer will also receive the poison pill, and not only the one that received it first.

When you start a single producer, you will see the following output:

```
Produced 1
Produced 2
....
```

When you start a single consumer, you will see the following output:

```
Consumed 1
Consumed 2
....
```

As you can see, the items produced on the queue by the producer are being consumed from that same queue by the consumer.

Because messages are produced 5 times faster than they are consumed, with a

single consumer the queue will keep growing. To improve throughput, you can start more consumers. If we start another one, we'll see each consumer takes care of half the messages. Consumer 1:

```
Consumed 20
Consumed 22
....
```

Consumer 2:

```
Consumed 21
Consumed 23
....
```

When you kill one of the consumers, the remaining consumer will process all elements again:

```
Consumed 40
Consumed 42
....
```

One thing to take care of that if there are many producers/consumers interacting with the queue, there will be a lot of contention and eventually the queue will become the bottleneck. One way of solving this problem is to introduce a stripe (essentially a list) of queues. But if you do, the ordering of messages send to different queues will not be guaranteed anymore. In a lot of cases a strict ordering isn't required and a stripe can be a simple solution to improve scalability.

Important: Realize that although the Hazelcast distributed queue preserves ordering of the messages (so the messages are taken from the queue in the same order they were put on the queue), if there are multiple consumers, the processing order is not guaranteed because the queue will not provide any ordering guarantees on messages after they are taken from the queue.

4.1.1 Capacity

In the previous example we showed a basic producer/consumer solution based on a distributed queue. Because the production of messages is separated from the consumption of messages, the speed of production is not influenced by the speed of consumption. If producing messages goes quicker than the consumption, then the queue will increase in size. If there is no bound on the capacity of the queue, then machines can run out of memory and you will get an OutOfMemoryError.

With the traditional BlockingQueue implementations, like the LinkedBlockingQueue,

a capacity can be set. When this is set and the maximum capacity is reached, placement of new items either fail or block, depending on type of the put operation. This prevents the queue from growing beyond a healthy capacity and the JVM from failing.

The Hazelcast queue also provides capacity control, but instead of having a fixed capacity for the whole cluster, Hazelcast provides a scalable capacity by setting the queue capacity per member using the queue property max-size. So if the capacity per member is 1000 and there are 5 members's, the total capacity is 5000. Therefor the capacity depends on the size of the cluster. To give our queue a capacity of 10 per member, we set the max-size:

```
<network>
    <join><multicast enabled="true"/></join>
</network>
<queue name="queue">
    <max-size>10</max-size>
</queue>
```

When we start a single producer, we'll see that 10 items are put on the queue and then the producer blocks. If we then start a single consumer, we'll immediately see that the producer will continue since the total capacity for the queue has doubled to 20 (2 JVM's times 10 items per JVM).

But since the producer produces 5 times as fast as the consumer, the queue will quickly reach its maximum capacity again and it will block. We can increase the capacity of the cluster by starting new consumers (both processing and the storage capacity increase) or just empty members (the storage capacity increases).

4.1.2 Backups

By default Hazelcast will make sure that there is one synchronous backup for the queue; so if the member hosting that queue fails, the backups on another member will be used so no entries are lost.

Backups can be controlled using the properties

1. backup-count: the number of synchronous backups, defaults to 1. So by default no entries will be lost if a member fails.
2. async-backup-count: the number of asynchronous backups, defaults to 0.

If you want increased high availability you can either increase the backup-count or the async-backup-count. If you want to have improved performance you could set the backup-count to 0, at the costs of potentially loosing entries on failure.

4.1.3 QueueStore

By default Hazelcast data-structures like the IQueue are not persistent:

1. If the cluster starts, the queues will not be populated by themselves.
2. Changes in the queue will not be made persistent, so if the cluster fails then entries will be lost.

In some cases this behavior is not desirable and luckily Hazelcast provide a mechanism for queue durability using the QueueStore which can connected to a more durable storage mechanism like a database for example. In Hazelcast 2 the Queue was implemented on top of the Hazelcast Map, so in theory you could make the queue persistent by configuring the MapStore of the backing map. In Hazelcast 3, the Queue is not implemented on top of a map anymore but luckily exposes a QueueStore directly.

4.2 IList

A List is a collection where every element only occurs ones and where the order of the element does matter. The Hazelcast com.hazelcast.core.IList implements the java.util.List. We'll demonstrate the IList by adding items to a list on one member and we print the element of that list on another member:

```
public class WriteMember {
   public static void main(String[] args) {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      IList<String> list = hz.getList("list");
      list.add("Tokyo");
      list.add("Paris");
      list.add("New York");
      System.out.println("Putting finished!");
   }
}
public class ReadMember {
   public static void main(String[] args) {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      IList<String> list = hz.getList("list");
      for (String s : list)
         System.out.println(s);
      System.out.println("Reading finished!");
   }
}
```

If you first run the WriteMember and after it has completed, start the ReadMember then the ReadMember will output the following:

```
Tokyo
Paris
```

```
New York
Reading finished!
```

As you can see, the data written to the List by the WriteMember is visible in the ReadMember and you also can see that the order is maintained. The List interface has various methods like the sublist that returns collections, it is important to understand that the returned collections are snapshots and are not backed up the by list. See [reference 'weak consistency' iterators at end of chapter]

4.3 ISet

A Set is a collection where every element only occurs once and where the order of the elements doesn't matter. The Hazelcast com.hazelcast.core.ISet implements the java.util.Set. I'll demonstrate the set by adding items in a Set on one member, and on another member we are going to print all the elements from that Set:

```
public class WriteMember {
   public static void main(String[] args) {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      ISet<String> set = hz.getSet("set");
      set.add("Tokyo");
      set.add("Paris");
      set.add("New York");
      System.out.println("Putting finished");
   }
}

public class ReadMember {
   public static void main(String[] args) {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      ISet<String> set = hz.getSet("set");
      for(String s: set)
         System.out.println(s);
      System.out.println("Reading finished!");
   }
}
```

If you first start the WriteMember and waiting for completion, you start the ReadMember; it will output the following:

```
Paris
Tokyo
New York
Reading finished!
```

As you can see, the data added by the WriteMember is visible in the ReadMember. As you also can see, the order is not maintained since order is not defined by the Set.

Just as with normal HashSet, the hashcode() and equals() method of the object are used and not the equals/hash of the byte array version of that object. This is different behaviour compared to the map; see [reference to equals/hash section in the map]

In Hazelcast the ISet (and same goes for the IList) is implemented as a collection within the MultiMap, where the id of the set is the key in the MultiMap and the value is the collection. This means that the ISet is not partitioned, so you can't scale beyond the capacity of a single machine and you can't control the partition where data from a set is going to be stored. If you want to have a distributed set that behaves more like the distributed map, one simple option is to implement a set based on a map, where the value can be some bogus value. It isn't possible to rely on the Map.keySet for returning a usable distributed set since it will return a non distributed snapshot of the keys.

4.4 Collection ItemListener

The IList, ISet and IQueue interfaces extend the com.hazelcast.core.ICollection interface. The nice thing is that Hazelcast enriches the existing collections api with the ability to listen to changes in the collections using the com.hazelcast.core.ItemListener. The ItemListener receives the ItemEvent which not only potentially contains the item, but also the member where the changed happened and the type of event (add or remove).

The following example shows an ItemListener that listens to all changes made in an IQueue:

```
public class ItemListenerMember {
   public static void main(String[] args) throws Exception {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      ICollection<String> q = hz.getQueue("queue");
      q.addItemListener(new ItemListenerImpl<String>(), true);
      System.out.println("ItemListener started");
   }

   private static class ItemListenerImpl<E>
         implements ItemListener<E> {

      @Override
      public void itemAdded(ItemEvent<E> e) {
         System.out.println("Item added:" + e.getItem());
      }

      @Override
```

```
        public void itemRemoved(ItemEvent<E> e) {
            System.out.println("Item removed:" + e.getItem());
        }
    }
}
```

We registered the ItemListenerImpl with the addItemListener method using the value 'true'. This is done to make sure that our ItemListenerImpl will get the value that has been added/removed. The reason this configuration option is available, is that in some cases you only want to be notified that a change happened, but you're not interested in the actual change and don't want to pay for sending the value over the line.

To see that the ItemListener really is working, we'll create a member that makes a change in the queue:

```
public class CollectionChangeMember{
    public static void main(String[] args) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        BlockingQueue<String> q = hz.getQueue("queue");
        q.put("foo");
        q.put("bar");
        q.take();
        q.take();
    }
}
```

First start up the ItemListenerMember and wait till it displays "ItemListener started". After that start the CollectionChangeMember and you will see the following output in the ItemListenerMember:

```
item added:foo
item added:bar
item removed:foo
item removed:bar
```

ItemListeners are useful if you need to react upon changes in collections. But realize that listeners are executed asynchronously, so it could be that at the time your listener runs, the collection has changed again.

Ordering All events are ordered, which means that listeners will receive and process the events in the order they are actually occurred.

4.5 Good to know

Iterator stability: Iterators on collections are weakly consistent; meaning that when a collection changes while creating the iterator, you could encounter duplicates or miss element. Changes on that iterator will not result in changes on the collection. An iterator doesn't need to reflect the actual state and will not throw a ConcurrentModifcationException.

Not Durable: In Hazelcast 2 the IQueue/IList/ISet were build on top of the Hazelcast distributed map and by accessing that map you could influence the collection's behavior including storage. This isn't possible anymore in Hazelcast 3. The IQueue does have its own QueueStore mechanism, but the List/Set have not. Perhaps this will be added in a later release.

Replication: The replication for IList and ISet can't be configured and will automatically have 1 synchronous backup and 0 asynchronous backups. Perhaps in the future this is going to be configurable.

Destruction: IQueue/ISet/IList instances are immediately 'destroyed' when they are empty and will not take up space. Listeners will remain registered, unless that collection is destroyed explicitly. Once an item is added to implicit destroyed collection, the collection will automatically be recreated.

No merge policy for the Queue: if a clusters containing some queue is split, then each subcluster is still able to access their own view of that queue. If these subclusters merge, the queue can't be merged and one of them is deleted.

Not partitioned: the IList/ISet/IQueue are not partitioned, so maximum size of the collection doesn't rely on the size of the cluster, but on the capacity of a single member since the whole queue will be kept in the memory of a single JVM.

This is a big difference compared to Hazelcast 2.x where they were partitioned. The Hazelcast team decided to drop this behavior since the 2.x implementation was not truly partitioned do to reliance on a single member where a lot of metadata for the collection was stored. This limitation needs to be taken into consideration when you are designing a distributed system. A few ways to solve this issue are to use a stripe of collections or to build your collection on top of the IMap. Another more flexible but probably more time consuming alternative is to write the collection on top of the new SPI functionality [reference to SPI chapter]

A potential solution for the IQueue, is to make a stripe of queues instead of a single queue. Since each collection in that stripe is likely to be assigned to a different partition than its neighbours, the queues will end up in different members. If ordering of items is not important, the item can be placed on an arbitrary queue. Else the right queue could be selected based on some property of the item; so that all items having the same property end up in the same queue.

Uncontrollable partition: It currently isn't possible to control the partition the collection is going to be placed on and this means that more remoting is required than strictly needed. In the future this will be possible so you can say:

```
String partitionKey = "foobar";
IQueue q1 = hz.getQueue(partitionKey,"q1");
IQueue q2 = hz.getQueue(partitionKey,"q2");
```

In this case q1 and q2 are going to be stored in the same partition.

4.6 What is next?

In this chapter we have seen various collections in action and we have seen how they can be configured.

Chapter 5

Distributed Map

In this chapter, you'll learn how to use one of the most versatile data structures in Hazelcast; the com.hazelcast.core.IMap. The IMap extends the java ConcurrentMap and therefor it also extends java.util.Map. The big difference between a normal Map implementation like the HashMap and the Hazelcast IMap implementation is that the later is a distributed data-structure.

Internally Hazelcast divides the map in partitions, and distributes the partitions evenly among the members in the cluster. The partition of map entry is based on the key of that entry: each key belongs to a single partition. By default Hazelcast will use 271 partitions for all partitioned data-structures. This value can be changed with the 'hazelcast.map.partition.count' property.

When a new member is added, the oldest members in the cluster decides which partitions are going to be moved to that new member. Once the partitions have been move, this member will take its share in the load. So to scale up a cluster, just add new members to the cluster.

When a member is removed, all the partitions that members owns, are moved to other members. So scaling down a cluster is simple, just remove members from the cluster. Apart from a 'soft' removal of the member, it can also happen that there is a 'hard' removal; e.g. the member crashes or it gets disconnected from the cluster due to network issues. Luckily Hazelcast provides various degrees of failover to deal with this situation. By default there will be one synchronous backup; so the failure of a single member will not lead to loss of data because a replica of that data is available on another member.

[todo: is this with elastic memory or just normal] On Youtube there is demo: http://www.youtube.com/watch?v=TOhbhKqJpvw where 4 Terabyte of data from 1 billion entries is stored on 100 Amazon EC2 instances, supporting to 1.3 million

of operations/second.

5.1 Creating a Map

Creating a distributed Map is very simple as the following example shows:

```
public class FillMapMember {

    public static void main(String[] args) {
        HazelcastInstance hzInstance = Hazelcast.
           newHazelcastInstance();
        Map<String, String> map = hzInstance.getMap("cities");
    }
}
```

In this example we create a basic 'cities' map which we'll use in the next sections.

You don't need to configure anything in the hazelcast.xml file; Hazelcast will use the 'default' Map configuration from the 'hazelcast-default.xml' to configure that map. If you do want to configure the map, you can use the follow example as a minimal map configuration in the hazelcast.xml:

```
<map name="cities"/>
```

Lazy creation: the Map is not created when the 'getMap' method is called. Only when the map instance is accessed, it will be created. This is useful to know if you are making use of the DistributedObjectListener and fail to receive creation events.

5.2 Reading/Writing

The Hazelcast Map implements the ConcurrentMap interface, so reading/writing key/values is very simple since you can use familiar methods like get/put etc.

To demonstrate this basic behaviour, the following Member creates a Map and writes some entries in that map:

```
public class FillMapMember {

    public static void main(String[] args) {
        HazelcastInstance hzInstance = Hazelcast.
           newHazelcastInstance();
        Map<String, String> map = hzInstance.getMap("map");
        map.put("1", "Tokyo");
        map.put("2", "Paris");
```

```
        map.put("3", "New York");
    }
}
```

As you can see the Map is retrieved using the hzInstance.getMap(mapName) and after that some entries are stored in that map. Reading the entries from that map also is very simple:

```
public class PrintAllMember {

    public static void main(String[] args) {
        HazelcastInstance hzInstance = Hazelcast.
           newHazelcastInstance();
        Map<String, String> map = hzInstance.getMap("map");
        for(Map.Entry<String,String> entry : map.entrySet())
            System.out.println(entry.getKey()+" "+entry.getValue()
               );
    }
}
```

If we first run the FillMapMember and then run the PrintAllMember, we get the following output:

```
3 New York
1 Tokyo
2 Paris
```

As you can see, the map updates from the FillMapMember are visible in the PrintAllMember.

Internally Hazelcast will serialize the key/values (see chapter Serialization) to byte arrays and store this in the underlying storage mechanism. This means changes made to a key/value after they are stored in the map, will not reflect on the stored state. Therefor the following idiom is broken:

```
Employee e  = employees.get(123);
e.setFired(true);
```

If you want this change to be stored in the map, you need to put the updated value back:

```
Employee e  = employees.get(123);
e.setFired(true);
employees.put(123,e);
```

5.3 InMemoryFormat

The IMap is a distributed data-structure, so a key/value can be read/written on a different machine than where the actual content is stored. To make this

possible, Hazelcast serialises the key/value to byte-arrays when they are is stored and deserializes the key/value when they are loaded. A serialized representation of an object is called the binary-format. For more information about serialization of keys/values see [chapter serialization].

The problem with storing an object in binary-format, is when queries (predicates) and entry-processors read the same value multiple times, because for each read, the object stored in binary-format, needs to be deserialized. This can have a big impact on performance. For these cases it would be better if the object would not be stored in binary-format, but in object-format: so the value instance and not to a byte-array.

That is why the IMap provides control on the format of the stored value using the in-memory-format setting. This option only is available for values; keys will always be stored in binary format. There are 3 in-memory-formats available and it very important that you understand them correctly:

1. BINARY: the value is stored in binary-format. Every time the value is needed it will be deserialized.
2. OBJECT: the value is stored in object-format. If a value is needed in an query/entry-processor, this value is used and no deserialization is needed.
3. CACHED: the value is always stored in binary format, but as soon as a query needs a value [todo:and entryprocessor?], the object is deserialized and cached next to the binary-format. The next time the value is needed in a query/entry-processor, no deserialization is needed.

The default in-memory-format is BINARY.

The big question of course is which one to use. You should consider using the OBJECT in-memory-format if the majority of your hazelcast usage is composed of queries/entry-processors. The reason is that no deserialization is needed when a value is used in a query/entry-processor because the object already is available in object-format. With the BINARY in-memory-format a deserialization is needed since the object only is available in binary-format.

If the majority of your operations are regular Map operations like put/get, you should consider the BINARY in-memory-format. This sounds counterintuitive because normal operations, like the get, rely on the object instance and with a binary format no instance is available. But when the OBJECT in-memory-format is used, the map will never return the store instance but will create a clone instead. This involves a serialize on the owning node followed by a deserialize on the caller

node. With the BINARY format, only a deserialize is needed and therefor is faster. For similar reasons a put with the BINARY in-memory-format will be faster than the OBJECT in-memory-format. When the OBJECT in-memory-format is used, the Map will not store the actual instance, but will make a clone and this involves a serialize followed by a deserialize. When the BINARY in-memory-format is used, only a deserialize is needed.

If you have a mixture of normal operations and queries/entry-processors, you might consider the CACHED in-memory-format since it will always have the binary format available and potentially have the object format available. The price that needs to be payed is increased memory usage since at any given moment it could be that the byte-array and object instance are kept in memory.

In the following example you can see a map configured with the OBJECT in-memory-format.

```
<map name="cities">
    <in-memory-format>OBJECT</in-memory-format>
</map>
```

It is important to realise that if a value is stored in object-format, a change on a returned value, doesn't effect the stored instance because a clone of the stored value is returned and not the actual instance. So changes made on an object after it is returned, will not reflect on the actual stored data. Also when a value is written to a map, if the value is stored in object format it will be a copy of the put value, not the original. So changes made on the object after it is stored, will not reflect on the actual stored data.

5.3.1 What happened to cache-value

If you have been using Hazelcast 2.x you might remember the cache-value property. This property has been dropped in Hazelcast 3. Just as with the InMemoryFormat, the cache-value make it possible to prevent unwanted deserialization. The big difference is that with the cache-value enabled it was possible to get the same instance on subsequent calls of e.g. Map.get, but with the InMemoryFormat this isn't possible. The reason to drop the feature is that returning the same instance leads to unexpected sharing of an object instance. With an immutable object like a String, this won't cause any problems, but with an mutable object this can lead to e.g. concurrency control issues.

5.4 Hashcode and equals

In most cases you probably will make use of some basic type like a Long, Integer or String as key. But in some cases you will need to create custom keys. But to do it correctly in Hazelcast, you need to understand how this mechanism [which mechanism?] works because it works differently compared to traditional map implementations. When you store a key/value in a Hazelcast map, instead of storing the Object, the object are serialized to byte arrays and these are stored. To use the hash/equals in Hazelcast you need to know the following rules:

1. for keys: the hash/equals is determined based on the content of the byte array, so equal keys need to result in equal byte arrays. See [serialization chapter; serializable for warning].
2. for values: the hash/equals is determined based on the in-memory-format; for BINARY the binary format is used. For OBJECT and CACHED the equals of the object is used.

As you can see the difference is subtile, but it is crucial to understand.

Below is an example of an problematic Pair class:

```
public final class Pair implements Serializable {
    private final String significant;
    private final String insignificant;

    public Pair(String significant, String insignificant) {
        this.significant = significant;
        this.insignificant = insignificant;
    }

    @Override
    public boolean equals(Object thatObj) {
        if (this == thatObj) {
            return true;
        }
        if (thatObj == null || getClass() != thatObj.getClass()) {
            return false;
        }
        Pair that = (Pair) thatObj;
        return this.significant.equals(that.significant);
    }

    @Override
    public int hashCode() {
       return significant.hashCode();
    }
}
```

This Pair has 2 fields; the significant field which used in the hash/equals implementation and the insignificant field which is not. If we make 2 keys:

```
Pair key1 = new Pair("a","1");
Pair key2 = new Pair("a","2");
```

then 'key1.equals(key2)' and 'key1.hashCode()==key2.hashCode()'. So if a value would be put in a map with key1, it should be retrieved using key2. But because the binary format of key1 (which contains 'a' and '1') is different than the binary format of key2 (which contain 'a' and '2'), the hash code and equals are different. This is demonstrated in the following program:

```
public class BrokenKeyMember {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();

        Map<Pair, String> normalMap = new HashMap<String,Pair>();
        Map<Pair, String> hzMap = hz.getMap("map");

        Pair key1 = new Pair("a", "1");
        Pair key2 = new Pair("a", "2");

        normalMap.put(key1, "foo");
        hzMap.put(key1, "foo");

        System.out.println("normalMap.get: " + normalMap.get(key2)
           );
        System.out.println("hzMap.get: " + hzMap.get(key2));

        System.exit(0);
    }
}
```

When this program is run, you will get the following output:

```
normalMap.get: foo
hzMap.get: null
```

As you can see, the Pair works fine for a HashMap, but doesn't work for a Hazelcast IMap.

So for a key it is very important that the binary-format of equal objects are the same. For values this depends on the in-memory-format setting. If we configure the following 3 maps in the hazelcast.xml:

```
<hazelcast>
    <map name="objectMap">
        <in-memory-format>OBJECT</in-memory-format>
    </map>

    <map name="cachedMap">
```

```
        <in-memory-format>CACHED</in-memory-format>
    </map>

    <map name="binaryMap">
        <in-memory-format>BINARY</in-memory-format>
    </map>
</hazelcast>
```

If we define 2 values v1 and v2 where the resulting byte-array is different, but the equals method will indicate that they are the same and we put v1 in each map and check for its existence using map.contains(v2):

```
public class BrokenValueMember {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();

        Map<String, Pair> normalMap = new HashMap<String,Pair>();
        Map<String, Pair> binaryMap = hz.getMap("binaryMap");
        Map<String, Pair> objectMap = hz.getMap("objectMap");
        Map<String, Pair> cachedMap = hz.getMap("cachedMap");

        Pair v1 = new Pair("a", "1");
        Pair v2 = new Pair("a", "2");

        normalMap.put("key", v1);
        binaryMap.put("key", v1);
        objectMap.put("key", v1);
        cachedMap.put("key", v1);

        System.out.println("normalMap.contains:" +
            normalMap.containsValue(v2));
        System.out.println("binaryMap.contains:" +
            binaryMap.containsValue(v2));
        System.out.println("objectMap.contains:" +
            objectMap.containsValue(v2));
        System.out.println("cachedMap.contains:" +
            cachedMap.containsValue(v2));
        System.exit(0);
    }
}
```

Then we get the following output:

```
normalMap.contains:true
binaryMap.contains:false
objectMap.contains:true
cachedMap.contains:true
```

As you can see, v1 is found using v2 in the normalMap, the cachedMap and the objectMap; this is because with these maps the equals is done based on the equals method of the object itself. But with the binaryMap the equals is done based on the binary-format and since v1 and v2 have different binary formats, v1 will not

be found using v2.

Even though the hashcode of a key/value is not used by Hazelcast to determine the partition the key/value will be stored in, it will be used by methods like Map.values() and Map.keySet() and therefor it is important that the hash and equals is implemented correctly. For more information see "Effective Java" chapter "Obey the general contract when overriding equals" and "Always override hashCode when you override equals".

5.5 Partition Control

Hazelcast makes it very easy to create distributed maps and access data in these maps. For example you could have a map with customers where the customerId is the key, and you could have a map with orders for a customer, where the orderId is the key. The problem is when the you frequently use the customer in combination with his orders, because chances are that the orders are stored in different partitions than the customer since the customer partition is determined on the customerId and the order partition on the orderId.

Luckily Hazelcast provides a solution to control the partition schema of your data so that all data can be stored in the same partition. If the data is partitioned correctly, your system will exhibit a strong locality of reference and this will reduce latency, increase throughput and improve scalability since less network hops and traffic is required.

To demonstrate this behaviour I'm going to implement a custom partitioning schema for a customer and his orders:

```
public class Customer implements Serializable {
    public final long id;

    public Customer(long id) {
        this.id = id;
    }
}

public final class Order implements Serializable {
    public final long orderId, customerId, articleId;

    public Order(long orderId, long customerId, long articleId) {
        this.orderId = orderId;
        this.customerId = customerId;
        this.articleId = articleId;
    }
```

```
}
```

To control the partition of the order, the OrderKey is introduced which implements PartitionAware. If a key implements this interface, instead of using the binary-format of the key to determine the correct partition, the binary-format of the result of 'getPartitionKey' method call is used. Because we want to partition on the customerId, the 'getPartitionKey' method will the customerId:

```
public final class OrderKey implements PartitionAware,
    Serializable {

    public final long orderId, customerId;

    public OrderKey(long orderId, long customerId) {
        this.orderId = orderId;
        this.customerId = customerId;
    }

    @Override
    public Object getPartitionKey() {
        return customerId;
    }
}
```

The equals and hashcode are not used in this particular example since Hazelcast will make use of the binary format of the key, however in practice you want to implement them, for more information see [hashcode and equals]

In the following example an order is placed with an OrderKey. At the end the partition-ids for a customer, the orderKey and the orderId are printed:

```
public class DataLocalityMember {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        Map<Long, Customer> customerMap = hz.getMap("customers");
        Map<OrderKey, Order> orderMap = hz.getMap("orders");

        long customerId = 100;
        long orderId = 200;
        long articleId = 300;
        Customer customer = new Customer(customerId);
        customerMap.put(customer.id, customer);
        OrderKey orderKey = new OrderKey(orderId, customer.id);
        Order order = new Order(orderKey.orderId, customer.id,
           articleId);
        orderMap.put(orderKey, order);

        PartitionService pService = hz.getPartitionService();
        Partition cPartition = pService.getPartition(customerId);
        Partition oPartition = pService.getPartition(orderKey);
        Partition wPartition = pService.getPartition(orderId);
```

```
        System.out.printf("Partition for customer: %s\n",
            cPartition.getPartitionId());
        System.out.printf("Partition for order with OrderKey: %s\n
            ",
            oPartition.getPartitionId());
        System.out.printf("Partition for order without OrderKey: %
            s\n",
            wPartition.getPartitionId());
    }
}
```

The output looks something like this:

```
Partition for customer: 124
Partition for order with OrderKey: 124
Partition for order without OrderKey: 175
```

As you can see, the partition of customer, is the same as the partition of the order of that customer. We can also see that the partition where an order would be stored using a naive orderId, is different than that of the customer. In this example we created the OrderKey that does the partitioning, but Hazelcast also provides a default implementation that can be used: the PartitionAwareKey.

Being able to control the partitioning schema of data is a very powerful feature and figuring out a good partitioning schema is an architectural choice that you want to get right as soon as possible. Once this is done correctly, it will be a lot easier to write a high performance and scalable system since the number of remote calls is limited.

Collocating data in a single partition often needs to be combined with sending the functionality to the partition that contains the collocated data. If for example an invoice would need to be created for the orders of customer, a Callable that creates the Invoice could be send using the IExecutorService.executeOnKeyOwner(invoiceCallable, customerId) method call. If you do not send the function to the correct partition, collocating data is not useful since a remote call is done for every piece of data. For more information about Executors and routing see ”chapter: Distributed Executor and Routing”.

5.6 High availability

In a production environment all kinds of things can go wrong. A machine could break down due to disk failure, the operating system could crash or it could get disconnected from the network. To prevent that the failure of a single member

leads to failure of the cluster, by default Hazelcast synchronously backs up all map entries on another Member. So if a member fails, no data is lost because the member containing the backup will take over.

The backup-count can be configured using the 'backup-count' property:

```
<map name="persons">
    <backup-count>1</backup-count>
</map>
```

The backup-count can be set to 0, if you favour performance over high availability. Also a higher value than 1 can be specified if you require increased availablity; but the maxium number of backups it 6.The maxium number of backups is 6. The default is 1, so in a lot of cases you don't need to specify it.

By default the backup operations are synchronous; so you have the guarantee that the backup(s) are updated before a method call like map.put completes. But this guarantee comes at the cost of blocking and therefor the latency increases. In some cases having a low latency is more important than having perfect backup guarantees, as long as the window for failure is small. That is why Hazelcast also supports asynchronous backups; where the backups are made at some point in time. This can be configured through the 'async-backup-count' property:

```
<map name="persons">
    <backup-count>0</backup-count>
    <async-backup-count>1<async-backup-count>
</map>
```

The async-backup-count defaults to 0. Unless you want to have asynchronous backups, it doesn't need to be configured.

Although backups can improve high availability, it will increase memory usage since the backups also are kept in memory. So for every backup, you will double the original memory consumption.

By default Hazelcast provides sequential consistency, meaning that when a map entry is read, the most recent written value is seen. This is done by routing the get request to the member that owns the key and therefor there won't be any out of sync copies. But sequential consistency comes at a price because if the value is read on an arbitrary cluster member, that Hazelcast needs to do a remote call to the member that owns the partition for that key. Hazelcast provides the option to increase performance by reducing consistency. This is done by by allowing to read from backup data at the price of potentially seeing stale data. This feature only

is available when there is at least 1 backup (doesn't matter if it a synchronous or asynchronous backup) and can be enabled by setting the read-backup-data property:

```
<map name="persons">
    <backup-count>0</backup-count>
    <async-backup-count>1</async-backup-count>
    <read-backup-data>true</read-backup-data>
</map>
```

In this example you can see a person map with a single asynchronous backup and where read backup data is enabled. This property defaults to false. Reading from the backup can improve performance a bit; if you have a 10 node cluster and read-backup-data is false, than there is 1 in 10 chance that the read will find the data locally. When there is a single backup and read-backup-data is false, there is an additional 1 in 10 chance that read will find the backup data locally. So in total there is a 1 in 5 chance that the data is found locally.

5.7 Eviction

By default all the map entries that are put in the map, will remain in that map. You can choose to delete them manually, but you can also rely on an eviction policy that takes care of deleting items automatically. This feature enables Hazelcast to be used as a distributed cache since hot data is kept in memory and cold data is evicted.

The eviction configuration can be done using the following parameters:

1. max-size: Maximum size of the map. When max size is reached, map is evicted based on the policy defined. Any integer between 0 and Integer.MAX VALUE. 0 means Integer.MAX_VALUE and the default is 0. There also is a policy attribute, that determines how the max-size is going to interpreted:
   (a) PER_NODE: maximum number of map entries in the JVM. This is the default policy.
   (b) PER_PARTITION: maximum number of map entries within a single partition. This is probably not a policy you will be using often since the storage size depending on the number of partitions a member is hosting. If the cluster is small it will be hosting more partitions and therefor map entries, than with a larger cluster.s
   (c) USED_HEAP_SIZE: Max used heap size in MB (mega-bytes) per JVM.
   (d) USED_HEAP_PERCENTAGE: Max used heap size as percentage of the

JVM heap size. If the JVM is configured with 1000 MB and and the max-size is 10, than with this policy the map the JVM is allowed to be 100 MB before map entries are evicted.

2. eviction-policy:
   (a) NONE: No items will be evicted, so the max-size is ignored. This is the default policy. If you want max-size to work, then you need to set an eviction-policy other than NONE. Of course, you still can combine it with time-to-live-seconds and max-idle-seconds.
   (b) LRU: Least Recently Used
   (c) LFU: Least Frequently Used.
3. time-to-live-seconds: Maximum number of seconds for each entry to stay in the map. Entries that are older than time-to-live-seconds and not updated for time-to-live-seconds will get automatically evicted from the map. Any integer between 0 and Integer.MAX_VALUE is valid. 0 means infinite and also is the default.
4. max-idle-seconds: Maximum number of seconds for each entry to stay idle in the map. Entries that are idle (not touched) for more than max-idle-seconds will get automatically evicted from the map. Entry is touched if get, put or containsKey is called. Any integer between 0 and Integer.MAX_VALUE. 0 means infinite. Default is 0.
5. eviction-percentage: When the maximum size is reached, the specified percentage of the map will be evicted. The default value is 25 percent. If the value is set to a too small value, then only a small number of map entries are evicted and this can lead to a lot of overhead if map entries are frequently inserted.

An example configuration:

```
<map name="articles">
    <max-size policy="PER_NODE">10000</max-size>
    <eviction-policy>LRU</eviction-policy>
    <max-idle-seconds>60</max-idle-seconds>
</map>
```

This configures an articles Map that will start to evict map entries from a member, as soon as the map size within that member exceeds 10000. It will then start to remove map entries that are least recently used. Also when map entries are not used for more than 60 seconds, they will be evicted as well.

It is also possible to evict a key manually by calling the IMap.evict(key) method. You might wonder what the difference is between this method and the IMap.delete(key); if there is no MapStore defined then there is no difference. Otherwise a IMap.delete

will call a delete on the MapStore and potentially delete the map entry from the database. With the evict method, the map entry is only removed from the map.

MapStore.delete(Object key: is not called when a MapStore is used an an map entry is evicted. So if the MapStore is connected to a database, no record entries are removed due to map entries being evicted.

5.8 Near Cache

All the map entries within a given partition, are owned by a single member. If a map entry is read, the member that owns the partition of the key is asked to read the value. But in some cases data needs to be read very frequently by members that don't own the key and therefor most request require remoting and this reduces performance and scalability. Normally it is best to partition the data, so that all relevant data is stored in the same partition and just send the operation to the machine owning the partition. But this is not always an option.

Luckily Hazelcast has a feature called the near cache, that makes map entries locally available by adding a local cache attached to the the map. Imagine a web-shop where articles can be ordered and where these articles are stored in a Hazelcast map. To enable local caching of frequently used articles, the near cache is configured like this:

```
<map name="articles">
    <near-cache/>
</map>
```

On the near cache the following properties can be configured:

1. max-size: the maximum number of cache entries per local cache. As soon as the maximum size has been reached, the cache will start to evict entries based on the eviction policy. max-size should be between 0 and Integer.MAX_SIZE, where 0 will be interpreted as Integer.MAX_SIZE. The default is Integer.MAX_SIZE, but it probably is better to explicitly configure max-size in combination with an eviction-policy, or else set time-to-live-seconds/max-idle-seconds to prevent OutOfMemoryErrors. The max-size of the near cache is independent of the max-size of the map itself.
2. eviction-policy: the policy used to evict members from the cache when the near cache is full. The following options are available:
   (a) NONE: No items will be evicted, so the max-size is ignored. If you want max-size to work you need to set an eviction-policy other than NONE.

Of course, you still can combine it with time-to-live-seconds and max-idle-seconds.

(b) LRU: Least Recently Used. This is the default policy.

(c) LFU: Least Frequently Used.

3. time-to-live-seconds: the number of seconds a map entry is allowed to remain in the cache. Valid values are 0 to Integer.MAX_SIZE, and 0 will be interpreted as infinite. The default is 0.
4. max-idle-seconds: the maximum number of seconds a map entry is allowed to stay in the cache without being read. max-idle-seconds should be between 0 and Integer.MAX_SIZE, where 0 will be interpreted as Integer.MAX_SIZE. The default is 0.
5. invalidate-on-change: should all the members listen to change of their cached entries and evict the entry when updated or deleted. Valid values are true/-false and defaults to true. [todo: what is the consequence of having millions of items in the cache? There will also be many listeners]
6. in-memory-format: the in-memory-format of the cache and defaults to BINARY. For more information see [Map InMemoryFormat].

An example configuration:

```
<map name="articles">
    <near-cache/>
        <max-size>10000</max-size>
        <eviction-policy>LRU</eviction-policy>
        <max-idle-seconds>60</max-idle-seconds>
    </near-cache>
</map>
```

This configures an articles Map with a near-cache, that will start to evict near-cache entries from a member, as soon as the near-cache size within that member exceeds 10000. It will then start to remove near-cache entries that are least recently used. Also when near cache entries are not used for more than 60 seconds, they will be evicted as well.

In the previous 'Eviction' section we talked about evicting items from the map, but it is important to understand that the near cache and map eviction are two different things. The near cache is a local map that contains frequently accessed map entries from any member, the local map will only contain map entries it owns. You can even combine the eviction and the near cache; although their settings are independent.

Some things worth considering when using a near cache:

1. increases memory usage since the near cache items need to be stored in the

memory of the member

2. reduces consistency, especially when 'invalidate-on-change' is false: it could be that a cache entry never is refreshed.
3. it is best used for read only data. Especially when 'invalidate-on-change' is enabled, there is a lot of remoting involved to invalidate the cache entry, when a map entry is updated.
4. the nearcache can also be enabled on the client. See [chapther client]
5. there is no functionality currently available to heat up the cache.

5.9 Concurrency Control

The Hazelcast map itself is thread-safe just like the ConcurrentHashMap or the Collections.synchronizedMap, but in some cased your thread safety requirements are bigger than what hazelcast provides out of the box. Luckily Hazelcast provides multiple concurrency control solutions; it can either be pessimistic using locks or optimistic using compare and swap operations. Todo: what about transactions?

Take a look at the following example; if run by multiple members in parallel the total amount would be [Talip: refer and encourage the use of executeOnKey API]

```
public class RacyUpdateMember {

    public static void main(String[] args) throws Exception {
        HazelcastInstance hzInstance = Hazelcast.
           newHazelcastInstance();
        IMap<String, Value> map = hzInstance.getMap("map");
        String key = "1";
        map.put(key, new Value());
        System.out.println("Starting");
        for (int k = 0; k < 1000; k++) {
            if(k%100 == 0) System.out.println("At: "+k);
            Value value = map.get(key);
            Thread.sleep(10);
            value.field++;
            map.put(key, value);
        }
        System.out.println("Finished! Result = " + map.get(key).
           field);
    }

    static class Value implements Serializable {
        public int field;
    }
}
```

5.9.1 Pessimistic Locking

The classic way to solve the race problem is to use a lock. In Hazelcast there are various ways to lock, but for this example we'll use the locking functionality provided by the map using the map.lock/map.unlock methods.

```
public class PessimisticUpdateMember {

    public static void main(String[] args) throws Exception {
        HazelcastInstance hzInstance = Hazelcast.
           newHazelcastInstance();
        IMap<String, Value> map = hzInstance.getMap("map");
        String key = "1";
        map.put(key, new Value());
        System.out.println("Starting");
        for (int k = 0; k < 1000; k++) {
            map.lock(key);
            try {
                Value value = map.get(key);
                Thread.sleep(10);
                value.field++;
                map.put(key, value);
            } finally {
                map.unlock(key);
            }
        }
        System.out.println("Finished! Result = " + map.get(key).
           field);
    }

    static class Value implements Serializable {
        public int field;
    }
}
```

Another way to lock is to acquire some predictable Lock object from Hazelcast. You could give every value its own lock, but you could also create a stripe of locks. Although it potentially can increase contention, it will reduce space.

5.9.2 Optimistic Locking

It is important to implement equals on the value, because this is used to determine of 2 objects are equal. With the ConcurrentHashMap it is based on object reference. On the keys the byte-array equals is used, but on the replace(key,oldValue,newValue) the equals is used. If you fail to forget it, your code will not work!

```
//This code is broken on purpose.
public class OptimisticMember {
```

```
    public static void main(String[] args) throws Exception {
        HazelcastInstance hzInstance = Hazelcast.
           newHazelcastInstance();
        IMap<String, Value> map = hzInstance.getMap("map");
        String key = "1";
        map.put(key, new Value());
        System.out.println("Starting");
        for (int k = 0; k < 1000; k++) {
            if(k%10==0) System.out.println("At: "+k);
            for (; ; ) {
                Value oldValue = map.get(key);
                Value newValue = new Value(oldValue);
                //   Thread.sleep(10);
                newValue.field++;
                if(map.replace(key, oldValue, newValue))
                    break;
            }
        }
        System.out.println("Finished! Result = " + map.get(key).
           field);
    }
    static class Value implements Serializable {
        public int field;
        public Value(){}
        public Value(Value that) {
            this.field = that.field;
        }
        public boolean equals(Object o){
            if(o == this)return true;
            if(!(o instanceof Value))return false;
            Value that  = (Value)o;
            return that.field == this.field;
        }
    }
}
```

Aba problem; it can be that the following thing happens. And explain how it can be solved by adding a version; although all the other fields will be equal, the version field will prevent objects from being seen as equal.

5.9.3 Pessimistic vs Optimistic

5.10 EntryProcessor

One of the new features of Hazelcast 3 is the EntryProcessor. It allows to send a function, the EntryProcessor, to a particular key or to all keys in an IMap. Once the EntryProcessor is completed, it is discarded, so it is not a durable mechanism like the EntryListener or the MapInterceptor.

Imagine that you have a map of employees, and you want to give every employee a bonus. In the example below you see a very naive implementation of this functionality:

```
public class Employee implements Serializable {
    private int salary;

    public Employee(int salary) {
        this.salary = salary;
    }

    public int getSalary() {
        return salary;
    }

    public void incSalary(int delta){
        salary+=delta;
    }
}

public class NaiveProcessingMember {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IMap<String,Employee> employees = hz.getMap("employees");
        employees.put("John", new Employee(1000));
        employees.put("Mark", new Employee(1000));
        employees.put("Spencer", new Employee(1000));

        for(Map.Entry<String,Employee> entry: employees.entrySet()
            ){
            String id = entry.getKey();
            Employee employee = employees.get(id);
            employee.incSalary(10);
            employees.put(entry.getKey(),employee);
        }

        for(Map.Entry<String,Employee> entry: employees.entrySet()
            ){
            System.out.println(entry.getKey()+
                " salary: "+entry.getValue().getSalary());
        }
        System.exit(0);
    }
}
```

It is naive for a few reasons. The first reason is that this functionality isn't very scalable; a single machine will need to pull all the employees to himself, transform it and write it back. So if your number of employees grows 2 times, it will probably take 2 times as much time. Another problem is that the current implementation is subject to race problems; imagine that a different process currently also is giving a raise of 10. Because the read and write of the employee is not atomic, since there

is no lock, it could be that one of the raises is overwritten and the employee only gets a single raise instead of a double raise.

That is why the EntryProcessor was added to Hazelcast. The EntryProcessor captures the logic that should be executed on an MapEntry. Hazelcast will send the EntryProcessor to each member in the cluster and each member will in parallel apply the EntryProcessor to all map entries. This means that the EntryProcessor is scalable; the more machines you add, the faster the processing will be completed. Another important feature of the EntryProcessor is that it will deal with race problem by acquiring exclusive access to the MapEntry when it is processing.

In the following example you can see the raise functionality implemented using a EntryProcessor:

```
public class EntryProcessorMember{

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IMap<String,Employee> employees = hz.getMap("employees");
        employees.put("John", new Employee(1000));
        employees.put("Mark", new Employee(1000));
        employees.put("Spencer", new Employee(1000));

        employees.executeOnEntries(new EmployeeRaiseEntryProcessor
            ());

        for(Map.Entry<String,Employee> entry: employees.entrySet()
            ){
            System.out.println(entry.getKey()+
                " salary: "+entry.getValue().getSalary());
        }
        System.exit(0);
    }

    static class EmployeeRaiseEntryProcessor
            extends AbstractEntryProcessor<String, Employee> {

        @Override
        public Object process(Map.Entry< String, Employee> entry)
            {
            entry.getValue().incSalary(10);
            return null;
        }
    }
}
```

If we run this program, we'll get the following output:

```
Mark salary: 20
John salary: 20
Spencer salary: 20
```

5.10.1 Process return value

In the example, the process method modifies the employee instance and returns null. The EntryProcessor can also be used to return a value for every map entry. If we wanted to calculate the sum of all salaries, the following EntryProcessor will return the salary of an employee:

```
static class GetSalaryEntryProcessor
        extends AbstractEntryProcessor<String, Employee> {

    public GetSalaryEntryProcessor(){
            super(false);
    }

    @Override
    public Object process(Map.Entry< String, Employee> entry) {
        return entry.getValue().getSalary();
    }
}
```

And it can be used like this:

```
Map<String,Object> salaries = employees.executeOnEntries(
    new GetSalaryEntryProcessor());
int total=0;
for(Object salary : salaries.values()){
    total+=(Integer)salary;
}
System.out.println("Total salary of all employees:"+x);
```

But you need to be really careful using this technique since the salaries map will be kept in memory and this can lead to an OutOfMemoryError. If you don't care about a returned map, it is best to let the process method return null. This will prevent the result for a single process invocation to be stored in the map.

If you are wondering why the GetSalaryEntryProcessor constructor calls the super with false, check the next section.

5.10.2 Backup processor

When the EntryProcessor is applied on a map, it will not only process all primary map entries, it will also process all backups. This is needed to prevent the primary map entries are going to contain different data than the backups. In the current examples, we made use of the AbstractEntryProcessor class instead of the EntryProcessor interface, which applies the same logic to primary and backups. But if you want, you can apply different logic on the primary than on the backup.

This can be useful if the value doesn't need to be changed, but you want to do a certain action. E.g. log, retrieve information. The previous example where the total salary of all employees is calculated, is such an situation. That is why the GetSalaryEntryProcessor constructor calls the super with false; this signals the AbstractEntryProcessor not to apply any logic to the backup, only to the primary. To fully understand how EntryProcessor works, lets have a look at the implementation of the AbstractEntryProcessor:

```
public abstract class AbstractEntryProcessor<K, V>
        implements EntryProcessor<K, V> {

    private final EntryBackupProcessor<K,V> entryBackupProcessor;

    public AbstractEntryProcessor(){
        this(true);
    }

    public AbstractEntryProcessor(boolean applyOnBackup){
       if(applyOnBackup){
          entryBackupProcessor = new EntryBackupProcessorImpl();
       }else{
          entryBackupProcessor = null;
       }
    }

    @Override
    public abstract Object process(Map.Entry<K, V> entry);

    @Override
    public final EntryBackupProcessor<K, V> getBackupProcessor() {
         return entryBackupProcessor;
    }

    private class EntryBackupProcessorImpl
            implements EntryBackupProcessor<K,V>{

        @Override
        public void processBackup(Map.Entry<K, V> entry) {
            process(entry);
        }
    }
}
```

The important method here is the getBackupProcessor. If we don't want to apply any logic in the backups, we can return null. This signals to Hazelcast that only the primary map entries need to be processed. If a we do want to apply logic on the backups, we need to return a EntryBackupProcessor instance. In this case the EntryBackupProcessor.processBackup method will make use of the process method, but if you provide a custom EntryProcessor implementation, you have complete freedom on how it should be implemented.

5.10.3 Threading

To understand how the EntryProcessor works, you need to understand how the threading works. Hazelcast will only allow a single thread, the partition thread, to be active in a partition. This means that by design it isn't possible that operations like IMap.put are interleaved with other Map operations, or with system operations like migration of a partition. The EntryProcessor will also be executed on the partition thread, and therefor, while the EntryProcessor is running, no other operations on that MapEntry can happen.

If the EntryProcessor would process all map entries in a given partition, and the number of map entries would be big, no other operations would be possible on this partition. It can even become more problematic, because there are only a limited number of partition threads and when they are all busy processing, the member won't be able to process any operation on all of its partitions. It will not only cause problems within the IMap, but on all DistributedObjects whose partitions are owned by that member.

To deal with this problem the EntryProcessor will process a batch of map entries a time and once this batch is complete, it will release the partition thread and reschedule itself. This gives other operations the opertunity to run and prevens them from starving. Hazelcast exposes 2 properties to control the batch size:

1. hazelcast.entryprocessor.batch.max.size: the maximum number of map entries executed within a single batch. Defaults to 10.000.
2. hazelcast.entryprocessor.batch.max.time.millis: the maximum number of miliseconds a batch is allowed to take. Defaults to 2000 miliseconds. Hazelcast will not abort a the processing of a Map entry, so make sure that processing a map entry doesn't take too much time.

Hazelcast will end the schedule the next batch as soon as the maximum number of map entries have been processor or when the max time has been exceeded.

Because of the threading model, doing an operation on a different partition than the current partition thread is not allowed, otherwise you could run into a deadlock. Luckily it is allowed to call operations on the current partition from the EntryProcessor, so if you have partitioned your DistributedObjects correctly, they can be accessed. E.g. you could access another map to retrieve information needed for processing.

If your EntryProcessor maintains local state, you need take into account that the

same EntryProcessor can be called concurrently by the partition threads.

5.10.4 Good to know

InMemoryFormat: If you are often using the EntryProcessor or queries, it might be an idea to use the InMemoryFormat.OBJECT or InMemoryFormat.CACHED. For the OBJECT memory format Hazelcast will not serialize/deserialize the entry and you are able to apply the EntryProcessor without serialialisation cost. So the value instance that is stored, is passed to the EntryProcessor and that instance will also be stored in the MapEntry (unless you create a new instance of course). For the InMemoryFormat.CACHED, the deserialization will only happens once if no cached instance if available. For more information see the [chapter: InMemoryFormat]

Process single key: if you want to execute the EntryProcessor on a single key, he IMap.executeOnKey method can be used. Of course you could do the same with an IExecutorService.executeOnKeyOwner, but you need to lock and potentially deal with more serialization.

Process using predicate: todo: need to wait till Enes implemented this feature for 3.1

Deletion: the EntryProcessor can also be used to delete items by setting the map entry value to null. In the following example you can see that all bad employees are being deleted using this approach:

```
class DeleteBadEmployeeEntryProcessor
            extends AbstractEntryProcessor<String, Employee> {
    @Override
    public Object process(Map.Entry< String, Employee> entry) {
        if(entry.getValue().isBad()){
            entry.setValue(null);
        }
        return null;
    }
}
```

PartitionAware: because the EntryProcessor needs to be serialized to be send to another machine, you can't pass it complex dependencies like the HazelcastInstance. When the HazelcastInstanceAware interface is implemented, the dependencies can be injected. For more information see [serialization: HazelcastInstanceAware]

5.11 MapInterceptor

todo

5.12 EntryListener

One of the new features of version 3.0 is the EntryListener. You can listen map entry events providing a predicate and so event will be fired for each entry validated by your query. IMap has a single method for listening map providing query.

```
public class ListeningMember {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IMap<String,String> map = hz.getMap("somemap");
        map.addEntryListener(new MyEntryListener(), true);
        System.out.println("EntryListener registered");
    }

    static class MyEntryListener implements EntryListener<String,
       String>{
        @Override
        public void entryAdded(EntryEvent<String, String> event) {
            System.out.println("entryAdded:"+event);
        }

        @Override
        public void entryRemoved(EntryEvent<String, String> event)
            {
            System.out.println("entryRemoved:"+event);
        }

        @Override
        public void entryUpdated(EntryEvent<String, String> event)
            {
            System.out.println("entryUpdated:"+event);
        }

        @Override
        public void entryEvicted(EntryEvent<String, String> event)
            {
            System.out.println("entryEvicted:"+event);
        }
    }
}
public class ModifyMember {
    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IMap<String, String> map = hz.getMap("somemap");
        String key = "" + System.nanoTime();
        String value = "1";
```

```
        map.put(key, value);
        map.put(key, "2");
        map.delete(key);
        System.exit(0);
    }
}
```

When you first start the ListeningMember and then the ModifyMember, the ListeningMember will output something like this:

```
entryAdded:EntryEvent {Address[192.168.1.100]:5702} key
   =251359212222282, oldValue=null, value=1, event=ADDED, by
   Member [192.168.1.100]:5702
entryUpdated:EntryEvent {Address[192.168.1.100]:5702} key
   =251359212222282, oldValue=1, value=2, event=UPDATED, by Member
    [192.168.1.100]:5702
entryRemoved:EntryEvent {Address[192.168.1.100]:5702} key
   =251359212222282, oldValue=2, value=2, event=REMOVED, by Member
    [192.168.1.100]:5702
```

5.12.1 Threading

To correctly use the EntryListener, it is very important to understand the threading model. Unlike the EntryProcessor, the EntryListener doesn't run on the partition threads but runs on an event thread; the same threads that are used by other collection listeners and ITopic messagelisteners. So with the EntryListener it is allowed to access other partitions. Just like other logic that runs on an event thread, you need to watch out for long running tasks because it could lead to starvation of other event listeners because they don't get a thread. But it can also lead to OOME because of event being queued quicker than they are being processed.

5.12.2 Good to know

HazelcastInstanceAware: When an EntryListener is send to a different machine, it will be serialized and then deserialized. This can be problematic if you need to access dependencies which can't be serialized. To deal with this problem, if the EntryListener implement HazelcastInstanceAware, the HazelcastInstance can be injected. For more information see [serialization: HazelcastInstanceAware].

5.13 Contuous Query

In the previous section we talked about the MapEntryListener that can be used to listen to changed with a map. One of the new additions to Hazelcast 3 is the Continous Predicate: a EntryListener that is registered using a predicate. This makes it possible to listen to the changes made to a specific map entries.

To demonstrate the continuous query, we are going to listen to the changes made to a person with a specific name. So lets create the Person class first:

```
public class Person implements Serializable{

    private final String name;

    public Person(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

The following step is to register a EntryListener using a predicate so that the continuous query is created:

```
public class ContinuousQueryMember {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IMap<String, String> map = hz.getMap("map");
        map.addEntryListener(new MyEntryListener(),
                new SqlPredicate("name=peter"), true);
        System.out.println("EntryListener registered");
    }

    static class MyEntryListener
            implements EntryListener<String, String> {
        @Override
        public void entryAdded(EntryEvent<String, String> event) {
            System.out.println("entryAdded:" + event);
        }

        @Override
        public void entryRemoved(EntryEvent<String, String> event)
            {
            System.out.println("entryRemoved:" + event);
        }
```

```
        @Override
        public void entryUpdated(EntryEvent<String, String> event)
            {
            System.out.println("entryUpdated:" + event);
        }

        @Override
        public void entryEvicted(EntryEvent<String, String> event)
            {
            System.out.println("entryEvicted:" + event);
        }
    }
}
```

As you can see, the query is created using a the SqlPredicate 'name=peter'. So the listener will be notififed as soon as a person with name 'peter' is modified. To demonstrate this, start the ContinuousQueryMember and then start the following member:

```
public class ModifyMember {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IMap<String, Person> map = hz.getMap("map");

        map.put("1", new Person("peter"));
        map.put("2", new Person("talip"));
        System.out.println("done");
        System.exit(0);
    }
}
```

When ModifyMember is done, the ContinuousQueryMember will show the following output:

```
entryAdded:EntryEvent {Address[192.168.178.10]:5702} key=1,
    oldValue=null, value=Person{name='peter'}, event=ADDED, by
    Member [192.168.178.10]:5702
```

As you can see the listener is only notified for peter, and not for talip.

5.14 Distributed Queries

Imagine that we have a Hazelcast IMap where the key is some id and the value is Person object and we want to retrieve all persons with a given name using the following naive implementation:

```
    public Set<Person> getWithNameNaive(String name){
        Set<Person> result = new HashSet<Person>();
        for(Person person: personMap.values()){
```

```
            if(person.name.equals(name)){
                result.add(person);
            }
        }
        return result;
    }
```

This is what you probably would write if the map would be an ordinary map. But when the map is distributed map, there are some performance and scalability problems with this approach:

1. It is not parallelizable. One member will iterate over all persons instead of spreading the load over multiple members. Because the search isn't parallelizable, the system can't scale; you can't add more members to the cluster to increase performance.
2. It is inefficient because all persons need to be pulled over the line before being deserialized into the memory of the executing member. So there is a unnecessary network traffic.

Luckily Hazelcast solves these problems by supporting predicates that are executed on top of a fork/join mechanism:

1. when the predicate is requested to be evaluated by the caller, it is forked to each member in the cluster
2. each member will filter all local map entries using the predicate. Before a predicate evaluates a map entry, the key/value of that entry are deserialized and passed to the predicate.
3. the caller joins on the completion of all members and merges the results into a single set

The fork/join approach is highly scalable because it parallelizable. By adding additional cluster members, the number of partitions per member is reduced and therefor the time a member needs to iterate over all its data, is reduced as well. Also the local filtering is parallelizable because a pool of 'partition threads' will evaluate segments of elements concurrently. And last but not least, the amount of network traffic is reduced drastically, since only filtered data is send instead of all data.

Hazelcast provides 2 API's for distributed queries:

1. Criteria API
2. Distributed SQL Query

5.14.1 Criteria API

To implement the Person search using the JPA like criteria API, you could do the following:

```
import static com.hazelcast.query.Predicates.*;

    ...

    public Set<Person> getWithName(String name) {
        Predicate namePredicate = equal("name", name);
        return (Set<Person>) personMap.values(namePredicate);
    }
```

The namePredicate verifies that the 'name' field has a certain value using the equal operator. After we have created the predicate, we apply it to the personMap by calling the 'IMap.values(Predicate)' method which takes care of sending it to all members in the cluster, evaluating it, and merging the result. Because the predicate is send over the line, it needs to be 'serializable'. See the [todo: serialize chapter] for more information.

The Predicate is not limited to values only. It can also apply be applied to the keySet, the entrySet and the localKeySet of the IMap.

equal operator

In the previous example we already saw the equal operator in action where it gets the name of the person object. When it is evaluated, it first tries to lookup an accessor method, so in case of 'name', the accessor methods it will try are 'isName()' and 'getName()'. If one found, it is called and the comparison is done. Of course an accessor method doesn't need to return a field, it could also be a synthetic accessor where some value is created on the fly. If no accessor is found, a field with the given name is looked up. If that exists, it is returned and otherwise a RuntimeException is thrown. Hazelcast doesn't care about the accessibility of a field or an accessor method, so you are not forced to make them public.

In some cases you need to traverse over an object structure, e.g. we want the street of the address the person lives at. With the equal operator this can be done like this: 'address.street'. This expression is evaluated from left to right and there is no limit on the number of steps involved. Also accessor methods can be used here. Another important thing is how the equal operator deals with null, especially with object traversal. As soon as null is found, it is used in the comparison.

And, Or and Not predicates

Predicates can be joined using the 'and' and 'or' predicate:

```
import static com.hazelcast.query.Predicates.*;

    ...

    public Set<Person> getWithNameAndAge(String name, int age) {
        Predicate namePredicate = equal("name", name);
        Predicate agePredicate = equal("name", age);
        Predicate predicate = and(namePredicate, agePredicate);
        return (Set<Person>) personMap.values(predicate);
    }

    public Set<Person> getWithNameOrAge(String name, int age) {
        Predicate namePredicate = equal("name", name);
        Predicate agePredicate = equal("age", age);
        Predicate predicate = or(namePredicate, agePredicate);
        return (Set<Person>) personMap.values(predicate);
    }
```

And of course we can't forget the 'not' predicate:

```
    public Set<Person> getNotWithName(String name) {
        Predicate namePredicate = equal("name", name);
        Predicate predicate = not(namePredicate);
        return (Set<Person>) personMap.values(predicate);
    }
```

Other operators

In the Predicates class you can find a whole collections of useful operators:

1. notEqual: checks if the result of an expression is not equal to a certain value.
2. instanceOf: checks if the result of an expression has a certain type
3. like: checks if the result of an expression matches some string pattern. % (percentage sign) is placeholder for many characters, _ (underscore) is placeholder for only one character.
4. greaterThan: checks if the result of an expression is greater than a certain value.
5. greaterEqual: checks if the result of an expression is greater or equal than a certain value.
6. lessThan: checks if the result of an expression is less than a certain value
7. lessEqual: checks if the result of an expression is than than or equal to a certain value.
8. between: checks if the result of an expression is between 2 values (this is inclusive).

9. in: checks if the result of an expression is an element of a certain collection.
10. isNot: checks if the result of an expression is false.
11. regex : checks if the result of an expression matches some regular expression.

If the predicates provided by Hazelcast are not enough, you can always write your own predicate by implementing the Predicate interface:

```
public interface Predicate<K, V> extends Serializable {
    boolean apply(Map.Entry<K, V> mapEntry);
}
```

The MapEntry not only contains the key/value, but also contains all kinds of metadata like the time it was created/expires/last-accessed etc.

PredicateBuilder

The syntax we used so far to create Predicates is clear but can be simplified by making use of the PredicateBuilder. It provides a fluent interface that can make building predicates simpler. But underwater the same functionality is being used. Here is an example where a predicate is build that selects all persons with a certain name and age using this PredicateBuilder:

```
public Set<Person> getWithNameAndAgeSimplified(String name,
    int age) {
     EntryObject e = new PredicateBuilder().getEntryObject();
     Predicate predicate = e.get("name").equal(name).and(e.get(
         "age").equal(age));
     return (Set<Person>) personMap.values(predicate);
}
```

As you can see, it can simplify things, especially if you have complex predicates. But it is a matter of taste which approach you prefer.

5.14.2 Distributed SQL Query

In the previous section the Criteria API was explained where expression/predicate objects are manually created. The pain can be reduced a bit by making use of the PredicateBuilder, but it still isn't perfect. That is why a DSL was added: the Distributed SQL Query, that is based on a SQL like language. But underwater the Criteria API is used.

The 'get with name' function we already implemented using the Criteria API, can be implementing using the Distributed SQL Query like this:

```
public Set<Person> getWithName(String name){
```

```
    Predicate predicate = new SqlPredicate(String.format("name = %
        s",name));
    return (Set<Person>) personMap.values(predicate);
}
```

As you can see, the SqlPredicate is a Predicate itself and therefor can be combined with the Criteria API. The language itself isn't case sensitive, but 'columns' used in the query are. Underneath you can see an overview of the DSL:

1. logical operators

   ```
   man and age>30
   man=false or age = 45 or name = 'Joe'
   man and (age >20 OR age < 30)
   not man
   ```

2. relational operators

   ```
   age <= 30
   name ="Joe"
   age != 30
   ```

3. between

   ```
   age between 20 and 33
   age not between 30 and 40
   ```

4. like

   ```
   name like 'Jo%' (true for 'Joe', 'Josh', 'Joseph' etc.)
   name like 'Jo_' (true for 'Joe'; false for 'Josh')
   name not like 'Jo_' (true for 'Josh'; false for 'Joe')
   name like 'J_s%' (true for 'Josh', 'Joseph'; false 'John', '
       Joe')
   ```

5. in

   ```
   age in (20, 30, 40)
   age not in (60, 70)
   ```

5.15 Indexes

To speed up queries, just like in databases, the Hazelcast map supports indexes. Using an index prevents from iterating over all values, in database terms this is called a full table scan, but directly jump to the interesting ones. There are 2 types of indexes:

1. Ordered: e.g. a numeric field where you want to do range searches like bigger than.
2. Unordered: e.g. a name field.

In the previous chapter we talked about a Person that has a name, age etc. To speed up searching on these fields, we can place an unordered index on name and an ordered index on age:

```
<map name="persons">
    <indexes>
        <index ordered="false">name</index>
        <index ordered="true">age</index>
    </indexes>
</map>
```

The ordered attribute defaults to 'false'.

To retrieve the index field of an Object, first an accessor method will be tried and if that doesn't exist, a direct field access is done. With the index accessor method you are not limited to returning a field, you can also create a synthetic accessor method where some value is calculated on the fly. The index field also supports object traversal, so you could create an index on the street of the address of a person using 'address.street'. There is no limitation on the depth of the traversal. Hazelcast doesn't care about the accessibility of the index field or accessor method, so you are not forced to make them public. An index field or an object containing an field, for the 'x.y' notation, is allowed to be null.

One big difference compared to Hazelcast 2, with Hazelcast 3 indexes can be created on the fly. There is even an option in the Management Center to create an index on an existing IMap.

The performance impact of using one or more indexes depends on a lot of factors; among them are the size of the map, the chance of finding the element with a full table scan etc. Also adding one or more indexes, make mutations to the map more expensive since the index needs to be updated as well. So it could be that if you have more mutations than searches, that the performance with an index is lower than without an index. Therefor it is recommended to test in a production like environment, using a representative size/quality of the dataset, which configuration is best for you. In the source code of the book you can 2 very rudimentary index benchmarks, one for updating and one for searching.

In Hazelcast versions prior to 3.0, indexing for String fields was done only for the first the 4 characters. With Hazelcast version 3.0+ indexing is done on the entire String.

In the example, the indexes are placed attributes of a basic datatypes like int and String. But the IMap allows indexes to be placed on an attribute of any type, as long as it implements Comparable. So you can create indexes on custom data-types.

5.16 Persistence

In the previous section we talked about backups that protect against member failure, so if one member goes down, another member takes over. But it does not protect you against cluster failure, e.g. when a cluster is hosted in a single datacenter, and it goes down. Luckily Hazelcast provides a solution loading and storing data externally, e.g. in a database. This can be done using:

1. com.hazelcast.core.MapLoader: useful for reading entries from an external datasource, but changes don't need to be written back.
2. com.hazelcast.core.MapStore: useful for reading and writing map entries from and to an external datasource. The MapStore interface extends the MapLoader interface.

And one instance per Map per Node will be created.

The following example shows an extremely basic HSQLDB implementation of the MapStore where we load/store a simple Person object with a name field:

```
public class PersonMapStore implements MapStore<Long, Person> {
    private final Connection con;

    public PersonMapStore() {
        try {
            con = DriverManager.getConnection("jdbc:hsqldb:
               mydatabase", "SA", "");
            con.createStatement().executeUpdate(
                    "create table if not exists person (id bigint,
                        name varchar(45))");
        } catch (SQLException e) {throw new RuntimeException(e);}
    }

    @Override
    public synchronized void delete(Long key) {
        try {
            con.createStatement().executeUpdate(
                    format("delete from person where id = %s", key
                       ));
        } catch (SQLException e) {throw new RuntimeException(e);}
    }

    @Override
    public synchronized void store(Long key, Person value) {
        try {
            con.createStatement().executeUpdate(
                    format("insert into person values(%s,'%s')",
                       key, value.name));
        } catch (SQLException e) {throw new RuntimeException(e);}
    }

    @Override
```

```
    public synchronized void storeAll(Map<Long, Person> map) {
        for (Map.Entry<Long, Person> entry : map.entrySet())
            store(entry.getKey(), entry.getValue());
    }

    @Override
    public synchronized void deleteAll(Collection<Long> keys) {
       for(Long key: keys) delete(key);
    }

    @Override
    public synchronized Person load(Long key) {
        try {
            ResultSet resultSet = con.createStatement().
               executeQuery(
                    format("select name from person where id =%s",
                        key));
            try {
                if (!resultSet.next()) return null;
                String name = resultSet.getString(1);
                return new Person(name);
            } finally {resultSet.close();}
        } catch (SQLException e) {throw new RuntimeException(e);}
    }

    @Override
    public synchronized Map<Long, Person> loadAll(Collection<Long>
        keys) {
        Map<Long, Person> result = new HashMap<Long, Person>();
        for (Long key : keys) result.put(key, load(key));
        return result;
    }

    Override
    public Set<Long> loadAllKeys() {
         return null;
    }
}
```

As you can see the implementation is quite simple and certainly can be improved, e.g. transactions, prevention against sql injection etc. Because the MapStore/MapLoader can be called by threads concurrently, this implementation make use of synchronization to deal with that correctly. Currently it relies on a course grained locked, but you could perhaps apply finer grained locking based on the key and a striped lock.

To connect the PersonMapStore to the persons map, we can configure it using the 'map-store' setting:

```
<map name="persons">
    <map-store enabled="true">
        <class-name>PersonMapStore</class-name>
    </map-store>
```

```
</map>
```

In the following code fragment you can see a member that writes a person to the map exits the JVM. And you can see a member that loads the person and prints it.

```
public class WriteMember {

    public static void main(String[] args) throws Exception {
        HazelcastInstance hzInstance = Hazelcast.
           newHazelcastInstance();
        IMap<Long, Person> personMap = hzInstance.getMap("
           personMap");
        personMap.put(1L, new Person("Peter"));
        System.exit(0);
    }
}
public class ReadMember {

    public static void main(String[] args) throws Exception {
        HazelcastInstance hzInstance = Hazelcast.
           newHazelcastInstance();
        IMap<Long, Person> personMap = hzInstance.getMap("
           personMap");
        Person p = personMap.get(1L);
        System.out.println(p);
    }
}
```

With the WriteMember you can see that the System.exit(0) is called at the end. This is done to release the HSQLDB so that it can be opened by the ReadMember. Calling System.exit is a safe way for Hazelcast to leave the cluster due to a shutdown hook, and it waits for all backup operations to complete.

A word of caution: the MapLoader/MapStore should NOT call Map/Queue/MultiMap/List/Set/etc operations, otherwise you might run into deadlocks.

Pre-Populating the map

With the MapLoader it is possible to pre-populate the Map so that when it is created, the important entries are loaded in memory. This can be done by letting the 'loadAllKeys' method return the Set of all 'hot' keys that need to be loaded for the partitions owned by the member. This also makes parallel loading possible, since each member can load its own keys. If the 'loadAll' method return null, as we did in the example, then the map will not be pre-populated. Also important to know is that Map is created lazily by Hazelcast, so only when one of the members

calls the 'HazelcastInstance.getMap(name)' the map is actually created and the MapLoader called. If your application requires that Map up front without really needing the content, you could wrap the map in a lazy proxy that calls the getMap method only when really needed.

I common mistake made is that the 'loadAllKeys' returns all keys in the database table. This could be problematic since you would pull the complete table in memory, but another important problem is that if each member returns the all keys, each member will load the complete table from the database. So if a you have 1.000.000 records in the database, and 10 members, then the total number of records loaded is 10.000.000 instead of 1.000.000. Of course the Map.size will still be 1.000.000. That is why a member should only load the records it owns, e.g. by adding the

[todo: do not return all keys in the database; since each member is going to do the same. Is there any form of protection against this? Because when new members are added, they will load the map and therefor the load-all-keys method is called. So how can you do this once only?] [todo: how to figure out in which partition you are] [todo: how to figure out which data to load from the database]

You need to be aware of that the map only knows about map entries that are in memory, only when a get is done for an explicit key, then the map entry is loaded from the MapStore. This behavior is called a read through. So if the loadAll would return a subset of the keys in the database, then e.g. the Map.size() will show only the size of this subset, not the record count in the database. And the same goes for queries; these will only be executed on the entries in memory, not on the records in the database.

To make sure that you only keep hot entries in memory, you can configure the 'time-to-live-seconds' property on the Map. When a Map entry isn't used and the time to live expires, it will automatically be removed from the map without calling the MapStore.delete.

Write Through vs Write Behind

Although the MapStore makes durability possible, it also comes at a cost: every time that a change is made in the map, a write through to the your persistence mechanism happens. Write through operations increase latency since databases cause latency (e.g. disk access). In Hazelcast it is possible to use a write behind

instead of a write through. When a change happens, the change is synchronously written to the backup partition (if that is configured), but the change to the database is done asynchronously. Enabling write behind can be done by configuring the 'write-delay-seconds' in the 'map-store' configuration section. It defaults to 0, which means a write through. A value higher than 0 indicates a write behind. Using write behind is not completely without danger, it could happen that the cluster fails before the write to the database has completed. In that case information could be lost.

[todo: what about consistency? Are the writes still done in order or out of order?]

MapLoaderLifecycleSupport

In some cases your MapLoader needs to be notified of lifecycle events. This can be done by letting the MapLoader implementation implement the com.hazelcast.core.MapLoaderLifecyc interface. This signals to Hazelcast that the implementation is interested in:

1. init: useful if you want to initialize resources like opening database connections. One of the parameters the init method receives is a Properties object. This is useful if you want to pass properties from the outside to the MapLoader implementation. If you make use of the xml configuration, in the map-store xml configuration you can specify the properties that need to be passed to the init method.
2. destroy: useful if need to cleanup resources like closing database connections.

5.16.1 Good to know

Serialize before store: the value is serialized before the MapStore.store is called. If you are retrieving the id or using optimistic locking in the database by adding a version field, this can cause problems because changes made on the entity, are done after the value has been serialized. So the existing bytearray will contain the old id/version; no matter what the store method updated.

5.17 MultiMap

In some cases you need to store multiple values for a single key. You could use a normal collection as value and store the 'real' values in this collection. This works fine if everything is done in memory, but in a distributed and concurrent

environment it isn't that easy. One problem with this approach is that the whole collection needs to be deserialized for an operation like add. Imagine a collection of 100 elements, then 100 elements need to be deserialized when the value is read, and 101 items are serialized when the value is written; so a total of 201 elements. This can cause a lot of overhead, cpu, memory, network usage etc. Another problem is that without additional concurrency control, e.g. using a lock or a replace call, it is likely to run into a lost update which can lead to issues like items not being deleted, or items getting lost. To solve these problems Hazelcast provides a MultiMap where multiple values can be stored under a single key.

The MultiMap doesn't implement the java.util.Map interface since the signatures of the methods are different. The MultiMap does have support for most of the IMap functionality (so locking, listeners etc), but it doesn't support indexing, predicates and the MapLoader/MapStore.

To demonstrate the MultiMap we are going to create 2 member. The PutMember will put data in the MultiMap:

```
public class PutMember {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        MultiMap<String, String> map = hz.getMultiMap("map");

        map.put("a", "1");
        map.put("a", "2");
        map.put("b", "3");
        System.out.println("PutMember:Done");
    }
}
```

And the PrintMember will print all entries in that MultiMap:

```
public class PrintMember {

    public static void main(String[] args) {
        HazelcastInstance hzInstance = Hazelcast.
           newHazelcastInstance();
        MultiMap<String,String> map = hzInstance.getMultiMap("map"
           );
        for(String key: map.keySet()){
            Collection<String> values = map.get(key);
            System.out.printf("%s -> %s\n",key,values);
        }
    }
}
```

If we first run PutMember and then run PrintMember, then PrintMember will show

```
b -> [3]
a -> [2, 1]
```

As you can see, there is a single value for key 'b' and 2 values for key 'a'.

Configuration

The MultiMap is configured with the MultiMapConfig using the following configuration options:

1. valueCollectionType: the collection type of the value. There are 2 options: SET and LIST. With a set duplicate and null values are not allowed and ordering is inrelevant. With the list duplicates and null values are allowed and ordering is relevant. Defaults to SET.
2. listenerConfigs: the EntryListeners for the MultiMap.
3. binary: if the value is stored in binary format (true) or in object format (false), defaults to true. todo: when should you choose
4. backupCount: the number of synchronous backups, defaults to 1.
5. asyncBackupCount: the number of asynchronous backups. Defaults to 0.
6. statisticsEnabled: if the statistics have been enabled, defaults to 0. The statistics can be accessed by calling the MultiMap.getLocalMultiMapStats() method.

5.17.1 Good to know:

Value collection not partitioned: the collection used as value, is not partitioned and stored on a single machine. So the maximum size of the value depends on the capacity of a single machine. So you need to be careful how much data is stored in the value collection.

Get returns copy: It is important to realise that map.get(key) returns a copy of the values at some moment in time. Changes to this collection will result in an a UnsupportedOperationException. So if you want to change the values, you need to do it through the MultiMap interface.

Removing items: you can remove items from the MultiMap. If the collection for a specific key is empty, this collection will not automatically be removed, so it can be that you need to clean up the MultiMap, to prevent memory leaks.

Collection copying: if a value for key K is stored on member1 because K is owned

by that member1, and member2 does a map.get(K), then the whole collection will transported from member1 to member 2. So if that value collection is big, it could lead to performance problems. A solution would be to send the operation to member1, so send the logic to the data instead of the data to the logic.

Map of maps: The MultiMap can't be used as a map of maps where there are 2 keys to find the value. We have some plans to add this in the near future, but if you can't wait, you could either create a composite key or use dynamically created maps, where the name of map is determined by the first key, and the second key is the key in that map.

5.18 Good to know

Snapshot: When Map.entrySet(), Map.keySet() or Map.values() is called, a snapshot of the current state of the map is returned. Changes that are made in the map, do not reflect on changes in these sets and vice versa. Also when changes are made on these collections, an UnsupportedOperationException is thrown.

Serialization: Although the IMap looks like an in memory data-structure like a HashMap, there are big differences. One of the differences is that (de)serialization needs to take place for a lot of operation. Also it could be that remoting is involved. This means that the IMap will not have the same performance characteristics as an in memory Map. To minimize serialization cost, make sure you correctly configure the InMemoryFormat.

Size: method is a distributed operation; a request is send to each member to return the number of map entries they contains. This means that abusing the size method could lead to performance problems.

Memory Usage: an completely empty IMap instance consumed ¿200 KB's of memory in the cluster with a default configured number of partitions. So having a lot of small maps could lead to unexpected memory problems. If you double the number of partitions, the memory usage will roughly double as well.

5.19 What is next

The Hazelcast IMap is a very feature rich data-structures and by properly configuring it will serve a lot of purposes.

Chapter 6

Distributed Executor Service

Java 5 was perhaps the most fundamental upgrade since Java was released. On a language level we got generics, static imports, enumerations, varargs, enhanced for loop and annotations. Although less known, Java 5 also got fundamental fixes for the Java Memory Model (JSR-133) and we got a whole new concurrency library (JSR-166) found in java.util.concurrent.

This library contains a lot of goodies; some parts you probably don't use on a regular based, but other parts you perhaps do. One of the features that was added is the java.util.concurrent.Executor. The idea is that you wrap functionality in a Runnable if you don't need to return a value, or in a Callable if you need to return a value, and submitting it to the Executor. A very basic example of the executor:

```
class EchoService{
   private final ExecutorService =
      Executors.newSingleThreadExecutor();

   public void echoAsynchronously(final String msg){
      executor.execute(new Runnable(){
         public void run() {
            System.out.println(msg);
         }
      });
   }
}
```

So while a worker thread is processing the task, the thread that submitted the task is free to work asynchronously. There is virtually no limit in what you can do in a task; complex database operations, intensive cpu or IO operations, render images etc.

However the problem in a distributed system is that the default implementation of the Executor, the ThreadPoolExecutor, is designed to run within a single JVM. In a distributed system you want that a task submitted in one JVM, can be processed

in another. Luckily Hazelcast provides the IExecutorService, that extends the java.util.concurrent.ExecutorService, but is designed to be used in a distributed environment. The IExecutorService is a new in Hazelcast 3.x.

Lets start with a very simple example of IExecutorService where a task is executed that does some waiting and echoes a message:

```
public class EchoTask implements Runnable, Serializable {
   private final String msg;

   public EchoTask(String msg) {
      this.msg = msg;
   }

   @Override
   public void run() {
      try {
         Thread.sleep(5000);
      } catch (InterruptedException e) {
      }
      System.out.println("echo:" + msg);
   }
}
```

This EchoTask implements the Runnable interface so that it can be submitted to the Executor. It also implements the Serializable interface because it could be that it is send to a different JVM to be processed. Instead of making the class Serializable, you could also rely on other serialization mechanisms; see [chapter serialization].

The next part is the MasterMember that is responsible for submitting (and executing) 1000 echo messages:

```
public class MasterMember {
   public static void main(String[] args) throws Exception {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      IExecutorService executor = hz.getExecutorService("exec");
      for (int k = 1; k <= 1000; k++) {
         Thread.sleep(1000);
         System.out.println("Producing echo task: " + k);
         executor.execute(new EchoTask("" + k));
      }
      System.out.println("EchoTaskMain finished!");
   }
}
```

First we retrieve the executor from the HazelcastInstance and then we slowly submit 1000 echo tasks. One of the differences between Hazelcast 2.x and 3.x is that the HazelcastInstance.getExecutorService() method has disappeared; you now always need to provide a name instead of relying on the default one. By default

Hazelcast configures the executor with 8 threads in the pool. For our example we only need 1, so we configure it in the hazelcast.xml file like this:

```
<executor-service name="exec">
   <pool-size>1</pool-size>
</executor-service>
```

Another difference is that the core-pool-size and keep-alive-seconds properties from Hazelcast 2.x have disappeared, so the pool will have a fixed size.

When the MasterMember is started you will get output like this:

```
Producing echo task: 1
Producing echo task: 2
Producing echo task: 3
Producing echo task: 4
Producing echo task: 5
echo:1
....
```

As you can see the production of messages is 1/second and the processing is 0.2/second (the echo task sleeps 5 seconds), this means that we produce work 5 times faster than we are able to process it. Apart from making the EchoTask faster, there are 2 dimension for scaling:

1. scale up
2. scale out

Both are explained below and in practice they are often combined.

6.1 Scaling up

Scaling up, also called vertical scaling, is done by increasing the processing capacity on a single JVM. Since each thread can process 0.2 messages/second and we produce 1 message/second, if the Executor would have 5 threads it can process messages as fast as they are produced.

When you scale up you need to look carefully at the JVM if it can handle the additional load. If not; you may need to increase its resources (either cpu, memory, disk etc). If you fail to do so the performance could degrade instead of improving.

Scaling up the ExecutorService in Hazelcast is very simple, just increment the maxPoolSize. Since we know that 5 threads is going to give maximum performance, lets set them to 5.

```
<executor-service name="exec">
   <pool-size>5</pool-size>
```

```
</executor-service>
```

When we run the MasterNode we'll see something like this:

```
Producing echo task: 1
Producing echo task: 2
Producing echo task: 3
Producing echo task: 4
Producing echo task: 5
echo:1
Producing echo task: 6
echo:2
Producing echo task: 7
echo:3
Producing echo task: 8
echo:4
...
```

As you can see, the tasks are being processed as quickly as they are being produced.

6.2 Scaling out

Scaling up was very simple since there is enough cpu and memory capacity. But often one or more of the resources will be the limiting factor. In practice increasing the computing capacity within a single machine will reach a point where it isn't cost efficient since the expenses go up quicker than the capacity improvements.

Scaling out, also called horizontal scaling, is orthogonal to scaling up; instead of increasing the capacity of the system by increasing the capacity of a single machine, we just add more machines. In our case we can safely start multiple Hazelcast members on the same machine since processing the task doesn't consume resources while the task waits. But in real systems you probably want to add more machines (physical or virtualized) to the cluster.

To scale up our echo example, we can add the following very basic slave member:

```
import com.hazelcast.core.*;
public class SlaveMember {
   public static void main(String[] args) {
      Hazelcast.newHazelcastInstance();
   }
}
```

We don't need to do anything else because this member will automatically participate in the executor that was started in the master node and start processing tasks.

If one master and slave are started, you will see that the slave member is processing tasks as well:

```
echo:31
echo:33
echo:35
```

So in only a few lines of code, we are now able to scale out! If you want, you can start more slave members, but with tasks being created at 1 task/second, maximum performance is reached with 4 slaves.

6.3 Routing

Till so far we didn't care about which member did the actual processing of the task; as long as a member picks it up. But in some cases you do want to have that control. Luckily the IExecutorService provides different ways to route tasks:

1. any member. This is the default configuration.
2. specific member
3. the member hosting a specific key
4. all or subset of members.

In the previous section we already covered routing to any member. In the following sections I'll explain the last 3 routing strategies. This is also where a big difference is visible between Hazelcast 2.x and 3.x, while 2.x relied on the DistributedTask, 3.x relies on explicit routing methods on the IExecutorService.

6.3.1 Executing on a specific member

In some cases you may want to execute a task on a specific member. As an example we are going to send an echo task to each member in the cluster. This is done by retrieving all members using the Cluster object and iterating over the cluster members. To each of the members we are going to send an echo message containing their own address.

```
public class MasterMember {
   public static void main(String[] args) {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      IExecutorService executorService = hz.getExecutorService("ex
         ");
      for (Member member : hz.getCluster().getMembers()) {
         EchoTask task = new EchoTask("echo" +
            member.getInetSocketAddress());
         executorService.executeOnMember(task, member);
      }
```

```
    }
}
```

When we start a few slaves and a master, we'll get output like:

```
Members [2] {
    Member [192.168.1.100]:5702 this
    Member [192.168.1.100]:5703
}
...
echo/192.168.1.100:5702
```

As you can see, the EchoTasks are executed on the correct member.

6.3.2 Executing on key owner

When an operation is executed in a distributed system, this operation often needs to access distributed resources. If these resources are hosted on a different member then where the task is running, scalability and performance may suffer due to remoting overhead. Luckily this problem can be solved by improving locality of reference.

In Hazelcast this can be done by placing the resources for a task in some partition and sending the task to the member that owns that partition. When you start designing a distributed system, perhaps the most fundamental step is designing the partitioning scheme.

As an example we are going to create a distributed system where there is some dummy data in a map and for every key in that map we are going to execute a verify task. This task will verify if it has been executed on the same member as where that partition for that key is residing:

```
public class VerifyTask implements
        Runnable, Serializable, HazelcastInstanceAware {
    private final String key;
    private transient HazelcastInstance hz;

    public VerifyTask(String key) {
        this.key = key;
    }

    @Override
    public void setHazelcastInstance(HazelcastInstance hz) {
        this.hz = hz;
    }

    @Override
    public void run() {
```

```
        IMap map = hz.getMap("map");
        boolean localKey = map.localKeySet().contains(key);
        System.out.println("Key is local:" + localKey);
    }
}
```

If you look at the run method, you can see it accesses the map, retrieves all the keys that are owned by this member using the IMap.localKeySet() method, checks if the key is contained in that key set and prints the result. Another thing that is interesting is that this task implements HazelcastInstanceAware. This signals to Hazelcast that when this class is deserialized for execution, it will inject the HazelcastInstance executing that task. For more information see [reference to serialization/HazelcastInstanceAware section]

The next step is the MasterMember which first creates a map with some entries; we only care about the key so the value is bogus. And then it iterates over the keys in the map and send a VerifyTask for each key:

```
public class MasterMember {
    public static void main(String[] args) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        Map<String, String> map = hz.getMap("map");
        for (int k = 0; k < 10; k++)
            map.put(UUID.randomUUID().toString(), "");
        IExecutorService executor = hz.getExecutorService("exec");
        for (String key : map.keySet())
            executor.executeOnKeyOwner(new VerifyTask(key), key);
    }
}
```

As you can see we are now relying on the 'executeOnKeyOwner' to execute a task on the member owning a specific key. To verify the routing, we start a few slaves first and then start a master and we'll see output like this:

```
key is local:true
key is local:true
...
```

As you can see, the tasks are executed on the same member as where the data is living.

An alternative way of executing a request on a specific member, that originates from Hazelcast 2.x, is to let the task implement the HazelcastPartitionAware interface and use the 'execute' or 'submit' method on the IExecutorService. The HazelcastPartitionAware exposes the 'getPartitionKey' method that is used by the executor to figure out the key of the partition to route to. If a null value is returned, any partition will do.

6.3.3 Executing on all or subset of members

In some cases you may want to execute a task on multiple or even on all members. You should use this functionality wisely since it will create load on multiple members, potentially all members, and therefore can reduce scalability.

In the following example there is a set of members and on these members there is a distributed map containing some entries. Each entry has some UUID as key and 1 as value. To demonstrate executing a task on all members, we are going to create a distributed sum operation that sums all values in the map:

```
public class SumTask implements
        Callable<Integer>, Serializable, HazelcastInstanceAware {

    private transient HazelcastInstance hz;

    @Override
    public void setHazelcastInstance(HazelcastInstance hz) {
        this.hz = hz;
    }

    @Override
    public Integer call() throws Exception {
        IMap<String, Integer> map = hz.getMap("map");
        int result = 0;
        for (String key : map.localKeySet()) {
            System.out.println("Calculating for key: " + key);
            result += map.get(key);
        }
        System.out.println("Local Result: " + result);
        return result;
    }
}
```

When this SumTask is called it retrieves the map and then iterates over all local keys, sums the values and returns the result.

The MasterMember will first create the map with some entries. Then it will submit the SumTask to each member and the result will be a map of Future instances. And finally we'll join all the futures and sum the result and print it:

```
public class MasterMember {
    public static void main(String[] args) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        Map<String,Integer> map = hz.getMap("map");
        for (int k = 0; k < 5; k++)
            map.put(UUID.randomUUID().toString(), 1);
        IExecutorService executor = hz.getExecutorService("exec");
        Map<Member,Future<Integer>> result =
            executor.submitToAllMembers (new SumTask());
        int sum = 0;
```

```
        for(Future<Integer> future: result.values())
            sum+=future.get();
        System.out.println("Result: " + sum);
    }
}
```

When we start 1 slave and then a master member, we'll see something like this for the slave:

```
Calculating for key: 6ed5fe89-b2f4-4644-95a3-19dffcc71a25
Calculating for key: 5c870a8c-e8d7-4a26-b17b-d94c71164f3f
Calculating for key: 024b1c5a-21d4-4f46-988b-67a567ae80c9
Local Result: 3
```

And master member:

```
Calculating for key: 516bd5d3-8e47-48fb-8f87-bb647d7f3d1f
Calculating for key: 868b2f1e-e03d-4f1a-b5a8-47fb317f5a39
Local Result: 2
Result: 5
```

In this example we execute a task on all members, but if you only want to execute a task on a subset of members, you can call the submitToMembers method and pass the subset of members.

Not possible to send Runnable to every partition: there is no direct support to send a runnable to every partition. If this is an issue, the SPI could be a solution since Operations can be routed to specific partitions. So you could build such an executor on top of the SPI.

6.4 Futures

The Executor interface only exposes a single 'void execute(Runnable)' method that can be called to have a Runnable asynchronously executed. But in some cases you need to synchronize on results, e.g. when using a Callable or just want to wait till a task completes. This can be done making use of the java.util.concurrent.Future in combination with one of the submit methods of the IExecutorService.

To demonstrate the future, we are going to calculate a Fibonacci number by wrapping the calculation in a callable and synchronizing on the result:

```
public class FibonacciCallable
        implements Callable<Long>, Serializable {

    private final int input;

    public FibonacciCallable(int input) {
```

```
        this.input = input;
    }

    @Override
    public Long call() {
        return calculate(input);
    }

    private long calculate(int n) {
        if (n <= 1) return n;
        else return calculate(n - 1) + calculate(n - 2);
    }
}
```

The next step is submitting the task and using a Future to synchronize on results:

```
public class MasterMember {
    public static void main(String[] args) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IExecutorService executor = hz.getExecutorService("exec");
        int n = Integer.parseInt(args[0]);
        Future<Long> future =
            executor.submit(new FibonacciCallable(n));
        try {
            long result = future.get(10, TimeUnit.SECONDS);
            System.out.println("Result: "+result);
        } catch (TimeoutException ex) {
            System.out.println("A timeout happened");
        }
    }
}
```

As you can see when we call the executorService.submit(Callable) method, we get back a Future as result. This future allows us to synchronize on completion or cancel the computation.

When we run this application with 5 as argument, the output will be:

```
Result: 5
```

when you run this application with 500 as argument, it will probably take more than 10 seconds to complete and therefore the future.get will timeout. When the timeout happens, a TimeoutException is thrown. If it doesn't on your machine, it could be that your machine is very quick and you need to use a smaller timeout. Unlike Hazelcast 2.x, unfortunately it isn't possible in Hazelcast 3.0 to cancel a future. [todo: add reason] One possible solution is to let the task periodically check if certain key in a distributed map exist. A task can then be cancelled by writing some value for that key. You need to take care of removing keys to prevent this map from growing; this could be done using the time to live setting.

6.5 Execution Callback

With a future it is possible to synchronize on task completion. In some cases you want to synchronize on the completion of the task before executing some logic, in the same thread that submitted the task. In some other cases you want this post completion logic to be executed asynchronously, so that the submitting thread doesn't block. Hazelcast provides a solution for this using the ExecutionCallback.

In the 'Future' topic an example is shown where a Fibonacci number is calculated and waiting on the completion of that operation is done using a Future. In the following example we are also going to calculate a Fibonacci number, but instead of waiting for that task to complete, we register an ExecutionCallback where we print the result asynchronously:

```
public class MasterMember {
   public static void main(String[] args){
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      IExecutorService executor = hz.getExecutorService("exec");
      ExecutionCallback<Long> callback =
            new ExecutionCallback<Long>() {
         public void onFailure(Throwable t) {
            t.printStackTrace();}
         public void onResponse(Long response) {
            System.out.println("Result: " + response);}
      };
      executor.submit(new FibonacciCallable(10), callback);
      System.out.println("Fibonacci task submitted");
    }
}
```

As you can see, the ExecutionCallback has 2 methods; one method that is called on a valid response and we print it. And the other method is called on failure and we print the stacktrace:

If you run this example you will see the following output:

```
Fibonacci task submitted
Result: 55
```

As you can see, the thread that submitted the tasks to be executed, was not blocked. You can also see that eventually the result of the Fibonacci calculation will be printed.

6.6 Good to know

Work-queue has no high availability: Each member will create one or more local ThreadPoolExecutors with ordinary work-queues that do the real work. When a task is submitted, it will be put on the work-queue of that ThreadPoolExecutor and will not be backed up by Hazelcast. If something would happen with that member, all unprocessed work will be lost.

Work-queue is not partitioned: Each member specific executor will have its own private work-queue, once an item is placed on that queue, it will not be taken by a different member. So it could be that one member has a lot of unprocessed work, and another is idle.

Work-queue by default has unbound capacity: And this can lead to OutOfMemory-Errors, because the number of queued tasks can grow without being limited. This can be solved by setting the ¡queue-capacity¿ property on the executor-service. If a new task is submitted while the queue is full, the call will not block but immediately throw a RejectedExecutionException that needs to be dealt with. Perhaps that in the future blocking with configurable timeout will be made available.

No Load Balancing: is currently available for tasks that can run on any member. In the future there probably is going to be a customizable load balancer interface where for example load balancing could be done on the number of unprocessed tasks, cpu load, memory load etc. If load-balancing is needed, it could be done by creating an IExecutorService proxy that wraps the one returned by Hazelcast. Using the members from the ClusterService or member information from SPI:MembershipAwareService, it could route 'free' tasks to a specific member based on load.

Destroying Executors: you need to be careful when shutting down an IExecutorService because it will shutdown all corresponding executor in every member and subsequent calls to proxy will result in a RejectedExecutionException. When the executor is destroyed and later a HazelcastInstance.getExecutorService is done with the id of the destroyed executor a new executor will be created as if the old one never existed.

Executors doesn't log exceptions: when a task fails with an exception (or an error), this exception will not be logged by Hazelcast. This is in line with the ThreadPoolExecutorService from Java and it can be really annoying when you are

spending a lot of time on why something doesn't work. It can easily be fixed; either add a try/catch in your runnable and log the exception. Or wrap the runnable/-callable in a proxy that does the logging; the last option will keep your code a bit cleaner.

HazelcastInstanceAware: When a task is deserialized, in a lot of cases you need to get access to the HazelcastInstance. This can be done by letting the task implement HazelcastInstanceAware. For more information see [chapter Serialization:HazelcastInstanceAware]

6.7 What is next

In this chapter we explored the distributed execution of tasks using the Hazelcast ExecutorService.....

Chapter 7

Distributed Topic

In the 'Distributed Collections' chapter we talked about the IQueue, which can be used to create point to point message solutions. In such a solution there can be multiple publishers, but there each message will be consumed by a single consumer. An alternative approach is the publish/subscribe mechanism, where a single message can be consumed by multiple subscribers.

Hazelcast provides a publish/subscribe mechanism in the form of the com.hazelcast.core.ITopic. It is a distributed mechanism for publishing messages to multiple subscribers. Any number of members can publish messages to a topic and any number of members can receive messages from the topics they are subscribed to. The message can be an ordinary POJO, although it must be able to serialize [see serialization chapter] since it needs to go over the line.

I'll show you how the distributed topic works based on a very simple example; there is a single topic that is shared between a publisher and a subscriber. The publisher publishes the current date on the topic:

```
public class PublisherMember {
    public static void main(String[] args){
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        ITopic<Date> topic = hz.getTopic("topic");
        topic.publish(new Date());
        System.out.println("Published");
        System.exit(0);
    }
}
```

And the subscriber acquires the same topic and adds a MessageListener to subscribe itself to the topic:

```
public class SubscribedMember {

    public static void main(String[] args){
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
```

```
        ITopic<Date> topic = hz.getTopic("topic");
        topic.addMessageListener(new MessageListenerImpl());
        System.out.println("Subscribed");
    }

    private static class MessageListenerImpl
                    implements MessageListener<Date> {

        @Override
        public void onMessage(Message<Date> m) {
            System.out.println("Received: "+m.getMessageObject());
        }
    }
}
```

We first start up the subscriber, and we'll see: "Subscribed" in the console. Then we start the publisher, and after it published the message "Published", the subscriber will output something like:

```
Received: Sat Feb 15 13:05:24 EEST 2013
```

To make it a bit more interesting, you can start multiple subscribers. If you run the publisher again, all subscribers will receiving the same published message.

In the 'SubscribedMember' example we dynamically subscrib to a topic. If you prefer a more static approach, you can also do it through the configuration:

```
<topic name="topic">
    <message-listeners>
        <message-listener>MessageListenerImpl</message-listener>
    </message-listeners>
</topic>
```

Hazelcast makes use of reflection to create an instance of the 'MessageListenerImpl'. For this to work, this class needs to have a no-arg constructor. If need more flexibility creating a MessageListener implementation, you could have a look at the programmatic configuration where you can pass an explicit instance instead of a class.

```
Config config = new Config();
TopicConfig topicConfig = new TopicConfig();
topicConfig.setName("topic");
MessageListener listener = new YourMessageListener(arg1,arg2,...);
topicConfig.addMessageListenerConfig(new ListenerConfig(listener))
    ;
config.addTopicConfig(topicConfig);
```

7.1 Message ordering

Hazelcast provides certain ordering guarantees on the delivery of messages. If a cluster member publishes a sequence of messages, then Hazelcast will guarantee this each MessageListener will receive these messages in the order they were published by that member. So if message m1, m2, m3 are published by member M, then each listener will receive the messages in the following order: m1, m2, m3.

Even though messages will be received in the same order, by default, nothing can't be said about the ordering of messages send by different members. Imagine that member M sends m1, m2, m3 and member N sends n1, n2, n2. Then it could happen that listener1 receives m1, m2, m3, n1, n2, n3 but listener2 receives n1, m1, n2, m2, n3, m3. This is valid because in both cases the ordering of messages send by M or N is not violated.

But in some cases you want that all listeners receive the messages in exactly the same order. So if listener1 recieves m1, m2, m3, n1, n2, n3 then listener2 will receive the messages in exactly the same order. To realize this ordering guarantee, Hazelcast provides a global-order-enabled setting. Be careful though, it will have a considerable impact on throughput and latency.

It is important to understand that global-order-enabled setting doesn't say anything about synchronization between listener1 and listener2. So it could be that listener1 already is processing n3 (the last message in the sequence) but listener is still busy processing m1 (the first message in the sequence).

7.2 Scaling up the MessageListener

Hazelcast uses the internal event system, which is also used by collections events, to execute message listeners; This event system has a striped executor where the correct index within that stripe is determined based on the topic-name.

This means that by default all messages send by member X and topic T will always be executed by the same thread. There are some limitation caused by this approach. The most important one is that if processing a messages takes a lot of time, the thread will not be able to process other events which could become problematic. Another problem is that because the ITopic relies on the event system, also other Topics could be starved from threads.

One thing you can do is to increase the number of threads running in the striped executor of the event system using property 'hazelcast.event.thread.count'; which defaults to 5 threads.

Another way to deal with the limitations is to offload the messages processing to another thread. One of the ways to implement this is to make use of the StripedExecutor:

```
public class SubscribedMember {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        ITopic<Date> topic = hz.getTopic("topic");
        topic.addMessageListener(new MessageListenerImpl("topic"))
            ;
        System.out.println("Subscribed");
    }

    private final static StripedExecutor executor = new
       StripedExecutor(
        Executors.newFixedThreadPool(10), 10
    );

    private static class MessageListenerImpl
                   implements MessageListener<Date> {

        private final String topicName;

        public MessageListenerImpl(String topicName){
            this.topicName = topicName;
        }

        @Override
        public void onMessage(final Message<Date> m) {
            StripedRunnable task = new StripedRunnable() {
                @Override
                public int getKey() {
                    return topicName.hashCode();
                }

                @Override
                public void run() {
                    System.out.println("Received: " +
                                       m.getMessageObject());
                }
            };
            executor.execute(task);
        }
    }
}
```

In this example the work is offloaded to the striped executor by making use of the StripedRunnable where he thread that executes message is determined based

on the topic-name. In this example the capacity workqueue of the executor is unbound, which could lead to OOME problems. In most cases it is better to give the workqueue a maximum capacity and a call either can fail directly or wait a bit before space becomes available (see the TimedRunnable to control the timeout). Be careful about blocking for a too long period because the onMessage is called by a Thread of the event system and hogging this thread could lead to problems in the system.

In this case there is only a single topic and we have a single executor. But you could decide to create an executor per ITopic. If you don't care about ordering of messages at all, instead of using a striped executor, you could make use of an ordinary executor.

7.3 Good to know

HazelcastInstanceAware. If you need to have access to the HazelcastInstance in the MessageListener, let it implement the HazelcastInstanceAware interface. Hazelcast will then inject the HazelcastInstance when the topic is created. One thing to watch out for however is that if you have passed an MessageListener instance to the Config instead of a class, and you create multiple HazelcastInstances with it, that the MessageListener will get different HazelcastInstances injected.

Not transactional. The ITopic is not transactional, so be careful when it is used inside a transaction. If the transaction fails after a message is send or consumed and the transaction is rolled back, the message sending or consumption will not be rolled back.

No garbage collection for the topics. So as long as the topics are not destroyed, they will be managed by Hazelcast and this can eventually lead to memory issues since the topics are stored in memory. There are a few ways to deal with this issue. One way is to send a message to the subscribers that they

One way to solve it is by creating a garbage collection mechanism for the ITopic. Create a topic statistics IMap with the topic name as key and a pair containing the last processed message count and timestamp as value. When a topic is retrieved from the HazelcastInstance, an entry can be placed in the topic statics map if it is missing. Periodically iterate over all topics from the topics statics map e..g using the IMap.localKeySet(). Then retrieve the local statistics from the ITopic using the

ITopic.getLocalTopicStats() method and check if there is a the number of processed messages has changed, using the information from the topic statistics map. If there is a difference, update the topic statics in the map; the new timestamp can be determined using the Cluster.getClusterTime() method. If there is no change, and period between the current timestamp and the last timestamp exceeds a certain threshold, the topic and the entry in the topic statistics map can be destroyed. It is important to understand that this solution isn't perfect, since is that it could happen that a message is send to a topic that has been destroyed. The topic will be recreated, but the subscribers are gone.

No durable subscriptions. If a subscriber goes offline, it will not receives messages that were send while it was offline.

No metadata. The message is an ordinary POJO and therefor doesn't contain any metadata like a timestamp or an address to reply to. Luckily this can be solved by wrapping the message in an envelop - a new POJO - and setting the metadata on that envelop.

No delivery guarantee: When messages are send to a subscriber, they are put on a queue first and then the message will be taken from that queue to be processed. If items are on the queue and member fails, all messages on that queue will be lost.

No queue per topic: Hazelcast doesn't publish the messages on a topic specific queue, but makes of a single queue; the event queue. By default the size of the queue is limited to 1.000.000, but can be changed using the 'hazelcast.event.queue.capacity' property.

When the capacity of the queue is reached, the calling thread will block until there is capaciy on the queue or a timeout happens. When the timeout happens, Hazelcast log a warning which included the topic name and the sender of the message, but the producer of this message remains agnostic what happened.

If the event queue is full, Hazelcast will block for a short period. The default is 250 ms, but can be changed using the 'hazelcast.event.queue.timeout.millis' property. Be careful with making the timeout too because you don't own the blocked thread, which could be an internal Hazelcast thread, e.g. for dealing with IO. When such a thread is blockin for a long period, it could lead to problems in other part of the system.

There are plans to redesign the ITopic so that:

1. each topic gets its own queue with a configurable capacity
2. each topic gets a configurable message-listener executors so you can control how many threads are processing message-listeners.

Statistics: If you are interested in obtaining ITopic statics, you can enabled statistics using the 'statistics-enabled'

```
<topic name="topic">
   <statistics-enabled>true</statistics-enabled>
</topic>
```

The statistics like total messages published/received can only be accessed from cluster members using topic.getLocalTopicStats. Topic statistics can't be retrieved by the client, because only a member has knowledge about what happened to his topic. If you need to have global statistics, you need to aggregate the statistics of all members.

Statistics: If you are interested in obtaining ITopic statics, you can enabled statistics using the 'statistics-enabled'

```
<topic name="topic">
   <statistics-enabled>true</statistics-enabled>
</topic>
```

The statistics like total messages published/received can only be accessed from cluster members using topic.getLocalTopicStats. Topic statistics can't be retrieved by the client, because only a member has knowledge about what happened to his topic. If you need to have global statistics, you need to aggregate the statistics of all members.

7.4 What is next

In this chapter we have seen the ITopic. From a high level there is some overlap with JMS, but the provided functionality is limited. On the other side, the ITopic is extremely easy to use, scalable and doesn't require message brokers to be running.

Chapter 8

Hazelcast Clients

Till so far the examples showed members that were full participants in the cluster; so they will known about others and they will host partitions. But in some cases you only want to connect to the cluster to read/write data from the cluster or execute operations, but you don't want to participate as a full member in the cluster; in other words you want to have a client.

With the client one can connect to the cluster purely as a client and not have any of the responsibilities a normal cluster member has. When a Hazelcast operation is performed by a client, it is forwarded to a cluster member where it will be processed. A client only needs to have the hazelcast-client.jar on the classpath (less than 200 kilobyte depending on the version) and the normal Hazelcast jar is not needed.

We are going to implement an example client where a message will be put on a queue by a client and taken from the queue by a full member. Lets start with the full member:

```
public class FullMember {
   public static void main(String[] args) throws Exception{
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      System.out.print("Full member up");
      BlockingQueue<String> queue = hz.getQueue("queue");
      for(;;)
         System.out.println(queue.take());
   }
}
```

Underneath you can see the Hazelcast client example:

```
public class Client {
   public static void main(String[] args) throws Exception {
      ClientConfig clientConfig = new ClientConfig().addAddress("
         127.0.0.1");
      HazelcastInstance client = HazelcastClient.
         newHazelcastClient(clientConfig);
```

```
        BlockingQueue<String> queue = client.getQueue("queue");
        queue.put("Hello");
        System.out.println("Message send by Client!");
        System.exit(0);
    }
}
```

The client HazelcastInstance is created based on the com.hazelcast.client.ClientConfig. This config is configured with 127.0.0.1 as address since the full member will be running on the same machine as the client. If no port is specified, port 5701,5702 and 5703 will be checked. If the cluster is running on a different port e.g. 6701, it can be specified like this '.addAddress("127.0.0.1:6701")'

To how the client running, first start the full member and wait till it is up. Then start the client and you will see that the server prints 'Hello'. If you look in the logging of the full member, you will see that the client never pops up as member of the cluster.

In this example we make use of programmatic configuration of the ClientConfig. It is also possible to configure a ClientConfig using configuration file:

1. using a properties based configuration file in combination with the com.hazelcast.client.config.Cl
2. using an XML based configuration file in combination with the com.hazelcast.client.config.XmlC

The advantage of configuring the Hazelcast client using a configuration file, is the client configuration can easily be pulled out of the code which makes the client more flexible. E.g. you could use a different configuration file for every environment you are working in (dev, staging, production). In some cases the static nature of the configuration files can be limiting if you need to have dynamic information e.g. the addresses. What you can do is first load the ClientConfig using a configuration file, and adjust the dynamic fields.

8.1 Reusing the client

A client is designed to be shared between threads and you want to prevent creating an instance per request because it quite a heavy object because:

1. it contains a thread pool that is used for internal administration like heartbeat checking, scheduling of refreshing partitions, firing events when members are added removed etc.
2. just like a database, a client has a connection pool. These connections are used when the client executes an operation on the cluster, when the operation begins a connection is taken from this pool and when the operation completes,

the connection is returned. Setting up connections is relatively expense since it will cause load on the server and

In most cases it is best to create the client in the beginning and keep reusing it throughout the lifecycle of the client-application.

8.2 Configuration Options

In the client example, we did a minimal configuration of the ClientConfig and relied on defaults, but there is a lot that can be configured:

1. addresses: the known addresses of the cluster. It doesn't need to include all addresses, only enough to make sure that at least 1 will be online. see [failover]
2. poolSize: the maximum number of pooled connections to the member and defaults to 100. Setting this value too low means that there potentially is contention because operations need to wait till a connection becomes available. Setting it too high means that client and cluster will suffer (one of the reasons is that a socket per connection is maintained).
3. connectionTimeout: the number of milliseconds to wait till releasing a connection to a non responsive member, defaults to 60000 ms (1 minute). Once a connection has been established, the client will periodically check the connection by sending a heartbeat and expecting a response. With the connectionTimeout you can specify the maximum period a connection is allowed to go without a response to the heartbeat. If the value is set to low, it could lead to often recreating connections. If the value is set too high. it can lead to dead members being detected very late.
4. connectionAttemptLimit: the maximum numbers of times to try the addresses to connect to the cluster, defaults to 2. When a client starts or a the client looses connecting with the cluster, it will try to make a connection with one of the cluster member addresses. In some cases a client can't connect to these addresses, e.g. the cluster is not yet up or not reachable. Instead of giving up, one can increase the attempt limit to create a connection. Also have a look at the connectionAttemptPeriod.
5. connectionAttemptPeriod: the period in milliseconds between attempts to find a member in the cluster, defaults to 3000 ms.
6. listeners: enables listening to cluster state. Currently only the LifecycleListener is supported.
7. loadBalancer: see LoadBalancing for more information. Defaults to RoundRobinLB.

8. smart: [todo: copied doc] If true, client will route the key based operations to owner of the key at the best effort. Note that it uses a cached version of PartitionService.getPartitions() and doesn't guarantee that the operation will always be executed on the owner. The cached table is updated every second. defaults to true. [todo: what happens when the operation is not send to the correct member, will the member send it to the correct member?][todo: is there any reason why I don't want the smart property to be set?] If set to false, a message will always be send to the same member.
9. redoOperation: [todo: copy of doc] If true, client will redo the operations that were executing on the server and client lost the connection. This can be because of network, or simply because the member died. However it is not clear whether the application is performed or not. For idempotent operations this is harmless, but for non idempotent ones retrying can cause to undesirable effects. Note that the redo can perform on any member. If false, the operation will throw RuntimeException that is wrapping IOException. Defaults to false. [todo: should this not be something on the operation level? e.g. by implementing some marker interface]
10. group config: see [group config section]
11. socketOptions: [todo]
12. serializationConfig: configures how to serialize and deserialize on the client side. For all classes that are deserialized to the client the same serialization needs to be configured as done for the cluster. For more information see [chapter serialization]
13. proxyFactoryConfig: [todo]
14. socketInterceptor: [todo]
15. classLoader: [todo]
16. credentials: can be used to do authentication and authorisation. This functionality is only available in the Enterprise version of Hazelcast.

8.3 LoadBalancing

. When a client connects to the cluster, it will have access to the full list of members and it will be kept in sync, even if the ClientConfig only has a subset of members. If a operation needs to be sent to a specific member, it will directly be sent to that member. If a operation can be executed on any member, Hazelcast does automatic load balancing over all members in the cluster.

One of the very new cool features of Hazelcast 3.0 is that the routing mechanism is pulled out into an interface:

```
public interface LoadBalancer {
    void init(Cluster cluster, ClientConfig config);
    Member next();
}
```

This means that if you have specific routing requirements, e.g. load balance on cpu load, memory load, queue sizes etc, these requirements can be met by creating a custom LoadBalancer implementation. I'm sure that in the next releases of Hazelcast some of these implementations will be provided out of the box. If you are going to implement a custom loadBalancer, you can listen to member changes using:

```
Cluster cluster = hz.getCluster();
cluster.addMembershipListener(thelistener);
```

The MembershipListener functionality makes it easy to to create a deterministic loadbalancer since the MembershipListener:

1. will not be called concurrently
2. init method will be called with a set of all current members
3. no events will be lost between calling init and memberAdded or memberRemoved
4. memberAdded and memberRemoved method will be called in the order the events happened within the cluster. There is a total ordering of membership events since they will be coordinated from the master node.

LoadBalancer instances should not be shared between clients; every client should gets its own instance and the loadbalancer can be configured from the ClientConfig.

8.4 Failover

In a production environment you want the client to support failover to increase high availability. This is realized by in 2 parts. The first part is configuring multiple member addresses in the ClientConfig. As long as one of these members is online, the client will be able to connect to the cluster and will know about all members in the cluster. The second part is the responsibility of the LoadBalancer implementation. It can register itself as a MembershipListener and receives a list of all members in the cluster and will be notified if members are added or are removed. The LoadBalancer can use this update to date list of member addresses for routing.

8.5 Group Configuration

To prevent clients from joining a cluster, it is possible to configure the cluster group the client is able to connect to. On this cluster group the group name and the password can be set:

```
ClientConfig config = new ClientConfig()
    .addAddress("127.0.0.1");
config.getGroupConfig()
    .setName("group1")
    .setPassword("thepassword");
HazelcastInstance client = HazelcastClient.newHazelcastClient(
   config);
```

The groupname defaults to 'dev' and the password defaults to 'dev-pass'. For more information see [network configuration: cluster groups].

8.6 Sharing classes

In some cases you need to share classes between the client and the server. Of course you can give all the classes from the server to the client, but often this is undesirable due to tight coupling, security, copyright issues etc. If you don't want to share all the classes of the server with the client, create a separate API project (in Maven terms this could be a module) containing all the shared classes and interfaces and share this project between client and server.

One word of advice; watch out with sharing domain objects between client and server. This can cause a tight coupling since the client starts to see the internals of your domain objects. A recommended practice is to introduce special objects that are optimized for client/server exchange; Data Transfer Objects (DTO's). They cause some duplication, but having some duplication is better to deal with than tight coupling, which can make a system very fragile.

8.7 SSL

In Hazelcast 3 it is possible to encrypt communication between client and cluster using SSL. This means that all network traffic, which not only includes normal operations like a map.put but also passwords in credentials and GroupConfig, can't be read and potentially modified.

keytool -genkey -alias hazelcast -keyalg RSA -keypass password -keystore hazelcast.ks -storepass password keytool -export -alias hazelcast -file hazelcast.cer -keystore hazelcast.ks -storepass password keytool -import -v -trustcacerts -alias hazelcast -keypass password -file hazelcast.cer -keystore hazelcast.ts -storepass password

Example SSL configuration of the server:

```
public class Client {
    public static void main(String[] args) throws Exception {
        System.setProperty("javax.net.ssl.keyStore",
            new File("hazelcast.ks").getAbsolutePath());
        System.setProperty("javax.net.ssl.trustStore",
             new File("hazelcast.ts").getAbsolutePath());
        System.setProperty("javax.net.ssl.keyStorePassword", "
           password");

        ClientConfig config = new ClientConfig();
        config.addAddress("127.0.0.1");
        config.setRedoOperation(true);
        config.getSocketOptions().setSocketFactory(new
           SSLSocketFactory());

        HazelcastInstance client = HazelcastClient.
           newHazelcastClient(config);
        BlockingQueue <String> queue = client.getQueue("queue");
        queue.put("Hello!");
        System.out.println("Message send by client!");
        System.exit(0);
    }
}
```

Example SSL configuration of the client:

```
public class Member {
    public static void main(String[] args) throws Exception {
        System.setProperty("javax.net.ssl.keyStore",
            new File("hazelcast.ks").getAbsolutePath());
        System.setProperty("javax.net.ssl.trustStore",
            new File("hazelcast.ts").getAbsolutePath());
        System.setProperty("javax.net.ssl.keyStorePassword", "
           password");

        Config config = new Config();
        config.getNetworkConfig().setSSLConfig(new SSLConfig().
           setEnabled(true));

        HazelcastInstance hz = Hazelcast.newHazelcastInstance(
           config);
        BlockingQueue<String> queue = hz.getQueue("queue");
        System.out.println("Full member up");
        for (; ; )
            System.out.println(queue.take());
    }
```

```
}
```

8.8 What happened to the lite member?

If you have been using Hazelcast 2.x you might remember the option to create either a lite member or a native member. A lite member is seen as part of the cluster, although it doesn't host any partitions. Because it is part of the cluster, it knows about the other members. And therefor it knows about routing requests to correct member, which could improve performance compared to the native client.

The lite member also had some serious drawbacks:

1. because lite members are seen as members in the cluster, they send heartbeats/pings to each other and check each others statuses continuously. So if the number of lite members compared to the number of real members is high, and clients join/leave frequently, it can influence the health of the cluster.
2. although a lite member doesn't host any partitions, it will run tasks from the Hazelcast executors. Personally I always found this undesirable since you don't want to have a client run cluster tasks.

With Hazelcast 3.x a client always knows about all members and therefor can route requests to the correct member, without being part of the cluster.

8.9 Good to know

Shutdown: If you don't need to client anymore, it is very important to shut it down using the 'shutdown' method or using the LifeCycleService:

```
client.getLifecycleService().shutdown();
```

The reason why the shutdown is important, especially for short lived clients, is that the shutdown releases resources. It will shutdown the client thread-pool and the connection-pool. When a connection is closed, the client/member socket is closed and the ports are released; making them available for new connections. Network traffic also is reduces sincc the heartbeat doesn't need to be send anymore. And the client resources running on the cluster, like the EndPoint or e.g. a distributed Locks, that has been acquired by the client, are released. If the client is not shutdown and resources like the Lock have not been released, every thread that wants to acquire the lock is going to deadlock.

SPI: The Hazelcast client can also call SPI operations, see [SPI chapter]. But you need to make sure that the client has access the the appropriate classes and interfaces.

2 way clients: There are cases where you have a distributed system split in different clusters, but where there is a need to communicate between the clusters. Instead of creating 1 big Hazelcast cluster, it could be split up in different groups. To be able to have each group communicate with each other group, create multiple clients. So if you have 2 groups A and B, then A has a client to B and B has a client to A.

HazelcastSerializationException: if you run into this exception with the message "There is no suitable serializer for class YourObject", then you have probably forgotten to configure the SerializationConfig for the client. See [serialization]. In a lot of cases, you want to copy paste the whole serialization configuration of the server to make sure that the client en server are able to serialize/deserialize the same classes.

8.10 What is next

In this short chapter we explained a few different ways to connect to a Hazelcast cluster using a client. But there are more client solutions available: like the C# client, C++ client, Memcache Client and the Rest Client. For more information check the Client chapter of the Hazelcast reference manual.

Chapter 9

Serialization

Till so far the examples have relied on standard Java serialization by letting the objects we store in Hazelcast, implement the java.io.Serializable interface. But Hazelcast has a very advanced serialization system that supports native java serialization like Serializable and Externalizable; useful if you don't own the class and therefore can't change its serialization mechanism. But it also supports custom serialization mechanisms like DataSerializable, Portable, ByteArraySerializer and ByteStreamSerializer.

Serialization in Hazelcast works like this: when an object needs to be serialized, e.g. because it is placed in a Hazelcast data-structure like map or queue, Hazelcast first checks if it is an instance of DataSerializable or Portable. If that fails it checks if is a well known type like String, Long, Integer, byte[], ByteBuffer, Date etc, since serialization for these types can be optimised. Then it checks for user specified types [reference to ByteArraySerializer/ByteStreamSerializer]. If that fails it will fall back on Java serialization (including the Externalizable). If this also fails, the serialization fails because the class can't be serialized. This sequence of steps is useful to determine which serialization mechanism is going to be used by Hazelcast if a class implements multiple interfaces, e.g. Serializable and Portable.

Whatever serialization technology is used, if a class definition is needed, Hazelcast will not automatically download it. So you need to make sure that your application has all the classes it needs on the classpath.

9.1 Serializable

The native Java serialization is the easiest serialization mechanism to implement since often a class only needs to implement the java.io.Serializable interface:

```
public class Person implements Serializable {
   private static final long serialVersionUID = 1L;

   private String name;

   public Person(String name) {
      this.name = name;
   }
}
```

When this class is serialized, all non static non transient fields will automatically be serialized.

Make sure that a serialVersionUID is added since this prevents the JVM from calculating one on the fly which can lead to all kinds of class compatibility issues. In the examples it not always added to reduce space, but for production code there is no excuse.

One of things you need to watch out for when making use of serialization, is that because you don't have exact control on how an Object is (de)serialized, you don't control the actual byte content. In most cases this won't be an issue, but if you are making use of a method that relies on the byte-content comparisons and the byte-content of equal objects is different, then you get unexpected behaviour. An example of such a method is the Imap.replace(key,expected,update) and an example of a serialized data-structure with unreliable byte-content is a HashMap. So if your 'expected' class directly or indirectly relies on a HashMap, the replace method could fail to replace keys.

9.2 Externalizable

Another serialization technique supported by Hazelcast is the java.io.Externalizable which provides more control on how the fields are serialized/deserialized and can also help to improve performance compared to standard Java serialization. An example of the Externalizable in action:

```
public class Person implements Externalizable {
   private String name;

   public Person(String name) {
      this.name = name;
   }

   @Override
   public void readExternal(ObjectInput in)
```

```
        throws IOException, ClassNotFoundException {
    this.name = in.readUTF();
  }

  @Override
  public void writeExternal(ObjectOutput out)
        throws IOException {
    out.writeUTF(name);
  }
}
```

As you can see the writing and reading of fields is explicit and reading needs to be done in the same order as writing. Unlike the Serializable, the serialVersionUID is not required.

9.3 DataSerializable

Although Java serialization is very easy to use, it comes at a price:

1. lack of control on how the fields are serialized/deserialized.
2. suboptimal performance due to streaming class descriptors, versions, keeping track of seen objects to deal with cycles etc. This causes additional cpu load and suboptimal size of serialized data.

That is why in Hazelcast 1 the DataSerializable serialization mechanism was introduced.

To see the DataSerializable in action, lets implement on the Person class:

```
public class Person implements DataSerializable {
  private String name;

  public Person(){}

  public Person(String name) {
    this.name = name;
  }

  @Override
  public void readData(ObjectDataInput in)
        throws IOException {
    this.name = in.readUTF();
  }

  @Override
  public void writeData(ObjectDataOutput out)
        throws IOException {
    out.writeUTF(name);
  }
}
```

As you can see it looks a lot like the Externalizable functionality since also an explicit serialization of the fields is required. And just like the Externalizable, the reading fields needs to be done in the same order as they are written. Apart from implementing the DataSerializable interface, no further configuration is needed. As soon as this Person class is going to be serialized, Hazelcast checks if it implements the DataSerializable interface.

One requirement for a DataSerializable class is that it has a no arg constructor. This is needed during deserialization because Hazelcast needs to create an instance. You can make this constructor private, so that it won't be visible to normal application code.

To see the DataSerializable in action, lets have a look at the following code:

```
public class Member {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        Map<String, Person> map = hz.getMap("map");
        map.put("Peter", new Person("Peter"));
        Person p = map.get("Peter");
        System.out.println(p);
    }
}
```

If you run this, you will see:

```
Person(name=Peter)
```

IdentifiedDataSerializable

One of the problem with DataSerializable is that it uses reflection to create an instance of the class. One of the new features of Hazelcast 3 is the IdentifiedDataSerializable which relies on a factory to create the instance and therefor is faster when deserializing, since deserialization relies on creating new instances.

The first step is to modify the Person class to implement the IdentifiedDataSerializable interface:

```
public class Person implements IdentifiedDataSerializable {
    private String name;

    public Person(){}

    public Person(String name) {
        this.name = name;
    }
```

```
    @Override
    public void readData(ObjectDataInput in)
            throws IOException {
        this.name = in.readUTF();
    }

    @Override
    public void writeData(ObjectDataOutput out)
            throws IOException {
        out.writeUTF(name);
    }

    @Override
    public int getFactoryId() {
        return PersonDataSerializableFactory.FACTORY_ID;
    }

    @Override
    public int getId() {
        return PersonDataSerializableFactory.PERSON_TYPE;
    }

    @Override
    public String toString() {
        return String.format("Person(name=%s)", name);
    }
}
```

As you can see it looks a lot like the Person class from the DataSerializable, but 2 additional methods are added: getFactoryId and getId. The getFactoryId should return a unique positive number and the getId should return a unique positive number within its corresponding PersonDataSerializableFactory. So you can have IdentifiedDataSerializable implementations that return the same id, as long as the getFactoryId is different. Personally I prefer to move the id's to the DataSerializableFactory implementation so you have a clear overview.

The next part is to create a PersonDataSerializableFactory which is responsible for creating an instance of the Person class.

```
public class PersonDataSerializableFactory
        implements DataSerializableFactory{

    public static final int ID = 1;

    public static final int PERSON_TYPE = 1;

    @Override
    public IdentifiedDataSerializable create(int typeId) {
        if(typeId == PERSON_TYPE){
            return new Person();
        }else{
```

```
            return null;
        }
    }
}
```

The create method is the only method that needs to be implemented. If you have many subclasses you might consider using a switch case statement instead of a bunch of 'if-else if' statements. If a type id is received of an unknown type, null can be returned or an exception can be thrown. If null is returned, Hazelcast will thrown an exception for you.

And the last part is the registration of the PersonDataSerializableFactory in the hazelcast.xml.

```
<hazelcast>
   <serialization>
      <data-serializable-factories>
         <data-serializable-factory
             factory-id="1">PersonDataSerializableFactory</data-
                serializable-factory>
      </data-serializable-factories>
   </serialization>
</hazelcast>
```

If you look closely you see that the PersonDataSerializableFactory.FACTORY_ID has the same value as the 'factory-id' field in the XML. This is very important since Hazelcast relies on these values to find the right DataSerializableFactory when deserializing.

To see the IdentifiedDataSerializable in action, have a look at the following example:

```
public class Member {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        Map<String, Person> map = hz.getMap("map");
        map.put("Peter", new Person("Peter"));
        Person p = map.get("Peter");
        System.out.println(p);
    }
}
```

If you run it, you will see:

```
Person(name=Peter)
```

9.4 Portable

With the introduction of Hazelcast 3.0 a new serialization mechanism is added: the Portable. The cool thing about the Portable is that object creation is pulled into user space, so you control the initialisation of the Portable instances and are not forced to use a no arg constructor. For example you could inject dependencies or you could even decide to move the construction of the Portable from a prototype bean in a Spring container.

To demonstrate how the Portable mechanism works, lets create a Portable version of the Person class:

```
public class Person implements Portable {
    private String name;

    Person(){
    }

    public Person(String name) {
        this.name = name;
    }

    @Override
    public int getClassId() {
        return PortableFactoryImpl.PERSON_CLASS_ID;
    }

    @Override
    public int getFactoryId() {
        return PortableFactoryImpl.FACTORY_ID;
    }

    @Override
    public void writePortable(PortableWriter writer)
            throws IOException {
        System.out.println("Serialize");
        writer.writeUTF("name", name);
    }

    @Override
    public void readPortable(PortableReader reader)
            throws IOException {
        System.out.println("Deserialize");
        this.name = reader.readUTF("name");
    }

    @Override
    public String toString() {
        return String.format("Person(name=%s)",name);
    }
}
```

As you can see, the write method include the field names, making it possible to read particular fields without being forced to read all. This is useful for querying and indexing because it reduces overhead because deserialization isn't needed. Unlike the DataSerializable, the order of reading and writing fields isn't important since it is based on name. Also a no-arg constructor is added so that it can be initialised from the PortableFactoryImpl, but if you place it in the same package you could give it a package friendly access modifier to reduce visibility.

The last 2 interesting methods are getClassId which returns the identifier of that class. And the other method is the getFactoryId which must return the id of the PortableFactory that is going to take care of serialising and deserializing.

The next step is the 'PortableFactory' which is responsible for creating a new Portable instance based on the class id. In our case, the implementation is very simple since we only have a single Portable class:

```
import com.hazelcast.nio.serialization.*;
public class PortableFactoryImpl implements PortableFactory {
    public final static int PERSON_CLASS_ID = 1;
    public final static int FACTORY_ID = 1;

    @Override
    public Portable create(int classId) {
        switch (classId) {
            case PERSON_CLASS_ID:
               return new Person();
        }
        return null;
     }
}
```

But in practice the switch case probably will be a lot bigger. If an unmatched classId is encountered, null should be returned which will lead to a HazelcastSerializationException. A class id needs to be unique within the corresponding PortableFactory and needs to be bigger than 0. You can declare the class id in the class to serialize, but personally I prefer to add it to the PortableFactory so you have a good overview of which id's there are.

A factory id needs to be unique and larger than 0. You probably will have more than 1 PortableFactory. To make sure that every factory gets a unique factory id, I prefer to make a single class/interface where all PortableFactory id's in your system are declared, e.g.:

```
public class PortableFactoryConstant {
    public final static int PERSON_FACTORY_ID = 1;
    public final static int CAR_FACTORY_ID = 2;
```

```
    ....
}
```

The getFactoryId should make use of these constants. This prevents looking all over the place if the factory id is unique.

The last step is to configure the PortableFactory in the Hazelcast configuration:

```
<serialization>
   <portable-factories>
      <portable-factory factory-id="1">PortableFactoryImpl</
         portable-factory>
   </portable-factories>
</serialization>
```

Hazelcast can have multiple PortableFactories. You need to make sure that the factory-id in the xml is the same as in the code.

Of course we also want to see it in action:

```
public class Member {

   public static void main(String[] args) {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      Map<String, Person> map = hz.getMap("map");
      map.put("Peter", new Person("Peter"));
      System.out.println(map.get("Peter"));
   }
}
```

When we run this PortableMember, we'll see the following output:

```
Serialize
Serialize
Deserialize
Person(name=Peter)
```

As you can see the Person is serialized when it is stored in the map and it is deserialized when it is read. You might ask why Serialize is called twice. This is because for every Portable class, the first time it is (de)serialized, Hazelcast generates a new class that supports the serialization/deserialization process. And for this generation process another serialization is executed to figure out the metadata (the fields and their types).

The names of the fields are case sensitive and need to be valid java identifiers and therefore should not contain '.' or '-' for example.

9.4.1 DataSerializable vs Portable

Portable supports versioning and is language/platform independent which makes it useful for client/cluster communication. Another advantage is that is very performant for map queries, since it avoids full serialization because data can be retrieved on the field level. Otherwise, if the serialization only is needed for intra cluster communication, then DataSerializable is still a good alternative.

Object traversal

If a Portable, has a Portable field, the write and read operations need to be forwarded to that object. For example if we would add a Portable address field to Person:

```
public void writePortable(PortableWriter writer)
      throws IOException {
   writer.writeUTF("name", name);
   writer.writePortable("address", address);
}
public void readPortable(PortableReader reader)
      throws IOException {
   this.name = reader.readUTF("name");
   this.address = reader.readPortable("address");
}
```

If the field is of type Portable and null, the PortableWriter.writePortable(String fieldName, Portable portable) method will complain about the null. This is because with a null value, the type of the field is not known and this causes problems with the platform independent nature of Portable. In that case the PortableWriter.writePortable(String fieldName, int classId, Portable portable) method can be called where an explicit class id needs to be passed.

If the object is not a Portable, primitive, array or String then there is no direct support for serialization. Of course, you could transform the object using Java serialization to a byte array, but this would mean that the platform independence is lost. A better solution is to create some form of String representation, potentially using XML, so that platform compatibility is maintained. The methods readUTF/writeUTF can perfectly deal with null Strings so passing null object references is no problem.

Serialize DistributedObject

Serialization of the DistributedObject is not provided out of the box, so you can't put for example an ISemaphore on an IQueue on one machine, and take it from another. But there are solutions to this problem.

One solution is to pass the id of the DistributedObject, perhaps in combination with the type. When deserializing, look up the object in the HazelcastInstance; e.g. in case of an IQueue you can call HazelcastInstance.getQueue(id) or the Hazelcast.getDistributedObject. Passing the type is useful if you don't know the type of the DistributedObject.

If you are deserializing your own Portable Distributed Object object and it receives an id that needs to be looked up; the class can implement the HazelcastInstanceAware interface: [reference to HazelcastInstanceAware section]. Since the HazelcastInstance is set after deserialization, you need to store the id's first and you could do the actual retrieval of the DistributedObjects in the setHazelcastInstance method.

9.4.2 Serializing raw data

When using the Portable functionality, the fieldname is added so that the fields can be retrieved easily and that te field can be indexed and used within queries without needing to deserialize the object. But in some cases this can be a lot of overhead. If this is an issue, it is possible to write raw data using the PortableWriter.getRawDataOutput() method and can be read using the PortableReader.getRawDataInput() method. Reading an writing raw data should be the last reading and writing operations on the PortableReader and PortableWriter.

Cycles

One thing to look out for, this also goes for the DataSerializable, are cycles between objects because it can lead to a stack overflow. Standard Java serialization protects against this, but since manual traversal is done in Portable objects, there is no out of the box protection. If this is an issue, you could store a map in a ThreadLocal that can be used to detect cycles and a special placeholder value could be serialized to end the cycle.

Subtyping

Subtyping with the Portable functionality is easy, let every subclass have its own unique type id and add these id's to the switch/case in the PortableFactory so that the correct class can be instantiated.

Versioning

In practice it can happen that multiple versions of same class are serialized and deserialized; imagine a Hazelcast client with an older Person class compared to the cluster. Luckily the Portable functionality supports versioning. In the configuration you can explicitly pass a version using the '¡portable-version¿' tag (defaults to 0):

```
<serialization>
   <portable-version>1</portable-version>
   <portable-factories>
      <portable-factory factory-id="1">PortableFactoryImpl</
         portable-factory>
   </portable-factories>
</serialization>
```

When a Portable instance is deserialized, apart from the serialized fields of that Portable also metadata like the class id and the version is stored. That is why it is important that every time you make a change in the serialized fields of a class, that the version is changed. In most cases incrementing the version is the simplest approach.

Adding fields to a Portable is simple although you probably need to work with default values if an old Portable os deserialized.

Removing fields can lead to problem if a new version of that Portable (with the removed field) is deserialized on a client which depends on that field.

Renaming fields is simpler because there the Portable mechanism doesn't rely on reflection; so there is no automatic mapping of fields on the class and fields in the serialized content.

An issue to watch out for is changing the field type, although Hazelcast can do some basic type upgrading (e.g. int to long or float to double).

Renaming the Portable is simple since the name of the Portable is not stored as metadata, but the class id (which is a number) is stored.

Luckily Hazelcast provides access to the metadata of the to deserialize object through the PortableReader; the version, available fields, the type of the fields etc can be retrieved. So you have full control on how the deserialization should take place.

HazelcastInstanceAware: the PortableFactory can't be combined with the HazelcastInstanceAware. There is a feature request so perhaps this functionality is going to be added in the future.

9.5 StreamSerializer

One of the additions the Hazelcast 3 is to use a stream for serializing and deserializing data by implementing the StreamSerializer. The StreamSerializer is not only practical if you want to create your own implementations, but you can also use it to adapt an external serialization library like JSON, protobuf, Kryo etc.

Lets start with a very simple object we are going to serialize using a StreamSerializer.

```
public class Person {
   private String name;
   public Person(String name) {
      this.name = name;
   }
}
```

As you can see, there are no interfaces to implement. There also is no need for a no argument constructor.

The next step is the StreamSerializer implementation for this Person class:

```
public class PersonStreamSerializer
       implements StreamSerializer<Person> {

   @Override
   public int getTypeId() {
      return 1;
   }

   @Override
   public void write(ObjectDataOutput out, Person person)
          throws IOException {
      out.writeUTF(person.getName());
```

```
    }

    @Override
    public Person read(ObjectDataInput in) throws IOException {
        String name = in.readUTF();
        return new Person(name);
    }

    @Override
    public void destroy() {
    }
}
```

As you can see the implementation is quite simple. The ObjectDataOutput and ObjectDataInput have methods available for primitives like int, boolean etc but also for String: writeUTF/readUTF which can safely deal with null and also for objects. See object traversal 9.5.

If practice classes probably have more fields. If you are writing the fields, make sure that they are read in the same order as they are written.

On thing worth mentioning is that the type id needs to be unique so that on deserialization, Hazelcast is able to figure out which serializer should be used to deserialize the object. Hazelcast has claimed the negative id's and will throw an error if your type id is smaller than 1.

A practical way to generate unique id's is to use a class (or interface) where you define all type id's in your system:

```
public final class MySerializationConstants {
    private static int ID = 1;
    public static final int PERSON_TYPE = ID++;
    public static final int CAR_TYPE = ID++;
    ...
```

And to use these type id's in the 'getTypeId' method:

```
public class PersonStreamSerializer
        implements StreamSerializer<Person> {

    @Override
    public int getTypeId() {
        return MySerializationConstants.PERSON_TYPE;
    }
    ...
```

It is very important never to change the order of the type id's when you have old deserialized instances somewhere. This is because a change of the order will change the actual value of the type id and Hazelcast will not be able to correctly

deserialize objects that were created using the old order.

The last step is the registration of the PersonStreamSerializer in the hazelcast.xml:

```
<serialization>
   <serializers>
      <serializer
          type-class="Person">PersonStreamSerializer</serializer>
   </serializers>
</serialization>
```

In this particular case we have registered for the 'Person' class the serializer 'PersonStreamSerializer. When Hazelcast is going to serialize an object, it lookups the serializer registered for the class of that object. Hazelcast is quite flexible; if it fails to find a serializer for a a particular class, it first tries to match based on superclasses and then on interfaces. So in theory you could create a single StreamSerializer that is able to deal with a class hierarchy if that StreamSerializer is registered for the root class of that class hierarchy. If you are going to make use of this approach, then you need to write sufficient data to the stream so that on deserialization you are able to figure out exactly which class needs to be instantiated.

It isn't possible to create StreamSerializers for well known types like the Long, String, primitive arrays etc since Hazelcast already registered ones.

The see the serializer in action:

```
public class Member {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        Map<String, Person> map = hz.getMap("map");
        Person person = new Person("peter");
        map.put(person.getName(), person);
        System.out.println(map.get(person.getName()));
    }
}
```

And you will get the following output:

```
Person{name='peter'}
```

Object Traversal

In practice you often need to deal with object graphs and luckily this is quite easy. To create the graph we add the Car class where each car has a colour and an owner.

```
public class Car  {
    private String color;
    private Person owner;

    public Car(Person owner,String color) {
        this.color = color;
        this.owner = owner;
    }

    public String getColor() {
        return color;
    }

    public Person getOwner() {
        return owner;
    }

    @Override
    public String toString() {
        return "Car{" +
                "color='" + color + '\'' +
                ", owner=" + owner +
                '}';
    }
}
```

The interesting part is the StreamSerializer for the car; especially the ObjectDataOutput.writeObject and ObjectDataInput.readObject methods.

```
public class CarStreamSerializer
       implements StreamSerializer<Car> {

    @Override
    public int getTypeId() {
        return MySerializationConstants.CAR_TYPE;
    }

    @Override
    public void write(ObjectDataOutput out, Car car)
           throws IOException {
        out.writeObject(car.getOwner());
        out.writeUTF(car.getColor());
    }

    @Override
    public Car read(ObjectDataInput in) throws IOException {
        Person owner = in.readObject();
        String color = in.readUTF();
        return new Car(owner,color);
    }

    @Override
    public void destroy() {
    }
}
```

When the writeObject is called, Hazelcast will lookup a serializer for the particular type. Hazelcast has serializers available for the wrapper types like Long, Boolean etc. And luckily the writeObject (and readObject) are perfectly able to deal with null.

The complete the example, the CarStreamSerializer also needs to be registered:

```
<serialization>
   <serializers>
      <serializer
         type-class="Person">PersonStreamSerializer</serializer>
      <serializer
         type-class="Car">CarStreamSerializer</serializer>
   </serializers>
</serialization>
```

And if you run the following example:

```
public class Member {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        Map<String, Car> map = hz.getMap("map");

        Person owner = new Person("peter");
        Car car = new Car(owner, "red");

        map.put("mycar", car);
        System.out.println(map.get("mycar"));
    }
}
```

You will get the following output:

```
Car{color='red', owner=Person{name='peter'}}
```

So traversing object graphs for serialization and reconstructing object graphs on deserialization is quite simple. One thing you need to watch out for is cycles, see [link: portable:cycles]

Collections

If the field of an object that needs to be serialized with the stream serializer, then currently there is no other solution than to write a custom serializer for that field. Support for collection serializers probably will be added in the near future, but for the time being you might have a look at the following 2 implementations to give you an impression. First the serializer for the LinkedList:

```
public class LinkedListStreamSerializer
```

```
        implements StreamSerializer<LinkedList> {

    @Override
    public int getTypeId() {
        return MySerializationConstants.LINKEDLIST_TYPE;
    }

    @Override
    public void write(ObjectDataOutput out, LinkedList l)
            throws IOException {
        out.writeInt(l.size());
        for(Object o: l){
            out.writeObject(o);
        }
    }

    @Override
    public LinkedList read(ObjectDataInput in)
            throws IOException {
        LinkedList l = new LinkedList();
        int size = in.readInt();
        for(int k=0;k<size;k++){
           l.add(in.readObject());
        }
        return l;
    }

    @Override
    public void destroy() {
    }
}
```

And now the serializer for the HashMap.

```
public class HashMapStreamSerializer
        implements StreamSerializer<HashMap> {

    @Override
    public int getTypeId() {
        return MySerializationConstants.HASHMAP_TYPE;
    }

    @Override
    public HashMap read(final ObjectDataInput in)
            throws IOException {
        int size = in.readInt();
        HashMap m = new HashMap(size);
        for(int k=0;k<size;k++){
            Object key = in.readObject();
            Object value = in.readObject();
            m.put(key,value);
        }
        return m;
    }

    @Override
```

```
    public void write(final ObjectDataOutput out, final HashMap m)
            throws IOException {
        out.writeInt(m.size());
        Set<Map.Entry> entrySet = m.entrySet();
        for(Map.Entry entry: entrySet){
            out.writeObject(entry.getKey());
            out.writeObject(entry.getValue());
        }
    }

    @Override
    public void destroy() {
    }
}
```

It is very important that you know which collections classes are being serialized. If there is no collection serializer registered, the system will default to the GlobalSerializer and this defaults normal serialization and this might not be the behaviour you are looking for.

9.5.1 Kryo StreamSerializer

Writing a customer serializer like a StreamSerializer can be a lot of work. Luckily there are a lot of serialization libraries that take care of all this work. Kryo is one of these librarties we use at Hazelcast and is quite fast, flexibile and results in small byte arrays. It can also deal with object cycles.

Lets start with a simple Person class:

```
public class Person{
    private String name;

    private Person(){}

    public Person(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return String.format("Person(name=%s)", name);
    }
}
```

As you can see there is no interface that needs to be implemented. It also is no problem if the class is implementing a serialization interface like Serializable since it will be ignored by Hazelcast.

The Kryo instance is not threadsafe and therefor you can't create PersonKryoSerializer with a Kryo instance as field. But since the Kryo instance is relatively expensive to create, we want to reuse the instance. That is why the Kryo instance is put on a threadlocal.

```
public class PersonKryoSerializer implements StreamSerializer<
    Person> {

    private static final ThreadLocal<Kryo> kryoThreadLocal
            = new ThreadLocal<Kryo>() {
        @Override
        protected Kryo initialValue() {
            Kryo kryo = new Kryo();
            kryo.register(Person.class);
            return kryo;
        }
    };

    @Override
    public int getTypeId() {
        return 2;
    }

    @Override
    public void write(ObjectDataOutput objectDataOutput, Person
        product)
            throws IOException {
        Kryo kryo = kryoThreadLocal.get();
        Output output = new Output((OutputStream) objectDataOutput
            );
        kryo.writeObject(output, product);
        output.flush();
    }

    @Override
    public Person read(ObjectDataInput objectDataInput)
            throws IOException {
        InputStream in = (InputStream) objectDataInput;
        Input input = new Input(in);
        Kryo kryo = kryoThreadLocal.get();
        return kryo.readObject(input, Person.class);
    }

    @Override
    public void destroy() {
    }
}
```

As you can see the PersonKryoSerializer is relatively simple to implement. The nice thing is that Kryo takes care of cycle detection and produces much smaller serialized data than Java serialization. For one of our customer we managed to the size of MapEntries down from 15 kilobyte average using Java Serialization, to less than 6 kilobyte. When we enabled Kryo compression, we even managed to get

it below 3 kilobyte.

The PersonKryoSerializer needs to be configured in Hazelcast:

```
<hazelcast>
    <serialization>
        <serializers>
            <serializer type-class="Person">PersonKryoSerializer</
               serializer>
        </serializers>
    </serialization>
</hazelcast>
```

And when we run the following example code:

```
public class Member {

    public static void main(String[] args) {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        Map<String, Person> map = hz.getMap("map");
        map.put("Peter", new Person("Peter"));
        Person p = map.get("Peter");
        System.out.println(p);
        System.exit(0);
    }
}
```

We'll see the following outout:

```
Person(name=Peter)
```

In the previous example we showed how Kryo can be implemented as a StreamSerializer. And the cool thing is that you can just plug in serializer for a particular class; no matter if that class already is implementing a different serialization strategy like Serializable. If you don't have the chance to implement Kryo as StreamSerializer, then you can also directly implement the serialization on the class. This could be done using the DataSerializable and (de)serialize each field using Kryo. This approach is especially useful if you are still working on Hazelcast 2.x. Kryo is not the only serializable library, you also might want to have a look at Jackson Smile, Protobuf etc.

9.6 ByteArraySerializer

An alternative to the StreamSerializer is the ByteArraySerializer. With the ByteArraySerializer the raw bytearray internally used by Hazelcast is exposed. This is practical if you are working with a serialization library that works with bytearrays instead of streams.

The following code example show the ByteArraySerializer in action:

```
public class PersonByteArraySerializer
        implements ByteArraySerializer<Person> {

    @Override
    public void destroy() {
    }

    @Override
    public int getTypeId() {
        return 1;
    }

    @Override
    public byte[] write(Person object) throws IOException {
        return object.getName().getBytes();
    }

    @Override
    public Person read(byte[] buffer) throws IOException {
        String name = new String(buffer);
        return new Person(name);
    }
}
```

The PersonByteArraySerializer can be configured in the same way as the StreamSerializer is configured:

```
<serialization>
   <serializers>
      <serializer
          type-class="Person">PersonByteArraySerializer</
             serializer>
   </serializers>
</serialization>
```

9.7 Global Serializer

The new Hazelcast serialization functionality also makes it possible to configure a global serializer in case there are no other serializers found:

```
<serialization>
   <serializers>
      <global-serializer>PersonStreamSerializer
      </global-serializer>
   </serializers>
</serialization>
```

There can only be a single global serializer and for this global serializer the StreamSerializer.getTypeId method doesn't need to return a relevant value.

The global serializer can also be a ByteArraySerializer.

9.8 HazelcastInstanceAware

In some cases when an object is deserialized, it needs access to the HazelcastInstance so that DistributedObjects can be accessed. This can be done by implementing HazelcastInstanceAware, e.g.:

```
public class Person implements
      Serializable, HazelcastInstanceAware {

   private static final long serialVersionUID = 1L;

   private String name;
   private transient  HazelcastInstance hz;

   public Person(String name) {
      this.name = name;
   }

   @Override
   public void setHazelcastInstance(HazelcastInstance hz) {
      this.hz = hz;
      System.out.println("hazelcastInstance set");
   }

   @Override
   public String toString() {
      return String.format("Person(name=%s)",name);
   }
}
```

After this person is deserialized, Hazelcast will check if the object implements HazelcastInstanceAware and will call the setHazelcastInstance method. The hz field needs be transient since it should not be serialized.

Injecting a HazelcastInstance into a domain object (an Entity) like Person isn't going to win you a beauty-contest. But it is a technique I often use in combination with Runnable/Callable implementations that are executed by an IExecutorService that sends them to another machine. After deserialization of such a task, the implementation of the run/call method often needs to access the HazelcastInstance.

A best practice for implementing the setHazelcastInstance method is only to set the HazelcastInstance field and not to be tempted to execute operations on the HazelcastInstance. The reason behind this is that for some HazelcastInstanceAware

implementations, the HazelcastInstance isn't fully up and running yet when it is injected.

You need to be careful with using the HazelcastInstanceAware on anything else than the 'root' object that is serialized. Hazelcast sometimes optimises local calls by skipping serialization and some serialization technologies, like Java serialization, don't allow for applying additional logic when an object graph is deserialized. In these cases only the root of the graph is checked if it implements HazelcastInstanceAware, but the graph isn't traversed.

9.8.1 UserContext

Obtaining other dependencies than a HazelcastInstance was a bit more complicated in Hazelcast 2.x. Often the only way out is to rely on some form of static field. But luckily Hazelcast 3 provides a new solution using the user context: a (Concurrent)Map that can be accessed from the HazelcastInstance using the getUserContext() method. In the user context arbitrary dependencies can be placed using some key as String.

Lets start with such an EchoService dependency we want to make available in a EchoTask which is going to be executed using an Hazelcast distributed executor:

```
public class EchoService{
    public void echo(String msg){
        System.out.println(msg);
    }
}
```

As you can see there are no special requirements for this dependency; no interfaces to implement. It is just an ordinary POJO.

This EchoService dependency needs to be injected in the UserContext and then it can be found when we execute the EchoTask.

```
public class Member {
    public static void main(String[] args){
        EchoService echoService = new EchoService();

        Config config = new Config();
        config.getUserContext().put("echoService",echoService);
        HazelcastInstance hz = Hazelcast.newHazelcastInstance(
            config);

        IExecutorService executor = hz.getExecutorService("
            echoExecutor");
        executor.execute(new EchoTask("hello"));
```

```
    }
}
```

As you can see injecting a dependency into the UserContext is quite simple. It is important to understand that Hazelcast doesn't provide support for injecting dependencies into the user context from the XML configuration since this would require knowledge of how to create the actual dependency. The Hazelcast configuration is purely meant as a configuration mechanism for Hazelcast and not as general purpose object container like Spring. So you need to add the dependencies to the UserContext programmatically.

The last part is retrieval of the dependency. The first thing that needs to be done is the implement the HazelcastInstanceAware interface to inject the HazelcastInstance. From this HazelcastInstance we can retrieve the UserContext by calling the getUserContext method:

```
public class EchoTask
             implements Runnable, Serializable,
                        HazelcastInstanceAware {

    private transient HazelcastInstance hz;
    private final String msg;

    public EchoTask(String msg) {
        this.msg = msg;
    }

    @Override
    public void run() {
        EchoService echoService =
                (EchoService)hz.getUserContext().get("echoService"
                   );
        echoService.echo(msg);
    }

    @Override
    public void setHazelcastInstance(HazelcastInstance hz) {
        this.hz = hz;
    }
}
```

If we run this code, we'll see:

```
hello
```

It is not only possible to configure user-context on the Config, but you can also directly configure the user-context of the HazelcastInstance. This is practical if you need to add dependencies on the fly. However don't forget to clean up what you put in the user-context, else you might run into resources problems like an

OutOfMemoryError.

Perhaps needless to mention; changes made in the user-context are local to a member only. Other members in the cluster are not going to observe changes in the user-context of one member. So if you need to have that EchoService available on each member, you need to add it to the user-context on each member.

Important to know is that when a HazelcastInstance is created using some Config instance, a new user-context ConcurrentMap is created and the content of the user-context of the Config copied. So changes made to the user-context of the HazelcastInstance will not reflect on other HazelcastInstance created using the same Config instance.

9.9 ManagedContext

In some cases when a serialized object is deserialized not all of its fields can be deserialized since they are 'transient'. These fields could be important data-structures like executors, database connections etc. Luckily Hazelcast provides a mechanism that is called when an object is deserialized and gives the ability to 'fix' the object, by setting missing fields, call methods, wrap it inside a proxy etc, so it can be used. This mechanism is called the 'ManagedContext' and can be configured on the SerializationConfig.

In the following example we have a DummyObject with a serializable field: ser and a transient field: trans:

```
class DummyObject implements Serializable {
    transient Thread trans = new Thread();
    String ser = "someValue";

    @Override
    public String toString() {
        return "DummyObject{" +
                "ser='" + ser + '\'' +
                ", trans=" + trans +
                '}';
    }
}
```

So when this object is deserialized, the serializable field will be set. But the transient field will be null. To prevent this from happening we can create a ManagedContext implementation that will restore this field:

```
class ManagedContextImpl implements ManagedContext {
```

```
    @Override
    public Object initialize(Object obj) {
        if (obj instanceof DummyObject) {
            ((DummyObject) obj).trans = new Thread();
        }
        return obj;
    }
}
```

When an object is deserialized, the initialise method will be called. In our case we are going to restore the transient field. To see the ManagedContextImpl in action, have a look at the following code:

```
public class Member {

   public static void main(String[] args) throws Exception {
      Config config = new Config();
      config.setManagedContext(new ManagedContextImpl());
      HazelcastInstance hz = Hazelcast.newHazelcastInstance(config
         );

      Map<String, DummyObject> map = hz.getMap("map");
      DummyObject input = new DummyObject();
      System.out.println(input);

      map.put("1", input);
      DummyObject output = map.get("1");
      System.out.println(output);
      System.exit(0);
   }
}
```

When this code is run, you will see output like:

```
DummyObject{ser='someValue', trans=Thread[Thread-2,5,main]}
DummyObject{ser='someValue', trans=Thread[Thread-3,5,main]}
```

As you can see the transient field has been restored to a new thread.

Hazelcast currently provides one ManagedContext implementation, the SpringManagedContext for integration with Spring. But if you are integrating with different products, e.g. Guice, you could provide your own ManagedContext implementation.

If you need to have dependencies in your ManagedContext, you can let it implement the HazelcastInstanceAware interface. Custom dependencies you can retrieve using the 9.8.1.

You need to be careful with using the ManagedContext on anything else than

the 'root' object that is serialized. Hazelcast sometimes optimises local calls by skipping serialization and some serialization technologies, like Java serialization, don't allow for applying additional logic when an object graph is deserialized. In these cases only the root of the object graph be offered to the ManagedContext, but the graph isn't traversed.

9.10 Good to know

Nested operations: DataSerializable and Portable instances are allowed to call operations on Hazelcast that leads to new serialize/deserialize operations and unlike Hazelcast 2.x it doesn't lead to StackOverflowErrors.

Thread-safety: the serialization infrastructure, e.g. classes implementing (Data)Serializable, Portable and support structures like TypeSerializer or PortableFactory, need to be threadsafe since the same instances will be access by concurrent threads.

Encryption for in memory storage: In some cases, having raw data in memory, is a potential security risk. This problem can be solved by modifying the serialization behaviour of the class so that it encrypts the data on writing and decrypts on reading. In some cases, e.g. storing a String in a map, the instance needs to be wrapped in a different type (e.g. EncryptedPortableString) to override the serialization mechanism.

Performance: Serialization and deserialization will have an impact on performance, no matter how fast the serialization solution is. That is why you need to be careful with operations on Hazelcast data-structures. For example iterating over a normal HashMap is very fast, since no serialization is needed, but iterating over a Hazelcast distributed map is a lot slower. Not only because there is potential remoting involved, but also because data needs to be deserialized. I have burned myself on this issue because it isn't always immediately obvious from the code.

Performance comparisons: for on overview of performance comparisons between the different serialization solutions, you could have a look at the following blogpost: http://tinyurl.com/k4dkgt2

Mixing serializers: If an object graph is serialized, it can happen that different parts of the graph are serialized using different serialization technologies. It could

be that some part are serialized with Portable, other parts with StreamSerializers and Serializers. Normally this won't be an issue, but if you need to exchange these classes with the outside world it is best to have ever serialized using Portable.

In Memory: Currently all serialization is done in memory. If you are dealing with large object graphs or large quantities of data, you need to keep this in the back of your mind. There is an feature request that makes it possible to use streams between members and members/clients and overcomes this in memory limitation. Hopefully this will be implemented in the near future.

Factory id's: of different serialization technologies e.g. Portable vs IdentifiedDataSerializable don't need to be unique.

9.11 What is next

In this chapter we have seen different forms of serialization; making serialization extremely flexible, especially with the Portable and the TypeSerializers. In most cases this will be more than sufficient. But if you ever run into a limitation you could, create a task in http://github.com/hazelcast/hazelcast and perhaps it will be added to the next Hazelcast release.

Chapter 10

Transactions

Till so far the examples didn't contain any transactions, but transactions can make life a lot easier since they provide:

1. Atomicity: without atomicity it could be that some of the operations on Hazelcast structures succeeds while other failed. Leaving the system in an inconsistent state.
2. Consistency: moves the state of the system from one valid state to the next.
3. Isolation: the transaction should behave as if it was the only transaction running. Normally there are all kinds of isolation levels that allow certain anomalies to happen.
4. Durability: makes sure that if a system crashes after a transaction commits, that nothing gets lost.

There are some fundamental changes in the transaction API of Hazelcast 3. In Hazelcast 2.x some of the data-structures could be used with or without a transaction, but in Hazelcast 3 transactions only are possible on explicit transactional data-structures, e.g. the TransactionalMap, TransactionalMultiMap, TransactionalQueue and the TransactionalSet. The reason behind this design choice is that not all operations can be made transactional, or if they can be made transactional, they would have huge performance/scalability implications. So to prevent running into surprises, transactions are only available on explicit transactional data-structures.

Another change in the transaction API, is that the TransactionContext is the new interface to use. It supports the same basic functionality as the Hazelcast 2.x Transaction, like begin, commit and rollback a transaction. But it also supports getting access to transactional data-structures like the TransactionalMap and TransactionalQueue.

Underneath you can find an example of the transaction API in practice:

```
public class TransactionalMember {
   public static void main(String[] args) {
      HazelcastInstance hz = Hazelcast.newHazelcastInstance();
      TransactionContext txCxt = hz.newTransactionContext();
      TransactionalMap<String, String> map = txCxt.getMap("map");
      txCxt.beginTransaction();
      try {
         map.put("1", "1");
         map.put("2", "2");
         txCxt.commitTransaction();
      } catch (Throwable t) {
         txCxt.rollbackTransaction();
      }
      System.out.println("Finished");
      System.exit(0);
   }
}
```

As you can see using a transaction is quite simple.

In the example you see that the transactional map is retrieved within the transaction. It isn't allowed to retrieve a transactional data-structure in one transaction and reuse it in another one. The retrieved object, in this case the TransactionalMap, is a proxy to the real data-structure and will contain transaction specific state like cache. So it should not be reused. Of cours the same transactional data-structure can be retrieved multiple times within the same transaction.

10.1 Configuring the TransactionalMap

the TransactionalMap is backed up by a normal IMap. And you can configure a TransactionalMap using the configuration mechanism of the IMap. Imagine that you have a TransactionalMap called 'employees', then you can configure this TransactionalMap using:

```
<map name="employees">
    ....
</map>
```

So a TransactionalMap will have all the configuration options you have on a normal IMap. The same goes for the TransactionalMultiMap which is backed up by a MultiMap.

Because the TransactionalMap is build on top of the IMap, the transactionalMap can be loaded as an IMap:

```
public class TransactionalMember {
   public static void main(String[] args) {
```

```
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        TransactionContext txCxt = hz.newTransactionContext();
        TransactionalMap<String,Employee> employees = context.getMap
            ("employees");
        employees.put("1",new Employee());
        txCxt.commitTransaction();

        IMap employeesIMap = hz.getMap("employees");
        System.out.println(employeesIMap.get("1"));
    }
}
```

You will see that the employee that was put in the TransactionalMap, being retrieved from the IMap. In practice you probably never want to do this.

If you have enabled JMX, then the TransactionalMap and TransactionalMultiMap will appear as a normal Map/MultiMap.

10.2 TransactionOptions

In some cases the default behavior of the Transaction isn't working and needs to be fine tuned. With the Hazelcast transaction API this can be done by using the TransactionOptions object and pass it to the HazelcastInstance.newTransactionContext(Transaction method.

Currently Hazelcast provides the following configuration options:

1. timeoutMillis: the number of milliseconds a transaction will hold a lock. Defaults to 2 minutes. In most cases this timeout is enough since transaction should be executed quickly. [todo:]what happens to the transaction when the the timeout expires?
2. TransactionType: the type of the transaction which is either TWO_PHASE or LOCAL. The TWO_PHASE is more reliable in case of failover because first all partitions that have been touched by the transaction are asked (in parallel) to prepare; so make sure that the data is still locked. Once that succeeds, the partitions are asked to write their changes and release the lock. With LOCAL transaction, all the [todo:]
3. durability: the number of backups for the transaction log, defaults to 1. See [partial commit failure for more info]

The following fragment makes use of the TransactionOptions to configure a TransactionContext which is LOCAL, has a timeout of 1 minutes and durability 1?

```
TransactionOptions txOptions = new TransactionOptions()
    .setTimeout(1, TimeUnit.MINUTES)
```

```
    .setTransactionType(TransactionOptions.TransactionType.TWO\
        _PHASE)
    .setDurability(1);
TransactionContext txCxt = hz.newTransactionContext(txOptions);
```

Not thread-safe the TransactionOptions object isn't thread-safe, so if you share it between threads, make sure it isn't modified after it is configured.

10.3 TransactionalTask

In the previous example we manually manage the transaction; we manually begin one and manually commit it when the operation is finished, and manually rollback the transaction on failure. This can cause a lot of code noise due to the repetitive boilerplate code. Luckily this can be simplified by making use of the TransactionalTask and the HazelcastInstance.executeTransaction(TransactionalTask) method. This method will automatically begin a transaction when the task starts and automatically commits it on success or does a rollback when a Throwable is thrown.

The previous example could be rewritten using the TransactionalTask like this:

```
public class TransactionalTaskMember {
    public static void main(String[] args) throws Throwable {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        hz.executeTransaction(new TransactionalTask() {
            @Override
            public Object execute(TransactionalTaskContext context)
                    throws TransactionException {
                TransactionalMap<String,String> map = context.getMap("
                    map");
                map.put("1", "1");
                map.put("2", "2");
                return null;
            }
        });
        System.out.println("Finished");
        System.exit(0);
    }
}
```

As you can see an anonymous inner class is used to create an instance of the TransactionalTask and to pass it to the HazelcastInstance.executeTransaction method to be executed using a transaction. It is important to understand that the execution of the task is done on the calling thread, so there is no hidden multi-threading going on.

If a RuntimeException/Error is thrown while executing the TransactionalTask,

the transaction is rolled back and the RuntimeException/Error will be rethrown. A checked exception is not allowed to be thrown by the TransactionalTask, so it needs to be caught and dealt with by either doing a rollback or by a commit. Often a checked exception is a valid business flow and the transaction still needs to be committed.

Just as with the raw TransactionContext, the TransactionalTask can be used in combination with the TransactionOptions by calling the HazelcastInstance.executeTransaction(Trans method like this:

```
TransactionOptions txOptions = new TransactionOptions();
...
hz.executeTransaction(txOptions, new TransactionalTask() {
      @Override
      public Object execute(TransactionalTaskContext context)
            throws TransactionException {
        ...
      }
  });
```

[language=java]

10.4 Partial commit failure

[todo] When a transaction is rolled back and non transactional data structures are modified these changes will not be rolled back.

When a transactional operation is executed on a member, this member will keep track of the changes by putting them in a transaction-log. If a transaction hits multiple members, each member will store a subset of the transaction-log. When the transaction is preparing for commit, this transaction change log will be replicated 'durability' times.

10.5 Transaction Isolation

When concurrent interacting threads, are modifying distributed data-structures in Hazelcast, you could run into race problems. These race problems are not caused by Hazelcast, but just by sloppy application code. These problems are notorious hard to solve since they are hard to reproduce and therefore hard to debug. In most cases the default strategy to deal with race problems is to apply manual locking.

Luckily the The Hazelcast transaction api, where isolation from other concurrent executing transaction will be coordinated for you. By default Hazelcast provides a REPEATABLE_READ isolation level, so dirty reads and non repeatable reads are not possible, but phantom reads are. This isolation level is sufficient for most use cases since it is a nice balance between scalability and correctness.

The REPEATABLE_READ isolation level is implemented by automatically locking all reads/writes during the execution of a transaction and these locks will be kept till either the transaction commits or aborts. Because the reads and writes are locked, other transactions can't modify the data that has been accessed, so next time you read it, you will read the same data.

No Dirty reads

One of the read anomalies that can happen in transactional systems is a dirty read. Imagine that there is a map with keys of type String and values of type Integer. If transaction-1 would modify key 'foo' and increment the value from 0 to 1, and transaction-2 would read key 'foo' and see value 1, but the transaction-1 would abort, then transaction-2 would have seen the value 1; a value that was never committed. This is called a dirty read. Luckily in Hazelcast dirty reads are not possible.

Read Your Writes

The Hazelcast transaction supports Read Your Writes (RYW) consistency, meaning that when a transaction makes an update and later reads that data, it will see its own updates. Of course other transactions are not able to see these uncommitted changes, else they would suffer from dirty reads.

No Serialized isolation level

Higher isolation level like SERIALIZED is not desirable due lack of scalability. With the SERIALIZED isolation level, the phantom read isn't allowed. Imagine an empty map and transaction 1 reads the size of this map it time t1 and will see 0. Then transaction 2 starts, inserts a map entry and commits. If transaction 1 would read the size and see 1, it is suffering from a phantom read.

Often the only way to deal with a phantom read is to lock the whole data-structure to prevent other transactions inserting/removing map entries, but this is undesirable because it would cause a cluster wide blockage of this data-structure. That is why Hazelcast doesn't provide protection against phantom reads and therefor the isolation level is limited to REPEATABLE_READ.

Non transactional data-structures

It is important to understand that if during the execution of a transaction a non transactional data-structure, e.g. a non transactional IMap instance, is accessed, the access is done oblivious to a running transaction. So it could be that if you read the same non transactional data-structure multiple times, you could observe changes. Therefore you really need to take care when you choose to access non transactional data-structures while executing a transaction because you could burn your fingers.

It isn't possible to access a transactional data-structure without a transaction. If that happens an IllegalStateException is thrown.

10.6 Locking

It is important to understand how the locking within Hazelcast transactions work; if a transaction is used and a key is read or written, the transaction will automatically lock the entry for that key for the remaining duration of the transaction.

Other transactions are not able to use that map entry since they also automatically acquire the lock when they do a read/write. Hazelcast doesn't have fine grained locks like a read write lock; the lock acquired is exclusive.

If within a transaction a lock can't be acquired, the operation will timeout after 30 seconds and throws an OperationTimeoutException. This provides protection against deadlocks. [todo: timeout configurable?] If a lock was acquired successfully, but its lock expires and therefore will be released, the transaction will happily continue executing operations. Only when the transaction is preparing for commit, this released lock will be detected and a TransactionException will be thrown.

If a transactional data-structure is accessed without a transaction, and IllegalStateException is thrown, so no locking will happen.

When an change is made in a transaction, it is deferred till the commit. When the transaction is committed, the changes are applied and all locks are released. When a transaction is aborted, the locks are released and the changes are discarded.

If you are automatically retrying a transaction that throws an OperationTimeoutException, without controlling the number of retries, it is possible that the system will run into a livelock. These are even harder to deal with than deadlocks because the system is appearing to do something since the threads are busy. But there is no or not much progress due to transaction rollbacks. Therefore it is best to limit the number of retries and perhaps throw some kind of Exception to indicate failure.

The lock timeout can be set using the TransactionOptions.timeoutMilliseconds [todo: reference].

10.7 Cashing and session

Hazelcast transactions provide support for caching reads/writes. If you have been using Hibernate then you probably know the Hibernate-Session, or if you have been using JPA then you probably know the EntityManager. One of the functions of the Hibernate-Session/EntityManager is that once a record is read from the database, if a subsequent read for the same record is executed, it can be retrieved from the session instead of going to the database. Hazelcast support similar functionality.

One big difference between the EntityManager and the Hazelcast transaction, is that the EntityManager will track your dirty objects and will update/insert the objects when the transaction commits. So normally you load one or more entities, modify them, commit the transaction and let the EntityManager deal with writing the changes back to the database. The Hazelcast transaction API doesn't work like this, so the following idiom is broken:

```
TransactionContext txCxt = hz.newTransactionContext();
TransactionalMap<String,Employee> employees = context.getMap("
    employees");
Employee employee = employee.get("123");
employee.fired = true;
txCxt.commitTransaction();
```

10.8 Performance

Although transactions may be easy to use, their usage can influence the application performance drastically due to locking and dealing with partial failed commits. Try to make keep transactions as short as possible, so that locks are held for the least amount of time and the least amount of data is locked. Also try to colocate data in the same partition if possible [todo reference to partitioning strategy]

10.9 Good to know

No readonly support Hazelcast transactions can't be configured as readonly. Perhaps this will be added in the future.

No support for transaction propagation. It isn't possible to created nested transaction. If you do, an IllegalStateException will be thrown.

Hazelcast client. Transactions can also be used from the Hazelcast client.

ITopic: there is no transactional ITopic. Perhaps this will be implemented in the future.

No thread locals: the Transaction API doesn't rely on thread local storage. So if you need to offload some work to a different thread, pass the TransactionContext to the other thread. The transactional data-structures can be passed as well, but the other thread could also retrieve them again since a transactional data-structure can be retrieved multiple times from the TransactionContext instance.

10.10 What is next

In this chapter we saw how to use transactions. You can also use Hazelcast transactions in JEE application using the JEE integration; see "J2EE Integration" in the Hazelcast manual for more information.

Chapter 11

Network configuration

Hazelcast can run perfectly within a single JVM and this is excellent for development and to speed up testing. But Hazelcast true strength becomes apparent when a cluster of JVM's running on multiple machines is created. Having a cluster of machines makes Hazelcast resilient to failure; if one machine fails, the data will failover to another machine as if nothing happened. It also makes Hazelcast scalable; just add extra machines to the cluster, to gain additional capacity. Creating clusters can be done by configuring the network settings.

To test the networking settings, we are going to make use of the following minimalistic Hazelcast member, which is going to load the configuration from a Hazelcast xml configuration file, unless specified otherwise:

```
public class Member {

    public static void main(String[] args) {
        Hazelcast.newHazelcastInstance();
    }
}
```

The basic structure of the hazelcast.xml file is this:

```
<hazelcast>
    ...
    <network>
        ...
    </network>
    ...
</hazelcast>
```

But for brevity reasons I'll leave out the enclosing hazelcast tags. You can find the complete sources for this book on the hazelcast website [todo: link to sources].

For a Hazelcast cluster to function correctly, all members must be able to contact each other member. Hazelcast doesn't support connecting 'over' a member that is

able to connect to another member.

11.1 Port

One of the most basic configuration settings is the port Hazelcast is going to use for communication between the members. This can be done by the 'port' property in the network configuration which defaults to 5701.

```
<network>
    <port>5701</port>
</network>
```

If you start the member, you will get output like:

```
INFO: [192.168.1.101]:5701 [dev] Address[192.168.1.101]:5701 is
   STARTED
```

As you can see, the port 5701 is being used. If another member has claimed port 5701, you will see that port 5702 is assigned. This is because Hazelcast by default will try 100 ports to find a free one it can bind to. So if you configured port to be 5701, Hazelcast tries ports between 5701 and 5801 (exclusive) until it finds a free. In some cases you want to control the number of ports tried, e.g. if you have a large number of Hazelcast instances running on a single machine or you only want a few ports to be used. This can be done by specifying the port-count attribute, which defaults to 100. In the following example you can see the port-count with value 200:

```
<network>
    <port port-count="200">5701</port>
</network>
```

In most cases you won't need to specify the 'port-count' attribute but it can be very practical for those rare cases you need to.

If you only want to make use of a single explicit port, you can disable automatic port increment using the 'auto-increment' attribute (which defaults to true):

```
<network>
    <port auto-increment="false">5701</port>
</network>
```

The 'port-count' property will be ignored when 'auto-increment' is false.

If you look carefully at the end of the logging, you'll see the following warning:

```
WARNING: [192.168.1.104]:5701 [dev] No join method is enabled!
   Starting standalone.
```

You will get this warning no matter how many members you start. The cause is that if you use the XML configuration, by default, no join mechanism is selected and therefor the members can't join to form a cluster. To specify a join mechanism, see [Join Mechanism]

11.2 Join Mechanism

Hazelcast supports 3 mechanisms for members to join the cluster:

1. tcp/ip-cluster
2. multicast
3. Amazon EC2

One of these mechanism needs to be enabled to form a cluster, else they will remain standalone. If you make use of programmatic Hazelcast configuration, by default multicast is enabled. If you make use of XML, none is enabled so you need to disable one. After joining the cluster, Hazelcast relies on TCP for internal communication.

11.3 Multicast

With multicast discovery a member will send a message to all members that listen on a specific multicast group. It is the easiest mechanism to use, but is not always available. Underneath you can see a very minimalistic multicast configuration:

```
<network>
   <join>
       <multicast enabled="true"/>
   </join>
</network>
```

If you start one member, you will see output like this:

```
Jan 22, 2013 2:06:30 PM com.hazelcast.impl.MulticastJoiner
INFO: [192.168.1.104]:5701 [dev]

Members [1] {
   Member [192.168.1.104]:5701 this
}
```

As you can see, the member is started and currently the cluster has a single member. If you start another member on the same machine, on the console of the first member the following will be added to the output:

```
Members [2] {
```

```
    Member [192.168.1.104]:5701 this
    Member [192.168.1.104]:5702
}
```

As you can see, the first member can see the second member. And if we look at the end of logging for the second member, we'll find something similar:

```
Members [2] {
    Member [192.168.1.104]:5701
    Member [192.168.1.104]:5702 this
}
```

We now have a 2 member Hazelcast cluster running on a single machine. It starts to become more interesting if you start multiple members on different machines.

The multicast configuration can be tuned using the following elements:

1. multicast-group: With multicast a member is part of the multicast group and will not receive multicast messages from other group. So by setting the multicast-group or the multicast-port, you can have separate Hazelcast clusters within the same network and it is a best practice to use separate groups if the same network is used for different purposes. The multicast group ip address doesn't conflict with normal unicast ip addresses since they have a specific range that is excluded from normal unicast usage: 224.0.0.0 to 239.255.255.255 (inclusive) and defaults:224.2.2.3. The address 224.0.0.0 is reserved and should not be used.
2. multicast-port: The port of the multicast socket where the Hazelcast member listens and sends discovery messages to. Unlike normal unicast sockets where only a single process can listen to a port, with multicast sockets multiple processes can listen to the same port. So you don't need to be worried about having multiple Hazelcast members that run on the same JVM are going to conflict. This property defaults to 54327
3. multicast-time-to-live: Set the default time-to-live for multicast packets sent out on this in order to control the scope of the multicasts. Defaults to 32 and a maximum of 255.
4. multicast-timeout-seconds: Specifies the time in seconds that a node should wait for a valid multicast response from another node running in the network before declaring itself as master node and creating its own cluster. This applies only to the startup of nodes where no master has been assigned yet. If you specify a high value, e.g. 60 seconds, it means until a master is selected, each node is going to wait 60 seconds before continuing, so be careful with providing a high value. If the value is set too low, it might be that nodes are

giving up too early and will create their own cluster. This property defaults to 2 seconds.

Below you can see a full example of the configuration:

```
<network>
   <join>
      <multicast enabled="true">
        <multicast-group>224.2.2.3</multicast-group>
        <multicast-port>54327</multicast-port>
        <multicast-time-to-live>32</multicast-time-to-live>
        <multicast-timeout-seconds>2</multicast-timeout-seconds>
      </multicast>
   </join>
</network>
```

11.3.1 Trusted interfaces

By default multicast join requests of all machines will be accepted, but in some cases you want to have more control. With the trusted-interfaces you can control the machines you want to listen to by registering their ip address as trusted member. If a join request of a machine is received that is not a trusted member, it will be ignored and it will be prevented to join cluster. Below you see an example where join only requests of '192.168.1.104' are allowed.

```
<network>
   <join>
      <multicast enabled="true">
         <trusted-interfaces>
            <interface>192.168.1.104</interface>
         </trusted-interfaces>
      </multicast>
   </join>
</network>
```

Hazelcast supports a wildcard on the last octet of the ip address e.g. '192.168.1.*' and also supports an ip range on the last octet, e.g. '192.168.1.100-110'. If you do not specify any trusted-interfaces, so the set of trusted interfaces is empty, no filtering will be applied.

If you have configured trusted interfaces but one or more nodes are not joining a cluster, it could be that your trusted interfaced configuration is too strict. Hazelcast will log on the finest level if a message is filtered out so you can see what is happening.

If you are making use of the programmatic configuration, the trusted interfaces are called trusted members.

11.3.2 Debugging multicast

If you don't see members joining, then it is likely because multicast is not available. A cause can be the firewall; you can test this by disabling the firewall or enable multicast in the firewall [see firewall section]. Another cause can be that it is disabled on the network or the network doesn't support it. On *NIX environments you can check if your network interface supports multicast by calling 'ifconfig — grep -i multicast', but it doesn't mean that it is available. To check if multicast is available, 'iperf' is a useful tool which is available for Windows/*NIX/OSX. To test multicast using multicast-group '224.2.2.3', open a terminal one 2 machines within the network and run the following in the first terminal 'iperf -s -u -B 224.2.2.3 -i 1' and 'iperf -c 224.2.2.3 -u -T 32 -t 3 -i 1' in the other terminal. If data is being transferred then multicast is working.

If you want to use multicast for local development and it isn't working, you can try the following unicast configuration:

```
<network>
   <join>
      <multicast enabled="false"/>
      <tcp-ip enabled="true"/>
   </join>
</network>
```

11.4 TCP/IP cluster

In the previous section we made use of multicast, but this is not always an option because in production environments it often is prohibited and in cloud environments it often is not supported. That is why there is another discovery mechanism: the TCP/IP cluster. The idea is that there should be a one or more well known members to connect to. Once members have connected to these well known members and joined the cluster, they will keep each other up to date with all member addresses.

Underneath is an example of a TCP/IP cluster configuration where such a well known member with IP '192.168.1.104':

```
<network>
   <join>
      <tcp-ip enabled="true">
         <member>192.168.1.104</member>
      </tcp-ip>
   </join>
```

```
</network>
```

Multiple members can be configured using a comma separated list, or multiple '¡member¿' entries. A range of IP's can be defined using syntax '192.168.1.100-200'. If no port is provided, Hazelcast will automatically try port 5701..5703. If depending on IP addresses is not desirable, it also is possible to provide the hostname. Instead of using multiple '¡member¿' you can also use '¡members¿'.

```
<network>
   <join>
      <tcp-ip enabled="true">
         <members>192.168.1.104,192.168.1.105</members>
      </tcp-ip>
   </join>
</network>
```

This is very useful in combinaiton with XML Variables [reference to XML variables]

By default Hazelcast will 'bind' (so accept incoming traffic) to all local network interfaces. If this is unwanted behavior the 'hazelcast.socket.bind.any' [todo:hazelcast.server.socket.bin can be set to false. In that case Hazelcast will first use the interfaces configured in the 'interfaces/interfaces' to resolve one interface to bind to. If none is found, it will use the interfaces in the 'tcp-ip/members' to resolve one interface to bind to. If no interface is found, it will default to localhost.

When there is a large number of IP's listed [todo: especially in virtualized environments] and members can't build up a cluster, the 'connection-timeout-seconds' attribute, which defaults to 5, can be set to a higher value. The first scan and delay between scans can be configured using property 'hazelcast.merge.first.run.delay.seconds' and respectively 'hazelcast.merge.next.run.delay.seconds'. By default Hazelcast will scan every 5 seconds.

11.4.1 Required member

If a member needs to be available before a cluster is started, there is an option to set the required member:

```
<tcp-ip enabled="true">
   <required-member>192.168.1.104</required-member>
   <member>192.168.1.104</member>
   <member>192.168.1.105</member>
</tcp-ip>
```

In this example a cluster will only start when member '192.168.1.104' is found. Once this member is found, it will become the master. That means 'required-member' is the address of expected master node.

11.5 EC2 Auto Discovery

Apart from multicast and the tcp/ip-cluster join mechanisms, there a third mechanism: Amazon AWS. This mechanism, which makes use of tcp/ip discovery behind the scenes, reads out EC2 instances within a particular region and have certain tag-keys/values or securitygroup. These instances will be the well known members of the cluster. So a single region is used to lets new nodes discover the cluster, but the cluster can span multiple regions (it can even span multiple cloud providers).

So lets start with a very simple setup:

```
<network>
   <join>
      <aws enabled="true">
         <access-key>my-access-key</access-key>
         <secret-key>my-secret-key</secret-key>
      </aws>
   </join>
</network>
```

And 'my-access-key' and 'my-secret-key' need to be replaced with your access key and secret key. Make sure that the started machines have a security group where the correct ports are opened (see firewall). And also make sure that the 'enabled="true"' section is added because if you don't add it, the aws configuration will not be picked up (it is disabled by default). To prevent hardcoding the access-key and secret-key, you could have a look at variables [reference to chapter "Learning the basics/Variables"]

The AWS section also has a few configuration options:

1. region: the region where the machines are running. Defaults to 'us-east-1'. If you run in a different region, you need to specify it, else the members will not discover each other.
2. tag-key,tag-value: allows you to limit the numbers of EC2 instances to look at by providing them with a unique tag-key/tag-value. This makes it possible to create multiple cluster in a single data center.
3. security-group-name: just like the tag-key,tag-value, it filters out EC2 instances. Doesn't need to be specified.

The aws tag accepts an attribute called "conn-timeout-seconds". The default value is 5 seconds and can be increased if you have many IP's listed and members can not properly build up the cluster.

In case you are using a different cloud provider than Amazon EC2, you can still make use of Hazelcast. What you can do it to use the programmatic api to configure a tcp-ip cluster and the well known members need to be retrieved from your cloud provider (e.g. using jclouds).

If you have problems connecting and you are not sure if the EC2 instances are being found correctly, then you could have a look at the AWSClient class. This client is used by Hazelcast to determine all the private ip addresses of EC2-instances you want to connect to. So if you feed it the configuration settings you are using, you can see if the EC2-instances are being found:

```
public static void main(String[] args)throws Exception{
    AwsConfig config = new AwsConfig();
    config.setSecretKey(..) ;
    config.setSecretKey(...);
    config.setRegion(...);
    config.setSecurityGroupName(...);
    config.setTagKey(..);
    config.setTagValue(...);
    AWSClient client = new AWSClient(config);
    List<String> ipAddresses = client.getPrivateIpAddresses();
    System.out.println("addresses found:"+ipAddresses);
    for(String ip: ipAddresses){
       System.out.println(ip);
    }
}
```

Another things you can do when your cluster is not being created correctly is to change the log level to finest/debug. Hazelcast will log which instances in a region it is encountering and will also tell if an instance is accepted or if it an rejected and the reason for rejection:

1. bad status: so when it isn't running.
2. non matching group-name
3. non matching tag-key/tag-value

A full step by step tutorial on how to start a demo application on Amazon EC2 can be found in the appendix.

11.6 Partition Group Configuration

Normally Hazelcast prevents the master and the backup partitions to be stored on the same JVM to guarantee high availability. But it can be that multiple Hazelcast members of the same cluster are running on the same machine; so when the machine fails it can still happen that both master and backup fail. Luckily Hazelcast provides a solution for this problem in the form of partition groups:

```
<hazelcast>
   <partition-group enabled="true" group-type="HOST_AWARE"/>
</hazelcast>
```

Using this configuration all members that share the same hostname/host-IP, will be part of the same group and therefor will not host both master and backup(s).

Another reason partition groups can be useful is that normally Hazelcast considers all machines to be equal and therefor will distribute the partitions evenly. But in some cases machines are not equal, e.g. different amount of memory available or slower cpu's, so could lead to a load imbalance. With a partition group you can make member-groups where each member-group is considered to have the same capacity and where each member is considered to have the same capacity as the other members in the same member-group. In the future perhaps a balance-factor will be added to relax these constraints. Underneath is an example where we define multiple member groups based on matching ip-addresses:

```
<hazelcast>
   <partition-group enabled="true" group-type="CUSTOM">
      <member-group>
         <interface>10.10.1.*</interface>
      </member-group>
      <member-group>
         <interface>10.10.2.*</interface>
      </member-group
   </partition-group>
</hazelcast>
```

In this example there are 2 member groups, where the first member-group contains all member with an IP 10.10.1.0-255 and the second member-group contains all member with an IP of 10.10.2.0-255. You can also this approach to create different groups for each data-center so that when the primary datacenter goes offline, the backup datacenter can take over.

11.7 Cluster Groups

Sometimes it desirable to have multiple isolated clusters on the same network instead of a single cluster. For example when a network is used for different environments or different applications. Luckily this can be done using groups:

```
<hazelcast>
   <group>
      <name>application1</name>
      <password>somepassword</password>
   </group>
</hazelcast>
```

The password is optional and defaults to 'dev-pass'. A group is something else than a partition-group; with the former you create isolated clusters and with the latter you control how partitions are being mapped to members. If you don't want to have a hard coded password, you could have a look at [Learning the Basics: Variables]

11.8 SSL

In a production environment often you want to prevent that the communication between Hazelcast members can be tampered or can be read by an intruder because it could contains sensitive information. Luckily Hazelcast provides a solution for that: SSL encryption.

The basic functionality is provided by `SSLContextFactory` interface and configurable through the the SSL section in network configuration. Luckily Hazelcast provides a default implementation called the `BasicSSLContextFactory` which we are going to use for the example:

```
<network>
   <join>
       <multicast enabled="true"/>
   </join>
   <ssl enabled="true">
      <factory-class-name>
         com.hazelcast.nio.ssl.BasicSSLContextFactory
      </factory-class-name>
      <properties>
         <property name="keyStore">keyStore.jks</property>
         <property name="keyStorePassword">password</property>
      </properties>
   </ssl>
</network>
```

The 'keyStore' is the path to the keyStore. [todo: is it possible to provide a classpath reference so it can be included in the jar, and is this a desirable practice?] and the 'keyStorePassword' is the password of the keystore. In the example code you can find an already created keystore and also the documentation to create one yourself.

When you start a member, you will see that SSL is enabled:

```
INFO: [192.168.1.104]:5701 [dev] SSL is enabled
```

There are some additional properties that can be set on the `BasicSSLContextFactory`:

1. keyManagerAlgorithm: defaults to 'SunX509'.
2. trustManagerAlgorithm: defaults to 'SunX509'.
3. protocol: defaults to 'TLS'

Another way to configure the keyStore and keyStorePassword is through the `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword` system properties. The recommended practice is to make a single keyStore file that is shared between all instances. It isn't possible to include the keystore within the jar.

11.9 Encryption

Apart from supporting SSL, Hazelcast also supports symmetric encryption based on the Java Cryptography Architecture (JCA). The main advantage of using the latter is that it is easier to set up because you don't need to deal with the keystore. The main disadvantage is that its less secure, because SSL relies on an on the fly created public/private key pair and the symmetric encryption relies on a constant password/salt.

SSL and symmetric encryption solutions will roughly have the same CPU and network bandwidth overhead since for the main data they rely on symmetric encryption (only the public key is encrypted using asymmetric encryption). Compared to non encrypted data, the performance degradation will be roughly 50 %.

The demonstrate the encryption, lets have a look at the following configuration:

```
<network>
   <join>
       <multicast enabled="true"/>
   </join>
   <symmetric-encryption enabled="true">
      <algorithm>PBEWithMD5AndDES</algorithm>
      <salt>somesalt</salt>
      <password>somepass</password>
```

```
      <iteration-count>19</iteration-count>
   </symmetric-encryption>
</network>
```

When we start 2 members using this configuration, we'll see that the symmetric encryption is activated:

```
Jan 20, 2013 9:22:08 AM com.hazelcast.nio.SocketPacketWriter
INFO: [192.168.1.104]:5702 [dev] Writer started with
   SymmetricEncryption
Jan 20, 2013 9:22:08 AM com.hazelcast.nio.SocketPacketReader
INFO: [192.168.1.104]:5702 [dev] Reader started with
   SymmetricEncryption
```

Since encryption relies on the JCA, additional algorithms can be used by enabling the Bouncy Castle JCA provider through property `hazelcast.security.bouncy.enabled`. Hazelcast used to have support for asymmetric encryption, but due its complex setup, this feature has been removed from Hazelcast 3.0. Currently there is no support for encryption between a native client and a cluster member.

11.10 Specifying network interfaces

Most server machines can have multiple network interfaces.

You can also specify which network interfaces that Hazelcast should use. Servers mostly have more than one network interface so you may want to list the valid IPs. Range characters ('*' and '-') can be used for simplicity. So 10.3.10.*, for instance, refers to IPs between 10.3.10.0 and 10.3.10.255. Interface 10.3.10.4-18 refers to IPs between 10.3.10.4 and 10.3.10.18 (4 and 18 included). If network interface configuration is enabled (disabled by default) and if Hazelcast cannot find an matching interface, then it will print a message on console and won't start on that member.

```
<hazelcast>
    <network>
        <interfaces enabled="true">
            <interface>10.3.16.*</interface>
            <interface>10.3.10.4-18</interface>
            <interface>192.168.1.3</interface>
        </interfaces>
    </network>
</hazelcast>
```

11.11 Firewall

When a Hazelcast member connects to another Hazelcast member, it binds to server port 5701 (see the port configuration section) to receive the inbound traffic. On the client side also a port needs to be opened for the outbound traffic. By default this will be an 'ephemeral' port since we it doesn't matter which port is being used as long as it is free. The problem is that the lack of control on the outbound port, can be a security issues, because the firewall needs to expose all ports for outbound traffic.

Luckily Hazelcast is able to control the outbound ports. For example if we want to allow the port range 30000-31000, it can be configured like this:

```
<network>
   <join>
      <multicast enabled="true"/>
   </join>
   <outbound-ports>
      <ports>30000-31000</ports>
   </outbound-ports>
</network>
```

To demonstrate the outbound ports configuration, start 2 Hazelcast members with this configuration and when the members are fully started, execute 'sudo lsof -i — grep java'. Below you can see the cleaned output of the command:

```
java 46117 IPv4 TCP *:5701 (LISTEN)
java 46117 IPv4 TCP 172.16.78.1:5701->172.16.78.1:30609 (
   ESTABLISHED)
java 46120 IPv4 TCP *:5702 (LISTEN)
java 46120 IPv4 TCP 172.16.78.1:30609->172.16.78.1:5701 (
   ESTABLISHED)
```

As you can see there are 2 java processes: 46117 and 46120 that listen on port 5701 and 5702 (inbound traffic). You can also see that java process 46120 is using port 30609 for outbound traffic.

Apart from specifying port ranges, you can also specify individual ports. Multiple port configurations can be combined either by separating them by comma or by providing multiple '¡ports¿' sections. So if you want to use port 30000,30005 and portrange 31000 till 32000, you could say the following '¡ports¿30000,30005,31000-32000¡/ports¿'.

11.11.1 iptables

If you are making use of iptables, the following rule can be added to allow for outbound traffic from ports 33000-31000:

```
iptables -A OUTPUT -p TCP --dport 33000:31000 -m state --state NEW
    -j ACCEPT
```

and to control incoming traffic from any address to port 5701:

```
iptables -A INPUT -p tcp -d 0/0 -s 0/0 --dport 5701 -j ACCEPT
```

and to allow incoming multicast traffic:

```
iptables -A INPUT -m pkttype --pkt-type multicast -j ACCEPT
```

11.12 Connectivity test

If you are having troubles because machines won't join a cluster, you might check the network connectity between the 2 machines. You can use a tool called iperf for that. On one machine you execute:

```
iperf -s -p 5701
```

This means that you are listening at port 5701.

At the other machine you execute the following command:

```
iperf -c 192.168.1.107 -d -p 5701
```

Where you replace '192.168.1.107' by the ip address of your first machine. If you run the command and you get output like this:

```
------------------------------------------------------------
Server listening on TCP port 5701
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 192.168.1.107, TCP port 5701
TCP window size: 59.4 KByte (default)
------------------------------------------------------------
[  5] local 192.168.1.105 port 40524 connected with 192.168.1.107
   port 5701
[  4] local 192.168.1.105 port 5701 connected with 192.168.1.107
   port 33641
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.2 sec  55.8 MBytes  45.7 Mbits/sec
[  5]  0.0-10.3 sec  6.25 MBytes  5.07 Mbits/sec
```

You know the 2 machines can connect to each other. However if you are seeing something like this:

```
Server listening on TCP port 5701
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
connect failed: No route to host
```

Then you know that you might have a network connection problem on your hands.

11.13 Good to know

11.14 What is next

The network configuration for Hazelcast is very extensive. There are some features like IPv6, network partitioning (split-brain syndrome), specifying network interfaces, socket interceptors, WAN replication, that have been left out, but can be found on the Hazelcast reference manual. Also the mailing-list is a very valuable source of information.