Mastering Hazelcast

The Essential Companion to the Reference Manual

Current to Hazelcast 3.2

PETER VEENTJER

Foreword by Greg Luck

Table of Contents

Acknowledgments

Special thanks go to the Hazelcast guys: Talip Ozturk, Fuad Malikov and Enes Akar who are technically responsible for Hazelcast and helped to answer my questions. But I really want to thank Mehmet Dogan, architect at Hazelcast, since he was my main source of information and put up with the zillion questions I have asked.

Also thanks to all committers and mailing list members for contributing to making Hazelcast such a great product.

Finally, I’m very grateful for my girlfriend, Ralitsa Spasova, for being a positive influence around me and making me a better person.

Foreword

Peter Veentjer leads the QuSP ("Quality, Stability and Performance") team at Hazelcast. In that role, he roves over the whole code base with an eagle eye and has built up deep expertise on Hazelcast. Peter is also a great communicator, wishing to spread his knowledge of and enthusiasm for Hazelcast to our user base. So it was natural for Peter to create Mastering Hazelcast.

In Mastering Hazelcast, Peter takes an in-depth look at fundamental Hazelcast topics. This book should be seen as a companion to the Reference Manual. The reference manual covers all Hazelcast features. Mastering Hazelcast gives deeper coverage over the most important topics. Each chapter has a **Good to Know** section, which highlights important concerns.

This book includes many code examples. These and more can be accessed from https://github.com/hazelcast/hazelcast-code-samples. A great way to learn Hazelcast is to download the examples and work with them as you read the chapters.

Like much of Hazelcast, this book is open source. Feel free to submit pull requests to add to and improve it. It is a living document that gets updated as we update Hazelcast.

Greg Luck
CEO Hazelcast

Preface

Writing concurrent systems has long been a passion of mine, so it is a logical step to go from concurrency control within a single JVM to concurrency control over multiple JVMs. A lot of the knowledge that is applicable to concurrency control in a single JVM also applies to concurrency over multiple JVMs. However, there is a whole new dimension of problems that make distributed systems even more interesting to deal with.

What is Hazelcast?

When you professionally write applications for the JVM, you will likely write server-side applications. Although Java has support for writing desktop applications, the server-side is where Java really shines.

Today, in the era of cloud computing, it is important that server-side systems are:

1. Scalable: just add and remove machines to match the required capacity.
2. Highly available: if one or more machines has failed, the system should continue as if nothing happened.
3. Highly performant: performance should be fast, and cost effective.

Hazelcast is an In-Memory Data Grid. It is:

1. highly available. It does not lose data after a JVM crash because it automatically replicates partition data to other cluster members. In the case of a member going down, the system will automatically failover by restoring the backup. Hazelcast has no master member that can form a single point of failure; each member has equal responsibilities.
2. lightning-fast. Each Hazelcast member can do thousands of operations per second.

Hazelcast on its own is elastic, but not automatically elastic; it will not automatically spawn additional JVMs to become members in the cluster when the load exceeds a certain upper threshold. Also, Hazelcast will not shutdown JVMs when the load drops below a specific threshold. You can achieve this by adding a glue code between Hazelcast and your cloud environment.

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

This freedom, combined with very well thought out APIs, makes Hazelcast a joy to use. In many cases, you simply use interfaces from java.util.concurrent, such as Executor, BlockingQueue or Map. In little time and with simple and elegant code, you can write a highly available, scalable and high-performing system.

Who should read this book

This book aims at developers and architects who 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 one of the JVM-based languages like Scala, Groovy or Clojure. Hazelcast also provides an almost identical API for .NET and C++.

If you are a developer that has no prior experience with Hazelcast, then this book will help you learn the basics to get up and running quickly. If you already have some experience, it will round out your knowledge. If you are a heavy Hazelcast user, it will give you insights into advanced techniques and things to consider.

What is in this book

This book shows you how to make use of Hazelcast by going through Hazelcast's most important features. Its focus is now on Hazelcast 3.2 but it is a living document and will be revised as new releases come out. Some of the improvements introduced with 3.2 are minor changes, although those changes can have a huge impact on your system. Other improvements are very big, such as the SPI (came out with 3.0) which lets you write your own distributed data structures if you are not happy with the ones provided by Hazelcast.

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

In **Chapter 2: Learning the Basics**, you will learn the basic steps to start Hazelcast instances, load and configure DistributedObjects, configure logging, and the other fundamentals of Hazelcast.

In **Chapter 3: 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 4: Distributed Collections**, you will learn how to make use of distributed collections like the IQueue, IList and ISet.

In **Chapter 5: Distributed Map**, you will learn about the IMap functionality. Since IMap functionality is very extensive, there is a whole topic about dealing with its configuration options, such as high availability, scalability, etc. You will also learn how to use Hazelcast as a cache and persist its values.

In **Chapter 6: 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 7: Distributed Topic**, you will learn about creating a publish/subscribe solution using the Distributed Topic functionality.

In **Chapter 8: Hazelcast Clients**, you will learn about setting up Hazelcast clients.

In **Chapter 9: Serialization**, you will learn more about the different serialization technologies that are supported by Hazelcast. Java Serializable and Externalizable, and also the native Hazelcast serialization techniques like DataSerializable and the new Portable functionality will be explained.

In **Chapter 10: Transactions**, you will learn about Hazelcast's transaction support, which prevents transactional data-structures from being left in an inconsistent state.

In **Chapter 11: Network Configuration**, you will learn about Hazelcast's network configuration. Different member discovery mechanisms like multicast, Amazon EC2, and security will be explained.

In **Chapter 12: SPI**, you will learn about using the Hazelcast SPI to make first class distributed services. This functionality is perhaps the most important new feature introduced with Hazelcast 3.0.

In **Chapter 13: Threading Model**, you will learn about using the Hazelcast threading model. This helps you write an efficient system without causing cluster stability issues.

In **Chapter 14: Performance Tips**, you will learn some tips to improve Hazelcast performance.

Online Book Resources

You can find the online version of this book at http://www.hazelcast.org/mastering-hazelcast/.

Code examples that are structured chapter by chapter in a convenient Maven project can be cloned using GitHub at https://github.com/hazelcast/hazelcast-code-samples. I recommend you run the examples as you read the book.

Please feel free to submit any errata as an issue to this repository, or send them directly to masteringhazelcast@hazelcast.com.

Online Hazelcast Resources

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

1. Hazelcast Project Website: http://hazelcast.org
2. Hazelcast Company Website: http://hazelcast.com
3. Hazelcast Documentation: http://hazelcast.org/documentation
4. Hazelcast Usergroup: http://groups.google.com/group/hazelcast
5. Hazelcast Source Code: https://github.com/hazelcast/hazelcast/
6. Hazelcast Code Examples:
   https://github.com/hazelcast/hazelcast-code-samples

Building distributed systems on Hazelcast is really a joy to do. I hope I can make you as enthusiastic

about it as I am. So let’s get started with building distributed applications that you can be proud of.

Chapter 1. Getting Started

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

1.1. Installing Hazelcast

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

For this book, we rely on the community edition of Hazelcast 3.2 which you can download from http://www.hazelcast.org/download/. If your project uses Maven, there is no need to install Hazelcast at all, see http://www.hazelcast.org/download/#maven. Otherwise, you should make sure that the Hazelcast JAR is added to your classpath. Apart from this JAR, there is no need to install Hazelcast. The lack of an installation process for Hazelcast saves quite a lot of time, time that can use 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 going 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</version>
   </dependency>
</dependencies>
```

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

The latest snapshot is even more recent because it is updated as soon as a change is merged in the Git repository. If you want to 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>
```

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

You can access the examples used in the book from this website: https://github.com/hazelcast/hazelcast-code-samples.

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

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

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

```
mvn clean install
```

Each example has one or more bash scripts to run it. Some users prefer to run them from their IDE.

1.4. Building Hazelcast

If you want to build Hazelcast yourself, perhaps because you want to provide a bug fix, 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 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, 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 you 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 of getting started with Hazelcast: how to create Hazelcast instances, how to load and configure distributed objects, how to configure logging, and so on.

2.1. Configuring Hazelcast

Hazelcast can be configured in three different ways.

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

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

2.1.1. Configuring Hazelcast XML

When you are running a Maven project, just create a folder named `resources` under `src/main/` and create a file called `hazelcast.xml`. The following shows an empty `hazelcast.xml` configuration file:

```
<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 configuration file 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 this 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.

2.1.2. Configuring for Multicast

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 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.

2.1.3. Resolving Hazelcast Configuration Files

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 approaches are used to resolve the configuration, in the given order:

1. Hazelcast checks if the `hazelcast.config` system property is set. If it is, then its value is used as the path to the configuration file. This is useful if you want the application to choose the Hazelcast configuration file at the time of 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, or it can be a classpath reference if it is prefixed with `classpath:`.

2. Hazelcast checks if there is a `hazelcast.xml` in the working directory.

3. Hazelcast checks if there is a `hazelcast.xml` on the classpath.

4. If all of the above options fail to provide a Hazelcast config to the application, the default Hazelcast configuration is loaded from the Hazelcast JAR.

One of the changes in Hazelcast 3.2 is that when a `hazelcast.xml` file contains errors, an exception will be thrown and no `HazelcastInstance` is created. Prior to the 3.2 release, no exception was thrown and a Hazelcast instance with the `hazelcast-default.xml` configuration was created. The problem with that approach was that you ended up with a `HazelcastInstance` with a different configuration than you expect.

2.1.4. Loading Hazelcast XML Configuration from Java

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

- `ClasspathXmlConfig`: Load the `config` from a classpath resource containing the XML configuration.

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

- `FileSystemXmlConfig`: Load the `config` from a file.

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

- `InMemoryXmlConfig`: Load the `config` from an in-memory string containing the XML configuration.

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

- `UrlXmlConfig`: Load the `config` from a 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);
```

2.1.5. Loading Configuration Programmatically

Another option to load a `HazelcastInstance` is via programmatic configuration:

```
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);
      ...
   }
}
```

2.1.6. Fluent Interface

The Hazelcast `Config` object has a fluent interface: 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 very useful for testing, and is a solution for the static nature of the XML configuration. You can easily create content for the programmatic configuration on the fly: for example, 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: for example, the network configuration.

2.1.7. No Static Default HazelcastInstance

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

2.1.8. Same Configuration

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. Differences between configurations can lead to problems. Either Hazelcast will detect the differences, or the distributed objects will be created with different configurations based on how the members were configured.

2.2. Multiple Hazelcast Instances

In most cases, you will have a single Hazelcast instance per JVM. However, multiple Hazelcast instances can also run in a single JVM. This is useful for testing, and is also needed for more complex setups: for example, application servers running multiple independent applications using Hazelcast. You can start multiple Hazelcast instances like this:

Creating multiple HazelcastInstances

```
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 in the above output, the created cluster has 2 members.

2.3. Loading a DistributedObject

In the previous sections, we saw how a `HazelcastInstance` can be created. In most cases, you want to load a `DistributedObject`, such as a queue, from this `HazelcastInstance`. So let's 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 objects using the SPI, you can use the `HazelcastInstance.getDistributedObject`. One thing worth mentioning is that most of the distributed objects defined in the configuration are created lazily; 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, see Distributed Collections: IQueue.

2.4. Unique Names for Distributed Objects

Some of the distributed objects will be static; they will be created and used through the application and the IDs of these objects will be known up front. Other distributed objects are created on the fly, and one of the problems is finding unique names when new data structures need to be created. One of the solutions to this problem is to use the `IdGenerator`, which will generate cluster wide unique IDs.

```
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. See Wildcard Configuration.

A distributed object created with a unique name often needs to be shared between members. You can do this by passing the ID to the other members, and you can use one of the `HazelcastInstance.get` methods to retrieve the `DistributedObject`. For more information, see Serialization: DistributedObject.

In Hazelcast, the name and type of the `DistributedObject` uniquely identifies that object:

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

In this example, two different distributed objects are created with the same name but different types. In normal applications, you want to prevent different types of distributed objects from sharing the same name. You can add the type to the name, such as `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 into 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 a singleton bean definition.

2.6. Destroying a DistributedObject

A `DistributedObject` can be destroyed using the `DistributedObject.destroy()` method, which clears and releases all resources for this object within the cluster. But, you should use this method with care

because 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 issue 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 two 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
```

The system will not report any error and will behave as if nothing has happened; the only difference is the creation of the new queue resource. Again, a lot of care needs to be taken when destroying distributed objects.

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. Wildcard configuration makes it possible to use the same configuration for multiple instances. For example, we could configure the previous `testmap` example with a value of 10 for `time-to-live-seconds` using a wildcard configuration like this:

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

By 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; the difference is that in Hazelcast, multiple gets of a data structure with the same ID will still result in the same instance, whereas with prototype beans new instances are returned.

2.7.1. Avoiding Ambiguous Configuration

It is important that you watch out for ambiguous configuration, as in the following example:

```
<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 loaded using `hz.getMap("map")` then Hazelcast will not throw an error or log a warning; instead, Hazelcast selects one of the maps. The selection does not depend on the definition order in the configuration file and is not based on the best-fitting match. You should make sure that your wildcard configurations are very specific. One of the ways to do it is including 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

Hazelcast has two types of distributed objects. 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 it will also use the name 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 two 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, as in the following example.

```
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 be used to control the partition of distributed objects, and can also be used to send a Runnable to the correct member using the `IExecutor.executeOnKeyOwner` method, as in Distributed Executor Service: Executing on Key Owner, and to control in which partition a map entry is stored, as in (see Map: Partition Control).

2.8.1. DistributedObject Names and Partition Key

If a `DistributedObject` name includes a partition key, then Hazelcast will use the base-name 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. Therefore, the following two semaphores are different.

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

2.8.2. Accessing DistributedObject Partition Keys

To access the partition key of a `DistributedObject`, you can call the `DistributedObject.getPartitionKey` method.

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

This method 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.

2.8.3. Obtaining DistributedObject Partition Keys

In the previous examples, the `foo` partition key was used. In many 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 option is to look at the `Random` class. The only thing you need to watch out for is to have the partition keys evenly distributed among the partitions.

[[using-@-in-name]] ==== Using @ in Partitioned DistributedObject Names

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

For more information about why you want to control partitioning, see Performance Tips: Partitioning Schema.

2.9. Properties

Hazelcast provides an option to configure certain properties which are not part of an explicit configuration section, such as 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 Properties section in the Hazelcast Reference Manual or have a look at the `GroupProperties` class.

Apart from properties in the `hazelcast.xml`, you can also pass properties using the command line: `java -Dproperty-name=property-value`. One thing to watch out for: you can't override properties in the `hazelcast.xml` or the programmatic configuration from the command line because the command line 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 increase the pool size since you have beefier machines there; in a development environment, you might want to set it to a low value.

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

You can use a different mechanism than the system properties, such as a property file or a database. You can do this 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);
```

The `Config` subclasses, like `FileSystemXmlConfig`, accept `Properties` in their constructors.

Although variables will give you a lot more flexibility, they have limitations: you can only parametrize but you can't add new XML sections. If your needs go beyond what the variables provide, you might consider using some kind of template engine like Velocity to generate your `hazelcast.xml` file. Another option is using programmatic configuration, either by creating a completely new `Config` instance or loading a `template` from XML and enhancing 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. You can set logging by adding a property in the `hazelcast.xml`:

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

Or you can set with the programmatic configuration:

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

Or you can configure it 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 by using the `LogListener` interface. See the Logging Configuration section in the Hazelcast Reference Manual for more information.

WARNING	If you are not making use of configuring logging from the command line, be very careful about 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. Some users make use of the command line version instead of the properties section for logging to avoid confusion.

If you are making use of `jdk` logging and you are annoyed that your log entry is spread over two lines, have a look at the `SimpleLogFormatter`, as in the following example.

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

2.12. Good to Know

Hazelcast config is 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 you don't have control of the creation of the Hazelcast instances, but you 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 when that was 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` is created, how it can be configured, and how a `DistributedObject` is created. In the following chapters, you will learn about all the different distributed objects 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 the concurrency library that was introduced in Java 5 under `java.util.concurrent`, such as `Executor`, `Lock` and `AtomicReference`.

This concurrency functionality is useful if you want to write a Java application that uses multiple threads, but the focus here is to provide synchronization in a single JVM and not distributed synchronization over multiple JVMs. Luckily, Hazelcast provides support for various distributed synchronization primitives, such as the `ILock`, `IAtomicLong`, etc. Apart from making synchronization between different JVMs possible, these primitives also support high availability: if one machine fails, the primitive remains usable for other JVMs.

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, such as `get`, `set`, `getAndSet`, `compareAndSet` and `incrementAndGet`. There is a big difference in performance since remote calls are involved.

This example demonstrates 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 `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, you have a few ways to deal with it depending on your requirements. You can create a stripe (essentially an array) of `IAtomicLong` instances to reduce pressure. Or you can keep changes local and only publish them to the `IAtomicLong` once a while. There are a few downsides; you could lose 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: it is a 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 adds 2 to 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 methods:

- `apply`: Applies the function to the value in the `IAtomicLong` without changing the actual value and returns the result.
- `alterAndGet`: Alters the value stored in the `IAtomicLong` by applying the function, storing the result in the `IAtomicLong` and returning the result.
- `getAndAlter`: Alters the value stored in the `IAtomicLong` by applying the function and returning the original value.
- `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, 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 race problem; the read and the write of the `IAtomicLong` are not atomic, so they could be 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 therefore 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, making this a lot more scalable.

3.1.2. Good to know

Replication: the IAtomicLong has 1 synchronous backup and zero asynchronous backups and is not configurable.

3.2. IdGenerator

In the previous section, the IAtomicLong was introduced. IAtomicLong can be used to generate unique IDs within a cluster. Although that will work, it probably isn't the most scalable solution since all members will content on incrementing the value. If you are only interested in unique IDs, 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 IDs to generate. This is done behind the scenes by using an IAtomicLong. A segment is claimed by incrementing that IAtomicLong by 10000. After claiming the segment, the IdGenerator can increment a local counter. Once all IDs in the segment are used, it will claim a new segment. The consequence of this approach is that only 1 in 10000 times is network traffic needed; 9999 out of 10000, the ID generation can be done in memory and therefore 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 10000 instead of 1 out of 1.

Let's 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 IDs. If you do see duplicates, it could be that the IdGeneratorMembers didn't form a cluster; see Network Configuration: Multicast.

There are some issues you need to be aware of.

- IDs generated by different members will be out of order.
- 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 IDs 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, for example, 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 cases 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 IDs have been generated and the desired starting value is bigger than 0.

3.2.1. Good to know

> *Replication*: the `IdGenerator` has 1 synchronous backup and zero asynchronous backups and is not configurable.

3.3. IAtomicReference

In the first section, the `IAtomicLong` was introduced. The `IAtomicLong` is very useful if you need to deal with a long, but in some cases you need to deal with a reference. That is why Hazelcast also supports

the `IAtomicReference`, which is the distributed version of the `java.util.concurrent.atomic.AtomicReference`.

Let's see the `IAtomicReference` in action:

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

        IAtomicReference<String> ref = hz.getAtomicReference("reference");
        ref.set("foo");
        System.out.println(ref.get());
        System.exit(0);
    }
}
```

If you execute this code, you will see:

```
foo
```

Just like the `IAtomicLong`, the `IAtomicReference` has methods that accept a function as argument, such as `alter`, `alterAndGet`, `getAndAlter` and `apply`. There are big advantages for using these methods.

3.3.1. Good to know

- From a performance point of view, it is better to send the function to the data then the data to the function. Often the function is a lot smaller than the value and therefore the function is cheaper to send over the line. Also, the function only needs to be transferred once to the target machine, while the value needs to be transferred twice.
- You don't need to deal with concurrency control. If you would do a load, transform, and store, you could run into a data race since another thread might have updated the value you are about to overwrite.
- When a function is executed on the `AtomicReference`, make sure that the function doesn't run too long. As long as that function is running, the whole partition is not able to execute other requests. Don't hog the operation thread.

Some issues you need to be aware of:

- The `IAtomicReference` works based on byte-content, not on object-reference. Therefore, if you are using the `compareAndSet` method, it is important that you do not change to the original value because its serialized content will then be different. It is also important to know that if you rely on Java serialization, sometimes (especially with hashmaps) the same object can result in different binary content.
- The `IAtomicReference` will always have 1 synchronous backup.
- All methods returning an object will return a private copy. You can modify it, but the rest of the world will be shielded from your changes. If you want these changes to be visible to the rest of the world, you need to write the change back to the `IAtomicReference`; but be careful with introducing a data race.
- The in-memory format of an `IAtomicReference` is binary. So the receiving side doesn't need to have the class definition available, unless it needs to be deserialized on the other side (for example, because a method like `alter` is executed). This deserialization is done for every call that needs to have the object instead of the binary content, so be careful with expensive object graphs that need to be deserialized.
- If you have an object graph or an object with many fields, and you only need to calculate some information or you only need a subset of fields, you can use the `apply` method. This way, the whole object doesn't need to be sent over the line, only the information that is relevant.

3.4. ILock

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

Hazelcast provides a distributed lock implementation and makes it possible to create a critical section

within a cluster of JVMs, so only a single thread from one of the JVMs in the cluster is allowed to acquire that lock. Other threads, no matter if they are on the same JVMs or not, will not be able to acquire the lock; 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 = hz.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 `"Data race 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 you use a lock (it doesn't matter if it is a Hazelcast lock or a 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, 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();
}
```

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.

Here are more general issues worth knowing about the Hazelcast lock.

- Hazelcast lock 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.

- As with the other Lock implementations, Hazelcast lock should always be acquired outside of a

`try/finally` block. Otherwise, the lock acquire can fail, but an unlock is still executed.

- Keep locks as short as possible. If locks are kept too long, it can lead to performance problems, or worse, deadlock.
- With locks it is easy to run into deadlocks. Having code you don't control or understand running inside your locks is asking for problems. Make sure you understand exactly the scope of the lock.
- To reduce the chance of a deadlock, you can use the `Lock.tryLock` method to control the waiting period. The `lock.lock()` method will not block indefinitely, but will timeout with an `OperationTimeoutException` after 300 seconds.
- Locks are automatically released when a member has acquired a lock and that member goes down. This prevents threads that are waiting for a lock from waiting indefinitely. This is also needed for failover to work in a distributed system. The downside is that if a member goes down that acquired the lock and started to make changes, other members could start to see partial changes. In these cases, either the system could do some self repair or a transaction might solve the problem.
- A lock must always be released by the same thread that acquired it, otherwise look at the `ISemaphore`.
- Locks are fair, so they will be granted in the order they are requested.
- There are no configuration options available for the lock.
- A lock can be checked if it is locked using the `ILock.isLocked` method, although the value could be stale as soon as it is returned.
- 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.
- The `Hazelcast.getLock` doesn't work on a name of type `String`, but can be a 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.
- Replication: the `ILock` has one synchronous backup and zero asynchronous backups and is not configurable.
- A lock is not automatically garbage collected. So if you create new locks over time, make sure to destroy them. If you don't, you can run into an `OutOfMemoryError`.

3.5. ICondition

With a Condition, it is possible to wait for certain conditions to happen: for example, wait for an item to be placed on a queue. Each lock can have multiple conditions, such as if an item is available in the queue and if room is available in the queue. In Hazelcast 3, the `ICondition`, which extends the `java.util.concurrent.locks.Condition`, has been added.

There is one 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. You don't want to rely on the order of their creation, so 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. Another member will set the value on that counter. Let's 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());
   }
}
```

First, the lock is acquired (`getLock`). Then, the counter is checked within a loop. 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`. Here, the Lock is acquired, the value is 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
```

3.5.1. Good to know

A few things worth knowing about the `ICondition`:

- 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.
- You can choose to signal only a single thread instead of all threads by calling the `ICondition.signal()` method instead of the `ICondition.signalAll()` method.
- 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 is often a good practice to wait with a timeout using the `await(long time, TimeUnit unit)` or `awaitNanos(long nanosTimeout)` method.
- Waiting threads are signaled in FIFO order.
- *Replication*: the `ICondition` has 1 synchronous backup and zero asynchronous 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, such as using a resource. Each semaphore has a number of permits, where each permit represents a single thread allowed to execute that activity concurrently. As soon as a thread wants 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 will look a lot like a lock. A big difference is that the semaphore has no concept of ownership. 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, while a semaphore can have more than 1.

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

- If a permit is available, the number of permits in the semaphore is decreased by one and the calling thread can continue.
- If no permit is available, the calling thread will block until a permit becomes available, a timeout happens, the thread is interrupted, or when the semaphore is destroyed and an `InstanceDestroyedException` is thrown.

The following example explains the semaphore. To simulate a shared resource, we have an `IAtomicLong` initialized with the value 0. This resource is going to be used 1000 times. When a thread starts to use that resource, it will be incremented, and when finished 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 threads. We can do this by configuring the `initial-permits` for the semaphore in the Hazelcast configuration file:

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

When you start the `SemaphoreMember` 5 times, you will see the 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
```

The maximum number of concurrent threads using that resource is always equal to or smaller than 3. As an experiment, you can remove the semaphore acquire/release statements and see for yourself in the output 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`: if a member goes and replication is enabled

(by default it is), then another member takes 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:

- `backup-count`: Number of synchronous replicas and defaults to 1.
- `async-backup-count`: Number of asynchronous replicas and defaults to 0.

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

Good to know

A few things worth knowing about the `ISemaphore`:

- Fairness. The `ISemaphore` 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.
- 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 to the Hazelcast Lock). If the permit would not be released, the system could run in a deadlock.
- The `acquire()` method doesn't timeout, unlike the Hazelcast `Lock.lock()` method. To prevent running into a deadlock, you can use one of timed acquire methods, like `ISemaphore.tryAcquire(int permits, long timeout, TimeUnit unit)`.
- The `initial-permits` is allowed to be negative, indicating 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 that are being performed by one or more threads are completed. A `CountDownLatch` can be seen as a gate containing a counter. Behind this gate, threads can wait till the counter reaches 0. 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 `com.hazelcast.core.ICountDownLatch`.

To explain the `ICountDownLatch`, imagine that there is a leader process that is executing some action that will eventually complete. 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 the `Leader` owner of that latch), does some waiting, and then calls countdown which notifies the listeners for that latch. In this example, we ignore the boolean return value of `trySetCount` since there will be only a single `Leader`, but in practice you probably want to 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);
  }
}
```

We retrieve the `ICountDownLatch` and then call `await` on it so the thread listens for when the `ICountDownLatch` reaches 0. In practice, a process that should have decremented the counter by calling the `ICountDownLatch.countDown` method can fail, and therefore the `ICountDownLatch` will never reach 0. To force you to deal with this situation, the await methods have timeouts to prevent waiting indefinitely.

If we first start a leader and then start 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 shows an `ICountDownLatch` with only a single step. If a process has n steps, you should initialize the `ICountdownLatch` with n instead of 1, and for each completed step, you should 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 reaches zero and this would result in the waiters waiting till the end of time. Because this behavior is undesirable, Hazelcast will automatically notify all listeners if the owner gets disconnected, and therefore listeners could be 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: for example, 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 the one you will use on a daily basis. Unlike Java's implementation, Hazelcast's `ICountDownLatch` count can be reset after a countdown has finished, but it cannot be reset during an active count.

Replication: the `ICountDownLatch` has 1 synchronous backup and zero asynchronous backups and is not configurable.

3.7.1. Good to know

Cluster Singleton Service: In some cases you need a thread that will only run on a limited number of members. Often only a single thread 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 easy 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, the first thing this service needs to do is to acquire the lock or a license and on success, the thread can start 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 exits the cluster (due to a crash, network disconnect, etc.), the lock is automatically released and the license is returned. Then, other cluster members that are waiting to acquire the lock/license can now have their turn.

3.8. What is next?

In this chapter, we looked at various synchronization primitives that are supported by Hazelcast. If you need a different one, you can try to build it on top of existing ones or you can create a custom one using the Hazelcast SPI. One thing that would be nice to add is 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, making it easy to integrate distributed collections into your system without making too many code changes. A distributed collection can be called concurrently from the same JVM, and can be called concurrently by different JVMs. 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 POJOs) to work at different speeds. The Hazelcast `com.hazelcast.core.IQueue`, which extends the `java.util.concurrent.BlockingQueue`, allows threads from the same JVM to interact with that queue. Since the queue is distributed, it also allows different JVMs to interact with it. 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 will 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 wait for 5 seconds. Then, it will consume the next message and stop when it receives the -1. This behavior is 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 consumers will receive the poison pill, not just 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, the queue will keep growing with a single consumer. 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 the elements again:

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

If there are many producers/consumers interacting with the queue, there will be a lot of contention and eventually the queue will become a bottleneck. One way you can solve this is to introduce a stripe (essentially a list) of queues. But if you do, the ordering of messages sent to different queues will no longer be guaranteed. In many cases, a strict ordering isn't required and a stripe can be a simple solution to improve scalability.

IMPORTANT	Although the Hazelcast distributed queue preserves ordering of the messages (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 the 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` implementation, such as the `LinkedBlockingQueue`, you can set a capacity. When this is set and the maximum capacity is reached, placement of new items either fails or blocks, depending on the type of the `put` operation. This prevents the queue from growing beyond a healthy capacity and the JVM from failing. It is important to understand that the `IQueue` is not a partitioned data structure like the `IMap`, so the content of the `IQueue` will not be spread over the members in the cluster. A single member in the cluster will be responsible for keeping the complete content of the `IQueue` in memory. Depending on the configuration, there will also be a backup which

keeps the whole queue in the memory.

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 using the queue property `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 see that the messages are being consumed and the producer will produce again.

4.1.2. Backups

By default, Hazelcast will make sure that there is one synchronous backup for the queue. 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 following properties.

- `backup-count`: Number of synchronous backups, defaults to 1. So by default, no entries will be lost if a member fails.
- `async-backup-count`: 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 can set the `backup-count` to 0, but at the cost of potentially losing entries on failure.

4.1.3. QueueStore

By default, Hazelcast data structures like the `IQueue` are not persistent.

- If the cluster starts, the queues will not be populated by themselves.
- 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. Luckily, Hazelcast provides a mechanism for queue durability using the `QueueStore`, which can connect to a more durable storage mechanism, such as a database. 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; instead, it exposes a `QueueStore` directly.

4.2. IList

A List is a collection where every element only occurs once and where the order of the elements 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 printing 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, you start the `ReadMember`, then the `ReadMember` will output the following:

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

The data that the `WriteMember` writes to the List is visible in the `ReadMember` and the order is maintained. The List interface has various methods (like the sublist) that returns collections, but it is important to understand that the returned collections are snapshots and are not backed up by the list. See Iterator Stability for a discussion of weak consistency.

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`. We'll demonstrate the Set by adding items in a Set on one member, and printing all the elements from that Set on another member:

```
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()` methods of the object are used and not the equals/hash of the byte array version of that object. This is a different behavior compared to the map; see Map: Hashcode and Equals.

In Hazelcast, the `ISet` (and 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 cannot 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, you can implement a Set based on a Map where the value is some bogus

value. It is not 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. 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 potentially contains the item, the member where the change is 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 for this configuration option is that in some cases, you only want to be notified when a change is 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` is really 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();
   }
}
```

If you start up the `ItemListenerMember` and wait till it displays `"ItemListener started"`, and then you start the `CollectionChangeMember`, 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: listeners will receive and process the events in the order they actually occurred.

4.4.1. Good to know

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

Not Durable: In Hazelcast 2, the `IQueue/IList/ISet` were built on top of the Hazelcast Distributed Map. By accessing that Map, you could influence the collections behavior, including storage. This is not possible in Hazelcast 3: the `IQueue` has its own QueueStore mechanism, but the List/Set does not. Perhaps this will be added in a later release.

Replication: The replication for `IList` and `ISet` cannot be configured and will automatically have 1 synchronous backup and 0 asynchronous backups. Perhaps in the future this will 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 the implicit destroyed collection, the collection will automatically be recreated.

No merge policy for the Queue: If a cluster containing a queue is split, then each subcluster will still able to access their own view of that queue. If these subclusters merge, the queue cannot be merged and one of them is deleted.

Not partitioned: The `IList/ISet/IQueue` are not partitioned, so the 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 due 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. You can solve this issue by using a stripe of collections or by building 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; see SPI.

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 neighbors, the queues will end up in different members. If ordering of items is not important, the item can be placed on an arbitrary queue. Otherwise, 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 is currently not possible to control the partition the collection is going to be placed on, so more remoting is required than is 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.

4.5. What is next?

In this chapter we have seen various collections in action and we have seen how they can be configured. In the following chapter, you will learn about Hazelcast Distributed Map.

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 therefore it also extends `java.util.Map`. Unlike a normal Map implementation like the HashMap, the Hazelcast `IMap` implementation is a distributed data structure.

Internally, Hazelcast divides the map into partitions and it distributes the partitions evenly among the members in the cluster. The partition of a map entry is based on the key of that entry: each key belongs to a single partition. By default, Hazelcast uses 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 member in the cluster decides which partitions are going to be moved to that new member. Once the partitions are moved, 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 member owned 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, there can be a 'hard' removal: for example, 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.

There is a demo on YouTube: http://www.youtube.com/watch?v=TOhbhKqJpvw. 4 Terabytes 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 will use in the following sections.

You do not 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 want to configure the map, you can use the following 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 use 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 simple since you can use familiar methods like `get` and `put`.

To demonstrate this basic behavior, the following Member creates a Map and writes some entries into 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 is 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
```

The map updates from the `FillMapMember` are visible in the `PrintAllMember`.

Internally, Hazelcast will serialize the key/values (see Serialization) to byte arrays and store them in the underlying storage mechanism. This means changes made to a key/value after they are stored in the Map will not be reflected on the stored state. Therefore, the following code 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 serializes the key/value to byte arrays when they are stored, and Hazelcast 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 Serialization.

Serializing and deserializing an object too frequently on one node can have a huge impact on performance. A typical use case would be Queries (predicate) and Entry Processors reading the same value multiple times. To eliminate this impact on performance, the objects should be stored in object format, not in binary format; this means that the value returned is the instance and not 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 is only available for values; keys will always be stored in binary format. You should understand the available in-memory formats:

- `BINARY`: the value is stored in binary format. Every time the value is needed, it will be deserialized.
- `OBJECT`: the value is stored in object format. If a value is needed in a query/entry-processor, this value is used and no deserialization is needed.

The default `in-memory-format` is `BINARY`.

The big question 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 is only available in binary format.

If the majority of your operations are regular Map operations like `put` or `get`, you should consider the `BINARY` in-memory format. This sounds counterintuitive because normal operations, such as `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 never returns the stored instance but creates a clone instead. This involves a serialization on the owning node followed by a deserialization on the caller node. With the `BINARY` format, only a deserialization is needed and therefore the process 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; this involves a serialization followed by a deserialization. When the `BINARY` in-memory format is used, only a deserialization is needed.

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>
```

If a value is stored in `OBJECT` in-memory format, a change on a returned value does not affect the stored instance because a clone of the stored value is returned, not the actual instance. Therefore, changes made on an object after it is returned will not be reflected 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. Therefore, changes made on the object after it is stored will not be reflected on the actual stored data.

5.3.1. Good to know:

Unsafe to use with `EntryProcessor` *in combination with queries:* If the `OBJECT` in-memory format is used, then the actual object instance is stored. When the `EntryProcessor` is used in combination with `OBJECT` in-memory format, then an `EntryProcessor` will have access to that object instance. A query also will have access to the actual object instance. However, queries are not executed on partition threads. Therefore, at any given moment, an `EntryProcessor` and an arbitrary number of query threads could access the same object instance. This can lead to data races and Java memory model violation.

Unsafe to use with MapReduce: If the `OBJECT` in-memory format is used in combination with MapReduce, you can run into the same data races and Java Memory Model violations as with the `EntryProcessor` in combination with queries.

5.3.2. What Happened to Cache-value

The `cache-value` property in Hazelcast 2.x has been dropped in Hazelcast 3. Just as with the `in-memory-format`, the `cache-value` makes it possible to prevent unwanted deserialization. When the `cache-value` was enabled, it was possible to get the same instance on subsequent calls like `Map.get`. This problem does not happen with the `in-memory-format`. The reason to drop `cache-value` 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 problems such as concurrency control issues.

5.4. Hashcode and Equals

In most cases, you probably will make use of basic types for a key, such as a `Long`, `Integer`, or `String`. But in some cases, you will need to create custom keys. To do it correctly in Hazelcast, you need to understand how hashcode and equals are implemented in Hazelcast, because it works differently compared to traditional Map implementations. Traditional users make their own implementation: for example, based on firstname/lastname for a person object. However, Hazelcast uses the binary representation of your object to determine the equals and hash. When you store a key/value in a Hazelcast Map, instead of storing the object, the object is serialized and stored to byte arrays. 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: Serializable.

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.

The difference is subtile, but it is crucial to understand.

Below is an example of a 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 is used in the hash/equals implementation and the insignificant field 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 `hashCode` 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
```

The `Pair` works fine for a `HashMap`, but doesn't work for a Hazelcast `IMap`.

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 three 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>
```

In the following code, we define two values, `v1` and `v2`, where the resulting byte array is different. The `equals` method will indicate that they are the same. 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
```

`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. 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 therefore it is important that the hash and equals are implemented correctly. For more information, the book "Effective Java" mentions that you should obey the general contract when overriding equals; 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 you frequently use the customer in combination with his orders, the orders may probably be stored in different partitions than the customer since the customer partition is determined with the `customerId` and the order partition is determined with 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 fewer network hops and traffic are required.

To demonstrate this behavior, the code below implements 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` 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 the partition of the `customerId`, the `getPartitionKey` method will use 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 example since Hazelcast will make use of the binary format of the key. In practice, you should implement them. For more information, see Hashcode and Equals.

In the following example, an order is placed with an `OrderKey`. At the end of the example, 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
```

The partition of the customer is the same as the partition of the order of that customer. Also, 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. For example, if an invoice needs to be created for the orders of a customer, a Callable that creates the Invoice could be sent using the `IExecutorService.executeOnKeyOwner(invoiceCallable, customerId)` method. 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 Distributed Executor Service and Distributed Executor Service: 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>
```

You can set `backup-count` to 0 if you favor performance over high availability. You can specify a higher value than 1 if you require increased availability; but the maximum 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; you are guaranteed that the backup(s) are updated before a method call like `map.put` completes. But this guarantee comes at the cost of blocking and therefore 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 are also kept in memory. So for every backup, you will double the original memory consumption.

By default, Hazelcast provides sequential consistency: 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 therefore there will be no out-of-sync copies. But sequential consistency comes at a price: if the value is read on an arbitrary cluster member, then 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 allowing reads to potentially see stale data. This feature is available only when there is at least 1 backup (synchronous or asynchronous). You can enable it 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 reading of backup data is enabled (the `read-backup-data` 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, there is a 1 in 10 chance that the read will find the data locally. When there is a single backup and `read-backup-data` is false, that adds another 1 in 10 chance that read will find the backup data locally. This totals to 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 delete them manually, but you can also rely on an eviction policy that deletes 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:

- `max-size`: Maximum size of the map. When maximum size is reached, the Map is evicted based on the policy defined. The value is an integer between 0 and Integer.MAX VALUE. 0 means Integer.MAX_VALUE and the default is 0. A policy attribute (`eviction-policy` seen below) determines how the `max-size` will be interpreted.
    - `PER_NODE`: Maximum number of map entries in the JVM. This is the default policy.
    - `PER_PARTITION`: Maximum number of map entries within a single partition. This is probably not a policy you will use often, because the storage size depends on the number of partitions that a member is hosting. If the cluster is small, it will host more partitions and therefore more map entries than with a larger cluster.
    - `USED_HEAP_SIZE`: Maximum used heap size in MB (mega-bytes) per JVM.

    - `USED_HEAP_PERCENTAGE`: Maximum used heap size as a percentage of the JVM heap size. If the JVM is configured with 1000 MB and the `max-size` is 10, this policy allows the map to be 100 MB before map entries are evicted.

- `eviction-policy`: Policy for evicting map entries.
    - `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`.
    - `LRU`: Least Recently Used.
    - `LFU`: Least Frequently Used.
- `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 are not updated for this duration will get automatically evicted from the map. The value can be any integer between 0 and Integer.MAX_VALUE. 0 means infinite, and 0 is the default.
- `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` method is called. The value can be any integer between 0 and Integer.MAX_VALUE. 0 means infinite, and 0 is the default.
- `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 value that is too small, then only that small amount of map entries are evicted, which can lead to a lot of overhead if map entries are frequently inserted.

Here is 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.

You can 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 no `MapStore` is defined, there is no difference. If a `MapStore` is defined, an `IMap.delete` will call a delete on the `MapStore` and potentially delete the map entry from the database. However, the evict method removes the map entry only from the map.

`MapStore.delete(Object key)` is not called when a `MapStore` is used and a 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 therefore most requests will require remoting. 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 you 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**. Near cache makes map entries locally available by adding a local cache attached to 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>
```

You can configure the following properties on the near cache:

- `max-size`: 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 is better to either explicitly configure `max-size` in combination with an `eviction-policy`, or set `time-to-live-seconds`/`max-idle-seconds` to prevent OutOfMemoryErrors. The `max-size` of the near cache is independent of that of the map itself.
- `eviction-policy`: Policy used to evict members from the cache when the near cache is full. The following options are available:
  - `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. You can combine NONE with `time-to-live-seconds` and `max-idle-seconds`.
  - `LRU`: Least Recently Used. This is the default policy.
  - `LFU`: Least Frequently Used.
- `time-to-live-seconds:` 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.
- `max-idle-seconds`: 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.

- `invalidate-on-change`: If true, all the members listen for change in their cached entries and evict the entry when it is updated or deleted. Valid values are true/false and the default is true.

- `in-memory-format`: In-memory format of the cache. Defaults to `BINARY`. For more information, see InMemoryFormat.

Here is 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`. It will evict near-cache entries from a member as soon as the `near-cache` size within that member exceeds 10000. It will then remove `near-cache` entries that are least recently used. When near cache entries are not used for more than 60 seconds, they will be evicted as well.

The previous Eviction section discussed evicting items from the map, but it is important to understand that 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, while 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:

- It increases memory usage since the near cache items need to be stored in the memory of the member.

- It reduces consistency, especially when `invalidate-on-change` is false: it could be that a cache entry is never refreshed.

- 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.

- It can also be enabled on the client. See Hazelcast Clients.

- 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`. In some cases, 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. You can also use the `executeOnKey` API, such as the `IMap.executeOnKey` method. Instead of dealing with pessimistic locking, such as `IMap.lock(key)`, or dealing with optimistic locking, such as `IMap.replace(key, oldvalue, newvalue)`, the `executeOnKey` takes care of concurrency control for you with very low overhead.

```
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: the `map.lock` and `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 can potentially increase contention, it will reduce space.

Good to know

- When the record is deleted, the lock associated to that record is deleted as well.
- When a map is deleted, all the locks associated to the records are deleted.
- A map lock doesn't support fairness, just like the regular `ILock`.
- The map lock is reentrant.
- Although it has the same infrastructure as an `ILock`, a map lock can't be explicitly retrieved using `HazelcastInstance.getLock`.
- You can lock a map entry of a non-existing key.
- When you unlock the map entry of a non-existing key, the map entry will automatically be deleted.

5.9.2. Optimistic Locking

It is important to implement object equals on the value, because that is used to determine if two 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 object equals is used. If you fail to use the correct equals, your code will not work!

```
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;
        }
    }
}
```

This code is broken on purpose. The problem can be solved by adding a `version` field; although all the other fields will be equal, the `version` field will prevent objects from being seen as equal.

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);
    }
}
```

The first reason this example is naive is that this functionality isn't very scalable; a single machine will need to pull all the employees to itself, transform it, and write it back. If your number of employees doubles, it will probably take twice as much time. Another problem is that the current implementation is subject to race problems; imagine that a different process currently gives an employee a raise of 10.

The read and write of the employee is not atomic since there is no lock, so 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 a map entry. Hazelcast will send the `EntryProcessor` to each member in the cluster, and then 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 problems by acquiring exclusive access to the map entry when it is processing.

In the following example, the raise functionality is 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) {
            Employee employee = entry.getValue();
            employee.incSalary(10);
            entry.setValue(employee);
            return null;
        }
    }
}
```

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

```
Mark salary: 1010
John salary: 1010
Spencer salary: 1010
```

5.10.1. Process Return Value

In the example, the process method modifies the `employee` instance and returns null. The `EntryProcessor` can also 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);
```

You need to be 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 from containing 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, such as log or retrieve information. The previous example, where the total salary of all employees is calculated, is such a 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, let's 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 we want to apply logic on the backups, we need to return an `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; therefore, while the `EntryProcessor` is running, no other operations on that map entry can happen.

It is important to understand that an `EntryProcessor` should run quickly because it is running on the partition thread. This means that other operations on the same partition will be blocked, and that other operations that use a different partition but are mapped to the same operation thread will also be blocked. Also, system operations such as partition migration will be blocked by a long running `EntryProcessor`. The same applies when an `EntryProcessor` is executed on a large number of entries; all entries are executed in a single run and will not be interleaved with other operations.

You need to take care to store mutable states in your `EntryProcessor`. For example, if a member contains partition 1 and 2 and they are mapped to partition threads 1 and 2, and if you are executing the entry processor on map entries in partition 1 and 2, then the same `EntryProcessor` will be used by different threads in parallel. It isn't a problem when you use `IMap.executeOnKey`, but it can be a problem with the other `IMap.execute` methods.

5.10.4. Good to know

InMemoryFormat: If you are often using the `EntryProcessor` or queries, it might be a good idea to use the `InMemoryFormat.OBJECT`. The `OBJECT` in-memory format in Hazelcast will not serialize/deserialize the entry, so you are able to apply the `EntryProcessor` without serialization cost. The value instance that is stored is passed to the `EntryProcessor`, and that instance will also be stored in the map entry (unless you create a new instance). For more information, see InMemoryFormat.

Process single key: If you want to execute the `EntryProcessor` on a single key, you can use the `IMap.executeOnKey` method. You could do the same with an `IExecutorService.executeOnKeyOwner`, but you would need to lock and potentially deal with more serialization.

Not Threadsafe: If state is stored in the `EntryProcessor` between process invocations, you need to understand that this state can be touched by different threads. This is because the same `EntryProcessor` instance can be used between different partitions that run on different threads. One potential solution is to put the state in a local thread.

Process using predicate: Deletion: You can delete items with the `EntryProcessor` 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;
    }
}
```

HazelcastInstanceAware: Because the `EntryProcessor` needs to be serialized to be sent 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. EntryListener

One of the new features of Hazelcast 3.0 is the `EntryListener`. You can listen for map entry events providing a predicate, and so the events 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 start the `ListeningMember` and then start 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
```

5.11.1. Threading

To correctly use the `EntryListener`, you must understand the threading model. Unlike the `EntryProcessor`, the `EntryListener` doesn't run on the partition threads. It runs on an event thread, the same threads that are used by other collection listeners and by `ITopic` message listeners. The `EntryListener` 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 since they don't get a thread. But it can also lead to OOME because of events being queued quicker than they are being processed.

5.11.2. Good to know

No events: When no EntryListeners are registered, no events will be sent, so you will not pay the price for something you don't use.

HazelcastInstanceAware: When an `EntryListener` is sent 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` implements `HazelcastInstanceAware`, you can inject the `HazelcastInstance`. For more information see Serialization: HazelcastInstanceAware.

5.12. Continuous Query

In the previous section, we talked about the `MapEntryListener`, which can be used to listen to changes in a map. One of the new additions to Hazelcast 3 is the Continuous Predicate: an `EntryListener` that is registered using a predicate. This makes it possible to listen to the changes made to 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 let's 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 na `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 the SqlPredicate `name=peter`. The listener will be notified as soon as a person with the 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`.

5.12.1. Good to know

Filtered at the source: The predicate of the continuous query is registered at the source: it is registered on each member that generates an event for a given partition. This means that if a predicate filters out an event, the event will not be sent over the line to the listener.

5.13. Distributed Queries

Imagine that we have a Hazelcast `IMap` where the key is some ID, the value is a `Person` object, and we want to retrieve all persons with a given name using the following (and 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 a distributed map, there are some performance and scalability problems with this approach.

- 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.
- 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 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 is parallelizable. By adding new cluster members, the number of partitions per member is reduced. Therefore, the time a member needs to iterate over all of 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 the amount of network traffic is reduced drastically, since only filtered data is sent instead of all data.

Hazelcast provides two APIs for distributed queries.

1. Criteria API
2. Distributed SQL Query

5.13.1. 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 sent over the line, it needs to be serializable. See Serialization for more information.

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

Equal Operator

In the previous example, we saw the equal operator in action where it gets the name of the person object. When it is evaluated, it first tries to look up an accessor method, so in case of `name`, the accessor methods that it will try are `isName()` and `getName()`. If one is found, it is called and the comparison is done. 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; 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: for example, you want the street of the address where the person lives. With the equal operator, you can do it like this: `address.street`. This expression is evaluated from left to right and there is no limit on the number of steps involved. Accessor methods can also 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` Operators

Predicates can be joined using the `and` and `or` operators:

```
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 here is 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 collection of useful operators.

- `notEqual`: Checks if the result of an expression is not equal to a certain value.
- `instanceOf`: Checks if the result of an expression has a certain type.
- `like`: Checks if the result of an expression matches some string pattern. % (percentage sign) is a placeholder for many characters, _ (underscore) is a placeholder for only one character.
- `greaterThan`: Checks if the result of an expression is greater than a certain value.
- `greaterEqual`: Checks if the result of an expression is greater than or equal to a certain value.
- `lessThan`: Checks if the result of an expression is less than a certain value.
- `lessEqual`: Checks if the result of an expression is less than or equal to a certain value.
- `between`: Checks if the result of an expression is between two values (this is inclusive).
- `in`: Checks if the result of an expression is an element of a certain collection.
- `isNot`: Checks if the result of an expression is false.
- `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 map entry 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 have used so far to create Predicates is clear. That syntax can be simplified more by making use of the `PredicateBuilder`. `PredicateBuilder` provides a fluent interface that can make building predicates simpler. But underneath, the same functionality is being used. Here is an example where a predicate is built that selects all persons with a certain name and age using `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, `PredicateBuilder` can simplify things, especially if you have complex predicates. It is a matter of taste which approach you prefer.

With the `PredicateBuilder`, it is possible to access the key. Imagine there is a key with field x and a value with field y. Then you could do the following to retrieve all entries with key.x = 10 and value.y = 20:

```
EntryObject e = new PredicateBuilder().getEntryObject();
Predicate predicate = e.key().get("x").equal(10).and(e.get("y").equal("20"));
```

5.13.2. Distributed SQL Query

In the previous section, the Criteria API was explained, where expression/predicate objects are manually created. This process can be simplified a bit by making use of the `PredicateBuilder`, but it still isn't perfect. That is why a DSL (Distributed SQL Query) was added which is based on an SQL-like language, and it uses the Criteria API underneath.

The `getWithName` function that we already implemented using the Criteria API can also be implemented using the Distributed SQL Query:

```
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` and therefore it can be combined with the Criteria API. The language isn't case sensitive, but "columns" used in the query are. Below, you can see an overview of the DSL:

- logical operators

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

- relational operators

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

- `between`

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

- `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')
```

- `in`

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

5.13.3. Good to know

No key access: With the SQL predicate, it isn't possible to access the key; use the `PredicateBuilder` for that.

Object traversal: With the SQL predicate, an object traversal can be done using `field.otherfield`. For example:

```
husband.mother.father.name=John
```

In this example, the name of the father of the mother of the husband should be `John`.

No arg methods: No arg methods can be called within a SQL predicate. In some cases, this is useful if you dynamically need to calculate a value based on some properties. The syntax is the same as for accessing a field.

5.14. Indexes

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

1. Ordered: for example, a numeric field where you want to do range searches like "bigger than".
2. Unordered: for example, 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. 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 a 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 a field (for the 'x.y' notation) is allowed to be null.

Hazelcast 3 has a big difference from Hazelcast 2: in Hazelcast 3, indexes can be created on the fly. Management Center even has an option to create an index on an existing `IMap`.

The performance impact of using one or more indexes depends on several factors; among them are the size of the map and the chance of finding the element with a full table scan. Other factors are adding one or more indexes, making 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. It is recommended that you test in a production-like environment, using a representative size/quality of the dataset, to see which configuration is best for you. In the source code of the book, you have 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 4 characters. With Hazelcast version 3.0+, indexing is done on the entire `String`.

In the example, the indexes are placed as attributes of basic data types 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.

5.15. Persistence

In the previous section, we talked about backups that protect against member failure: if one member goes down, another member takes over. But it does not protect you against cluster failure: for example, when a cluster is hosted in a single datacenter, and it goes down. Luckily, Hazelcast provides a solution loading and storing data externally, such as 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.

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;
    }
}
```

The implementation is simple and certainly can be improved, such as transactions, prevention against SQL injection, etc. Because the `MapStore/MapLoader` can be called by threads concurrently, this implementation makes 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 and then exits the JVM. Then, 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`, 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.

5.15.1. Pre-Populating the Map

With the `MapLoader`, you can pre-populate the Map so that when it is created, the important entries are loaded in memory. You can do this 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 returns null, as it does in the example, then the map will not be pre-populated. Map is created lazily by Hazelcast, so the map is actually created and the `MapLoader` called only when one of the members calls the `HazelcastInstance.getMap(name)`. If your application returns the map up front without needing the content, you could wrap the map in a lazy proxy that calls the `getMap` method only when it is really needed.

A common mistake made is having the `loadAllKeys` return all keys in the database table. This can be problematic since you would pull the complete table into memory, but another important problem is that if each member returns all the keys, each member will load the complete table from the database. So, if 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.

You need to be aware that the map only knows about map entries that are in the memory; when a get is done for an explicit key, then the map entry is loaded from the `MapStore`. This behavior is called "read through". So if the `loadAll` returns a subset of the keys in the database, then the `Map.size()` will show only the size of this subset, not the record count in the database. 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 the 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 having to call `MapStore.delete`.

5.15.2. 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 your persistence mechanism happens. Write through operations increase latency since databases cause latency (for example, disk access). In Hazelcast, you can 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. You can enable write behind by configuring the `write-delay-seconds` in the `map-store` configuration section. `write-delay-seconds` 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.

5.15.3. MapLoaderLifecycleSupport

In some cases, your `MapLoader` needs to be notified of lifecycle events. You can do this by having your `MapLoader` implement the `com.hazelcast.core.MapLoaderLifecycleSupport` interface.

- `init`: Useful if you want to initialize resources, such as 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.

- `destroy`: Useful if you need to cleanup resources, such as closing database connections.

5.15.4. Good to know

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

5.16. 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 the 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 such as 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, for 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, such as using a lock or a replace call, you could run into a lost update which can lead to issues like items not being deleted or 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 (locking, listeners, etc.), but it doesn't support indexing, predicates, and the `MapLoader/MapStore`.

To demonstrate the `MultiMap`, we are going to create two members. The `PutMember` will put data into 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`.

5.16.1. Configuration

The `MultiMap` is configured with the `MultiMapConfig` using the following configuration options.

- `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 irrelevant. With the list, duplicates and null values are allowed and ordering is relevant. Defaults to SET.
- `listenerConfigs`: The entry listeners for the `MultiMap`.
- `binary`: If the value is stored in binary format (true) or in object format (false). Defaults to true.
- `backupCount`: The number of synchronous backups. Defaults to 1.
- `asyncBackupCount`: The number of asynchronous backups. Defaults to 0.
- `statisticsEnabled`: If true, the statistics have been enabled.

The statistics can be accessed by calling the `MultiMap.getLocalMultiMapStats()` method.

5.16.2. Good to know:

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

Get returns copy: `map.get(key)` returns a copy of the values at some moment in time. Changes to this collection will result in an `UnsupportedOperationException`. 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 you may 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 member1, and member2 does a `map.get(K)`, then the whole collection will be transported from member1 to member2. If that value collection is big, it could lead to performance problems. A solution would be to send the operation to member1: send the logic to the data instead of sending 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.

5.16.3. 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 differences. For example, (de)serialization needs to take place for a lot of operations. Also remoting could be 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 `in-memory-format`.

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

Memory Usage: A completely empty `IMap` instance consumes >200 KBs 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.

5.17. What is next

The Hazelcast `IMap` is a data structure that is rich in features and by properly configuring, it will serve a lot of purposes as you saw in this chapter. In the following chapter, you will learn about Hazelcast Distributed Executor Service.

Chapter 6. Distributed Executor Service

Java 5 was perhaps the most fundamental upgrade to Java since it was first released. On a language level, we got generics, static imports, enumerations, varargs, and enhancements for loops 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 basis, other parts you probably do. One of the added features 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 then it is submitted to the Executor. Here is 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: you can perform complex database operations, perform intensive CPU or I/O operations, render images, etc.

However, the problem in a distributed system is that the default implementation of the Executor, which is 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 JVM. Luckily, Hazelcast provides the `IExecutorService`, which extends the `java.util.concurrent.ExecutorService`. It is designed to be used in a distributed environment. The `IExecutorService` is new in Hazelcast 3.x.

Let's start with a 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 sent to a different JVM to be processed. Instead of making the class Serializable, you could also rely on other serialization mechanisms; see 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`. Then we slowly submit 1000 echo tasks. One of the differences between Hazelcast 2.x and Hazelcast 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 one, so we configure it in the `hazelcast.xml` file like this:

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

Another difference from Hazelcast 2.x is that the `core-pool-size` and `keep-alive-seconds` properties 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
....
```

The production of messages is 1 per second and the processing is 0.2 per 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 dimensions for scaling:

1. Scale up
2. Scale out

Both are explained below. 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 in the example can process 0.2 messages/second and we produce 1 message/second, if the Executor has 5 threads it can process messages as fast as they are produced.

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

Scaling up the `ExecutorService` in Hazelcast is simple, just increment the maximum pool size. Since we know that 5 threads is going to give maximum performance, let's 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 were enough CPU and memory capacity. But one or more of the resources can often 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. 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 per second, maximum performance is reached with 4 slaves.

6.3. Routing

Until now, we didn't care which member did the actual processing of the task, as long as a member picks it up. But in some cases you want to have that control. Luckily, the `IExecutorService` provides different ways to route tasks.

1. Any member. This is the default configuration.
2. A specific member.
3. The member hosting a specific key.
4. All or subset of the members.

In the previous section, we already covered the first way: routing to any member. In the following sections, we'll explain the last 3 routing strategies. This is 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 will 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 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, that 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 a partition and sending the task to the member that owns that partition. When you design a distributed system, perhaps the most fundamental step is designing the partitioning scheme.

As an example, we will create a distributed system where there is dummy data in a map. For every key in that map, we will execute a verify task. This task will verify if it has been executed on the same member as where the partition for that key resides.

```
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 that 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. This task implements `HazelcastInstanceAware`, signaling to Hazelcast that when this class is deserialized for execution, it will inject the `HazelcastInstance` executing that task. For more information, see Serialization: HazelcastInstanceAware.

The next step is the `MasterMember`. First, it creates a map with some entries; we only care about the key so the value is bogus. Then, it iterates over the keys in the map and sends 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);
   }
}
```

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

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

The tasks are executed on the same member as where the data resides.

From Hazelcast 2.x, an alternative way of executing a request on a specific member has been 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 the executor uses 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 members, or even on all members. Use this functionality wisely, since it will create a load on multiple members, potentially all members, and therefore it can reduce scalability.

The following example has a set of members. On these members, there is a distributed map containing some entries. Each entry has a UUID as key and 1 as value. To demonstrate executing a task on all members, we will 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. The result will be a map of `Future` instances. And finally we'll join all the futures, 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 one 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 for the 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. 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. You could build such an executor on top of the SPI.

6.3.4. 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: for example, when you use a `Callable` or you just want to wait till a task completes. You can do this by using the `java.util.concurrent.Future` in combination with one of the submit methods of the `IExecutorService`.

To demonstrate the `Future`, we will 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 to submit the task and use 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");
      }
   }
}
```

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 the 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 timeout on your machine, it could be that your machine is very quick and you need to use a smaller timeout. Unlike Hazelcast 2.x, in Hazelcast 3.0 it isn't possible to cancel a future. One possible solution is to let the task periodically check if a certain key in a distributed map exists. A task can then be cancelled by writing some value for that key. You need to take care removing keys to prevent this map from growing; you can do this by using the time to live setting.

6.4. 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, and in the same thread that submitted the task. In 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 Futures section, an example was shown where a Fibonacci number is calculated. Waiting for the completion of that operation is done using a Future. In the following example, we 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");
    }
}
```

The `ExecutionCallback` has 2 methods. One method is called on a valid response and prints it. The other method is called on failure and it prints the stacktrace.

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

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

The thread that submitted the tasks to be executed was not blocked. Eventually, the result of the Fibonacci calculation will be printed.

6.5. Good to know

Work-queue has no high availability: Each member creates one or more local ThreadPoolExecutors with ordinary work queues that do the real work. When a task is submitted, it is put on the work queue of that `ThreadPoolExecutor` and is not backed up by Hazelcast. If something happens 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: This can lead to OutOfMemoryErrors because the number of queued tasks can grow without being limited. You can solve this 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 it immediately throws a `RejectedExecutionException` that needs to be dealt with. Perhaps in the future, blocking with configurable timeout will be made available.

No Load Balancing: This is currently available for tasks that can run on any member. In the future, there will probably be a customizable load balancer interface where load balancing could be done on the number of unprocessed tasks, CPU load, memory load, etc. If load balancing is needed, you can create 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 executors in every member, and therefore 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, then 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, but it can be annoying when you are spending a lot of time trying to find out why something doesn't work. This can easily be fixed. You can add a try/catch in your runnable and log the exception. You can also 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 access the `HazelcastInstance`. This can be done by letting the task implement `HazelcastInstanceAware`. For more information, see Serialization: HazelcastInstanceAware.

6.6. What is next

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

```
 [[distributed-topic]]
== Distributed Topic
```

In the Distributed Collections chapter, we talked about the `IQueue`, which you can use to create point-to-point message solutions. In such solutions, there can be multiple publishers, but 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: `com.hazelcast.core.ITopic` 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) since it needs to go over the line.

We'll show you how the distributed topic works based on a simple example where a single topic 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);
    }
}
```

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());
        }
    }
}
```

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

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

To make it 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 subscribe to a topic. If you prefer a more static approach, you can also subscribe to a topic through the configuration file.

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

Hazelcast uses reflection to create an instance of the `MessageListenerImpl`. For this to work, this class needs to have a no-arg constructor. If you 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);
```

6.7. Message Ordering

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

Even though messages will be received in the same order by default, nothing can be done about the ordering of messages sent by different members. Imagine that member M sends m1, m2, m3 and member N sends n1, n2, n2. Then listener1 could receive m1, m2, m3, n1, n2, n3, but listener2 could receive 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 all listeners to receive the messages in exactly the same order. If listener1 receives m1, m2, m3, n1, n2, and n3 in that order, then you want listener2 to receive the messages in exactly the same order. To realize this ordering guarantee, Hazelcast provides a `global-order-enabled` setting. Be careful, it will have a considerable impact on throughput and latency.

The `global-order-enabled` setting doesn't do 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).

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

You can deal with this by increasing the number of threads running in the striped executor of the event system. Use the property `hazelcast.event.thread.count`, which defaults to 5 threads.

You can also deal with this by offloading the message processing to another thread. You can implement this with 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 with the `StripedRunnable`, where the thread that executes the message is determined based on the topic name. In this example, the capacity

work queue of the executor is unbound, which could lead to OOME problems. In most cases, it is better to give the work queue a maximum capacity so that a call can either fail directly or wait until space becomes available (see the TimedRunnable to control the timeout). Be careful about blocking for too long 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 the messages, you could use an ordinary executor instead of a striped executor.

6.9. Good to know

Threading model: The `ITopic` is built on top of the event system. For more information regarding the threading model of the `ITopic`, see Threading Model.

HazelcastInstanceAware: If you need to access the `HazelcastInstance` in the MessageListener, have it implement the `HazelcastInstanceAware` interface. Hazelcast will then inject the `HazelcastInstance` when the topic is created. Watch out if you have passed a `MessageListener` instance to the `Config` instead of a class, and you create multiple HazelcastInstances with it, because 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 sent or consumed and the transaction is rolled back, the message sending or consumption will not be rolled back.

No garbage collection: There is no garbage collection for the topics. As long as the topics are not destroyed, they will be managed by Hazelcast. This can 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 create a garbage collection mechanism for the `ITopic`. Create a topic statistics `IMap` with the topic name as key, a pair containing the last processed message count, and timestamp as value. When a topic is retrieved from the `HazelcastInstance`, place an entry in the topic statistics map if it is missing. Periodically iterate over all topics from the topics statistics map, such as using the `IMap.localKeySet()` method. Then retrieve the local statistics from the `ITopic` using the `ITopic.getLocalTopicStats()` method and check if 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 the time 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. This solution isn't perfect, since it could happen that a message is sent 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 receive messages that were sent while it was offline.

No metadata: The message is an ordinary POJO and therefore it 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 queue per topic: Hazelcast doesn't publish the messages on a topic specific queue. Instead, it makes a single stripe of queues: the event queue. By default, the size of the queue is limited to 1,000,000, but you can change that with the `hazelcast.event.queue.capacity` property.

When the capacity of the queue is reached, the calling thread will block until either there is capacity on the queue or a timeout happens. When the timeout happens, Hazelcast logs a warning which includes the topic name and the sender of the message. However, the producer of this message remains agnostic about what happened.

If the event queue is full, Hazelcast will block for a short period. The default is 250 ms, but you can change that using the `hazelcast.event.queue.timeout.millis` property. Be careful making the timeout: you don't own the blocked thread, and that could be an internal Hazelcast thread, such as for dealing with I/O. When such a thread blocks for a long period, it could lead to problems in other parts of the system.

There are plans to redesign the `ITopic` so that:

- Each topic gets its own queue with a configurable capacity.
- 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` statistics, you can enable statistics using the `statistics-enabled` property.

```
<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 its topic. If you need to have global statistics, you need to aggregate the statistics of all members.

6.10. What is next

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

Chapter 7. Hazelcast Clients

So far in this book, the examples showed members that were full participants in the cluster. Those members will know 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; 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, the operation is forwarded to a cluster member where it will be processed. A client needs both the Hazelcast core JAR and the Hazelcast client JAR on the classpath.

We will implement an example client where a message is put on a queue by a client and taken from the queue by a full member. Let's 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());
   }
}
```

Below, you 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 the address since the full member will be running on the same machine as the client. If no port is specified, ports 5701, 5702 and 5703 will be checked. If the cluster is running on a different port, such as 6701, it can be specified like this: `.addAddress("127.0.0.1:6701")`.

To see how the client is running, first start the full member and wait till it is up. Then start the client.

You will see that the server prints `Hello`. If you look in the log for the full member, you will see that the client never pops up as a member of the cluster.

In this example, we use programmatic configuration for the `ClientConfig`. You can also configure a `ClientConfig` using the configuration file by:

- using a properties based configuration file in combination with the `com.hazelcast.client.config.ClientConfigBuilder`.
- using an XML based configuration file in combination with the `com.hazelcast.client.config.XmlClientConfigBuilder`.

The advantage of configuring the Hazelcast client using a configuration file is that you can easily pull the client configuration out of the code, which makes the client more flexible. For example, 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, such as the addresses. For that, you can first load the `ClientConfig` using a configuration file, and then adjust the dynamic fields.

If you create a `HazelcastInstance` using the following code:

```
HazelcastInstance hz = HazelcastClient.newHazelcastClient()
```

Then Hazelcast will use the following sequence of steps to determine the client configuration file to use.

1. Hazelcast checks if there is a system property `hazelcast.client.config`. If it exists, it is used. This means that you can configure the configuration to use from the command line using `-Dhazelcast.client.config=/foo/bar/client.xml`. You can also refer to a classpath resource using `-Dhazelcast.client.config=classpath:client.xml`. This makes it possible to bundle multiple configurations in your JAR and select one on startup.
2. Checks if there is a file called `hazelcast-client.xml` in the working directory.
3. Checks if there is a file called `hazelcast-client.xml` on the classpath.
4. Defaults to `hazelcast-client-default.xml`, which is provided by Hazelcast.

If you don't configure anything, the client will use the default configuration.

7.1. Reusing the Client

A client is designed to be shared between threads. You want to prevent creating an instance per request because clients are heavy objects.

- A client contains a thread pool that is used for internal administration like heartbeat checking,

scheduling of refreshing partitions, firing events when members are added and removed, etc.

- A client has a single connection to the cluster for communication, just like cluster members have among each other. This connection is an expensive resource and it is best to reuse it.

In most cases, it is best to create the client in the beginning and keep reusing it throughout the lifecycle of the client application.

7.2. Configuration Options

In the client example, we did a minimal configuration of the `ClientConfig` and relied on defaults. There is a lot more that you can configure.

- `addressList`: Known addresses of the cluster. It does not need to include all addresses, only enough to make sure that at least one will be online. See Failover.

- `connectionTimeout`: Time in 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 that a connection is allowed to go without a response to the heartbeat. If the value is set too low, it could lead to connections often being recreated. If the value is set too high, it can lead to dead members being detected very late.

- `connectionAttemptLimit`: Maximum number of times to try using the addresses to connect to the cluster. Defaults to 2. When a client starts or a client loses the connection with the cluster, it will try to make a connection with one of the cluster member addresses. In some cases, a client cannot connect to these addresses; for example, the cluster is not yet up or it is not reachable. Instead of giving up, one can increase the attempt limit to create a connection. Also have a look at the `connectionAttemptPeriod`.

- `connectionAttemptPeriod`: Period in milliseconds between attempts to find a member in the cluster. Defaults to 3000 milliseconds.

- `listeners`: Enables listening to the cluster state. Currently, only the `LifecycleListener` is supported.

- `loadBalancer`: See LoadBalancing for more information. Defaults to RoundRobinLB.

- `smart`: If true, the client will route the key based operations to the owner of the key at the best effort. Note that it uses a cached version of `PartitionService.getPartitions()` and it does not guarantee that the operation will always be executed on the owner. The cached table is updated every second. Defaults to true.

- `redoOperation`: If true, the client will redo the operations that were executing on the server when the 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 operations, retrying can cause undesirable effects. Note that the redo can perform on any member. If false, the operation will throw the `RuntimeException` that is

wrapping `IOException`. Defaults to false.

- `group`: See Group Configuration.
- `socketOptions`: Configures the network socket options with the methods `setKeepAlive(x)`, `setTcpNoDelay(x)`, `setReuseAddress(x)`, `setLingerSeconds(x)`, and `setBufferSize(x)`.
- `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 Serialization.
- `socketInterceptor`: Allows you to intercept socket connections before a node joins to cluster or a client connects to a node. This provides the ability to add custom hooks to join and perform connection procedures.
- `classLoader`: In Java, you can configure a custom classLoader. It will be used by the serialization service and to load any class configured in configuration, such as event listeners or ProxyFactories.
- `credentials`: Can be used to do authentication and authorization.

This functionality is only available in the Enterprise version of Hazelcast.

7.3. 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 an operation needs to be sent to a specific member, it will be sent directly to that member. If an operation can be executed on any member, Hazelcast does automatic load balancing over all members in the cluster.

One of the new features came with 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 (such as load balance on CPU load, memory load, or queue sizes), you can meet these requirements by creating a custom `LoadBalancer` implementation. In future releases of Hazelcast, some of these implementations might be provided out of the box. If you implement a custom `LoadBalancer`, you can listen to member changes using the following example.

```
Cluster cluster = hz.getCluster();
cluster.addMembershipListener(thelistener);
```

The `MembershipListener` functionality makes it easy to create a deterministic `LoadBalancer` for the following reasons.

- The `MembershipListener` will not be called concurrently.
- The `MembershipListener` `init` method will be called with a set of all current members.
- No events will be lost between calling `init` and calling `memberAdded` or `memberRemoved`.
- The `memberAdded` and `memberRemoved` methods will be called in the order that 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. The load balancer can be configured from the `ClientConfig`.

7.4. Failover

In a production environment, you want the client to support failover to increase high availability. This is realized in two 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 it will be notified if members are added or removed. The `LoadBalancer` can use this update list of member addresses for routing.

7.5. Group Configuration

To prevent clients from joining a cluster, you can configure the cluster group to which the client is able to connect. On this cluster group, you can set the group name and the password.

```
ClientConfig config = new ClientConfig()
    .addAddress("127.0.0.1");
config.getGroupConfig()
    .setName("group1")
    .setPassword("thepassword");
HazelcastInstance client = HazelcastClient.newHazelcastClient(config);
```

The group name defaults to `dev` and the password defaults to `dev-pass`. For more information, see Network Configuration: Cluster Groups.

7.6. Sharing Classes

In some cases, you need to share classes between the client and the server. You could 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 then share this project between the 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 (DTOs). DTOs cause some duplication, but having some duplication is better to deal with than tight coupling, which can make a system very fragile.

7.7. SSL

In Hazelcast 3, you can encrypt communication between client and cluster using SSL. This means that the whole network traffic, which includes normal operations like a `map.put` and includes passwords in credentials and `GroupConfig`, cannot 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());
    }
}
```

7.8. 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 does not host any partitions. Because it is part of the cluster, a lite member knows about the other members. Therefore, it knows about routing requests to the correct member, which could improve performance compared to the native client.

The lite member had some serious drawbacks.

- Because lite members are seen as members in the cluster, they send heartbeats/pings to each other and they 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.
- Although a lite member does not host any partitions, it will run tasks from the Hazelcast executors. This can be undesirable since you don't want to have a client running the cluster tasks.

With Hazelcast 3.x, a client always knows about all members. Therefore, a client can route requests to the correct member without being part of the cluster.

7.9. Good to know

Shutdown: If you don't need a 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 client 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 is also reduced since the heartbeat does not need to be sent anymore. And the client resources running on the cluster are released, such as the EndPoint or distributed Locks that have been acquired by the client. 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. But you need to make sure that the client has access to the appropriate classes and interfaces.

2 way clients: There are cases where you have a distributed system split in different clusters, but there is a need to communicate between the clusters. Instead of creating one big Hazelcast cluster, it could be split up in different groups. To be able to have each group communicate with the other groups, create multiple clients. If you have two groups A and B, then A should have a client to B and B should have 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 many cases, you want to copy/paste the whole serialization configuration of the server to make sure that the client and server are able to serialize/deserialize the same classes.

None smart clients and load balancing: If a client is not smart, it will randomize the members list and try to connect to one of these members until it succeeds. So if you have a 16 node cluster, and 2 members are configured in the client, all load will go through these members. The consequence is that load isn't equally spread over the members. Try to add as many members in the client configuration as possible to balance the load better.

7.10. 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 .NET client, C++ client, Memcache Client and the Rest Client. For more information, check the **Clients** chapter of the Hazelcast Reference Manual.

Chapter 8. Serialization

So far, the examples in this book 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, such as `Serializable` and `Externalizable`. This is 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`.

In Hazelcast, when an object needs to be serialized (for example, because the object is placed in a Hazelcast data structure like a map or queue), Hazelcast first checks if the object is an instance of `DataSerializable` or `Portable`. If that fails, Hazelcast checks if the object is a well known type, such as `String`, `Long`, `Integer`, `byte[]`, `ByteBuffer`, or `Date`, since serialization for these types can be optimized. Then, Hazelcast checks for user specified types, such as ByteArraySerializer and `ByteStreamSerializer`. If that fails, Hazelcast will fall back on Java serialization (including the Externalizable). If this also fails, the serialization fails because the class cannot 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, such as `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.

8.1. Serializable

The native Java serialization is the easiest serialization mechanism to implement, since a class often 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 to add `serialVersionUID` since this prevents the JVM from calculating one on the fly, and that can lead to all kinds of class compatibility issues. In the examples, it is not always added to reduce space, but for production code there is no excuse not to add it.

When you use serialization, 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 using a method that relies on the byte-content comparisons, and the byte-content of equal objects is different, then you get unexpected behavior. 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.

8.2. Externalizable

Another serialization technique supported by Hazelcast is the `java.io.Externalizable`. It provides more control on how the fields are serialized/deserialized and it can also help to improve performance compared to standard Java serialization. Here is 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);
   }
}
```

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.

8.3. DataSerializable

Although Java serialization is very easy to use, it comes at a price.

- Java serialization has lack of control on how the fields are serialized/deserialized.
- It also has 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, let's 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);
   }
}
```

This `DataSerializable` looks a lot like the `Externalizable` functionality since an explicit serialization of the fields is required. Just like the `Externalizable`, the reading of the 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 will be serialized, Hazelcast checks if it implements the `DataSerializable` interface.

One requirement for a `DataSerializable` class is that it has a no-argument 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, let's 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)
```

8.3.1. IdentifiedDataSerializable

One of the problems 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`. It relies on a factory to create the instance and therefore 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);
    }
}
```

This `IdentifiedDataSerializable` `Person` class looks a lot like the `Person` class from the `DataSerializable`, but two 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. You could move the IDs to the `DataSerializableFactory` implementation to 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 FACTORY_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 you need to implement. If you have many subclasses, you might consider using a `switch case` statement instead of a bunch of `if-else` statements. If a type ID is received of an unknown type, you can return null or throw an exception. If null is returned, Hazelcast will throw an exception for you.

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)
```

8.4. Portable

With the introduction of Hazelcast 3.0, a new serialization mechanism was added: the `Portable`. The cool thing about the `Portable` is that object creation is pulled into user space, so you control the initialization of the `Portable` instances and you are not forced to use a no-argument 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, let's 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);
   }
}
```

The `write` method include the field names, making it possible to read particular fields without being forced to read all of them. This is useful for querying and indexing: 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-argument constructor is added so that it can be initialized from the `PortableFactoryImpl`; if you place it in the same package, you could give it a package friendly access modifier to reduce visibility.

The last two interesting methods are `getClassId`, which returns the identifier of that class, and `getFactoryId`, which must return the ID of the `PortableFactory` that will take care of serializing 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;
     }
}
```

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 you could add it to the `PortableFactory` to have a good overview of which IDs are there.

A factory ID needs to be unique and larger than 0. You probably will have more than one `PortableFactory`. To make sure that every factory gets a unique factory ID, you could make a single class/interface where all `PortableFactory` IDs in your system are declared, as shown below.

```
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 portable factories. 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)
```

The `Person` is serialized when it is stored in the map and it is deserialized when it is read. `Serialize` is called twice because for every `Portable` class, the first time it is (de)serialized, Hazelcast generates a new class that supports the serialization/deserialization process. 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. Therefore, they should not contain characters such as `.` or `-`.

8.4.1. DataSerializable vs. Portable

Portable supports versioning and is language/platform independent. This makes `Portable` useful for client/cluster communication. Another advantage is that `Portable` can be more performant for map queries: it avoids full serialization because data can be retrieved on the field level. Otherwise, if the serialization is needed only for intra-cluster communication, then `DataSerializable` is still a good alternative. The disadvantage of `Portable` is that of all metadata: the fields that are available are part of

the payload of every serialized object. So the amount of data transferred with a `Portable` is a lot more than with a `DataSerializable`. If you want to use the fastest serialization mechanism, it is best to have a look at the `IdentifiedDataSerializable`, since no field metadata is send over the line.

8.4.2. Object Traversal

If a `Portable` has a `Portable` field, the write and read operations need to be forwarded to that object. For example, we could 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 unknown and this causes problems with the platform independent nature of `Portable`. In that case, you can call the `PortableWriter.writePortable(String fieldName, int classId, Portable portable)` method, 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 platform independence is lost. A better solution is to create some form of `String` representation, potentially using XML, to maintain platform compatibility. The methods `readUTF/writeUTF` can perfectly deal with null Strings, so passing null object references is no problem.

8.4.3. Serialize DistributedObject

Serialization of the `DistributedObject` is not provided out of the box: for example, you can't put 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`: for example, in case of an `IQueue`, you can call `HazelcastInstance.getQueue(id)` or `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 and it receives an ID that needs to be looked up, the class can implement the HazelcastInstanceAware interface. Since the `HazelcastInstance`

is set after deserialization, you need to store the IDs first, and then you could do the actual retrieval of the distributed objects in the `setHazelcastInstance` method.

8.4.4. Serializing Raw Data

When using the `Portable` functionality, the field name is added so that the fields can be retrieved easily and the field can be indexed and used within queries without needing to deserialize the object. In some cases, this can cause a lot of overhead. If overhead is an issue, you can write raw data using the `PortableWriter.getRawDataOutput()` method and read it using the `PortableReader.getRawDataInput()` method. Reading and writing raw data should be the last reading and writing operations on the `PortableReader` and `PortableWriter`.

8.4.5. Cycles

One thing to look out for, which also goes for `DataSerializable`, are cycles between objects: they can lead to a stack overflow. Standard Java serialization protects against this, but since manual traversal is done in `Portable` objects, there is no protection out of the box. 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.

8.4.6. Subtyping

Subtyping with the `Portable` functionality is easy: let every subclass have its own unique type ID, and then add these IDs to the switch/case in the `PortableFactory` so that the correct class can be instantiated.

8.4.7. Versioning

In practice, multiple versions of the same class could be serialized and deserialized, such as 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`, metadata like the class id and the version are also 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 also changed. In most cases, incrementing the version is the simplest approach.

Adding fields to a `Portable` is simple. However, you probably need to work with default values if an old `Portable` is deserialized.

Removing fields can lead to problems if a new version of that `Portable` (with the removed field) is deserialized on a client that depends on that field.

Renaming fields is simpler because the `Portable` mechanism does not 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 (for example, `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 object to be deserialized 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` cannot be combined with the `HazelcastInstanceAware`. There is a feature request, so perhaps this functionality will be added in the future.

8.5. StreamSerializer

One of the additions to Hazelcast 3 is to use a stream for serializing and deserializing data by implementing the `StreamSerializer`. The `StreamSerializer` is practical if you want to create your own implementations, and you can also use it to adapt an external serialization library, such as JSON, protobuf, Kryo, etc.

Let's start with a very simple object we will 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 is also 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() {
   }
}
```

The implementation is quite simple. The `ObjectDataOutput` and `ObjectDataInput` have methods available for primitives like `int`, `boolean`, etc., and also for `String`; `writeUTF/readUTF` can safely deal with null and also for objects. See Object Traversal.

In 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.

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 IDs and will throw an error if your type ID is smaller than 1.

A practical way to generate unique IDs is to use a class (or interface) where you define all type IDs 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 you use these type IDs 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 IDs when you have old deserialized instances somewhere. This is because a change of the order will change the actual value of the type ID, so 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 case, we have registered the serializer `PersonStreamSerializer` for the `Person` class. When Hazelcast is going to serialize an object, it looks up the serializer registered for the class for that object. Hazelcast is quite flexible; if it fails to find a serializer for a particular class, it first tries to match based on superclasses and then on interfaces. You could create a single `StreamSerializer` that can deal with a class hierarchy if that `StreamSerializer` is registered for the root class of that class hierarchy. If you use this approach, then you need to write sufficient data to the stream so that on deserialization, you can figure out exactly which class needs to be instantiated.

It is not possible to create StreamSerializers for well known types like `Long`, `String`, primitive arrays, etc., since Hazelcast already registers them.

Here is 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'}
```

8.5.1. Object Traversal

In practice, you often need to deal with object graphs. Luckily, this is quite easy. To create the graph, we add the `Car` class. Each car has a color 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 look up a serializer for the particular type. Hazelcast has serializers available for the wrapper types like `Long`, `Boolean`, etc. Luckily, the `writeObject` (and `readObject`) are perfectly able to deal with null.

To 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>
```

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'}}
```

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 Cycles.

8.5.2. Collections

If the field of an object needs to be serialized with the stream serializer, then currently there is no other solution except 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 two implementations. 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 collection classes are being serialized. If there is no collection serializer registered, the system will default to the `GlobalSerializer`, which defaults to normal serialization. This might not be the behavior you are looking for.

8.5.3. Kryo StreamSerializer

Writing a customer serializer, such as a `StreamSerializer`, can be a lot of work. Luckily, there are a lot of serialization libraries for this. Kryo is one of these libraries we use at Hazelcast and it is quite fast and flexible, and it results in small byte arrays. It can also deal with object cycles.

Let's 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);
    }
}
```

No interface needs to be implemented. It is no problem if the class implements a serialization interface like `Serializable` since it will be ignored by Hazelcast.

The `Kryo` instance is not threadsafe and therefore you can't create `PersonKryoSerializer` with a `Kryo` instance as a 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 local thread.

```
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() {
    }
}
```

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 customers, we managed to reduce the size of map entries from a 15 kilobyte average using Java Serialization, to less than a 6 kilobyte average. When we enabled Kryo compression, we managed to get it below 3 kilobytes.

The `PersonKryoSerializer` needs to be configured in Hazelcast.

```
<hazelcast>
    <serialization>
        <serializers>
            <serializer type-class="Person">PersonKryoSerializer</serializer>
        </serializers>
    </serialization>
</hazelcast>
```

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 output:

```
Person(name=Peter)
```

In the previous example, we showed how Kryo can be implemented as a `StreamSerializer`. The cool thing is that you can just plug in a serializer for a particular class; no matter if that class already implements a different serialization strategy such as Serializable. If you don't have the chance to implement Kryo as `StreamSerializer`, then you can also directly implement the serialization on the class. You can do this by using the `DataSerializable` and (de)serializing 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.

8.6. 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 that the `StreamSerializer` is configured.

```
<serialization>
   <serializers>
      <serializer
          type-class="Person">PersonByteArraySerializer</serializer>
   </serializers>
</serialization>
```

8.7. Global Serializer

The new Hazelcast serialization functionality also makes it possible to configure a global serializer in case no other serializers are found.

```
<serialization>
   <serializers>
      <global-serializer>PersonStreamSerializer
      </global-serializer>
   </serializers>
</serialization>
```

There can only be a single global serializer. For this global serializer, the `StreamSerializer.getTypeId` method does not need to return a relevant value.

The global serializer can also be a `ByteArraySerializer`.

8.8. HazelcastInstanceAware

In some cases, when an object is deserialized, it needs access to the `HazelcastInstance` so that distributed objects can be accessed. You can do this by implementing `HazelcastInstanceAware`, as in the following example.

```
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 to 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 you can 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 execute operations on the `HazelcastInstance`. The reason behind this is that for some `HazelcastInstanceAware` implementations, the `HazelcastInstance` isn't fully up and running when it is injected.

You need to be careful with using the `HazelcastInstanceAware` on anything other than the `root` object that is serialized. Hazelcast sometimes optimizes local calls by skipping serialization. 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.

8.8.1. UserContext

Obtaining dependencies other than a `HazelcastInstance` was more complicated in Hazelcast 2.x. Often the only way was to rely on some form of static field. 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`.

Let's start with an `EchoService` dependency that we want to make available in an `EchoTask` which will be executed using an Hazelcast distributed executor.

```
public class EchoService{
    public void echo(String msg){
        System.out.println(msg);
    }
}
```

There are no special requirements for this dependency, and no interfaces to implement. It is just an ordinary POJO.

This `EchoService` dependency needs to be injected into the `UserContext` so 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"));
    }
}
```

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 a general purpose object container like Spring. Therefore, 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 to implement the `HazelcastInstanceAware` interface that injects 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 possible to configure user-context on the Config, and you can also directly configure the user-context of the `HazelcastInstance`. This is practical if you need to add dependencies on the fly. Do not forget to clean up what you put in the user-context, else you might run into resource problems like an `OutOfMemoryError`.

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. If you need to have that `EchoService` available on each member, you need to add it to the user-context on each member.

It is important to know that when a `HazelcastInstance` is created using a `Config` instance, a new user-context `ConcurrentMap` is created and the content of the user-context of the `Config` is copied. Therefore, changes made to the user-context of the `HazelcastInstance` will not reflect on other `HazelcastInstance` created using the same `Config` instance.

Chapter 9. ManagedContext

In some cases, when a serialized object is deserialized, not all of its fields can be deserialized because 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 you the ability to fix the object by setting missing fields and call methods, wrapping it inside a proxy, etc., so it can be used. This mechanism is called the `ManagedContext` and you can configure it on the `SerializationConfig`.

In the following example, we have a `DummyObject` with a serializable field named `ser` and a transient field named `trans`.

```
class DummyObject implements Serializable {
    transient Thread trans = new Thread();
    String ser = "someValue";

    @Override
    public String toString() {
        return "DummyObject{" +
                "ser='" + ser + '\'' +
                ", trans=" + trans +
                '}';
    }
}
```

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 `initialize` 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 the output like:

```
DummyObject{ser='someValue', trans=Thread[Thread-2,5,main]}
DummyObject{ser='someValue', trans=Thread[Thread-3,5,main]}
```

The transient field has been restored to a new thread.

Hazelcast currently provides one ManagedContext implementation: the `SpringManagedContext` for integration with Spring. If you are integrating with different products, such as 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. You can retrieve custom dependencies using the UserContext.

You need to be careful using the ManagedContext on anything other than the root object that is serialized. Hazelcast sometimes optimizes local calls by skipping serialization. Also, 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 can be offered to the ManagedContext, but the graph isn't traversed.

9.1. Good to know

Nested operations: `DataSerializable` and `Portable` instances are allowed to call operations on Hazelcast that lead to new serialize/deserialize operations. Unlike Hazelcast 2.x, it does not lead to StackOverflowErrors.

Thread-safety: The serialization infrastructure, such as classes implementing `DataSerializable`, `Portable` and support structures like `TypeSerializer` or `PortableFactory`, need to be threadsafe since the same instances will be accessed 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 behavior of the class so that it encrypts the data on writing and decrypts on reading. In some cases, such as storing a `String` in a map, the instance needs to be wrapped in a different type (for example, `EncryptedPortableString`) to override the serialization mechanism.

Compression: The `SerializationConfig` has a `enableCompression` property which enables compression for `java.io.Serializable` and `java.io.Externalizable` objects. If your classes make use of a different serialization implementation, there is no out of the box support of compression.

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. This is because potential remoting is involved, and also because data needs to be deserialized. Some users have burned themselves on this issue because it isn't always immediately obvious from the code.

Performance comparisons: For an 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, different parts of the graph could be serialized using different serialization technologies. It could be that some parts 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 everything 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 mind. There is a feature request that makes it possible to use streams between members and members/clients and to overcome this memory limitation. Hopefully, this will be implemented in the near future.

Factory IDs: Different serialization technologies, such as `Portable` vs. `IdentifiedDataSerializable`, don't need to be unique.

9.2. What is Next

In this chapter we have seen different forms of serialization that make 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

In this book's previous chapters, the examples have not contained any transactions. Transactions can make life a lot easier since they provide:

- Atomicity. Without atomicity, some of the operations on Hazelcast structures could succeed while other fail, leaving the system in an inconsistent state.
- Consistency. This moves the state of the system from one valid state to the next.
- 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.
- Durability: This makes sure that if a system crashes after a transaction commits, that nothing gets lost.

There are fundamental changes in the transaction API for Hazelcast 3. In Hazelcast 2.x, some of the data structures could be used with or without a transaction. In Hazelcast 3, transactions are possible only on explicit transactional data structures, such as the `TransactionalMap`, `TransactionalMultiMap`, `TransactionalQueue` and the `TransactionalSet`. The reason behind this design choice is that not all operations can be made transactional: if they were made transactional, they would have huge performance/scalability implications. 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, such as begin, commit, and rollback. But it also supports accessing transactional data structures like the `TransactionalMap` and `TransactionalQueue`.

Here is 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);
   }
}
```

Using a transaction is simple. In the example, the transactional map is retrieved within the transaction. It is not 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 it will contain transaction specific states, such as cache. Therefore, that object should not be reused. Of course, 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`. You can configure a `TransactionalMap` using the configuration mechanism of the IMap. If you have a `TransactionalMap` called `employees`, then you can configure this `TransactionalMap` using:

```
<map name="employees">
    ....
</map>
```

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"));
   }
}
```

The employee that was put in the `TransactionalMap` is 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 does not work and needs to be fine tuned. With the Hazelcast transaction API, you can do this by using the `TransactionOptions` object and passing it to the `HazelcastInstance.newTransactionContext(TransactionOptions)` method.

Currently, Hazelcast provides the following configuration options.

1. `timeoutMillis`: Time in milliseconds a transaction will hold a lock. Defaults to 2 minutes. In most cases, this timeout is enough since the transaction should be executed quickly.

2. `TransactionType`: Either `LOCAL` or `TWO_PHASE`. See TransactionType.

3. `durability`: Number of backups for the transaction log, defaults to 1. See Partial Commit Failure for more infomation.

The following fragment makes use of the `TransactionOptions` to configure a `TransactionContext` which is `TWO_PHASE`, has a timeout of 1 minute, and a durability of 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.

10.2.1. TransactionType

With the TransactionType, you can influence how much guarantee you get when a member crashes when a transaction is committing. Hazelcast provides two TransactionTypes. Their names are a bit confusing.

1. `LOCAL`: Unlike the name suggests, `LOCAL` is a two phase commit. First, all cohorts are asked to prepare if everyone agrees. Then, all cohorts are asked to commit. The problem happens if during the commit phase one or more members crash; the system could be left in an inconsistent state because some of the members might have committed and others might not.

2. `TWO_PHASE`: The two phase commit is more than the classic two phase commit (if you want a regular two phase commit, use `LOCAL`). Before it commits, it copies the commit log to other members, so in case of member failure, another member can complete the commit.

So which one should you use? It depends. `LOCAL` will perform better but `TWO_PHASE` will provide better consistency in case of failure.

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 we 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 using the `TransactionalTask` and the `HazelcastInstance.executeTransaction(TransactionalTask)` method. This method automatically begins a transaction when the task starts and automatically commits it on success or performs a rollback when a Throwable is thrown.

The previous example could be rewritten to use 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);
   }
}
```

An anonymous inner class is used to create an instance of the `TransactionalTask` and to pass it to the `HazelcastInstance.executeTransaction` method, where it will 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 is 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`, you can use the `TransactionalTask` in combination with the `TransactionOptions` by calling the `HazelcastInstance.executeTransaction(TransactionOptions,TransactionalTask)` method like in this example.

```
TransactionOptions txOptions = new TransactionOptions();
...
hz.executeTransaction(txOptions, new TransactionalTask() {
     @Override
     public Object execute(TransactionalTaskContext context)
           throws TransactionException {
       ...
     }
  });
```

10.4. Partial Commit Failure

When a transaction is rolled back and non-transactional data structures are modified, these modifications 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 modify distributed data structures in Hazelcast, you could run into race problems. These race problems are not caused by Hazelcast, but by sloppy application code. These problems are notoriously hard to solve; 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, in the Hazelcast transaction API, isolation from other concurrent executing transactions will be coordinated for you. By default, Hazelcast provides a REPEATABLE_READ isolation level, so dirty reads and non-repeatable reads are not possible, although 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 preventing the isolation anomalies of dirty reads and unrepeatable reads.

10.5.1. No Dirty Reads

One of the read anomalies that can happen in transactional systems is a dirty read. Imagine a map with keys of type `String` and values of type `Integer`. If transaction-1 modifies key `foo` and increments the value from 0 to 1, and transaction-2 reads key `foo` and sees value 1, and then transaction-1 aborts, then transaction-2 sees the value 1, a value that was never committed. This is called a dirty read.

The dirty reads are prevented in Hazelcast by deferring the write till the commit of the transaction. All changes are stored locally in the transaction and therefore are invisible to other transactions. Only when the transaction commits and the cohorts have agreed with the the commit, are the actual changes actually written. This means that in Hazelcast, it is impossible to see the changes of a transaction in progress since all these changes are tracked locally in the transaction, and these are not visible to other transactions.

10.5.2. No Unrepeatable Reads

The repeatable read is implemented by caching all reads in the transaction. So if two subsequent reads of the same key in a TransactionalMap execute, the second read will see exactly the same content as the first read; no matter if a different transaction has updated that key. For more information, see Caching and Session.

10.5.3. 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. Other transactions are not able to see these uncommitted changes, otherwise they would suffer from dirty reads.

10.5.4. No Serialized Isolation Level

Higher isolation levels, like SERIALIZED, are not desirable due to lack of scalability. With the SERIALIZED isolation level, the phantom read isn't allowed. Imagine an empty map where transaction-1 reads the size of this map at time t1 and sees 0. Then transaction-2 starts, inserts a map entry and commits. If transaction-1 reads the size and sees 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. This is undesirable because it would cause a cluster wide blockage of this data structure. That is why Hazelcast does not provide protection against phantom reads, and therefore the isolation level is limited to REPEATABLE_READ.

10.5.5. Non-transactional Data Structures

If a non-transactional data structure (such as a non-transactional `IMap` instance) is accessed during the execution of a transaction, the access is done oblivious to a running transaction. So if you read the same non-transactional data structure multiple times, you could observe changes. Therefore, you need to take care when you choose to access non-transactional data structures while executing a transaction.

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. For example, if a `map.put` is done, the transaction will automatically lock the entry for that key for the remaining duration of the transaction. If another transaction wants to do an update on the same key, it also wants to acquire the lock and will wait till the lock is released or the transaction runs into a timeout (see TransactionOptions.timeoutMillis).

Reads on a map entry will not acquire the lock and reads will also not block if another transaction has acquired that lock; therefore, one transaction is able to read map entries locked by another transaction. Reads don't block writes and writes don't block reads. If you automatically want to acquire the lock when reading, check the `IMap.getForUpdate` method. This provides the same locking semantics as a `map.put` and can be compared with the `select for update` SQL statement.

Hazelcast doesn't have fine grained locks like a `readwritelock`; the lock acquired is exclusive. If a lock can't be acquired within a transaction, the operation will timeout after 30 seconds and throws an `OperationTimeoutException`. This provides protection against deadlocks. 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 is detected and a `TransactionException` is thrown. When a transaction aborts or commits, all locks are automatically released. Also, when a transaction expires, its locks will automatically be released.

If you are automatically retrying a transaction that throws an `OperationTimeoutException`, and you do not control the number of retries, it is possible that the system will run into a livelock. Livelocks are even harder to deal with than deadlocks because the system appears to do something since the threads are busy. But there is either not much or no 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.

10.7. Caching and Session

Hazelcast transactions provide support for caching reads/writes. If you have been using Hibernate, then you probably know the Hibernate-Session. 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 supports a 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 code is broken.

```
TransactionContext txCxt = hz.newTransactionContext();
TransactionalMap<String,Employee> employees = context.getMap("employees");
Employee employee = employees.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 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 co-locate data in the same partition if possible.

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 create nested transactions. 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. 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 a JEE application using the JEE integration; see "J2EE Integration" in the Hazelcast Reference Manual for more information.

Chapter 11. Network Configuration

Hazelcast can run perfectly within a single JVM. This is excellent for development and to speed up testing. But the true strength of Hazelcast becomes apparent when a cluster of JVMs 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. You can create clusters by configuring the network settings.

To test the networking settings, we are going to make use of the following minimalistic Hazelcast member, which loads 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>
```

For brevity reasons, this example leaves out the enclosing Hazelcast tags. You can find the complete sources for this book on the Hazelcast website. For a Hazelcast cluster to function correctly, all members must be able to contact every 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 uses for communication between the members. You can set this with the `port` property in the network configuration. It defaults to 5701.

```
<network>
    <port>5701</port>
</network>
```

If you start the member, you will get output like the following.

```
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 by default, Hazelcast will try 100 ports to find a free one that it can bind to. If you configured port to 5701, Hazelcast tries ports between 5701 and 5801 (exclusive) until it finds a free port. In some cases, you want to control the number of ports tried. For example, you could have a large number of Hazelcast instances running on a single machine or you only want a few ports to be used. You can do this by specifying the `port-count` attribute, which defaults to 100. In the following example you can see the `port-count` with a value of 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 in those rare cases where 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) as shown below.

```
<network>
    <port auto-increment="false">5701</port>
</network>
```

The `port-count` property will be ignored when `auto-increment` is false.

If you look 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 therefore the members can't join to form a cluster. To specify a join mechanism, see Join Mechanism.

11.2. Join Mechanism

Hazelcast supports three mechanisms for members to join the cluster.

1. TCP/IP-cluster
2. Multicast
3. Amazon EC2

One of these mechanisms needs to be enabled to form a cluster, else they will remain standalone. If you use programmatic Hazelcast configuration, `multicast` is enabled by default. If you use XML configuration, none is enabled so you need to enable one. After joining the cluster, Hazelcast relies on TCP for internal communication.

11.2.1. Multicast

With `multicast` discovery, a member will send a message to all members that listen to a specific `multicast` group. It is the easiest mechanism to use, but it is not always available. Here is an example of 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
}
```

The member is started. Currently, the cluster has a single member. If you start another member on the same machine, the following will be added to the output on the console of the first member .

```
Members [2] {
   Member [192.168.1.104]:5701 this
   Member [192.168.1.104]:5702
}
```

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 two-member Hazelcast cluster running on a single machine. It becomes more interesting if you start multiple members on different machines.

You can tune the `multicast` configuration 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 groups. By setting the `multicast-group` or the `multicast-port`, you can have separate Hazelcast clusters within the same network, so 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 of 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 where it sends discovery messages. 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. You don't need to worry that multiple Hazelcast members running on the same JVM will conflict. This property defaults to 54327.

3. `multicast-time-to-live`: Sets the default time-to-live for `multicast` packets sent out 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 start-up of nodes where no master has been assigned yet. If you specify a high value such as 60 seconds, it means until a master is selected, each node is going to wait 60 seconds before continuing. Be careful with providing a high value. Also avoid setting a too low value since nodes could give up too early and 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>
```

11.2.2. Trusted Interfaces

By default, `multicast` join requests of all machines will be accepted, but in some cases you want to have more control. With `trusted-interfaces`, you can control the machines you want to listen to by registering their IP address as a trusted member. If a join request is received for a machine that is not a trusted member, it will be ignored and it will be prevented from joining the cluster. Below is an example where only join 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, such as `192.168.1.*`, and also supports an IP range on the last octet, such as `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, your trusted interfaced configuration may be too strict. Hazelcast will log on the finest level if a message is filtered out so you can see what is happening.

If you use the programmatic configuration, the trusted interfaces are called trusted members.

11.2.3. Debugging Multicast

If you don't see members joining, it is likely that `multicast` is not available. A cause can be the firewall; you can test this by disabling the firewall or enabling `multicast` in the firewall (see Firewall). 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 on two machines within the network, then run the following in the first terminal `iperf -s -u -B 224.2.2.3 -i 1` and run `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>
```

11.3. TCP/IP Cluster

In the previous section, we used `multicast`, but in production environments `multicast` often is prohibited, and in cloud environments `multicast` 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.

Here is an example of a TCP/IP cluster configuration with 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>
```

You can configure multiple members using a comma separated list, or with multiple `<member>` entries. You can define a range of IPs using the syntax `192.168.1.100-200`. If no port is provided, Hazelcast will automatically try the ports 5701..5703. If you do not want to depend on IP addresses, you can provide the hostname. Instead of using more than one `<member>` to configure members, 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 combination with XML variables (see Learning The Basics: Variables).

By default, Hazelcast will `bind` (accept incoming traffic) to all local network interfaces. If this is an unwanted behavior, you can set the `hazelcast.socket.bind.any` 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, Hazelcast 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 a large number of IPs are listed and members can't build up a cluster, you can set the `connection-timeout-seconds` attribute, which defaults to 5, to a higher value. You can configure first scan and delay between scans using the property `hazelcast.merge.first.run.delay.seconds` and respectively `hazelcast.merge.next.run.delay.seconds`. By default, Hazelcast will scan every 5 seconds.

11.3.1. 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 the expected master node.

11.4. EC2 Auto Discovery

Apart from Multicast and the TCP/IP-cluster join mechanisms, there is 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 has certain tag-keys/values or a security group. These instances will be the well known members of the cluster. A single region is used to let new nodes discover the cluster, but the cluster can span multiple regions (it can even span multiple cloud providers). Let's 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>
```

`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 Learning the Basics: Variables.

The AWS section has a few configuration options.

- `region`: Region where the machines are running. Defaults to `us-east-1`. If you run in a different region, you need to specify it, otherwise the members will not discover each other.
- `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 clusters in a single data center.
- `security-group-name`: Just like the `tag-key`,`tag-value`, it filters out EC2 instances. This doesn't need to be specified.

The `aws` tag accepts an attribute called `conn-timeout-seconds`. The default value is 5 seconds. You can increase it if you have many IPs 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 use Hazelcast. You can use the programmatic API to configure a TCP/IP cluster. The well known members need to be retrieved from your cloud provider (for example, 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. If you feed it the configuration settings that 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 thing you can do when your cluster is not being created correctly is 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 rejected and the reason for rejection.

- Bad status: for when it isn't running.
- Non-matching group-name.
- 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.

11.5. Partition Group Configuration

Normally, Hazelcast prevents the master and the backup partitions from being stored on the same JVM to guarantee high availability. But multiple Hazelcast members of the same cluster can run on the same machine, and thus when the machine fails, both master and backup can fail. 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 therefore 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 therefore will distribute the partitions evenly. But in some cases machines are not equal, such as different amounts of memory

available or slower CPUs, and that could lead to a load imbalance. With a partition group, you can make member groups where each `member-group` has the same capacity and where each member has 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. Here 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 two 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 use this approach to create different groups for each data center so that when the primary data center goes offline, the backup data center can take over.

11.6. Cluster Groups

Sometimes it is 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, you can do this 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 other 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.

11.7. SSL

In a production environment, you often want to prevent the communication between Hazelcast

members from being tampered with or being read by an intruder because the communication could contain sensitive information. Hazelcast provides a solution for that: SSL encryption.

The basic functionality is provided by the `SSLContextFactory` interface and it is configurable through the the SSL section in network configuration. 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 and the `keyStorePassword` is the password of the keystore. In the example code, you can find an already created keystore; you can also find how to create one yourself in the documentation. 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 you can set on the `BasicSSLContextFactory`:

- `keyManagerAlgorithm`: Defaults to `SunX509`.
- `trustManagerAlgorithm`: Defaults to `SunX509`.
- `protocol`: Defaults to `TLS`.

Another way you can 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.

11.8. 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 it is 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 have roughly the same CPU and network bandwidth overhead because 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%. To demonstrate the encryption, let's 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 two 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 the property `hazelcast.security.bouncy.enabled`. Hazelcast used to support 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.

11.9. 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. You can use range characters (`and -) for simplicity. For instance, 10.3.10.` 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 (it is disabled by default) and if Hazelcast cannot find an matching interface, then it will print a message on the 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>
```

11.10. 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 it doesn't matter which port is being used as long as the port is free. The problem is that the lack of control on the outbound port can be a security issue, 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, we can configure like this:

```
<network>
   <join>
      <multicast enabled="true"/>
   </join>
   <outbound-ports>
      <ports>30000-31000</ports>
   </outbound-ports>
</network>
```

To demonstrate the outbound ports configuration, start two Hazelcast members with this configuration. When the members are fully started, execute `sudo lsof -i | grep java`. Below you can see the cleaned output of that 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)
```

There are 2 java processes, 46117 and 46120, that listen to ports 5701 and 5702 (inbound traffic). You can see that java process 46120 uses port 30609 for outbound traffic.

Apart from specifying port ranges, you can also specify individual ports. You can combine multiple

port configurations either by separating them with commas or by providing multiple `<ports>` sections. If you want to use port 30000, 30005 and port range 31000 till 32000, you could say the following: `<ports>30000,30005,31000-32000</ports>`.

11.10.1. iptables

If you are using 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
```

To also 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
```

11.11. Connectivity Test

If you are having troubles because machines won't join a cluster, you might check the network connectivity between the two machines. You can use a tool called `iperf` for that. On one machine, you execute the following command:

```
iperf -s -p 5701
```

This means that you are listening to port 5701.

On the other machine, you execute the following command:

```
iperf -c 192.168.1.107 -d -p 5701
```

Replace `192.168.1.107` with 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
```

Then you know the 2 machines can connect to each other. However, if you see 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.

11.12. Good to know

~~~~

Single TCP/IP connection: There is only a single TCP/IP connection between two members, not two. Also, between client and member, there is a single TCP/IP connection. If you run the following program:

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

And then you run the following command:

```
netstat -anp --tcp | grep -i java
```
~~~~

You get roughly the following output:

```
tcp6       0      0 :::5701                 :::*                    LISTEN
28420/java
tcp6       0      0 :::5702                 :::*                    LISTEN
28420/java
tcp6       0      0 192.168.1.100:5701      192.168.1.100:57220     ESTABLISHED
28420/java
tcp6       0      0 192.168.1.100:57220     192.168.1.100:5701      ESTABLISHED
28420/java
```

There is a single TCP/IP connection: 192.168.1.100:5701 < 192.168.1.100:57220.

11.13. 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 of this book. But you can find them in the Hazelcast Reference Manual. Also, the mailing list is a very valuable source of information.

Chapter 12. SPI

One of the most exciting new features of Hazelcast 3 is the new SPI module (see the `com.hazelcast.spi` package). SPI, Service Provider Interface, makes it possible to write first class distributed services and data structures yourself. This SPI is used by the Hazelcast team to build all the distributed data structures like `IMap`, `IExecutorService` and `IAtomicLong`. The SPI API is exposed to the end user, so you can write your own distributed data structures if you need something special. You also could write more complex services, such as an Actor library; a POC actor library has been buit on top of Hazelcast where the actors automatically scale and are highly available.

In this chapter, we build a distributed counter.

```
public interface Counter{
    int inc(int amount);
}
```

This counter will be stored in Hazelcast and it can be called by different members. This counter will also be scalable: the capacity for the number of counters scales with the number of members in the cluster. The counter will be highly available: if a member hosting that counter fails, a backup will already be available on a different member and the system will continue as if nothing happened. We are going to do this step by step; in each section, a new piece of functionality is going to be added.

12.1. Getting Started

In this section, we are going to show you a very basic `CounterService` whose lifecycle is managed by Hazelcast. In itself that is not extremely interesting, but it is the first part that needs to be in place for the Counter functionality. The `CounterService` is the gateway into the Hazelcast internals. Through this gateway, you will be able to create proxies, participate in partition migration, etc.

The `CounterService` needs to implement the `com.hazelcast.spi.ManagedService` interface.

```
public class CounterService implements ManagedService {
   private NodeEngine nodeEngine;

   public void init(NodeEngine e, Properties p) {
      System.out.println("CounterService.init");
      this.nodeEngine = e;
   }

   public void shutdown() {
      System.out.println("CounterService.shutdown");
   }

   public void reset() {
   }
}
```

The following methods need to be implemented.

- `init`: This method is called when this `CounterService` is initialized. It gives you the ability to do some initializing. The `NodeEngine` gives access to the internals of Hazelcast like the `HazelcastInstance`, `PartitionService`, etc. Through the `Properties` object, you can pass in your own properties.
- `shutdown`: This method is called when `CounterService` is shutdown. It gives the ability to clean up resources.
- `reset`: This method is called when the members have run into the split-brain problem. This occurs when disconnected members that have created their own cluster are merged back into the main cluster. Services can also implement the `SplitBrainHandleService` to indicate that they can take part in the merge process. For the `CounterService`, we are going to implement as a no-op.

The next step is to enable the `CounterService`; in our case, we are going to configure that with `hazelcast.xml`.

```
<network>
   <join><multicast enabled="true"/> </join>
</network>
<services>
   <service enabled="true">
      <name>CounterService</name>
      <class-name>CounterService</class-name>
   </service>
</services>
```

You can see that two properties are set.

1. `name`: This needs to be a unique name because it will be used to look up the service when a remote call is made. In our case, we'll call it `CounterService`. Please realize that this name will be sent with every request, so the longer the name, the more data needs to be (de)serialized and sent over the line. Don't make it too long, but also don't reduce it to something that is not understandable.

2. `class`: Class of the service, in this case, `CounterService`. The class needs to have a no-arg constructor, otherwise the object can't be initialized.

We also enabled `multicast` discovery since we'll rely on that later.

You can also pass properties, which will be passed to the `init` method. You can do this using the following syntax:

```
<service enabled="true">
   <name>CounterService</name>
   <class-name>CounterService</class-name>
   <properties>
      <someproperty>10</someproperty>
   </properties>
</service>
```

If you want to parse more complex XML, you might want to have a look at the `com.hazelcast.spi.ServiceConfigurationParser` which will give you access to the XML DOM tree.

Of course, we want to see this in action.

```
public class Member {
   public static void main(String[] args) {
      Hazelcast.newHazelcastInstance();
   }
}
```

If we start it, we'll see the following output.

```
CounterService.init
```

The `CounterService` is started as part of the startup of the `HazelcastInstance`. If you shutdown the `HazelcastInstance`, for example, by using Control-C, then you will see:

```
CounterService.shutdown
```

12.2. Proxy

In the previous section, we created a `CounterService` that starts when Hazelcast starts, but apart from that it doesn't do anything yet. In this section, we connect the `Counter` interface to the `CounterService`, we do a remote call on the member hosting the eventual counter data/logic, and we return a dummy result. In Hazelcast, remoting is done through a Proxy: on the `client` side, you get a proxy which exposes your methods. When a method is called, the proxy creates an operation object, sends this operation to the machine responsible to execute that operation, and eventually sends the result.

First, we let the `Counter` implement the `DistributedObject` interface to indicate that it is a distributed object. Some additional methods will be exposed, such as `getName`, `getId`, and `destroy`.

```
public interface Counter extends DistributedObject {
    int inc(int amount);
}
```

The next step is enhancing the `CounterService`. Apart from implementing the `com.hazelcast.spi.ManagedService` interface, it now also implements the `com.hazelcast.spi.RemoteService` interface. Through this interface, a client can get a handle of a `Counter` proxy.

```
public class CounterService implements ManagedService, RemoteService {
   public static final String NAME = "CounterService";

   private NodeEngine nodeEngine;

   @Override
   public DistributedObject createDistributedObject(String objectName) {
      return new CounterProxy(objectName, nodeEngine, this);
   }

   @Override
   public void destroyDistributedObject(String objectName) {
      //no-op
   }

   ....
```

The methods of the `ManagedService` were left out, but you can find the full source in the examples for the book. The `createDistributedObject` returns a `CounterProxy`. This proxy is a local representation to (potentially) remote managed data and logic. It is important to realize that caching the proxy instance and removing the proxy instance is done outside of this service, so we don't need to take care of it ourselves.

The next part is the `CounterProxy` implementation.

```
public class CounterProxy
        extends AbstractDistributedObject<CounterService>
    implements Counter {

   private final String name;

   public CounterProxy(String name, NodeEngine ne, CounterService cs) {
      super(ne, cs);
      this.name = name;
   }

   @Override
   public String getServiceName() {
      return CounterService.NAME;
   }

   @Override
   public String getName() {
      return name;
   }

   @Override
   public int inc(int amount) {
      NodeEngine nodeEngine = getNodeEngine();
      IncOperation operation = new IncOperation(name, amount);
      int partitionId = nodeEngine.getPartitionService().getPartitionId(name);
      InvocationBuilder builder = nodeEngine.getOperationService()
             .createInvocationBuilder(CounterService.NAME, operation, partitionId);
      try {
         final Future<Integer> future = builder.invoke();
         return future.get();
      } catch (Exception e) {
         throw ExceptionUtil.rethrow(e);
      }
   }
}
```

The `CounterProxy` does not contain counter state; it is just a local representative of remote data/functionality. Therefore, the `CounterProxy.inc` method needs to be invoked on the machine for hosting the partition that contains the real counter. This can be done using the Hazelcast SPI, which takes care of sending operations to the correct machine, executing the operation, and returning the results.

If you take a closer look at the `inc` method, the first thing it does is create the `IncOperation` with the

given name and the amount. Next, it gets the `partitionId`; this is done based on the name so that all operations for a given name will always result in the same `partitionId`. Then, it creates an `InvocationBuilder` based on the operation and the `partitionId` using the `InvocationBuilder`. This is where the connection is made between the operation and the partition.

The last part is invoking the `Invocation` and waiting for its result. This is done using a `Future`, which gives us the ability to synchronize on completion of that remote executed operation and to get the results. In this case, we do a simple get since we don't care about a timeout; for real systems, it is often better to use a timeout since most operations should complete in a certain amount of time. If they don't complete, it could be a sign of problems; waiting indefinitely could lead to stalling systems without any form of error logging.

If the execution of the operation fails with an exception, an `ExecutionException` is thrown and needs to be dealt with. Hazelcast provides a utility function for that: `ExceptionUtil.rethrow(Throwable)`. If you want to keep the checked exception, you need to deal with exception handling yourself, and the `ExceptionUtil` is not of much use. A nifty improvement for debugging is that if a remote exception is thrown, the stacktrace includes the remote side and the local side. This makes it possible to figure out what went wrong on both sides of the call.

If the exception is an `InterruptedException`, you can do two things. Either propagate the `InterruptedException` since it is a good practice for blocking methods like shown below, or just use the `ExceptionUtil.rethrow` for all exceptions.

```
try {
   final Future<Integer> future = invocation.invoke();
   return future.get();
} catch(InterruptedException e){
   throw e;
} catch(Exception e){
   throw ExceptionUtil.rethrow(e);
}
```

In this case, we don't care about the `InterruptedException` and therefore we catch all exceptions and let them be handled by the `ExceptionUtil`: it will be wrapped in a `HazelcastException` and the interrupt status will be set.

Currently, it isn't possible to abort an operation by calling the `future.cancel` method. Perhaps this will be added in a later release. This is also the reason why Executor Futures are not working since the executor is built on top of the SPI.

Let's do the part of the example that has been missing so far: the `IncOperation`.

```
class IncOperation extends AbstractOperation
    implements PartitionAwareOperation {

    private String objectId;
    private int amount, returnValue;

    // Important to have a no-arg constructor for deserialization
    public IncOperation() {
    }
    public IncOperation(String objectId, int amount) {
        this.amount = amount;
        this.objectId = objectId;
    }
    @Override
    public void run() throws Exception {
        System.out.println("Executing " + objectId + ".inc() on: "
        + getNodeEngine().getThisAddress());
        returnValue = 0;
    }
    @Override
    public boolean returnsResponse() {
        return true;
    }
    @Override
    public Object getResponse() {
        return returnValue;
    }
    @Override
    protected void writeInternal(ObjectDataOutput out) throws IOException {
        super.writeInternal(out);
        out.writeUTF(objectId);
        out.writeInt(amount);
    }
    @Override
    protected void readInternal(ObjectDataInput in) throws IOException {
        super.readInternal(in);
        objectId = in.readUTF();
        amount = in.readInt();
    }
}
```

The first three methods —`run`, `returnsResponse` and `getResponse`— are part of the execution. The `run` method is responsible for the actual execution; in this case, it is an empty placeholder. Since the `inc` operation is going to return a response, the `returnsResponse` method returns true. If your method is asynchrononous and does not need to return a response, it is better to return false because that is faster. The actual response we stored in the `returnValue` field is retrieved using the `getResponse` method.

Another important part of the `IncOperation` is that it implements the `PartitionAwareOperation` interface. This is an indicator for the `OperationService` that this operation should be executed on a certain partition. In our case, the `IncOperation` should be executed on the partition hosting our counter.

Because the `IncOperation` needs to be serialized, the `writeInternal` and `readInternal` methods need to be overwritten so that the `objectId` and `amount` are serialized and will be available when this operation runs. For deserialization, it is also mandatory that the operation has a no-arg constructor.

Of course, we want to run the code.

```
public class Member {
   public static void main(String[] args) {
      HazelcastInstance[] instances = new HazelcastInstance[2];
      for (int k = 0; k < instances.length; k++)
         instances[k] = Hazelcast.newHazelcastInstance();

      Counter[] counters = new Counter[4];
      for (int k = 0; k < counters.length; k++)
      counters[k] = instances[0].getDistributedObject(CounterService.NAME, k+"counter");

      for (Counter counter: counters)
         System.out.println(counter.inc(1));

      System.out.println("Finished");
   }
}
```

In this example, we start a `HazelcastInstance`.

```
Executing 0counter.inc() on: Address[192.168.1.103]:5702
0
Executing 1counter.inc() on: Address[192.168.1.103]:5702
0
Executing 2counter.inc() on: Address[192.168.1.103]:5701
0
Executing 3counter.inc() on: Address[192.168.1.103]:5701
0
Finished
```

We can see that our counters are being stored in different members (check the different port numbers). We can also see that the increment does not do any real logic yet since the value remains at 0. We will solve this in the next section.

In this example we managed to get the basics up and running, but some things are not correctly implemented. For example, in the current code, a new proxy is always returned instead of a cached

one. Also, the `destroy` is not correctly implemented on the `CounterService`. In the following examples, these issues will be resolved.

12.3. Container

In this section, we upgrade the functionality so that it features a real distributed counter. Some kind of data structure will hold an integer value and can be incremented, and we will also cache the proxy instances and deal with proxy instance destruction.

The first thing we do is that for every partition in the system, we create a `Container` which will contain all counters and proxies for a given partition.

```
public static class Container {
   final Map<String, Integer> values = new HashMap<String, Integer>();

   private void init(String objectName) {
      values.put(objectName, 0);
   }

   private void destroy(String objectName){
      values.remove(objectName);
   }
}
```

Hazelcast guarantees that within a single partition, only a single thread will be active. So we don't need to deal with concurrency control while accessing a container.

The examples in this chapter rely on `Container` instance per partition, but you have complete freedom as to how to do that. A different approach used in Hazelcast is to drop the `Container` and let the `CounterService` have a map of counters.

```
final ConcurrentMap<String, Integer> counters =
   new ConcurrentHashMap<String, Integer>();
```

You can use the ID of the counter as key and an Integer as value. The only thing you need to take care of is that if operations for a specific partition are executed, you only select the values for that specific partition. This can be as simple as the following example.

```
for(Map.Entry<String,Integer> entry: counters.entrySet()){
   String id = entry.getKey();
   int partitinId = nodeEngine.getPartitionService().getPartitionId(id);
   if(partitionid == requiredPartitionId){
      ...do operation
   }
}
```

It is personal taste which solution you prefer. The container approach is nice because there will not be any mutable shared state between partitions. It also makes operations on partitions simpler, since you don't need to filter out data that does not belong to a certain partition.

The next step is to integrate the `Container` in the `CounterService`.

```
public class CounterService implements ManagedService, RemoteService {
    public final static String NAME = "CounterService";
    Container[] containers;
    private NodeEngine nodeEngine;

    @Override
    public void init(NodeEngine ne, Properties properties) {
        this.nodeEngine = nodeEngine;
        containers = new Container[ne.getPartitionService().getPartitionCount()];
        for (int k = 0; k < containers.length; k++)
            containers[k] = new Container();
    }

    @Override
    public CounterProxy createDistributedObject(String objectName) {
        int partitionId = nodeEngine.getPartitionService().getPartitionId(objectName);
        Container container = containers[partitionId];
        container.init(objectName);
        return new CounterProxy(objectName, nodeEngine, this);
    }

    @Override
    public void destroyDistributedObject(String objectName) {
        int partitionId = nodeEngine.getPartitionService().getPartitionId(objectName);
        Container container = containers[partitionId];
        container.destroy(objectName);
    }

    ...
```

In the `init` method, a container is created for every partition. The next step is the

`createDistributedObject` method; apart from creating the proxy, we also initialize the value for that given proxy to `0`, so that we don't run into a `NullPointerException`. In the `destroyDistributedObject` method, the value for the object is removed. If we don't clean up, we'll end up with memory that isn't removed and that can potentially can lead to an OOME.

The last step is connecting the `IncOperation.run` to the container.

```
class IncOperation extends AbstractOperation
    implements PartitionAwareOperation {

    ...

    @Override
    public void run() throws Exception {
    System.out.println("Executing " + objectId + ".inc() on: "
        + getNodeEngine().getThisAddress());

        CounterService service = getService();
        CounterService.Container container =
            service.containers[getPartitionId()];
        Map<String, Integer> valuesMap = container.values;
        Integer counter = valuesMap.get(objectId);
        counter += amount;
        valuesMap.put(objectId, counter);
        returnValue = counter;
    }

    ...
```

The container can easily be retrieved using the `partitionId`: the range of partition IDs is 0 to `partitionCount` (exclusive), so it can be used as an index on the container array. Once the container has been retrieved, we can access the value. This example moved all `inc` logic in the `IncOperation`, but you may prefer to move the logic to the `CounterService` or to the Partition.

Let's run the following example code.

```
public class Member {
    public static void main(String[] args) {
        HazelcastInstance[] instances = new HazelcastInstance[2];
        for (int k = 0; k < instances.length; k++)
            instances[k] = Hazelcast.newHazelcastInstance();

        Counter[] counters = new Counter[4];
        for (int k = 0; k < counters.length; k++)
            counters[k] = instances[0].getDistributedObject(
                CounterService.NAME, k+"counter");

        System.out.println("Round 1");
        for (Counter counter: counters)
            System.out.println(counter.inc(1));

        System.out.println("Round 2");
        for (Counter counter: counters)
            System.out.println(counter.inc(1));

        System.out.println("Finished");
        System.exit(0);
    }
}
```

It outputs the following.

```
Round 1
Executing 0counter.inc() on: Address[192.168.1.103]:5702
1
Executing 1counter.inc() on: Address[192.168.1.103]:5702
1
Executing 2counter.inc() on: Address[192.168.1.103]:5701
1
Executing 3counter.inc() on: Address[192.168.1.103]:5701
1
Round 2
Executing 0counter.inc() on: Address[192.168.1.103]:5702
2
Executing 1counter.inc() on: Address[192.168.1.103]:5702
2
Executing 2counter.inc() on: Address[192.168.1.103]:5701
2
Executing 3counter.inc() on: Address[192.168.1.103]:5701
2
Finished
```

This means that we now have a basic distributed counter up and running!

12.4. Partition Migration

In our previous example, we managed to create real distributed counters. The only problem is that when members leave or join the cluster, the content of the partition containers does not migrate to different members. In this section, we are going to do that with partition migration.

The first thing is to add 3 new operations to the `Container`.

```
class Container {
   void clear() {
      values.clear();
   }

   void applyMigrationData(Map<String, Integer> migrationData) {
      values.putAll(migrationData);
   }

   Map<String, Integer> toMigrationData() {
      return new HashMap(values);
   }
   ...
}
```

1. `toMigrationData`: This method is called when Hazelcast wants to start the migration of the partition on the member that currently owns the partition. The result of the `toMigrationData` is partition data in a form that can be serialized to another member.

2. `applyMigrationData`: This method is called when the `migrationData` that is created by the `toMigrationData` method is applied to a member that is going to be the new partition owner.

3. `clear`: This method is called for two reasons. One reason is when the partition migration has succeeded and the old partition owner can get rid of all the data in the partition. The other reason is when the partition migration operation fails and the `new` partition owner needs to roll back its changes.

The next step is to create a `CounterMigrationOperation` that will be responsible for transferring the `migrationData` from one machine to anther and to call the `applyMigrationData` on the correct partition of the new partition owner.

```
public class CounterMigrationOperation extends AbstractOperation {

   Map<String, Integer> migrationData;

   public CounterMigrationOperation() {
   }

   public CounterMigrationOperation(Map<String, Integer> m) {
      this.migrationData = m;
   }

   @Override
   public void run() throws Exception {
      CounterService service = getService();
      Container container = service.containers[getPartitionId()];
      container.applyMigrationData(migrationData);
   }

   @Override
   protected void writeInternal(ObjectDataOutput out) throws IOException {
      out.writeInt(migrationData.size());
      for (Map.Entry<String, Integer> entry : migrationData.entrySet()) {
         out.writeUTF(entry.getKey());
         out.writeInt(entry.getValue());
      }
   }

   @Override
   protected void readInternal(ObjectDataInput in) throws IOException {
      int size = in.readInt();
      migrationData = new HashMap<String, Integer>();
      for (int i = 0; i < size; i++)
         migrationData.put(in.readUTF(), in.readInt());
   }
}
```

During the execution of a migration, no other operations will be running in that partition. Therefore, you don't need to deal with thread-safety.

The last part is connecting all the pieces. This is done by implementing `MigrationAwareService` as an additional interface on the `CounterService` which will signal Hazelcast that our service is able to participate in partition migration.

```
public class CounterService implements ManagedService,
   RemoteService, MigrationAwareService {

   ...

   @Override
   public void beforeMigration(PartitionMigrationEvent e) {
      //no-op
   }

   @Override
   public void clearPartitionReplica(int partitionId) {
      Container container = containers[partitionId];
      container.clear();
   }

   @Override
   public Operation prepareReplicationOperation(
      PartitionReplicationEvent e) {
      Container container = containers[e.getPartitionId()];
      Map<String, Integer> data = container.toMigrationData();
      return data.isEmpty() ? null : new CounterMigrationOperation(data);
   }

   @Override
   public void commitMigration(PartitionMigrationEvent e) {
      if (e.getMigrationEndpoint() == MigrationEndpoint.SOURCE) {
         Container c = containers[e.getPartitionId()];
         c.clear();
      }
   }

    @Override
    public void rollbackMigration(PartitionMigrationEvent e) {
       if (e.getMigrationEndpoint() == MigrationEndpoint.DESTINATION) {
          Container c = containers[e.getPartitionId()];
          c.clear();
       }
    }
}
```

By implementing the `MigrationAwareService`, some additional methods are exposed.

- `beforeMigration`: This method is called before any migration is done.
- `prepareMigrationOperation`: This method returns all the data in the partition that is going to be

moved. It migrates only the master and the first backup data, since `CounterService` supports only one backup.

- `commitMigration`: This method commits the migrated data. In this case, committing means that we clear the container for the partition of the old owner. Even though we don't have any complex resources like threads, database connections, etc., clearing the container is advisable to prevent memory issues. This method is called on both the primary and the backup. If this node is on the source side of migration (partition is migrating FROM this node) and the migration type is MOVE (partition is migrated completely, not copied to a backup node), then the method removes partition data from this node. If this node is the destination or the migration type is copy, then the method does nothing. If this node is on the destination side of migration (partition is migrating TO this node) then this method removes partition data from this node. If this node is on the source, then the methods does nothing.

- `rollbackMigration`: Rolls back a migration.

```
public class Member {
   public static void main(String[] args) throws Exception {
      HazelcastInstance[] instances = new HazelcastInstance[3];
      for (int k = 0; k < instances.length; k++)
         instances[k] = Hazelcast.newHazelcastInstance();

      Counter[] counters = new Counter[4];
      for (int k = 0; k < counters.length; k++)
         counters[k] = instances[0].getDistributedObject(
            CounterService.NAME, k + "counter");

      for (Counter counter : counters)
         System.out.println(counter.inc(1));

      Thread.sleep(10000);

      System.out.println("Creating new members");

      for (int k = 0; k < 3; k++)
         Hazelcast.newHazelcastInstance();

      Thread.sleep(10000);

      for (Counter counter : counters)
         System.out.println(counter.inc(1));

      System.out.println("Finished");
      System.exit(0);
   }
}
```

If we execute the example, we'll get output like this:

```
Executing 0counter.inc() on: Address[192.168.1.103]:5702
Executing backup 0counter.inc() on: Address[192.168.1.103]:5703
1
Executing 1counter.inc() on: Address[192.168.1.103]:5703
Executing backup 1counter.inc() on: Address[192.168.1.103]:5701
1
Executing 2counter.inc() on: Address[192.168.1.103]:5701
Executing backup 2counter.inc() on: Address[192.168.1.103]:5703
1
Executing 3counter.inc() on: Address[192.168.1.103]:5701
Executing backup 3counter.inc() on: Address[192.168.1.103]:5703
1
Creating new members
Executing 0counter.inc() on: Address[192.168.1.103]:5705
Executing backup 0counter.inc() on: Address[192.168.1.103]:5703
2
Executing 1counter.inc() on: Address[192.168.1.103]:5703
Executing backup 1counter.inc() on: Address[192.168.1.103]:5704
2
Executing 2counter.inc() on: Address[192.168.1.103]:5705
Executing backup 2counter.inc() on: Address[192.168.1.103]:5704
2
Executing 3counter.inc() on: Address[192.168.1.103]:5704
Executing backup 3counter.inc() on: Address[192.168.1.103]:5705
2
Finished
```

The counters have moved: `0counter` moved from `192.168.1.103:5702` to `192.168.1.103:5705`, but it is incremented correctly. Our counters can now move around in the cluster. If you need to have more capacity, add a machine and the counters will be redistributed. If you have surplus capacity, shut down the instance and the counters will be redistributed.

12.5. Backups

In this last section, we deal with backups; we make sure that when a member fails, then the data of the counter is available on another node. This is done by replicating that change to another member in the cluster. With the SPI, you can do this by letting the operation implement the `com.hazelcast.spi.BackupAwareOperaton` interface. Below, you can see this interface being implemented on the `IncOperation`.

```
class IncOperation extends AbstractOperation
    implements PartitionAwareOperation, BackupAwareOperation {
   ...

   @Override
   public int getAsyncBackupCount() {
      return 0;
   }

   @Override
   public int getSyncBackupCount() {
      return 1;
   }

   @Override
   public boolean shouldBackup() {
      return true;
   }

   @Override
   public Operation getBackupOperation() {
      return new IncBackupOperation(objectId, amount);
   }
}
```

Some additional methods need to be implemented. The `getAsyncBackupCount` and `getSyncBackupCount` methods signal how many asynchronous and synchronous backups are needed. In our case, we only want a single synchronous backup and no asynchronous backups. Here, the number of backups is hard coded, but you could also pass the number of backups as parameters to the `IncOperation`, or you could let the methods access the `CounterService`. The `shouldBackup` method tells Hazelcast that our `Operation` needs a backup. In our case, we are always going to make a backup, even if there is no change. In practice, you only want to make a backup if there is actually a change: in case of the `IncOperation`, you want to make a backup if `amount` is null.

The last method is the `getBackupOperation`, which returns the actual operation that is going to make the backup; the backup itself is an operation and will run on the same infrastructure. If a backup should be made and, for example, `getSyncBackupCount` returns 3, then three `IncBackupOperation` instances are created and sent to the three machines containing the backup partition. If there are fewer machines available than backups that need to be created, Hazelcast will just send a smaller number of operations. But it could be that if the cluster is too small, you don't get the same high availability guarantees you specified.

Let's have a look at the `IncBackupOperation`.

```
public class IncBackupOperation
    extends AbstractOperation implements BackupOperation {
   private String objectId;
   private int amount;

   public IncBackupOperation() {
   }

   public IncBackupOperation(String objectId, int amount) {
      this.amount = amount;
      this.objectId = objectId;
   }

   @Override
   protected void writeInternal(ObjectDataOutput out) throws IOException {
      super.writeInternal(out);
      out.writeUTF(objectId);
      out.writeInt(amount);
   }

   @Override
   protected void readInternal(ObjectDataInput in) throws IOException {
      super.readInternal(in);
      objectId = in.readUTF();
      amount = in.readInt();
   }

   @Override
   public void run() throws Exception {
      CounterService service = getService();
      System.out.println("Executing backup " + objectId + ".inc() on: "
        + getNodeEngine().getThisAddress());
      Container c = service.containers[getPartitionId()];
      c.inc(objectId, amount);
   }
}
```

Hazelcast will also make sure that a new `IncOperation` for that particular key will not be executing before the (synchronous) backup operation has completed.

We want to see the backup functionality in action.

```
public class Member {
   public static void main(String[] args) throws Exception {
      HazelcastInstance[] instances = new HazelcastInstance[2];
      for (int k = 0; k < instances.length; k++)
         instances[k] = Hazelcast.newHazelcastInstance();

      Counter counter = instances[0].getDistributedObject(CounterService.NAME,
"counter");
      counter.inc(1);
      System.out.println("Finished");
      System.exit(0);
    }
}
```

When we run this `Member`, we'll get the following output.

```
Executing counter0.inc() on: Address[192.168.1.103]:5702
Executing backup counter0.inc() on: Address[192.168.1.103]:5701
Finished
```

The `IncOperation` has executed, and the backup operation has executed. The operations have been executed on different cluster members to guarantee high availability. One of the experiments you could do is to modify the test code so you have a cluster of members in different JVMs and see what happens with the counters when you kill one of the JVMs.

12.6. Good to know

Don't hog the operation thread: Don't execute long-running operations on the operation thread, especially not on partition specific operation threads. This could cause major problems in the system because the partition (and other partitions running on that same thread) will wait for your operation to complete. Operations should be very short.

12.7. What is next

In this chapter we step-by-step created a distributed data structure on top of Hazelcast. Although Hazelcast provides a whole collection of very usable distributed data structures, the addition of the SPI changes Hazelcast from being a simple data grid to a data grid infrastructure where your imagination is the only limit. In distributed applications, there will always be the need to create special data structures that are useful for particular use cases. With the SPI, you can now build these data structures yourself.

Chapter 13. Threading Model

This chapter discusses the threading model of Hazelcast. This will help you understand how to write an efficient system and how to not cause cluster stability issues.

13.1. I/O Threading

Hazelcast uses a pool of threads for I/O: a single thread does not perform all the I/O, multiple threads do. On each cluster member, the I/O threading is split up into three types of I/O-threads.

1. accept-I/O-threads: These threads accept requests.
2. read-I/O-threads: These threads read data from other members/clients.
3. write-I/O-threads: These threads write data to other members/clients.

You can configure the number of I/O-threads using the `hazelcast.io.thread.count` system property, which defaults to 3 per member. This means that if 3 is used, in total there are 7 I/O-threads: 1 accept-I/O-thread, 3 read-I/O-threads and 3 write-I/O-threads. Each I/O-thread has its own `Selector` instance and waits on `Selector.select` if there is nothing to do.

In case of the I/O-read-thread, when sufficient bytes for a packet have been received, the Packet object is created. This Packet is then sent to the System where it is de-multiplexed. If the Packet header signals that it is an operation/response, it is handed over to the operation service (please see Operation Threading). If the Packet is an event, it is handed over to the event service (please see Event Threading).

If the packet is a request (an operation sent by a client), the I/O-thread reads the partition ID. If set, the packet is placed on the correct partition-aware operation thread; it does not matter if the partition is on the member or not. If the partition is on the member, the operation is executed by the partition-aware operation thread. Otherwise, the partition-aware operation thread sends the operation to the correct machine by handing it over to the correct I/O-thread. The partition-aware operation thread is immediately released for the next operation; the response is returned to the client using an asynchronous callback mechanism.

If the packet is a request and the partition ID is not set, the request is put on the executor in the `ClientEngineImpl`. This executor can be configured using the `hazelcast.clientengine.thread.count` property, which defaults to the number of cores times 20.

13.2. Event Threading

Hazelcast uses a shared event system to deal with components that rely on events like topic, collections listeners, and near cache. Each cluster member has an array of event threads and each thread has its own work queue (a regular `BlockingQueue` implementation). When an event is produced, either locally or remote, an event thread is selected (depending on if there is a message ordering) and the event is

placed in the work queue for that event thread.

You can set the following properties to alter the behavior of the system.

1. `hazelcast.event.thread.count`: Number of event threads in this array. Its default value is 5.
2. `hazelcast.event.queue.capacity`: Capacity of the work queue. Its default value is 1000000.
3. `hazelcast.event.queue.timeout.millis`: Timeout for placing an item on the work queue. Its default value is 250.

If you process a lot of events and you have many cores, changing the value of `hazelcast.event.thread.count` property to a higher value is a good idea. This way, more events can be processed in parallel.

Multiple components share the same event queues. If there are 2 topics, say A and B, for certain messages they may share the same queue(s) and hence the same event thread. If there are a lot of pending messages produced by A, then B needs to wait. Also, when processing a message from A takes a long time and the event thread is used for that, B will suffer from this. That is why it is better to offload processing to a dedicated thread (pool) so that systems are better isolated.

If events are produced at a higher rate than they are consumed, the queue will grow in size. To prevent overloading the system and running into an `OutOfMemoryException`, the queue is given a capacity of 1 million items. When the maximum capacity is reached, the items are dropped. This means that the event system is a "best effort" system. There is no guarantee that you are going to get an event. It can also be that Topic A has a lot of pending messages, and therefore B cannot receive messages because the queue has no capacity and messages for B are dropped. Another reason events are not reliable is that the JVM of the receiver crashes, all the messages in the event- queues will be lost.

13.3. IExecutor Threading

Executor threading is straightforward. When a task is received to be executed on Executor E, then E will have its own `ThreadPoolExecutor` instance and the work is put on the work queue of this executor. Executors are fully isolated, but they will share the same underlying hardware: most importantly, the CPUs.

You can configure the `IExecutor` using the `ExecutorConfig` (programmatic configuration) or using `<executor>` (declarative configuration).

13.4. Operation Threading

There are two types of threading operations.

1. Operations that are aware of a certain partition, such as `IMap.get(key)`
2. Operations that are not partition aware, such as the

`IExecutorService.executeOnMember(command,member)` operation.

Each of these types has a different threading model that is explained below.

13.4.1. Partition-aware Operations

To execute partition-aware operations, an array of operation threads is created. By default, the size of this array is 2 times the number of cores with a minimum of 2. You can change it using the `hazelcast.operation.thread.count` property.

Each operation thread has its own work queue; it will consume messages from that work queue. If a partition-aware operation needs to be scheduled, the right thread is found using the formula below.

```
threadIndex = partitionId % partition-thread-count
```

After the `threadIndex` is determined, the operation is put in the work queue of that operation thread. This means three things.

1. A single operation thread executes operations for multiple partitions; if there are 271 partitions and 10 partition threads, then roughly every operation thread will execute operations for 27 partitions.

2. Each partition belongs to only 1 operation thread. All operations for a partition will always be handled by exactly the same operation thread.

3. No concurrency control is needed to deal with partition-aware operations because once a partition-aware operation is put on the work queue of a partition-aware operation thread, you get the guarantee that only 1 thread is able to touch that partition.

Because of this threading strategy, there are two forms of false sharing you need to be aware of:

1. With false sharing of the partition, two completely independent data structures share the same partitions: for example, if there is a map `employees` and a map `orders`, it could be that an `employees.get(peter)` (running on partition 25) is blocked by a `map.get` of `orders.get(1234)` (also running on partition 25). So, if independent data structures share the same partition, a slow operation on one data structure can slow the other data structures.

2. With false sharing of the partition-aware operation thread, each operation thread is responsible for executing operations on a number of partitions. For example, thread-1 could be responsible for partitions 0,10,20..., and thread-2 could be responsible for partitions 1,11,21...,etc. If an operation for partition 1 is taking a lot of time, it will block the execution of an operation of partition 11 because both of them are mapped to exactly the same operation thread.

You need to be careful with long running operations because you could be starving operations of a thread. The general rule is that the partition thread should be released as soon as possible because operations are not designed to execute long running operations. That is why it is very dangerous to execute a long running operation —for example, using `AtomicReference.alter` or a `IMap.executeOnKey`— because these operations will block others operations to be executed.

Currently, there is no support for work stealing. Different partitions that map to the same thread may need to wait till one of the partitions is finished, even though there are other free partition operation threads available.

Example:

Take a 3 node cluster. Two members will have 90 primary partitions and one member will have 91 primary partitions. Let's say you have one CPU and 4 cores per CPU. By default, 8 operation threads will be allocated to serve 90 or 91 partitions.

13.4.2. Non-Partition-Aware Operations

To execute non-partition-aware operations, such as `IExecutorService.executeOnMember(command,member)`, generic operation threads are used. When the Hazelcast instance is started, an array of operation threads is created. The size of this array defaults to the number of cores divided by 2, with a minimum of 2. It can be changed using the `hazelcast.operation.generic.thread.count` property.

A non-partition-aware operation thread will never execute an operation for a specific partition. Only partition-aware operation threads execute partition-aware operations.

Unlike the partition-aware operation threads, all the generic operation threads share the same work queue: `genericWorkQueue`.

If a non-partition-aware operation needs to be executed, it is placed in that work queue and any generic operation thread can execute it. The big advantage is that you automatically have work balancing since any generic operation thread is allowed to pick up work from this queue.

The disadvantage is that this shared queue can be a point of contention. We do not practically see this in production because performance is dominated by I/O and the system is not executing very many non-partition-aware operations.

13.4.3. Priority Operations

In some cases, the system needs to execute operations with a higher priority, such as an important system operation. To support priority operations, we do the following:

1. For partition-aware operations, each partition thread has its own work queue. But apart from that, it also has a priority work queue. It will always check this priority queue before it processes work from its normal work queue.

2. For non-partition-aware operations, next to the `genericWorkQueue`, there also is a `genericPriorityWorkQueue`. When a priority operation needs to be executed, it is put in this `genericPriorityWorkQueue`. And just like the partition-aware operation threads, a generic operation thread will first check the `genericPriorityWorkQueue` for work.

Because a worker thread will block on the normal work queue (either partition specific or generic), a priority operation may not picked up because it will not be put on the queue where it is blocking. We

always send a `kick the worker` operation that does nothing but trigger the worker to wake up and check the priority queue.

13.5. Operation-response and Invocation-future

When an operation is invoked, a `Future` is returned. Let's take the example code below.

```
GetOperation operation = new GetOperation( mapName, key )
Future future = operationService.invoke( operation )
future.get)
```

The calling side blocks for a reply. In this case, `GetOperation` is set in the work queue for the partition of `key`, where it eventually is executed. On execution, a response is returned and placed on the work queue of the response thread. This thread will signal the `future` and notifies the blocked thread that a response is available. In the future, we will expose this `Future` to the outside world, and we will provide the ability to register a completion listener so you can do asynchronous calls.

13.6. Local Calls

When a local partition-aware call is done, an operation is made and handed over to the work queue of the correct partition operation thread, and a future is returned. When the calling thread calls `get` on that future, it will acquire a lock and wait for the result to become available. When a response is calculated, the future is looked up, and the waiting thread is notified.

In the future, this will be optimized to reduce the amount of expensive systems calls, such as `lock.acquire`/`notify` and the expensive interaction with the operation queue. Probably, we will add support for a caller-runs mode, so that an operation is directly executed on the calling thread.

13.7. Queries

Unlike regular operations such as `IMap.get`, which run on operation threads, there is also a group of operations that do not run on partition threads, but run on query threads: for example, the Collection<V> `IMap.values(Predicate predicate)` method. This means there is a separate thread pool that executes queries so queries can be run in parallel with regular operations. You can influence the size of the query thread pool by creating an `ExecutorConfig` (either programmatically or through XML) for the `hz:query` executor.

13.8. Map Loader

There are two methods of the `MapLoader` interface that are offloaded to different thread pools. One method is the `MapLoader.loadAllKeys()`, which loads all keys from the database that should be put in the `IMap`. Once this phase is completed, the keys are assigned to their partition for each partition upon which the `MapLoader.loadAll(keys)` is called. Therefore, loading of the actual data can be done in

parallel. The `MapLoader.loadAll` is executed on an executor with the name `hz:map-loadAllKeys` and the `MapLoader.loadAll(keys)` is executed on an executor with the name `hz:map-load`. The second one is interesting because if you want to load more data in parallel, this is the executor to tweak. To configure this executor, you can create an executor with the name `hz:map-load` and fine tune, for example, the thread pool size.

Unlike regular partition operations, the `MapLoader.loadAll(keys)` is not executed on a partition thread. This means that regular operations for that partition that access different data structures, instead of being blocked, can still be executed.

Chapter 14. Performance Tips

14.1. Cluster Design

Hazelcast assumes that the cluster is homogeneous: all nodes within a cluster have equal memory, equal storage capacity, equal network bandwidth, etc. It will not look at the resources of a member to determine which partitions can be moved in or out. In practice, this means that if your cluster is not homogeneous, then a lite member with few resources is responsible for the same load as a heavyweight member with many resources. If this is an issue, it is probably best to set up multiple clusters and to use clients to let clusters communicate with each other. There are plans to support heterogeneous members in the cluster, for example, by making use of sub-clusters, but this isn't concrete yet.

14.2. Map Performance Tips

Here are a few tips to improve map performance.

1. Try for zero or minimum backup count.
2. Use asynchronous backup instead of synchronous backup.
3. Use `IMap.set(k, v)` instead of `put(k,v)` so that the old value doesn't have to be returned (and thus, no deserialization).
4. Avoid locks and transactions.
5. If you really need `MapStore`, then consider having write-behind; write-through will slow you down.

14.3. Local stats

Some of the distributed objects, like the `IMap`, `IQueue` and `ITopic`, have statistics that can be accessed as in the following example.

```
IMap map = hz.getMap();
LocalMapStats stats = map.getLocalMapStats()
```

You can use these statistics to access all kinds of metrics. For example, in the case of `IMap`, you can see the number of map entries in that member, and the total number of `put` operations executed.

14.4. JMX Monitoring

Add the following system properties to enable JMX agent.

`-Dcom.sun.management.jmxremote`

`-Dcom.sun.management.jmxremote.port=portNo` (to specify jmx port) (optional)

`-Dcom.sun.management.jmxremote.authenticate=false` (to disable jmx auth) (optional)

Enable the Hazelcast property `hazelcast.jmx` using Hazelcast configuration (API, XML, Spring) or by setting the system property `-Dhazelcast.jmx=true`.

Use `jconsole`, `jvisualvm` (with the MBean plugin) or another JMX compliant monitoring tool.

JMX also exposes the local stats.

14.5. Other

The `SystemLogServices` retains some logging information that you can use with the Management Center. If you are not using it, you can disable it by setting `hazelcast.system.log.enabled` to false.

Chapter 15. Appendix

15.1. Hazelcast on EC2 Tutorial

This is a step-by-step tutorial on how to configure a very basic Hazelcast cluster inside Amazon EC2. This tutorial expects that you have ssh available (on Windows you can use Putty) and that you have an Amazon EC2 account. You also need to have your Amazon access key and secret key. The access key should look similar to this:

BNELSKS9B8LDLE8NXKKA

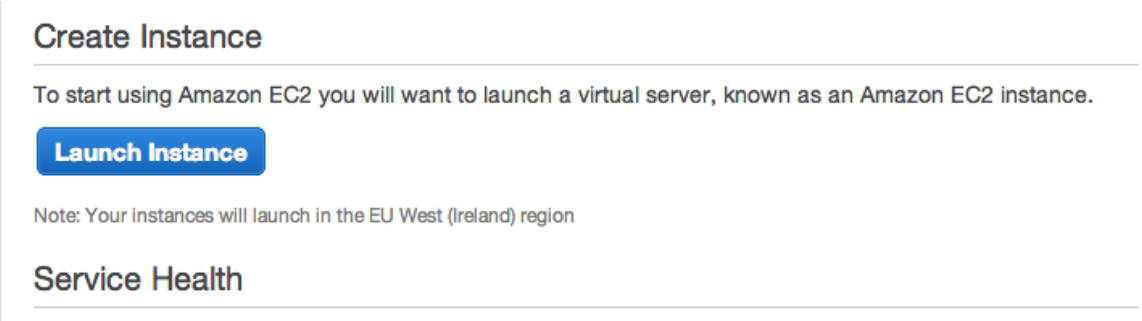
And your secret key should look similar to this:

384c32KDLLDM44l3l3lddudueIEEL/Uldlx395uM

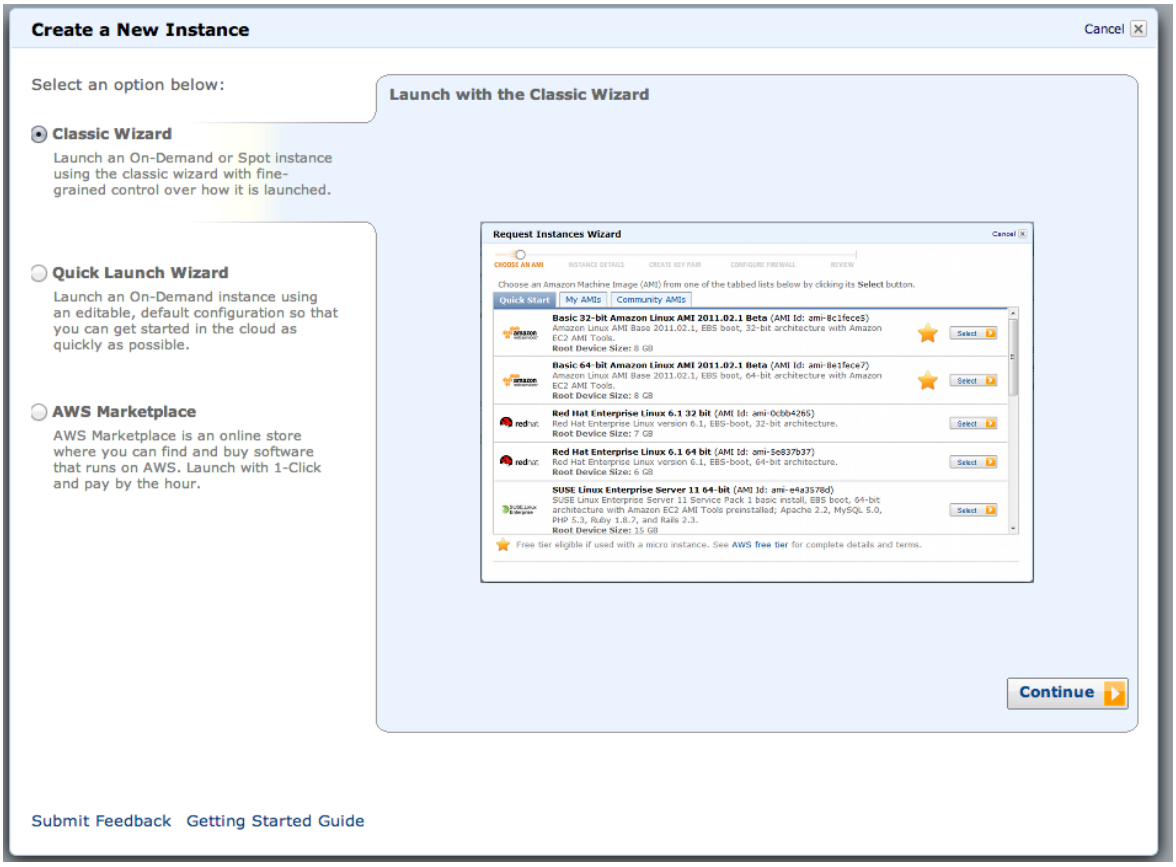
First, we login to the management console. For example:

https://console.aws.amazon.com/ec2/v2/home?region=eu-west-1.

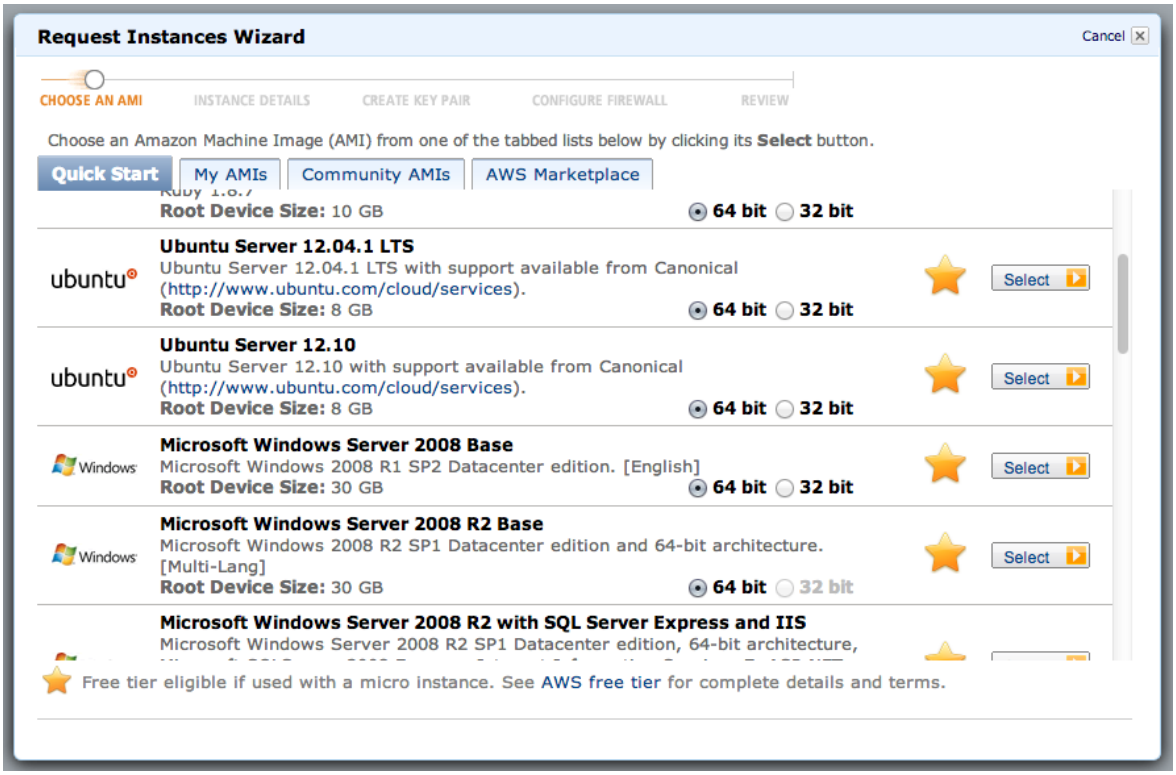
There, we press the "Launch Instance" button.



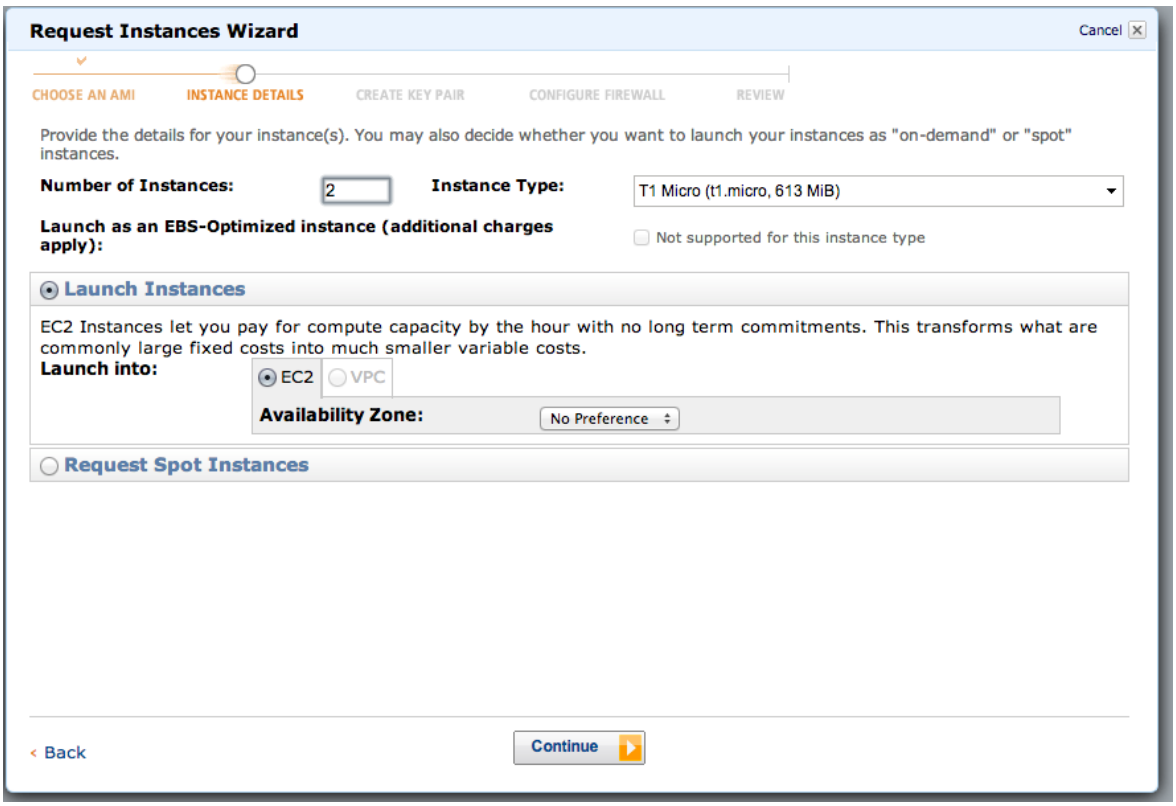
Select "Classic Wizard".

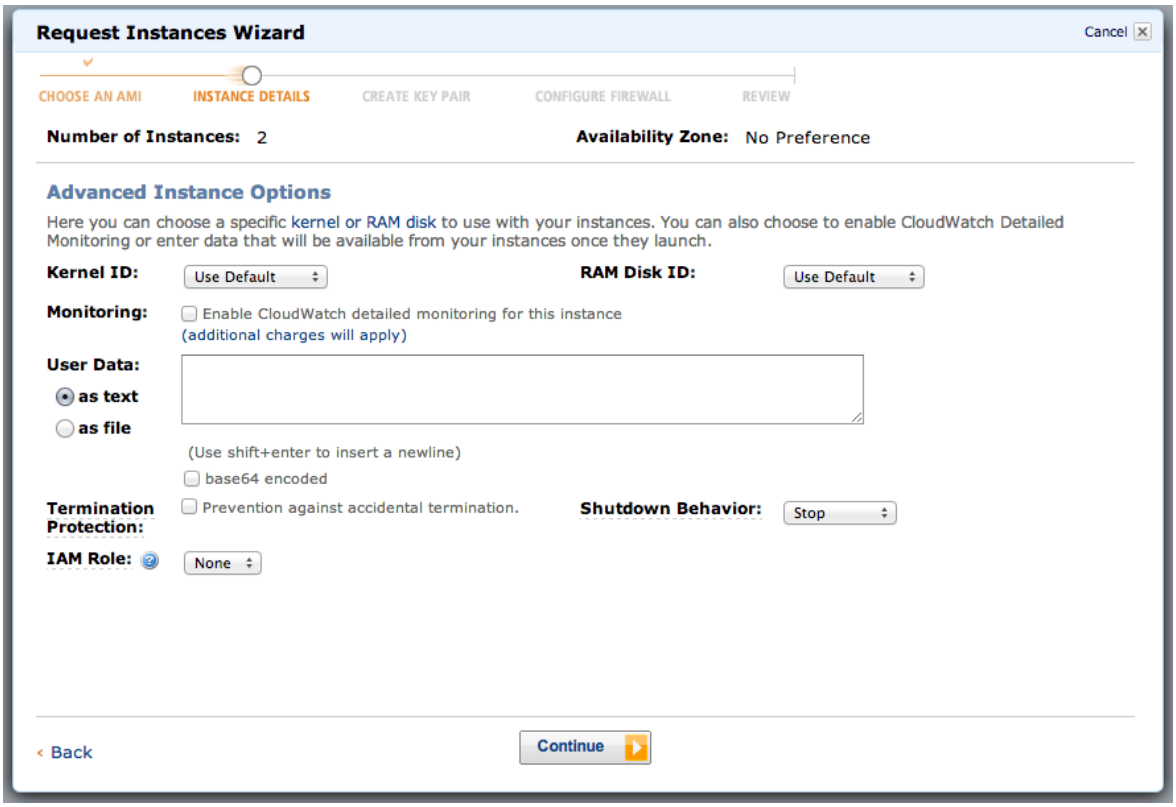


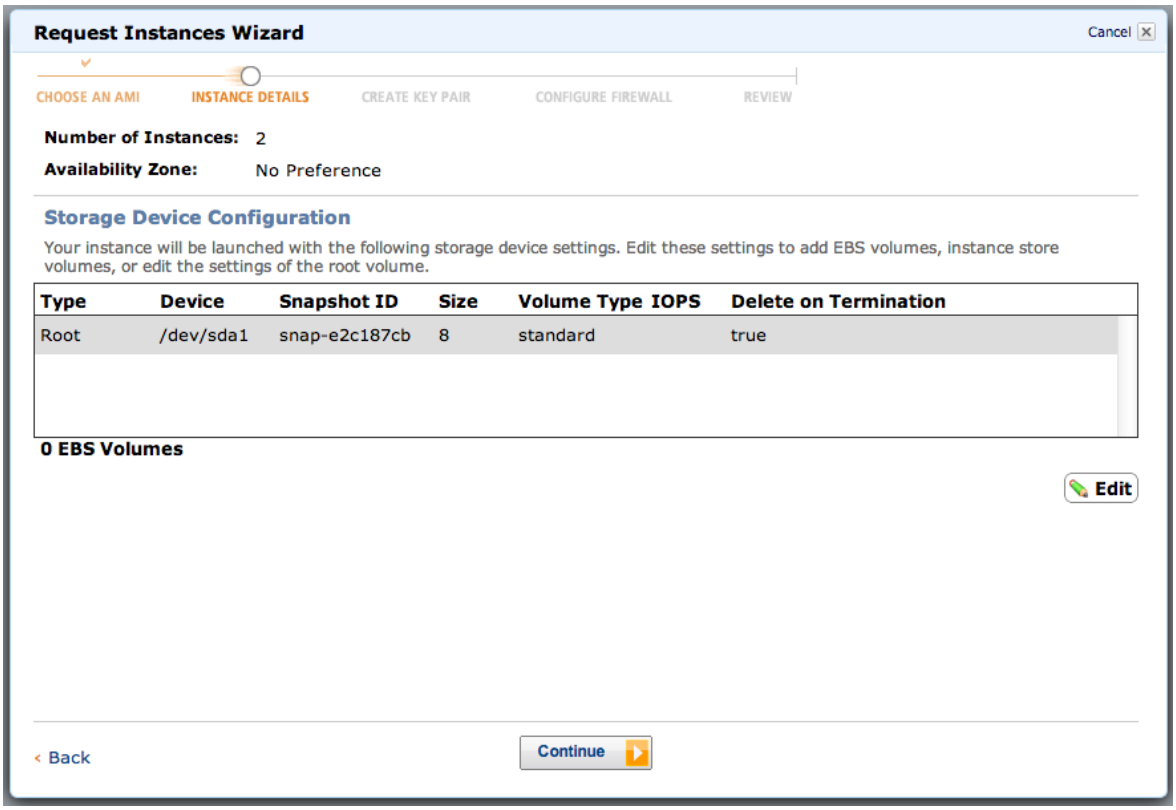
Select "Ubuntu Server 12.10".

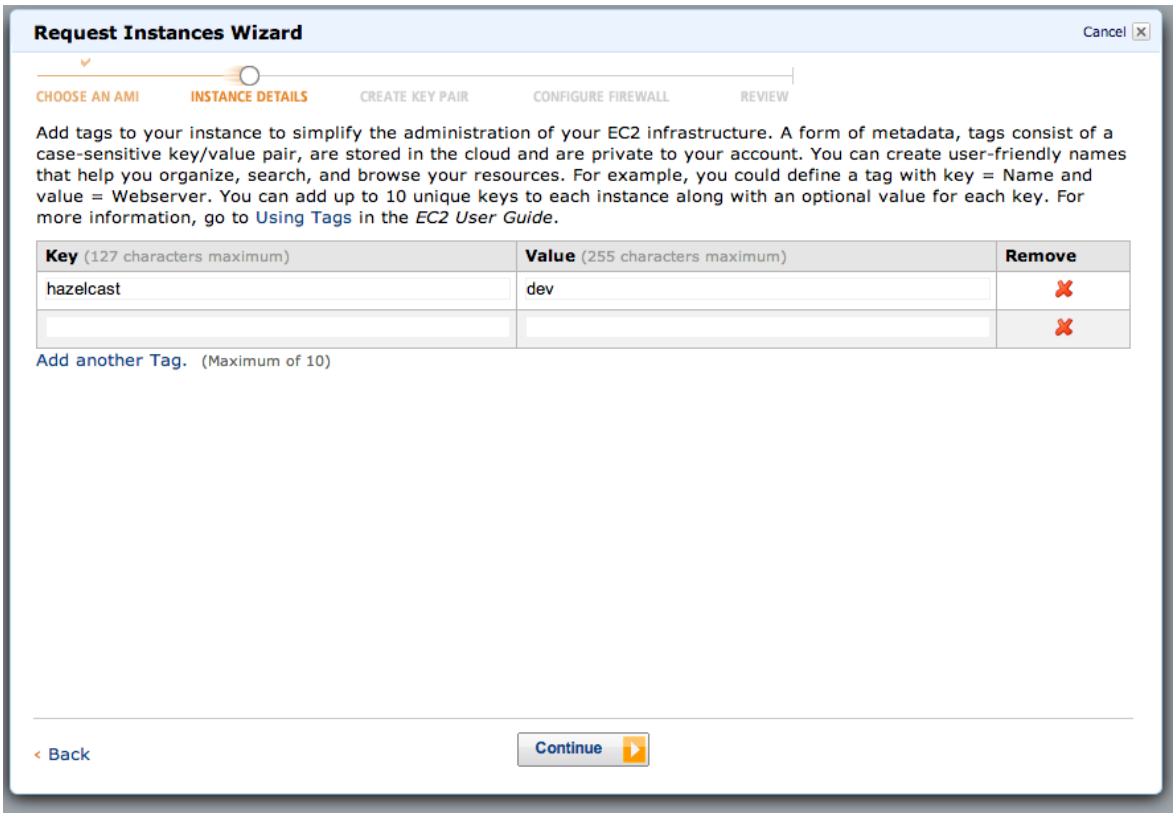


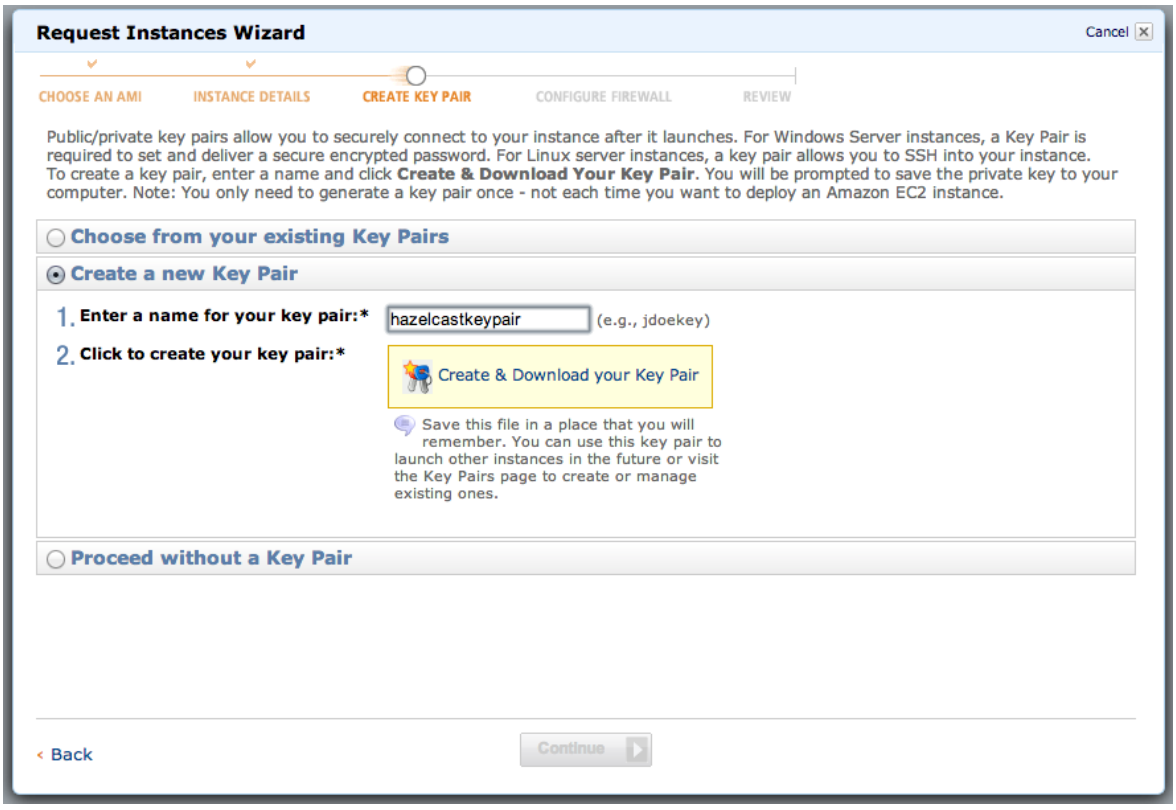
We are going to create two ubuntu servers at once. Enter 2 for "Number of instances" and press the "Continue" button.

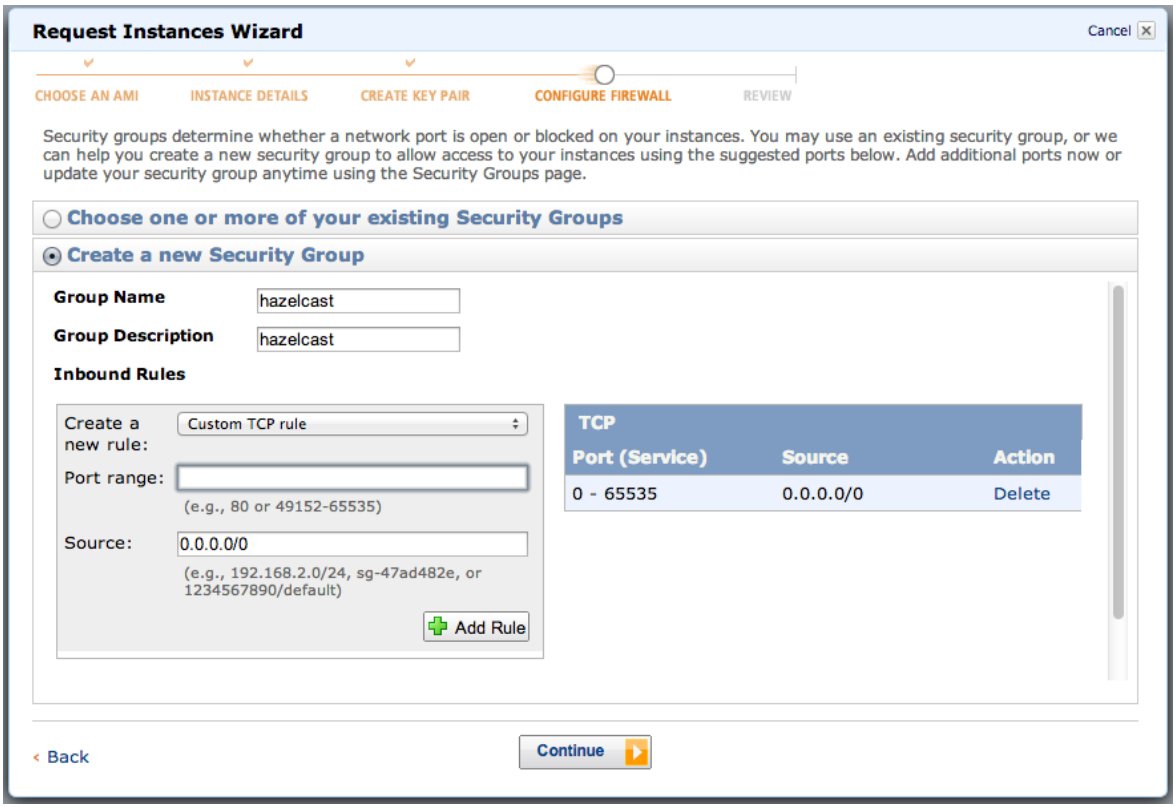


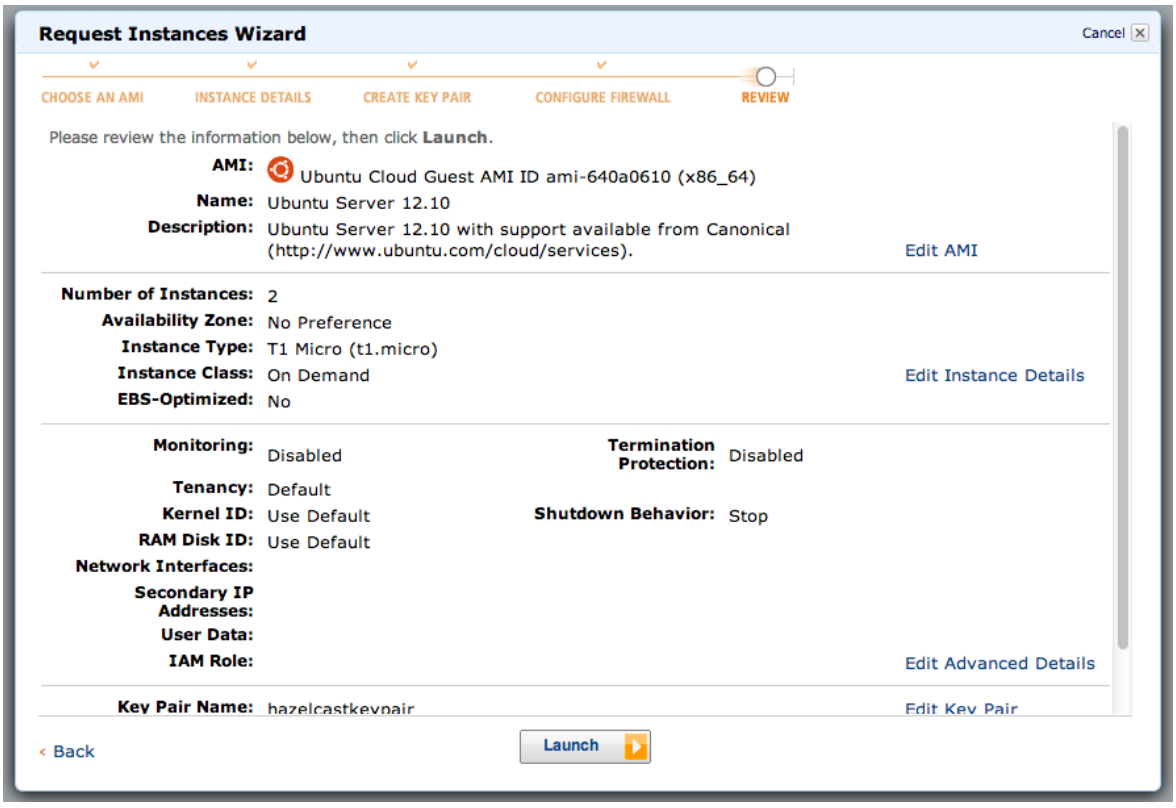


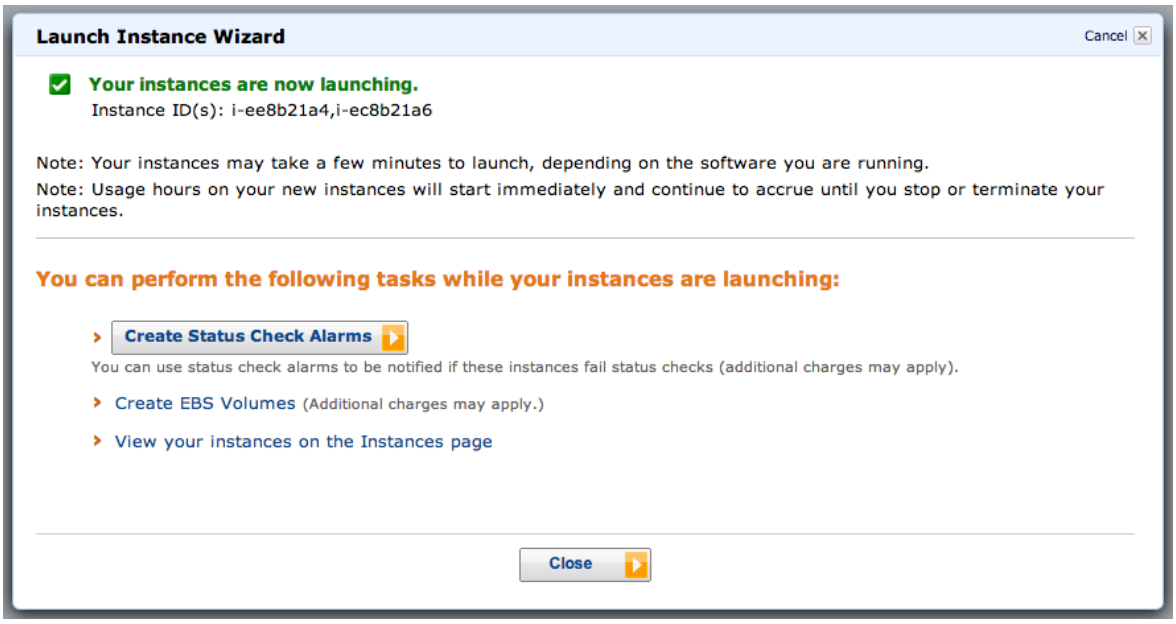

Request Instances Wizard
Cancel
CHOOSE AN AMI
INSTANCE DETAILS
CREATE KEY PAIR
CONFIGURE FIREWALL
REVIEW
Number of Instances: 2
Availability Zone: No Preference
Storage Device Configuration
Your instance will be launched with the following storage device settings. Edit these settings to add EBS volumes, instance store volumes, or edit the settings of the root volume.
Type Device Snapshot ID Size Volume Type IOPS Delete on Termination
Root /dev/sda1 snap-e2c187cb 8 standard true
0 EBS Volumes
Edit
Back
Continue


Request Instances Wizard
Cancel
CHOOSE AN AMI
INSTANCE DETAILS
CREATE KEY PAIR
CONFIGURE FIREWALL
REVIEW
Add tags to your instance to simplify the administration of your EC2 infrastructure. A form of metadata, tags consist of a case-sensitive key/value pair, are stored in the cloud and are private to your account. You can create user-friendly names that help you organize, search, and browse your resources. For example, you could define a tag with key = Name and value = Webserver. You can add up to 10 unique keys to each instance along with an optional value for each key. For more information, go to Using Tags in the EC2 User Guide.
Key (127 characters maximum)
Value (255 characters maximum)
Remove
hazelcast
dev
Add another Tag. (Maximum of 10)
Back
Continue


Request Instances Wizard
Cancel
CHOOSE AN AMI
INSTANCE DETAILS
CREATE KEY PAIR
CONFIGURE FIREWALL
REVIEW
Public/private key pairs allow you to securely connect to your instance after it launches. For Windows Server instances, a Key Pair is required to set and deliver a secure encrypted password. For Linux server instances, a key pair allows you to SSH into your instance. To create a key pair, enter a name and click Create & Download Your Key Pair. You will be prompted to save the private key to your computer. Note: You only need to generate a key pair once - not each time you want to deploy an Amazon EC2 instance.
Choose from your existing Key Pairs
Create a new Key Pair
1. Enter a name for your key pair:*
hazelcastkeypair
(e.g., jdoekey)
2. Click to create your key pair:*
Create & Download your Key Pair
Save this file in a place that you will remember. You can use this key pair to launch other instances in the future or visit the Key Pairs page to create or manage existing ones.
Proceed without a Key Pair
Back
Continue


Request Instances Wizard
Cancel
CHOOSE AN AMI
INSTANCE DETAILS
CREATE KEY PAIR
CONFIGURE FIREWALL
REVIEW
Security groups determine whether a network port is open or blocked on your instances. You may use an existing security group, or we can help you create a new security group to allow access to your instances using the suggested ports below. Add additional ports now or update your security group anytime using the Security Groups page.
Choose one or more of your existing Security Groups
Create a new Security Group
Group Name
hazelcast
Group Description
hazelcast
Inbound Rules
Create a new rule:
Custom TCP rule
Port range:
(e.g., 80 or 49152-65535)
Source:
0.0.0.0/0
(e.g., 192.168.2.0/24, sg-47ad482e, or 1234567890/default)
Add Rule
TCP
Port (Service)
Source
Action
0 - 65535
0.0.0.0/0
Delete
Back
Continue


Request Instances Wizard
Cancel
CHOOSE AN AMI
INSTANCE DETAILS
CREATE KEY PAIR
CONFIGURE FIREWALL
REVIEW
Please review the information below, then click Launch.
AMI: Ubuntu Cloud Guest AMI ID ami-640a0610 (x86_64)
Name: Ubuntu Server 12.10
Description: Ubuntu Server 12.10 with support available from Canonical (http://www.ubuntu.com/cloud/services).
Edit AMI
Number of Instances: 2
Availability Zone: No Preference
Instance Type: T1 Micro (t1.micro)
Instance Class: On Demand
EBS-Optimized: No
Edit Instance Details
Monitoring: Disabled
Termination Protection: Disabled
Tenancy: Default
Kernel ID: Use Default
Shutdown Behavior: Stop
RAM Disk ID: Use Default
Network Interfaces:
Secondary IP Addresses:
User Data:
IAM Role:
Edit Advanced Details
Key Pair Name: hazelcastkeypair
Edit Key Pair
Back
Launch


Launch Instance Wizard
Cancel
Your instances are now launching.
Instance ID(s): i-ee8b21a4,i-ec8b21a6
Note: Your instances may take a few minutes to launch, depending on the software you are running.
Note: Usage hours on your new instances will start immediately and continue to accrue until you stop or terminate your instances.
You can perform the following tasks while your instances are launching:
Create Status Check Alarms
You can use status check alarms to be notified if these instances fail status checks (additional charges may apply).
Create EBS Volumes (Additional charges may apply.)
View your instances on the Instances page
Close