# Contents

1 Preface ................................................................. 17
   1.1 Hazelcast Editions ................................................. 17
   1.2 Hazelcast Architecture .......................................... 17
   1.3 Licensing ........................................................ 17
   1.4 Trademarks ..................................................... 17
   1.5 Customer Support ............................................... 19
   1.6 Release Notes .................................................. 19
   1.7 Contributing to Hazelcast ...................................... 19
   1.8 Phone Home ..................................................... 19
   1.9 Typographical Conventions ..................................... 20

2 Document Revision History ........................................... 21

3 Getting Started ....................................................... 23
   3.1 Installation ..................................................... 23
      3.1.1 Hazelcast .................................................. 23
      3.1.2 Hazelcast Enterprise ...................................... 23
   3.2 Starting the Member and Client ................................ 25
      3.2.1 Deploying On Amazon EC2 .................................. 26
   3.3 Configuring Hazelcast .......................................... 26

4 Hazelcast Overview .................................................... 29
   4.1 Sharding in Hazelcast ........................................... 30
   4.2 Hazelcast Topology .............................................. 30
   4.3 Why Hazelcast? .................................................. 30
   4.4 Data Partitioning ................................................ 32
      4.4.1 How the Data is Partitioned ................................. 34
      4.4.2 Partition Table ............................................ 34
      4.4.3 Repartitioning ............................................. 34
   4.5 Use Cases ....................................................... 34
   4.6 Resources ....................................................... 35
5 Hazelcast Clusters

5.1 Discovering Cluster Members .................................................. 37
  5.1.1 Discovering Members by Multicast ...................................... 37
  5.1.2 Discovering Members by TCP ............................................ 38
  5.1.3 Discovering Members within EC2 Cloud ................................ 39

5.2 Creating Cluster Groups ....................................................... 40

6 Distributed Data Structures ..................................................... 41

6.1 Map ................................................................. 42
  6.1.1 Map Overview ......................................................... 42
  6.1.2 Map Backups .......................................................... 45
  6.1.3 Map Eviction .......................................................... 46
  6.1.4 In Memory Format ..................................................... 49
  6.1.5 Map Persistence ....................................................... 50
  6.1.6 Near Cache ............................................................ 55
  6.1.7 Map Locks .............................................................. 57
  6.1.8 Entry Statistics ....................................................... 59
  6.1.9 Map Listener .......................................................... 60
  6.1.10 Interceptors ........................................................... 62
  6.1.11 Preventing Out of Memory Exceptions ............................... 65

6.2 Queue ................................................................. 66
  6.2.1 Queue Overview ....................................................... 66
  6.2.2 Sample Queue Code ................................................... 67
  6.2.3 Bounded Queue ........................................................ 68
  6.2.4 Queue Persistence ..................................................... 69
  6.2.5 Configuring Queue .................................................... 70

6.3 MultiMap ............................................................... 71
  6.3.1 Sample MultiMap Code ............................................... 71
  6.3.2 Configuring MultiMap ................................................ 71

6.4 Set ................................................................. 72
  6.4.1 Sample Set Code ...................................................... 72
  6.4.2 Event Registration and Configuration for Set ....................... 72

6.5 List ............................................................... 73
  6.5.1 Sample List Code ...................................................... 73
  6.5.2 Event Registration and Configuration for List ....................... 74

6.6 Ringbuffer ............................................................ 74
  6.6.1 IQueue vs. Ringbuffer ................................................ 75
  6.6.2 Capacity ............................................................... 75
  6.6.3 Synchronous and Asynchronous Backups ............................. 76
  6.6.4 Time to live .......................................................... 76
10.1.5 Placing a Remote Call - Proxy .................................................. 145
10.1.6 Creating Containers .............................................................. 150
10.1.7 Partition Migration ............................................................... 154
10.1.8 Creating Backups ................................................................. 159
10.2 WaitNotifyService ................................................................. 161

11 Transactions ................................................................. 163
11.1 Transaction Interface .......................................................... 163
  11.1.1 LOCAL versus TWO PHASE .................................................. 164
11.2 XA Transactions ................................................................. 164
11.3 J2EE Integration ................................................................. 165
  11.3.1 Sample Code for J2EE Integration ........................................ 166
  11.3.2 Resource Adapter Configuration ......................................... 166
  11.3.3 Sample Glassfish v3 Web Application Configuration ................. 166
  11.3.4 Sample JBoss AS 5 Web Application Configuration .................. 167
  11.3.5 Sample JBoss AS 7 / EAP 6 Web Application Configuration .......... 167

12 Hazelcast JCache ............................................................. 171
12.1 JCache Overview ............................................................... 171
12.2 Setup and Configuration ..................................................... 171
  12.2.1 Application Setup .......................................................... 171
  12.2.2 Quick Example ............................................................. 173
  12.2.3 JCache Configuration ....................................................... 174
12.3 JCache Providers ............................................................... 176
  12.3.1 Provider Configuration ..................................................... 176
  12.3.2 JCache Client Provider .................................................... 177
  12.3.3 JCache Server Provider .................................................... 177
12.4 Introduction to the JCache API ............................................. 177
  12.4.1 JCache API Walk-through ................................................ 177
  12.4.2 Roundup of Basics ........................................................ 179
  12.4.3 Factory and FactoryBuilder .............................................. 180
  12.4.4 CacheLoader ............................................................... 180
  12.4.5 CacheWriter ............................................................... 181
  12.4.6 JCache EntryProcessor .................................................... 182
  12.4.7 CacheEntryListener ....................................................... 183
  12.4.8 ExpirePolicy .............................................................. 185
12.5 Hazelcast JCache Extension - ICache .................................... 185
  12.5.1 Scopes and Namespaces ................................................... 185
  12.5.2 Retrieving an ICache Instance .......................................... 188
  12.5.3 ICache Configuration ..................................................... 189
12.5.4 Async Operations .................................................. 190
12.5.5 Custom ExpiryPolicy ............................................... 191
12.5.6 JCache Eviction ..................................................... 192
12.5.7 JCache Near Cache ............................................... 195
12.5.8 Additional Methods .................................................. 198
12.5.9 BackupAwareEntryProcessor ...................................... 199
12.6 JCache Specification Compliance .................................... 200

13 Integrated Clustering .................................................. 203
13.1 Hibernate Second Level Cache ..................................... 203
  13.1.1 Sample Code for Hibernate .................................. 203
  13.1.2 Supported Hibernate Versions .................................. 203
  13.1.3 Hibernate Configuration ....................................... 203
  13.1.4 Hazelcast Configuration for Hibernate ....................... 204
  13.1.5 RegionFactory Options ......................................... 205
  13.1.6 Hazelcast Modes for Hibernate Usage ....................... 206
  13.1.7 Hibernate Concurrency Strategies ............................. 206
  13.1.8 Advanced Settings ............................................. 207
13.2 Web Session Replication ............................................ 207
  13.2.1 Filter Based Web Session Replication ....................... 208
  13.2.2 Spring Security Support ...................................... 210
  13.2.3 Tomcat Based Web Session Replication ....................... 213
  13.2.4 Jetty Based Web Session Replication ....................... 217
13.3 Spring Integration .................................................. 221
  13.3.1 Supported Versions ............................................. 221
  13.3.2 Spring Configuration ........................................... 222
  13.3.3 Spring Managed Context with @SpringAware .................... 225
  13.3.4 Spring Cache .................................................... 227
  13.3.5 Hibernate 2nd Level Cache Config ............................ 228
  13.3.6 Best Practices ................................................... 229

14 Storage .................................................. 231
14.1 High-Density Memory Store ....................................... 231
  14.1.1 Configuring Hi-Density Memory Store ....................... 231
14.2 Elastic Memory (High-Density Memory First Generation) ....... 232
14.3 Sizing Practices .................................................. 233
15 Hazelcast Java Client

15.1 Hazelcast Clients Feature Comparison ................................................. 235
15.2 Java Client Overview ................................................................. 237
  15.2.1 Java Client Dependencies ......................................................... 237
  15.2.2 Getting Started with Client API ................................................ 237
  15.2.3 Java Client Operation modes ..................................................... 238
  15.2.4 Failure Handling ................................................................. 238
  15.2.5 Supported Distributed Data Structures ........................................ 238
  15.2.6 Client Services ................................................................. 239
  15.2.7 Client Listeners ................................................................. 241
  15.2.8 Client Transactions ............................................................ 241
15.3 Java Client Configuration ............................................................. 241
  15.3.1 Client Network Configuration .................................................... 241
  15.3.2 Client Load Balancer Configuration .......................................... 247
  15.3.3 Client Near Cache Configuration .............................................. 247
  15.3.4 Client Group Configuration ..................................................... 248
  15.3.5 Client Security Configuration .................................................. 248
  15.3.6 Client Serialization Configuration ............................................. 248
  15.3.7 Client Listener Configuration .................................................. 248
  15.3.8 ExecutorPoolSize .............................................................. 249
  15.3.9 ClassLoader ........................................................................ 249
15.4 Client System Properties ............................................................... 249
15.5 Sample Codes for Client ................................................................. 249

16 Other Client Implementations .......................................................... 251

16.1 C++ Client ........................................................................... 251
  16.1.1 How to Setup ..................................................................... 251
  16.1.2 Platform Specific Installation Guides .................................... 252
  16.1.3 Code Examples .................................................................. 252
16.2 .NET Client ........................................................................ 256
  16.2.1 Client Configuration ............................................................. 259
  16.2.2 Client Startup .................................................................. 259
16.3 REST Client .......................................................................... 259
16.4 Memcache Client ................................................................. 262
  16.4.1 Unsupported Operations ......................................................... 263
17 Serialization

17.1 Serialization Overview .................................................. 265
17.2 Serialization Interfaces .................................................. 265
17.3 Comparison Table .......................................................... 266
17.4 Serializable & Externalizable ............................................. 266
17.5 DataSerializable ............................................................. 267
  17.5.1 IdentifiedDataSerializable ......................................... 269
17.6 Portable .................................................................. 270
  17.6.1 Versions ................................................................ 272
  17.6.2 Null Portable Serialization ........................................ 273
  17.6.3 DistributedObject Serialization .................................. 273
17.7 Custom Serialization ....................................................... 273
  17.7.1 StreamSerializer ....................................................... 273
  17.7.2 ByteArraySerializer .................................................. 276
17.8 HazelcastInstanceAware Interface ....................................... 276

18 Management ................................................................ 279

18.1 Statistics API per Node .................................................... 279
  18.1.1 Map Statistics .......................................................... 279
  18.1.2 Multimap Statistics ..................................................... 282
  18.1.3 Queue Statistics ......................................................... 285
  18.1.4 Topic Statistics .......................................................... 286
  18.1.5 Executor Statistics ...................................................... 287
18.2 JMX API per Node ............................................................ 287
18.3 Monitoring with JMX ......................................................... 293
18.4 Cluster Utilities ............................................................... 294
  18.4.1 Cluster Interface ......................................................... 294
  18.4.2 Member Attributes ...................................................... 295
  18.4.3 Cluster-Member Safety Check ..................................... 296
  18.4.4 Cluster Quorum .......................................................... 297
18.5 Management Center ......................................................... 299
  18.5.1 Introduction .............................................................. 299
  18.5.2 Tool Overview .......................................................... 300
  18.5.3 Home Page .............................................................. 303
  18.5.4 Caches ................................................................. 304
  18.5.5 Maps ................................................................. 306
  18.5.6 Queues ............................................................... 309
  18.5.7 Topics .............................................................. 311
  18.5.8 MultiMaps ............................................................ 312
  18.5.9 Executors ............................................................. 312
## 20 Performance

20.1 Data Affinity .......................................................... 353
20.2 Back Pressure .......................................................... 356
20.3 Threading Model ....................................................... 357
  20.3.1 I/O Threading .................................................. 357
  20.3.2 Event Threading .............................................. 358
  20.3.3 IExecutor Threading ....................................... 358
  20.3.4 Operation Threading ....................................... 358
20.4 SlowOperationDetector ........................................... 360
  20.4.1 Logging of Slow Operations .................................. 361
  20.4.2 Purging of Slow Operation Logs .............................. 361
20.5 Hazelcast Performance on AWS .................................... 361
  20.5.1 Selecting EC2 Instance Type .................................. 361
  20.5.2 Dealing with Network Latency ................................ 362
  20.5.3 Selecting Virtualization .................................... 362

## 21 Hazelcast Simulator

21.1 Key Concepts .......................................................... 363
21.2 Installing Simulator ................................................. 364
  21.2.1 Firewall settings ........................................... 364
  21.2.2 Setup of local machine (Coordinator) ....................... 365
  21.2.3 Setup of remote machines (Agents, Workers) ............... 365
  21.2.4 Setup of public/private key pair ........................... 365
21.3 Setting Up For Amazon EC2 ......................................... 366
21.4 Setting Up For Google Compute Engine ............................ 366
21.5 Setting Up Machines Manually .................................... 367
21.6 Executing a Simulator Test ......................................... 367
  21.6.1 Creating and Editing Properties File ....................... 368
  21.6.2 Running the Test ........................................... 369
  21.6.3 Running the Test with a Script ............................. 373
  21.6.4 Using Maven Archetypes ................................... 373
21.7 Provisioner ............................................................ 374
  21.7.1 Accessing the Provisioned Machine ......................... 374
21.8 Coordinator ........................................................... 375
  21.8.1 Controlling Hazelcast Declarative Configuration .......... 375
  21.8.2 Controlling Test Duration .................................. 375
  21.8.3 Controlling Client And Workers ............................. 375
21.9 Communicator .......................................................... 376
  21.9.1 Example ....................................................... 376
  21.9.2 Message Types ................................................ 376
## CONTENTS

- 21.9.3 Message Addressing .......................................................... 376
- 21.10 Simulator.Properties File Description ....................................... 377
- 21.11 Performance and Benchmarking ............................................... 379

## 22 WAN

- 22.1 WAN Replication ............................................................... 381
  - 22.1.1 Configuring WAN Replication .......................................... 381
  - 22.1.2 WAN Replication Additional Information ............................ 382
- 22.2 Enterprise WAN Replication .................................................. 383
  - 22.2.1 Replication implementations ........................................... 383
  - 22.2.2 WAN Replication Batch Size ........................................... 383
  - 22.2.3 WAN Replication Batch Frequency .................................... 384
  - 22.2.4 WAN Replication Operation Timeout .................................. 384
  - 22.2.5 WAN Replication Queue Capacity ..................................... 384
  - 22.2.6 Enterprise WAN Replication Additional Information ............... 385

## 23 Hazelcast Configuration

- 23.1 Configuration Overview ...................................................... 387
- 23.2 Using Wildcard ................................................................. 389
- 23.3 Using Variables .................................................................. 390
- 23.4 Composing Declarative Configuration ..................................... 390
- 23.5 Network Configuration ........................................................ 392
  - 23.5.1 Public Address .............................................................. 393
  - 23.5.2 Port ............................................................................ 393
  - 23.5.3 Outbound Ports ............................................................ 394
  - 23.5.4 Reuse Address ............................................................. 394
  - 23.5.5 Join .......................................................................... 395
  - 23.5.6 Interfaces .................................................................... 397
  - 23.5.7 SSL ........................................................................... 398
  - 23.5.8 Socket Interceptor .......................................................... 398
  - 23.5.9 Symmetric Encryption .................................................... 398
  - 23.5.10 IPv6 Support ............................................................... 398
- 23.6 Group Configuration .............................................................. 399
- 23.7 Map Configuration ............................................................... 399
  - 23.7.1 Map Store .................................................................... 401
  - 23.7.2 Near Cache ................................................................... 401
  - 23.7.3 Indexes ....................................................................... 402
  - 23.7.4 Entry Listeners .............................................................. 402
- 23.8 MultiMap Configuration ........................................................ 402
- 23.9 Queue Configuration ............................................................. 402
CONTENTS

23.10 Topic Configuration ........................................... 403
23.11 Reliable Topic Configuration ................................. 404
23.12 List Configuration ............................................ 405
23.13 Set Configuration ............................................. 406
23.14 Ringbuffer Configuration ..................................... 406
23.15 Semaphore Configuration ..................................... 407
23.16 Executor Service Configuration .............................. 407
23.17 Cache Configuration ........................................... 408
23.18 Serialization Configuration ................................... 409
23.19 MapReduce Jobtracker Configuration ...................... 410
23.20 Services Configuration ....................................... 411
23.21 Management Center Configuration .......................... 412
23.22 WAN Replication Configuration .............................. 412
23.23 Enterprise WAN Replication Configuration .................. 413
   23.23.1 IMap and ICache WAN Configuration .................... 414
23.24 Partition Group Configuration ............................... 416
23.25 Listener Configurations ..................................... 417
23.26 Logging Configuration ........................................ 421
23.27 System Properties ........................................... 423

24 Network Partitioning - Split Brain Syndrome ............... 427
  24.1 Understanding Partition Recreation ........................ 427
  24.2 Understanding Backup Partition Creation .................. 427
  24.3 Understanding The Update Overwrite Scenario ............. 427
  24.4 What Happens When The Network Failure Is Fixed .......... 428
  24.5 How Hazelcast Split Brain Merge Happens .................. 428
  24.6 Specifying Merge Policies ................................... 429

25 License Questions ................................................ 431
  25.1 Embedded Dependencies ..................................... 431
  25.2 Runtime Dependencies ....................................... 431

26 Common Exception Types ........................................ 433

27 Frequently Asked Questions .................................. 435
  27.1 Why 271 as the default partition count? .................... 435
  27.2 Is Hazelcast thread safe? .................................... 435
  27.3 What happens when a node goes down? ....................... 435
  27.4 How do I test the connectivity? ............................... 436
  27.5 How do I choose keys properly? .............................. 436
  27.6 How do I reflect value modifications? ....................... 436
27.7 How do I test my Hazelcast cluster? ........................................... 436
27.8 Does Hazelcast support hundreds of nodes? .............................. 438
27.9 Does Hazelcast support thousands of clients? ............................. 438
27.10 What is the difference between old LiteMember and new Smart Client? .................. 438
27.11 How do you give support? ......................................................... 438
27.12 Does Hazelcast persist? ............................................................. 438
27.13 Can I use Hazelcast in a single server? .................................... 438
27.14 How can I monitor Hazelcast? ................................................... 439
27.15 How can I see debug level logs? ................................................. 439
27.16 What is the difference between client-server and embedded topologies? .................. 439
27.17 How do I know it is safe to kill the second node? ....................... 440
27.18 When do I need Native Memory solutions? ................................. 440
27.19 Is there any disadvantage of using near-cache? .......................... 440
27.20 Is Hazelcast secure? ................................................................. 440
27.21 How can I set socket options? ................................................... 440
27.22 Periodically see client disconnections during idle time? ................ 440
27.23 How to get rid of “java.lang.OutOfMemoryError: unable to create new native thread”? .. 441
27.24 Does repartitioning wait for Entry Processor? .............................. 441
27.25 Why do Hazelcast instances on different machines not see each other? .................. 441
27.26 What Does “Replica: 1 has no owner” Mean? .............................. 442

28 Glossary .................................................................................. 443
Chapter 1

Preface

Welcome to the Hazelcast Reference Manual. This manual includes concepts, instructions and samples to guide you on how to use Hazelcast and build Hazelcast applications.

As the reader of this manual, you must be familiar with the Java programming language and you should have installed your preferred IDE.

1.1 Hazelcast Editions

This Reference Manual covers all editions of Hazelcast. Throughout this manual:

- **Hazelcast** refers to the open source edition of Hazelcast in-memory data grid middleware. It is also the name of the company providing the Hazelcast product.
- **Hazelcast Enterprise** - a commercially licensed edition of Hazelcast which provides high-value enterprise features in addition to Hazelcast.
- **Hazelcast Enterprise HD** - a commercially licensed edition of Hazelcast which provides High-Density (HD) Memory Store and Hot Restart Persistence features in addition to Hazelcast Enterprise.

1.2 Hazelcast Architecture

You can see the features for all Hazelcast editions in the following architecture diagram.

For more information on Hazelcast’s Architecture, please see the whitepaper An Architect’s View of Hazelcast.

1.3 Licensing

Hazelcast is free provided under the Apache 2 license. Hazelcast Enterprise is commercially licensed by Hazelcast, Inc.

For more detailed information on licensing, please see the License Questions appendix.

1.4 Trademarks

Hazelcast is a registered trademark of Hazelcast, Inc. All other trademarks in this manual are held by their respective owners.
Figure 1.1: image
1.5  Customer Support

Support for Hazelcast is provided via GitHub, Mail Group and StackOverflow.

For information on the commercial support for Hazelcast and Hazelcast Enterprise, please see hazelcast.com.

1.6  Release Notes

Please refer to the Release Notes document for the new features, enhancements and fixes performed for each Hazelcast release. You can also find information on upgrading Hazelcast from previous releases in the Release Notes document.

1.7  Contributing to Hazelcast

You can contribute to the Hazelcast code, report a bug or request an enhancement. Please see the following resources.

- Developing with Git: Document that explains the branch mechanism of Hazelcast and how to request changes.
- Hazelcast Contributor Agreement form: Form that each contributing developer needs to fill and send back to Hazelcast.
- Hazelcast on GitHub: Hazelcast repository where the code is developed, issues and pull requests are managed.

1.8  Phone Home

Hazelcast uses phone home data to learn about usage of Hazelcast.

Hazelcast member instances call our phone home server initially when they are started and then for every 24 hours. This applies to all the instances joined to the cluster.

What is sent in?
The following information is sent in a phone home:

- Hazelcast version
- Local Hazelcast member UUID
- Download ID
- A hash value of the cluster ID
- Cluster size bands for 5, 10, 20, 40, 60, 100, 150, 300, 600 and > 600
- Number of connected clients bands of 5, 10, 20, 40, 60, 100, 150, 300, 600 and > 600
- Cluster uptime
- Member uptime
- Hazelcast Enterprise specific:
  - Number of clients by language (Java, C++, C#)
  - Flag for Hazelcast Enterprise
  - Hash value of license key
  - Native memory usage

Phone Home Code
The phone home code itself is open source. Please see here.

Disabling Phone Homes
Set the hazelcast.phone.home.enabled system property to false either in the config or on the Java command line. Please see the System Properties section for information on how to set a property.
Phone Home URLs
For versions 1.x and 2.x: http://www.hazelcast.com/version.jsp.
For versions 3.x up to 3.6: http://versioncheck.hazelcast.com/version.jsp.
For versions after 3.6: http://phonehome.hazelcast.com/ping.

1.9 Typographical Conventions

Below table shows the conventions used in this manual.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bold font</strong></td>
<td>- Indicates part of a sentence that require the reader’s specific attention. - Also indicates property/parameter values.</td>
</tr>
<tr>
<td><em>italic font</em></td>
<td>- When italicized words are enclosed with “&lt;” and “&gt;”, indicates a variable in command/variable syntax.</td>
</tr>
<tr>
<td><code>monospace</code></td>
<td>- Indicates files, folders, class and library names, code snippets, and inline code words in a sentence.</td>
</tr>
<tr>
<td>RELATED INFORMATION</td>
<td>- Indicates a resource that is relevant to the topic, usually with a link or cross-reference.</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>Indicates information that is of special interest or importance, e.g. an additional action required in a particular situation.</td>
</tr>
<tr>
<td>element &amp; attribute</td>
<td>Mostly used in the context of declarative configuration, i.e. configuration performed by the <code>Hazelcast</code> configuration file.</td>
</tr>
</tbody>
</table>
Chapter 2

Document Revision History

This chapter lists the changes made to this document from the previous release.

NOTE: Please refer to the Release Notes for the new features, enhancements and fixes performed for each Hazelcast release. You can also find information on upgrading Hazelcast from previous releases in the Release Notes document.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1 - Preface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 2 - What’s New in Hazelcast 3.5</td>
<td>Upgrading from 3.x</td>
<td>Added information on how to contribute to Hazelcast.</td>
</tr>
<tr>
<td>Chapter 3 - Getting Started</td>
<td>Deploying On Amazon EC2</td>
<td>Added as a new section.</td>
</tr>
<tr>
<td>Chapter 4 - Hazelcast Overview</td>
<td>Data Partitioning</td>
<td>Separated from Getting Started as a new section.</td>
</tr>
<tr>
<td>Chapter 5 - Hazelcast Clusters</td>
<td>Creating Cluster Groups</td>
<td>Added as a new section explaining how to separate a Hazelcast cluster.</td>
</tr>
<tr>
<td>Chapter 6 - Distributed Data Structures</td>
<td>Map</td>
<td>The content of the section, previously read as Entry Listener improved and its name changed to Map Listener.</td>
</tr>
<tr>
<td></td>
<td>Replicated Map</td>
<td>Description of the new element write-coalescing added to the Write-Behind section.</td>
</tr>
<tr>
<td></td>
<td>RingBuffer</td>
<td>Incremental Key Loading added as a new section.</td>
</tr>
<tr>
<td></td>
<td>Reliable Topic</td>
<td>MapEviction added as a new section.</td>
</tr>
<tr>
<td>Chapter 7 - Distributed Events</td>
<td>Partition Lost Listener</td>
<td>The whole chapter improved by adding sections describing each listener.</td>
</tr>
<tr>
<td>Chapter 8 - Distributed Computing</td>
<td>Execution Member Selector</td>
<td>Added as a new section.</td>
</tr>
<tr>
<td>Chapter 9 - Distributed Query</td>
<td>Paging Predicate</td>
<td>Added a note related to random page access.</td>
</tr>
<tr>
<td>Chapter 11 - Transactions</td>
<td>Local versus Two Phase</td>
<td>Added a note related to REPEATABLE_READ isolation level.</td>
</tr>
<tr>
<td>Chapter 12 - Hazelcast JCache</td>
<td>JCache Near Cache</td>
<td>Added as a new section explaining the trade-offs between these two transaction types.</td>
</tr>
<tr>
<td>Chapter 13 - Integrated Clustering</td>
<td>Tomcat Based Web Session Replication</td>
<td>Added introduction paragraphs.</td>
</tr>
<tr>
<td></td>
<td>Web Session Replication</td>
<td>Updated the Overview paragraphs.</td>
</tr>
<tr>
<td></td>
<td>Sizing Practices</td>
<td>Added transient-attributes as a new section.</td>
</tr>
<tr>
<td>Chapter 14 - Storage</td>
<td></td>
<td>Added as a new section.</td>
</tr>
<tr>
<td>Chapter 15 - Hazelcast Java Client</td>
<td>Hazelcast Clients Feature Comparison</td>
<td>Separated from the formerly known &quot;Clients&quot; chapter to be a chapter of its own.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added an important note related to the new Java Client library in the chapter introduction.</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>Chapter 16 - Other Client Implementations</td>
<td>C++, .NET, Memcache and REST client sections separated from the formerly known “Clients” chapter.</td>
<td></td>
</tr>
<tr>
<td>Chapter 18 - Management</td>
<td>JMX API per Node</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Management Center</td>
<td></td>
</tr>
<tr>
<td>Chapter 19 - Security</td>
<td>ClusterLoginModule</td>
<td></td>
</tr>
<tr>
<td>Chapter 20 - Performance</td>
<td>Hazelcast Performance on AWS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Back Pressure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SlowOperationDetector</td>
<td></td>
</tr>
<tr>
<td>Chapter 21 - Hazelcast Simulator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 22 - WAN</td>
<td>WAN Replication Queue Capacity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enterprise WAN Replication</td>
<td></td>
</tr>
<tr>
<td>Chapter 23 - Hazelcast Configuration</td>
<td>Configuration Overview</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Using Variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System Properties</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enterprise WAN Replication Configuration</td>
<td></td>
</tr>
<tr>
<td>Chapter 25 - License Questions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 26 - Common Exception Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 27 - FAQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 28 - Glossary</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two new bean definitions added.

Added more information on the time travel data files to the Time Travel section.

The Enterprise Integration section added.

Added as a new section that provides best practices to improve the Hazelcast performance on Amazon Web Service.

Added as a new section explaining Back Pressure.

Added as a new section explaining SlowOperationDetector, a monitoring feature that collects information of all slow operations.

Added as a new chapter providing comprehensive information on the Hazelcast Simulator feature.

The previous heading title (WAN Replication Queue Size) and the system property name (hazelcast.enterprise.wanrep.queuesize) changed to WAN Replication Queue Capacity and hazelcast.enterprise.wanrep.queue.capacity.

Added as a new section describing the elements and attributes of Enterprise WAN Replication configuration.

Improved by adding missing configuration elements and attributes. Added introduction paragraphs to the chapter.

Added a note related to the invalid configurations.

Added as a new section explaining how to use variables in declarative configuration.

Added the new system properties hazelcast.unsafe.mode, hazelcast.io.input.thread.count and hazelcast.io.output.thread.count. Updated the definition of hazelcast.io.thread.count.

Added as a new section describing Enterprise WAN Replication Configuration.

Added as a new chapter describing license information of dependencies.

Added as a new chapter.

Added new questions/answers.

Added new glossary items.
Chapter 3

Getting Started

This chapter explains how to install Hazelcast, start a Hazelcast member and client, and gives Hazelcast configuration fundamentals.

3.1 Installation

The following sections explains the installation of Hazelcast and Hazelcast Enterprise.

3.1.1 Hazelcast

You can find Hazelcast in standard Maven repositories. If your project uses Maven, you do not need to add additional repositories to your pom.xml or add hazelcast-<version>.jar file into your classpath (Maven does that for you). Just add the following lines to your pom.xml:

```xml
<dependencies>
    <dependency>
        <groupId>com.hazelcast</groupId>
        <artifactId>hazelcast</artifactId>
        <version>3.5.2</version>
    </dependency>
</dependencies>
```

As an alternative, you can download and install Hazelcast yourself. You only need to:

- Unzip hazelcast-<version>.zip file.
- Add hazelcast-<version>.jar file into your classpath.

3.1.2 Hazelcast Enterprise

There are two Maven repositories defined for Hazelcast Enterprise:

```xml
<repository>
    <id>Hazelcast Private Snapshot Repository</id>
    <url>https://repository-hazelcast-l337.forge.cloudbees.com/snapshot/</url>
</repository>
```
Hazelcast Enterprise customers may also define dependencies, a sample of which is shown below.

```xml
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast-enterprise-tomcat6</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast-enterprise-tomcat7</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast-enterprise</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast-enterprise-all</artifactId>
  <version>${project.version}</version>
</dependency>
```

### 3.1.2.1 Setting the License Key

To use Hazelcast Enterprise, you need to set the license key in configuration.

- **Declarative Configuration**

  ```xml
  <hazelcast>
  ...
  <license-key>HAZELCAST_ENTERPRISE_LICENSE_KEY</license-key>
  ...
  </hazelcast>
  
  - **Client Declarative Configuration**

    ```xml
    <hazelcast-client>
    ...
    <license-key>HAZELCAST_ENTERPRISE_LICENSE_KEY</license-key>
    ...
    </hazelcast-client>
    
    - **Programmatic Configuration**

      ```java
      Config config = new Config();
      config.setLicenseKey("HAZELCAST_ENTERPRISE_LICENSE_KEY");
      ```

    - **Spring XML Configuration**
3.2 Starting the Member and Client

Having installed Hazelcast, you can get started. In this short tutorial, you perform the following activities.

1. Create a simple Java application using the Hazelcast distributed map and queue.
2. Run our application twice to have a cluster with two members (JVMs).
3. Connect to our cluster from another Java application by using the Hazelcast Native Java Client API.

Let’s begin.

- The following code starts the first Hazelcast member and creates and uses the `customers` map and queue.

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import java.util.Map;
import java.util.Queue;

public class GettingStarted {
    public static void main( String[] args ) {
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        Map<Integer, String> customers = hazelcastInstance.getMap( "customers" );
        customers.put( 1, "Joe" );
        customers.put( 2, "Ali" );
        customers.put( 3, "Avi" );
        System.out.println( "Customer with key 1: " + customers.get(1) );
        System.out.println( "Map Size:" + customers.size() );
        Queue<String> queueCustomers = hazelcastInstance.getQueue( "customers" );
        queueCustomers.offer( "Tom" );
        queueCustomers.offer( "Mary" );
        queueCustomers.offer( "Jane" );
        System.out.println( "First customer: " + queueCustomers.poll() );
        System.out.println( "Second customer: " + queueCustomers.peek() );
        System.out.println( "Queue size: " + queueCustomers.size() );
    }
}
```

- Run this `GettingStarted` class a second time to get the second member started. The members form a cluster and the output is similar to the following.
CHAPTER 3. GETTING STARTED

Members [2] {
    Member [127.0.0.1:5701]
    Member [127.0.0.1:5702] this
}

• Now, add the `hazelcast-client-<version>.jar` library to your classpath. This is required to use a Hazelcast client.

• The following code starts a Hazelcast Client, connects to our cluster, and prints the size of the `customers` map.

```java
package com.hazelcast.test;

import com.hazelcast.client.config.ClientConfig;
import com.hazelcast.client.HazelcastClient;
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.IMap;

public class GettingStartedClient {
    public static void main(String[] args) {
        ClientConfig clientConfig = new ClientConfig();
        HazelcastInstance client = HazelcastClient.newHazelcastClient(clientConfig);
        IMap map = client.getMap("customers");
        System.out.println("Map Size:" + map.size());
    }
}
```

• When you run it, you see the client properly connecting to the cluster and printing the map size as 3.

Hazelcast also offers a tool, Management Center, that enables you to monitor your cluster. To use it, deploy the `mancenter-<version>.war` included in the ZIP file to your web server. You can use it to monitor your maps, queues, and other distributed data structures and members. Please see the Management Center section for usage explanations.

By default, Hazelcast uses Multicast to discover other members that can form a cluster. If you are working with other Hazelcast developers on the same network, you may find yourself joining their clusters under the default settings. Hazelcast provides a way to segregate clusters within the same network when using Multicast. Please see the Creating Cluster Groups for more information. Alternatively, if you do not wish to use the default Multicast mechanism, you can provide a fixed list of IP addresses that are allowed to join. Please see the Join Configuration section for more information.

RELATED INFORMATION
You can also check the video tutorials here.

3.2.1 Deploying On Amazon EC2

You can deploy your Hazelcast project onto Amazon EC2 environment using Third Party tools such as Vagrant and Chef.

You can find a sample deployment project (`amazon-ec2-vagrant-chef`) with step by step instructions in the `hazelcast-integration` folder of the `hazelcast-code-samples` package. Please refer to this sample project for more information.

3.3 Configuring Hazelcast

When Hazelcast starts up, it checks for the configuration as follows:
3.3. CONFIGURING HAZELCAST

- First, it looks for the `hazelcast.config` system property. If it is set, its value is used as the path. This is useful if you want to be able to change your Hazelcast configuration: you can do this because it is not embedded within the application. You can set the `config` option with the following command:
  - `Dhazelcast.config=<path to the hazelcast.xml>`.
  The path can be a normal one or a classpath reference with the prefix CLASSPATH.
- If the above system property is not set, Hazelcast then checks whether there is a `hazelcast.xml` file in the working directory.
- If not, then it checks whether `hazelcast.xml` exists on the classpath.
- If none of the above works, Hazelcast loads the default configuration, i.e. `hazelcast-default.xml` that comes with `hazelcast.jar`.

When you download and unzip `hazelcast-<version>.zip`, you will see a `hazelcast.xml` in the `/bin` folder. This is the declarative configuration file for Hazelcast. Part of this XML file is shown below.

```xml
<group>
  <name>dev</name>
  <password>dev-pass</password>
</group>
<management-center enabled="false"/>http://localhost:8080/mancenter</management-center>
<network>
  <port auto-increment="true" port-count="100">5701</port>
  <outbound-ports>
    <!--
    Allowed port range when connecting to other nodes.
    0 or * means use system provided port.
    -->
    <ports>0</ports>
  </outbound-ports>
  <multicast enabled="true">
    <multicast-group>224.2.2.3</multicast-group>
    <multicast-port>54327</multicast-port>
  </multicast>
  <tcp-ip enabled="false"/>
</network>
```

For most users, default configuration should be fine. If not, you can tailor this XML file according to your needs by adding/removing/modifying properties.

Besides declarative configuration, you can configure your cluster programmatically. Just instantiate a `Config` object and add/remove/modify properties.

You can also use wildcards while configuring Hazelcast. Please refer to the Using Wildcard section for details.

Hazelcast also offers System Properties to tune some aspects of it. Please refer to the System Properties section for details.

**RELATED INFORMATION**

Please refer to the Hazelcast Configuration chapter for more information.
Hazelcast Overview

Hazelcast is an open source In-Memory Data Grid (IMDG). It provides elastically scalable distributed In-Memory computing, widely recognized as the fastest and most scalable approach to application performance. Hazelcast does this in open source. More importantly, Hazelcast makes distributed computing simple by offering distributed implementations of many developer friendly interfaces from Java such as Map, Queue, ExecutorService, Lock, and JCache. For example, the Map interface provides an In-Memory Key Value store which confers many of the advantages of NoSQL in terms of developer friendliness and developer productivity.

In addition to distributing data In-Memory, Hazelcast provides a convenient set of APIs to access the CPUs in your cluster for maximum processing speed. Hazelcast is designed to be lightweight and easy to use. Since Hazelcast is delivered as a compact library (JAR) and since it has no external dependencies other than Java, it easily plugs into your software solution and provides distributed data structures and distributed computing utilities.

Hazelcast is highly scalable and available (100% operational, never failing). Distributed applications can use Hazelcast for distributed caching, synchronization, clustering, processing, pub/sub messaging, etc. Hazelcast is implemented in Java and has clients for Java, C/C++, .NET and REST. Hazelcast also speaks memcache protocol. It plugs into Hibernate and can easily be used with any existing database system.

If you are looking for In-Memory speed, elastic scalability, and the developer friendliness of NoSQL, Hazelcast is a great choice.

**Hazelcast is simple**

Hazelcast is written in Java with no other dependencies. It exposes the same API from the familiar Java util package, exposing the same interfaces. Just add `hazelcast.jar` to your classpath, and you can quickly enjoy JVMs clustering and you can start building scalable applications.

**Hazelcast is peer-to-peer**

Unlike many NoSQL solutions, Hazelcast is peer-to-peer. There is no master and slave; there is no single point of failure. All nodes store equal amounts of data and do equal amounts of processing. You can embed Hazelcast in your existing application or use it in client and server mode where your application is a client to Hazelcast nodes.

**Hazelcast is scalable**

Hazelcast is designed to scale up to hundreds and thousands of nodes. Simply add new nodes and they will automatically discover the cluster and will linearly increase both memory and processing capacity. The nodes maintain a TCP connection between each other and all communication is performed through this layer.

**Hazelcast is fast**

Hazelcast stores everything in-memory. It is designed to perform very fast reads and updates.

**Hazelcast is redundant**

Hazelcast keeps the backup of each data entry on multiple nodes. On a node failure, the data is restored from the backup and the cluster will continue to operate without downtime.
4.1 Sharding in Hazelcast

Hazelcast shards are called Partitions. By default, Hazelcast has 271 partitions. Given a key, we serialize, hash and mode it with the number of partitions to find the partition the key belongs to. The partitions themselves are distributed equally among the members of the cluster. Hazelcast also creates the backups of partitions and distributes them among nodes for redundancy.

**RELATED INFORMATION**

*Please refer to the Data Partitioning section for more information on how Hazelcast partitions your data.*

4.2 Hazelcast Topology

If you have an application whose main focal point is asynchronous or high performance computing and lots of task executions, then embedded deployment is very useful. In this type, nodes include both the application and data. See the below illustration.

You can have a cluster of server nodes that can be independently created and scaled. Your clients communicate with these server nodes to reach to the data on them. Hazelcast provides native clients (Java, .NET and C++), Memcache clients and REST clients. See the below illustration.

4.3 Why Hazelcast?

**A Glance at Traditional Data Persistence**

Data is at the core of software systems. In conventional architectures, a relational database persists and provides access to data. Applications are talking directly with a database which has its backup as another machine. To increase performance, tuning or a faster machine is required. This can cost a large amount of money or effort.

There is also the idea of keeping copies of data next to the database, which is performed using technologies like external key-value stores or second level caching. This helps to offload the database. However, when the database is saturated or the applications perform mostly “put” operations (writes), this approach is of no use because it insulates the database only from the “get” loads (reads). Even if the applications are read-intensive, there can be consistency problems: when data changes, what happens to the cache, and how are the changes handled? This is when concepts like time-to-live (TTL) or write-through come in.
However, in the case of TTL, if the access is less frequent than the TTL, the result will always be a cache miss. On the other hand, in the case of write-through caches; if there are more than one of these caches in a cluster, then we again have consistency issues. This can be avoided by having the nodes communicating with each other so that entry invalidations can be propagated.

We can conclude that an ideal cache would combine TTL and write-through features. And, there are several cache servers and in-memory database solutions in this field. However, those are stand-alone single instances with a distribution mechanism to an extent provided by other technologies. This brings us back to square one: we would experience saturation or capacity issues if the product is a single instance or if consistency is not provided by the distribution.

And, there is Hazelcast

Hazelcast, a brand new approach to data, is designed around the concept of distribution. Hazelcast shares data around the cluster for flexibility and performance. It is an in-memory data grid for clustering and highly scalable data distribution.

One of the main features of Hazelcast is not having a master node. Each node in the cluster is configured to be the same in terms of functionality. The oldest node (the first node created in the node cluster) manages the cluster members, i.e. automatically performs the data assignment to nodes. If the oldest node dies, the second oldest node will manage the cluster members.

Another main feature is the data being held entirely in-memory. This is fast. In the case of a failure, such as a node crash, no data will be lost since Hazelcast distributes copies of data across all the nodes of cluster.

As shown in the feature list in the Hazelcast Overview, Hazelcast supports a number of distributed data structures and distributed computing utilities. This provides powerful ways of accessing distributed clustered memory and accessing CPUs for true distributed computing.

Hazelcast’s Distinctive Strengths

- It is open source.
- It is a small JAR file. You do not need to install software.
- It is a library, it does not impose an architecture on Hazelcast users.
- It provides out of the box distributed data structures (i.e. Map, Queue, MultiMap, Topic, Lock, Executor, etc.).
- There is no “master”, so no single point of failure in Hazelcast cluster; each node in the cluster is configured to be functionally the same.
• When the size of your memory and compute requirement increases, new nodes can be dynamically joined to the cluster to scale elastically.
• Data is resilient to node failure. Data backups are distributed across the cluster. This is a big benefit when a node in the cluster crashes: data will not be lost.
• Nodes are always aware of each other: they communicate, unlike traditional key-value caching solutions.
• You can build your own custom distributed data structures using the Service Programming Interface (SPI) if you are not happy with the data structures provided.

Finally, Hazelcast has a vibrant open source community enabling it to be continuously developed.

Hazelcast is a fit when you need:

• analytic applications requiring big data processing by partitioning the data,
• to retain frequently accessed data in the grid,
• a cache, particularly an open source JCache provider with elastic distributed scalability,
• a primary data store for applications with utmost performance, scalability and low-latency requirements,
• an In-Memory NoSQL Key Value Store,
• publish/subscribe communication at highest speed and scalability between applications,
• applications that need to scale elastically in distributed and cloud environments,
• a highly available distributed cache for applications,
• an alternative to Coherence, Gemfire and Terracotta.

4.4 Data Partitioning

As you read in the Sharding in Hazelcast section, Hazelcast shards are called Partitions. Partitions are memory segments, where each of those segments can contain hundreds or thousands of data entries, depending on the memory capacity of your system.

By default, Hazelcast offers 271 partitions. When you start a node, that node owns those 271 partitions. The following illustration shows the partitions in a single node Hazelcast cluster.

When you start a second node on that cluster (creating a 2-node Hazelcast cluster), the partitions are distributed as shown in the following illustration.

In the illustration, the partitions with black text are primary partitions, and the partitions with blue text are replica partitions (backups). The first node has 135 primary partitions (black), and each of these partitions are
backed up in the second node (blue). At the same time, the first node also has the replica partitions of the second node’s primary partitions.

As you add more nodes, Hazelcast one-by-one moves some of the primary and replica partitions to the new nodes, making all nodes equal and redundant. Only the minimum amount of partitions will be moved to scale out Hazelcast. The following is an illustration of the partition distributions in a 4-node Hazelcast cluster.

Hazelcast distributes the partitions equally among the members of the cluster. Hazelcast creates the backups of partitions and distributes them among nodes for redundancy.
4.4.1 How the Data is Partitioned

Hazelcast distributes data entries into the partitions using a hashing algorithm. Given an object key (for example, for a map) or an object name (for example, for a topic or list):

- the key or name is serialized (converted into a byte array),
- this byte array is hashed, and
- the result of the hash is mod by the number of partitions.

The result of this modulo - $\text{MOD}(\text{hash result}, \text{partition count})$ - gives the partition in which the data will be stored.

4.4.2 Partition Table

When you start a node, a partition table is created within it. This table stores the information for which partitions belong to which nodes. The purpose of this table is to make all nodes in the cluster aware of this information, making sure that each node knows where the data is.

The oldest node in the cluster (the one that started first) periodically sends the partition table to all nodes. In this way, each node in the cluster is informed about any changes to the partition ownership. The ownerships may be changed when, for example, a new node joins the cluster, or when a node leaves the cluster.

⚠️ **NOTE:** If the oldest node goes down, the next oldest node sends the partition table information to the other nodes.

You can configure the frequency (how often) that the node sends the partition table the information by using the `hazelcast.partition.table.send.interval` system property. The property is set to every 15 seconds by default.

4.4.3 Repartitioning

Repartitioning is the process of redistribution of partition ownerships. Hazelcast performs the repartitioning in the following cases:

- When a node joins to the cluster.
- When a node leaves the cluster.

In these cases, the partition table in the oldest node is updated with the new partition ownerships.

4.5 Use Cases

Some example usages are listed below. Hazelcast can be used:

- To share server configuration/information to see how a cluster performs,
- To cluster highly changing data with event notifications (e.g. user based events) and to queue and distribute background tasks,
- As a simple Memcache with near cache,
- As a cloud-wide scheduler of certain processes that need to be performed on some nodes,
- To share information (user information, queues, maps, etc.) on the fly with multiple nodes in different installations under OSGI environments,
- To share thousands of keys in a cluster where there is a web service interface on an application server and some validation,
• As a distributed topic (publish/subscribe server) to build scalable chat servers for smartphones,
• As a front layer for a Cassandra back-end,
• To distribute user object states across the cluster, to pass messages between objects and to share system data structures (static initialization state, mirrored objects, object identity generators),
• As a multi-tenancy cache where each tenant has its own map,
• To share datasets (e.g. table-like data structure) to be used by applications,
• To distribute the load and collect status from Amazon EC2 servers where front-end is developed using, for example, Spring framework,
• As a real time streamer for performance detection,
• As storage for session data in web applications (enables horizontal scalability of the web application).

4.6 Resources

• Hazelcast source code can be found at Github/Hazelcast.
• Hazelcast API can be found at Hazelcast.org/docs/Javadoc.
• Code samples can be downloaded from Hazelcast.org/download.
• More use cases and resources can be found at Hazelcast.com.
• Questions and discussions can be posted at Hazelcast mail group.
Chapter 5

Hazelcast Clusters

This chapter describes Hazelcast clusters and the ways cluster members use to form a Hazelcast cluster.

5.1 Discovering Cluster Members

A Hazelcast cluster is a network of cluster members that run Hazelcast. Cluster members (also called nodes) automatically join together to form a cluster. This automatic joining takes place with various discovery mechanisms that the cluster members use to find each other. Hazelcast uses the following discovery mechanisms.

- Multicast
- TCP
- EC2 Cloud

Each discovery mechanism is explained in the following sections.

**NOTE:** After a cluster is formed, communication between cluster members is always via TCP/IP, regardless of the discovery mechanism used.

5.1.1 Discovering Members by Multicast

With the multicast auto-discovery mechanism, Hazelcast allows cluster members to find each other using multicast communication. The cluster members do not need to know the concrete addresses of the other members, they just multicast to all the other members for listening. It depends on your environment if multicast is possible or allowed.

To set your Hazelcast to multicast auto-discovery, set the following configuration elements. Please refer to the multicast element section for the full description of the multicast discovery configuration elements.

- Set the enabled attribute of the multicast element to “true”.
- Set multicast-group, multicast-port, multicast-time-to-live, etc. to your multicast values.
- Set the enabled attribute of both tcp-ip and aws elements to “false”.

The following is an example declarative configuration.

```xml
<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>
  </join>
</network>
```
Pay attention to the `multicast-timeout-seconds` element. `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 the leader node (the first node joined to the cluster) and creating its own cluster. This only applies to the startup of nodes where no leader has been assigned yet. If you specify a high value to `multicast-timeout-seconds`, such as 60 seconds, it means that until a leader is selected, each node will wait 60 seconds before moving on. Be careful when providing a high value. Also be careful not to set the value too low, or the nodes might give up too early and create their own cluster.

### 5.1.2 Discovering Members by TCP

If multicast is not the preferred way of discovery for your environment, then you can configure Hazelcast to be a full TCP/IP cluster. When you configure Hazelcast to discover members by TCP/IP, you must list all or a subset of the members’ hostnames and/or IP addresses as cluster members. You do not have to list all of these cluster members, but at least one of the listed members has to be active in the cluster when a new member joins.

To set your Hazelcast to be a full TCP/IP cluster, set the following configuration elements. Please refer to the `tcp-ip` element section for the full description of the TCP/IP discovery configuration elements.

- Set the `enabled` attribute of the `multicast` element to “false”.
- Set the `enabled` attribute of the `aws` element to “false”.
- Set the `enabled` attribute of the `tcp-ip` element to “true”.
- Set your `member` elements within the `tcp-ip` element.

The following is an example declarative configuration.

```xml
<hazelcast>
  ...
  <network>
    ...
    <join>
      <multicast enabled="false"/>
    </join>
    <tcp-ip enabled="true">
      <member>machine1</member>
      <member>machine2</member>
      <member>machine3:5799</member>
      <member>192.168.1.0-7</member>
      <member>192.168.1.21</member>
    </tcp-ip>
    ...
    <join>
      ...
    </join>
  </network>
  ...
</hazelcast>
```
As shown above, you can provide IP addresses or hostnames for member elements. You can also give a range of IP addresses, such as 192.168.1.0-7.

Instead of providing members line by line as shown above, you also have the option to use the members element and write comma-separated IP addresses, as shown below.

<members>192.168.1.0-7,192.168.1.21</members>

If you do not provide ports for the members, Hazelcast automatically tries the ports 5701, 5702, and so on.

By default, Hazelcast binds to all local network interfaces to accept incoming traffic. You can change this behavior using the system property hazelcast.socket.bind.any. If you set this property to false, Hazelcast uses the interfaces specified in the interfaces element (please refer to the Interfaces Configuration section). If no interfaces are provided, then it will try to resolve one interface to bind from the member elements.

5.1.3 Discovering Members within EC2 Cloud

Hazelcast supports EC2 Auto Discovery. It is useful when you do not want to provide or you cannot provide the list of possible IP addresses.

To configure your cluster to use EC2 Auto Discovery, set the following configuration elements. Please refer to the aws element section for the full description of the EC2 Auto Discovery configuration elements.

- Add the hazelcast-cloud.jar dependency to your project. Note that it is also bundled inside hazelcast-all.jar. The Hazelcast cloud module does not depend on any other third party modules.
- Disable join over multicast and TCP/IP: set the enabled attribute of the multicast element to “false”, and set the enabled attribute of the tcp-ip element to “false”.
- Set the enabled attribute of the aws element to “true”.
- Within the aws element, provide your credentials (access and secret key), your region, etc.

The following is an example declarative configuration.

<join>
    <multicast enabled="false"/>
</multicast>
<tcp-ip enabled="false"/>
</tcp-ip>
<aws enabled="true">
    <access-key>my-access-key</access-key>
    <secret-key>my-secret-key</secret-key>
    <iam-role>s3access</iam-role>
    <region>us-west-1</region>
    <host-header>ec2.amazonaws.com</host-header>
    <security-group-name>hazelcast-sg</security-group-name>
    <tag-key>type</tag-key>
    <tag-value>hz-nodes</tag-value>
</aws>
</join>

5.1.3.1 Debugging

When needed, Hazelcast can log the events for the instances that exist in a region. To see what has happened or to trace the activities while forming the cluster, change the log level in your logging mechanism to FINEST or DEBUG. After this change, you can also see in the generated log whether the instances are accepted or rejected, and the reason the instances were rejected. Note that changing the log level in this way may affect the performance of the cluster. Please see the Logging Configuration section for information on logging mechanisms.

RELATED INFORMATION

You can download the white paper “Hazelcast on AWS: Best Practices for Deployment”* from Hazelcast.com.*
5.2 Creating Cluster Groups

You can create cluster groups. To do this, use the `group` configuration element.

By specifying a group name and group password, you can separate your clusters in a simple way. Example groupings can be by `development`, `production`, `test`, `app`, etc. The following is an example declarative configuration.

```xml
<hazelcast>
  <group>
    <name>app1</name>
    <password>app1-pass</password>
  </group>
  ...
</hazelcast>
```

You can also define the cluster groups using the programmatic configuration. A JVM can host multiple Hazelcast instances. Each Hazelcast instance can only participate in one group. Each Hazelcast instance only joins to its own group, it does not mess with other groups. The following code example creates three separate Hazelcast instances: h1 belongs to the app1 cluster, while h2 and h3 belong to the app2 cluster.

```java
Config configApp1 = new Config();
cfgApp1.getGroupConfig().setName( "app1" ).setPassword( "app1-pass" );

Config configApp2 = new Config();
cfgApp2.getGroupConfig().setName( "app2" ).setPassword( "app2-pass" );

HazelcastInstance h1 = Hazelcast.newHazelcastInstance( configApp1 );
HazelcastInstance h2 = Hazelcast.newHazelcastInstance( configApp2 );
HazelcastInstance h3 = Hazelcast.newHazelcastInstance( configApp2 );
```
Chapter 6

Distributed Data Structures

As mentioned in the Overview section, Hazelcast offers distributed implementations of Java interfaces. Below is the Java interface list with links to each section in this manual.

- **Standard utility collections:**
  - **Map:** The distributed implementation of `java.util.Map` lets you read from and write to a Hazelcast map with methods like get and put.
  - **Queue:** The distributed queue is an implementation of `java.util.concurrent.BlockingQueue`. You can add an item in one machine and remove it from another one.
  - **RingBuffer:** The distributed RingBuffer is implemented for reliable eventing system.
  - **Set:** The distributed and concurrent implementation of `java.util.Set`. It does not allow duplicate elements and does not preserve their order.
  - **List:** Very similar to Hazelcast List, except that it allows duplicate elements and preserves their order.
  - **MultiMap:** This is a specialized Hazelcast map. It is distributed, where multiple values under a single key can be stored.
  - **ReplicatedMap:** This does not partition data, i.e. it does not spread data to different cluster members. Instead, it replicates the data to all nodes.

- **Topic:** Distributed mechanism for publishing messages that are delivered to multiple subscribers; this is also known as a publish/subscribe (pub/sub) messaging model. Please see the Topic section for more information.

- **Concurrency utilities:**
  - **Lock:** Distributed implementation of `java.util.concurrent.locks.Lock`. When you lock using Hazelcast Lock, the critical section that it guards is guaranteed to be executed by only one thread in the entire cluster.
  - **Semaphore:** Distributed implementation of `java.util.concurrent.Semaphore`. When performing concurrent activities, semaphores offer permits to control the thread counts.
  - **AtomicLong:** Distributed implementation of `java.util.concurrent.atomic.AtomicLong`. Most of AtomicLong’s operations are available. However, these operations involve remote calls and hence their performances differ from AtomicLong, due to being distributed.
  - **AtomicReference:** When you need to deal with a reference in a distributed environment, you can use Hazelcast AtomicReference. This is the distributed version of `java.util.concurrent.atomic.AtomicReference`.
  - **IdGenerator:** You use Hazelcast IdGenerator to generate cluster-wide unique identifiers. ID generation occurs almost at the speed of `AtomicLong.incrementAndGet()`.
  - **CountDownLatch:** Distributed implementation of `java.util.concurrent.CountDownLatch`. Hazelcast CountDownLatch is a gate keeper for concurrent activities, enabling the threads to wait for other threads to complete their operations.

Common Features of all Hazelcast Data Structures:
42

CHAPTER 6. DISTRIBUTED DATA STRUCTURES
• If a member goes down, its backup replica (which holds the same data) will dynamically redistribute the data,
including the ownership and locks on them, to the remaining live nodes. As a result, no data will be lost.
• There is no single cluster master that can cause single point of failure. Every node in the cluster has equal
rights and responsibilities. No single node is superior. There is no dependency on an external ‘server’ or
‘master’.

Here is an example of how you can retrieve existing data structure instances (map, queue, set, lock, topic, etc.) and
how you can listen for instance events, such as an instance being created or destroyed.
import java.util.Collection;
import com.hazelcast.config.Config;
import com.hazelcast.core.*;
public class Sample implements DistributedObjectListener {
public static void main(String[] args) {
Sample sample = new Sample();
Config config = new Config();
HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance(config);
hazelcastInstance.addDistributedObjectListener(sample);
Collection<DistributedObject> distributedObjects = hazelcastInstance.getDistributedObjects();
for (DistributedObject distributedObject : distributedObjects) {
System.out.println(distributedObject.getName() + "," + distributedObject.getId());
}
}
@Override
public void distributedObjectCreated(DistributedObjectEvent event) {
DistributedObject instance = event.getDistributedObject();
System.out.println("Created " + instance.getName() + "," + instance.getId());
}
@Override
public void distributedObjectDestroyed(DistributedObjectEvent event) {
DistributedObject instance = event.getDistributedObject();
System.out.println("Destroyed " + instance.getName() + "," + instance.getId());
}
}

6.1
6.1.1

Map
Map Overview

Hazelcast Map (IMap) extends the interface java.util.concurrent.ConcurrentMap and hence java.util.Map.
It is the distributed implementation of Java map. You can perfrom operations like reading and writing from/to a
Hazelcast map with the well known get and put methods.
6.1.1.1

How Distributed Map Works

Hazelcast will partition your map entries and almost evenly distribute onto all Hazelcast members. Each member
carries approximately “(1/n * total-data) + backups”, n being the number of nodes in the cluster. For example, if
you have a node with 1000 objects to be stored in the cluster, and then you start a second node, each node will
both store 500 objects and back up the 500 objects in the other node.
Let’s create a Hazelcast instance (node) and fill a map named Capitals with key-value pairs using the following
code.


public class FillMapMember {
    public static void main( String[] args ) {
        HazelcastInstance hzInstance = Hazelcast.newHazelcastInstance();
        Map<String, String> capitalcities = hzInstance.getMap( "capitals" );
        capitalcities.put( "1", "Tokyo" );
        capitalcities.put( "2", "Paris" );
        capitalcities.put( "3", "Washington" );
        capitalcities.put( "4", "Ankara" );
        capitalcities.put( "5", "Brussels" );
        capitalcities.put( "6", "Amsterdam" );
        capitalcities.put( "7", "New Delhi" );
        capitalcities.put( "8", "London" );
        capitalcities.put( "9", "Berlin" );
        capitalcities.put( "10", "Oslo" );
        capitalcities.put( "11", "Moscow" );
        ...
        capitalcities.put( "120", "Stockholm" )
    }
}

When you run this code, a node is created with a map whose entries are distributed across the node’s partitions. See the below illustration. For now, this is a single node cluster.

**NOTE:** Please note that some of the partitions will not contain any data entries since we only have 120
objects and the partition count is 271 by default. This count is configurable and can be changed using the system property `hazelcast.partition.count`. Please see the System Properties section.

Now, let's create a second node by running the above code again. This will create a cluster with 2 nodes. This is also where backups of entries are created; remember the backup partitions mentioned in the Hazelcast Overview section. The following illustration shows two nodes and how the data and its backup is distributed.

As you see, when a new member joins the cluster, it takes ownership and loads some of the data in the cluster. Eventually, it will carry almost “(1/n * total-data) + backups” of the data, reducing the load on other nodes.

HazelcastInstance::getMap returns an instance of `com.hazelcast.core.IMap` which extends the `java.util.concurrent.ConcurrentMap` interface. Methods like `ConcurrentMap.putIfAbsent(key, value)` and `ConcurrentMap.replace(key, value)` can be used on the distributed map, as shown in the example below.

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import java.util.concurrent.ConcurrentMap;

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();

Customer getCustomer( String id ) {
    ConcurrentMap<String, Customer> customers = hazelcastInstance.getMap( "customers" );
    Customer customer = customers.get( id );
    if (customer == null) {
        customer = new Customer( id );
        customer = customers.putIfAbsent( id, customer );
    }
}
```
public boolean updateCustomer( Customer customer ) {
    ConcurrentMap<String, Customer> customers = hazelcastInstance.getMap( "customers" );
    return ( customers.replace( customer.getId(), customer ) != null );
}

public boolean removeCustomer( Customer customer ) {
    ConcurrentMap<String, Customer> customers = hazelcastInstance.getMap( "customers" );
    return customers.remove( customer.getId(), customer );
}

All ConcurrentMap operations such as put and remove might wait if the key is locked by another thread in the local or remote JVM. But, they will eventually return with success. ConcurrentMap operations never throw a java.util.ConcurrentModificationException.

Also see:

- Data Affinity section.
- Map Configuration with wildcards.
- Map Configuration section for a full description of Hazelcast Distributed Map configuration.

### 6.1.2 Map Backups

Hazelcast distributes map entries onto multiple JVMs (cluster members). Each JVM holds some portion of the data.

Distributed maps have 1 backup by default. If a member goes down, you do not lose data. Backup operations are synchronous, so when a map.put(key, value) returns, it is guaranteed that the entry is replicated to one other node. For the reads, it is also guaranteed that map.get(key) returns the latest value of the entry. Consistency is strictly enforced.

#### 6.1.2.1 Sync Backup

To provide data safety, Hazelcast allows you to specify the number of backup copies you want to have. That way, data on a JVM will be copied onto other JVM(s). You select the number of backup copies using the backup-count property.

```xml
<hazelcast>
  <map name="default">
    <backup-count>1</backup-count>
  </map>
</hazelcast>
```

When this count is 1, a map entry will have its backup on one other node in the cluster. If you set it to 2, then a map entry will have its backup on two other nodes. You can set it to 0 if you do not want your entries to be backed up, e.g. if performance is more important than backing up. The maximum value for the backup count is 6.

Hazelcast supports both synchronous and asynchronous backups. By default, backup operations are synchronous and configured with backup-count. In this case, backup operations block operations until backups are successfully copied to backup nodes (or deleted from backup nodes in case of remove) and acknowledgements are received. Therefore, backups are updated before a put operation is completed. Sync backup operations have a blocking cost which may lead to latency issues.
6.1.2.2 Async Backup

Asynchronous backups, on the other hand, do not block operations. They are fire & forget and do not require acknowledgements; the backup operations are performed at some point in time. Async backup is configured using the `async-backup-count` property. An example is shown below.

```xml
<hazelcast>
  <map name="default">
    <backup-count>0</backup-count>
    <async-backup-count>1</async-backup-count>
  </map>
</hazelcast>
```

**NOTE:** Backups increase memory usage since they are also kept in memory.

**NOTE:** A map can have both sync and async backups at the same time.

6.1.2.3 Read Backup Data

By default, Hazelcast has one sync backup copy. If `backup-count` is set to more than 1, then each member will carry both owned entries and backup copies of other members. So for the `map.get(key)` call, it is possible that the calling member has a backup copy of that key. By default, `map.get(key)` will always read the value from the actual owner of the key for consistency. It is possible to enable backup reads (read local backup entries) by setting the value of the `read-backup-data` property to `true`. Its default value is `false` for strong consistency. Enabling backup reads can improve performance.

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

This feature is available when there is at least 1 sync or async backup.

6.1.3 Map Eviction

Unless you delete the map entries manually or use an eviction policy, they will remain in the map. Hazelcast supports policy based eviction for distributed maps. Currently supported policies are LRU (Least Recently Used) and LFU (Least Frequently Used).

Map eviction works based on the size of a partition. For example, once you specify a size using the `PER_NODE` attribute for `max-size` (please see Configuring Map Eviction), Hazelcast internally calculates the maximum size for every partition. Eviction process starts according to this calculated per-partition maximum size when you try to put an entry. Below section gives an example scenario.

6.1.3.1 Example Map Eviction Scenario

Assume that you have the following figures:

- Partition count: 200
- Entry count for each partition: 100
6.1. MAP

- max-size (PER_NODE): 20000
- eviction-percentage (please see Configuring Map Eviction): 10%

The total number of entries here is 20000 (partition count * entry count for each partition). This means you are at the eviction threshold since you set the max-size to 20000. When you try to put an entry:

1. Entry goes to the relevant partition.
2. Partition checks whether the eviction threshold is reached (max-size).
3. If reached, approximately 10 (100 * 10%) entries are evicted from that particular partition.

As a result of this eviction process, when you check the size of your map, it is ~19990 (20000 - ~10). After this eviction, subsequent put operations will not trigger the next eviction until the map size is again close to the max-size.

**NOTE:** Above scenario is just an example to describe how the eviction process works. Hazelcast finds the most optimum number of entries to be evicted according to your cluster size and selected policy.

6.1.3.2 Configuring Map Eviction

The following is an example declarative configuration for map eviction.

```xml
<hazelcast>
  <map name="default">
    ...
    <time-to-live-seconds>0</time-to-live-seconds>
    <max-idle-seconds>0</max-idle-seconds>
    <eviction-policy>LRU</eviction-policy>
    <max-size policy="PER_NODE">5000</max-size>
    <eviction-percentage>25</eviction-percentage>
    <min-eviction-check-millis>100</min-eviction-check-millis>
    ...
  </map>
</hazelcast>
```

Let’s describe each element.

- **time-to-live**: Maximum time in seconds for each entry to stay in the map. If it is not 0, entries that are older than this time and not updated for this time are evicted automatically. Valid values are integers between 0 and Integer.MAX_VALUE. Default value is 0, which means infinite. If it is not 0, entries are evicted regardless of the set eviction-policy.

- **max-idle-seconds**: Maximum time in seconds for each entry to stay idle in the map. Entries that are idle for more than this time are evicted automatically. An entry is idle if no get, put or containsKey is called. Valid values are integers between 0 and Integer.MAX_VALUE. Default value is 0, which means infinite.

- **eviction-policy**: Valid values are described below.
  - NONE: Default policy. If set, no items will be evicted and the property max-size will be ignored. You still can combine it with time-to-live-seconds and max-idle-seconds.
  - LRU: Least Recently Used.
  - LFU: Least Frequently Used.

- **max-size**: Maximum size of the map. When maximum size is reached, the map is evicted based on the policy defined. Valid values are integers between 0 and Integer.MAX_VALUE. Default value is 0. If you want max-size to work, set the eviction-policy property to a value other than NONE. Its attributes are described below.
CHAPTER 6. DISTRIBUTED DATA STRUCTURES

- **PER_NODE**: Maximum number of map entries in each JVM. This is the default policy.
  
  `<max-size policy="PER_NODE">5000</max-size>

- **PER_PARTITION**: Maximum number of map entries within each partition. Storage size depends on the partition count in a JVM. This attribute should not be used often. Avoid using this attribute with a small cluster: if the cluster is small it will be hosting more partitions, and therefore map entries, than that of a larger cluster. Thus, for a small cluster, eviction of the entries will decrease performance (the number of entries is large).
  
  `<max-size policy="PER_PARTITION">27100</max-size>

- **USED_HEAP_SIZE**: Maximum used heap size in megabytes for each JVM.
  
  `<max-size policy="USED_HEAP_SIZE">4096</max-size>

- **USED_HEAP_PERCENTAGE**: Maximum used heap size percentage for each JVM. If, for example, JVM is configured to have 1000 MB and this value is 10, then the map entries will be evicted when used heap size exceeds 100 MB.
  
  `<max-size policy="USED_HEAP_PERCENTAGE">10</max-size`

- **FREE_HEAP_SIZE**: Minimum free heap size in megabytes for each JVM.
  
  `<max-size policy="FREE_HEAP_SIZE">512</max-size>

- **FREE_HEAP_PERCENTAGE**: Minimum free heap size percentage for each JVM. If, for example, JVM is configured to have 1000 MB and this value is 10, then the map entries will be evicted when free heap size is below 100 MB.
  
  `<max-size policy="FREE_HEAP_PERCENTAGE">10</max-size>

  * **eviction-percentage**: When max-size is reached, the specified percentage of the map will be evicted. For example, if set to 25, 25% of the entries will be evicted. Setting this property to a smaller value will cause eviction of a smaller number of map entries. Therefore, if map entries are inserted frequently, smaller percentage values may lead to overheads. Valid values are integers between 0 and 100. The default value is 25.

  * **min-eviction-check-millis**: The minimum time in milliseconds which should elapse before checking whether a partition of the map is evictable or not. In other terms, this property specifies the frequency of the eviction process. The default value is 100. Setting it to 0 (zero) makes the eviction process run for every put operation.

**NOTE:** When map entries are inserted frequently, the property **min-eviction-check-millis** should be set to a number lower than the insertion period in order not to let any entry escape from the eviction.

### 6.1.3.3 Sample Eviction Configuration

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

In the above sample, **documents** map starts to evict its entries from a member when the map size exceeds 10000 in that member. Then, the entries least recently used will be evicted. The entries not used for more than 60 seconds will be evicted as well.

### 6.1.3.4 Evicting Specific Entries

The eviction policies and configurations explained above apply to all the entries of a map. The entries that meet the specified eviction conditions are evicted.

But you may want to evict some specific map entries. In this case, you can use the **ttl** and **timeunit** parameters of the method **map.put()**. A sample code line is given below.

```
myMap.put( "1", "John", 50, TimeUnit.SECONDS )
```

The map entry with the key “1” will be evicted 50 seconds after it is put into **myMap**.
6.1.3.5 Evicting All Entries

The method `evictAll()` evicts all keys from the map except the locked ones. If a MapStore is defined for the map, `deleteAll` is not called by `evictAll`. If you want to call the method `deleteAll`, use `clear()`.

A sample is given below.

```java
public class EvictAll {

    public static void main(String[] args) {
        final int numberOfKeysToLock = 4;
        final int numberOfEntriesToAdd = 1000;

        HazelcastInstance node1 = Hazelcast.newHazelcastInstance();
        HazelcastInstance node2 = Hazelcast.newHazelcastInstance();

        IMap<Integer, Integer> map = node1.getMap(EvictAll.class.getCanonicalName());
        for (int i = 0; i < numberOfEntriesToAdd; i++) {
            map.put(i, i);
        }
        for (int i = 0; i < numberOfKeysToLock; i++) {
            map.lock(i);
        }
        // should keep locked keys and evict all others.
        map.evictAll();

        System.out.printf("# After calling evictAll...
        ");
        System.out.printf("# Expected map size\t: %d\n", numberOfKeysToLock);
        System.out.printf("# Actual map size\t: %d\n", map.size());
    }
}
```

NOTE: Only EVICT_ALL event is fired for any registered listeners.

6.1.4 In Memory Format

IMap has an in-memory-format configuration option. By default, Hazelcast stores data into memory in binary (serialized) format. But sometimes, it can be efficient to store the entries in their object form, especially in cases of local processing like entry processor and queries. By setting in-memory-format in map’s configuration, you can decide how the data will be stored in memory. You have the following format options.

- **BINARY** (default): This is the default option. The data will be stored in serialized binary format. You can use this option if you mostly perform regular map operations, such as `put` and `get`.

- **OBJECT**: The data will be stored in deserialized form. This configuration is good for maps where entry processing and queries form the majority of all operations and the objects are complex ones, making the serialization cost respectively high. By storing objects, entry processing will not contain the deserialization cost.

Regular operations like `get` rely on the object instance. When the **OBJECT** format is used and a `get` is performed, the map does not return the stored instance, but creates a clone. Therefore, this whole `get` operation includes a serialization first on the node owning the instance, and then a deserialization on the node calling the instance. When the **BINARY** format is used, only a deserialization is required; this is faster.
Similarly, a put operation is faster when the BINARY format is used. If the format was OBJECT, map would create a clone of the instance, and there would first a serialization and then deserialization. When BINARY is used, only a deserialization is needed.

**NOTE:** If a value is stored in OBJECT format, a change on a returned value does not affect the stored instance. In this case, the returned instance is not the actual one but a clone. Therefore, changes made on an object after it is returned will not reflect on the actual stored data. Similarly, when a value is written to a map and the value is stored in OBJECT format, it will be a copy of the put value. Therefore, changes made on the object after it is stored will not reflect on the stored data.

### 6.1.5 Map Persistence

Hazelcast allows you to load and store the distributed map entries from/to a persistent data store such as a relational database. To do this, you can use Hazelcast’s MapStore and MapLoader interfaces.

When you provide a MapLoader implementation and request an entry (IMap.get()) that does not exist in memory, MapLoader’s load or loadAll methods will load that entry from the data store. This loaded entry is placed into the map and will stay there until it is removed or evicted.

When a MapStore implementation is provided, an entry is also put into a user defined data store.

**NOTE:** Data store needs to be a centralized system that is accessible from all Hazelcast Nodes. Persistence to local file system is not supported.

**NOTE:** Also note that, the MapStore interface extends the MapLoader interface as you can see in the interface code.

Following is a MapStore example.

```java
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);
        }
    }

    public synchronized void delete(Long key) {
        System.out.println("Delete:" + key);
        try {
            con.createStatement().executeUpdate("delete from person where id = %s", key);
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

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

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

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

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);
    }
}

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;
}

public Iterable<Long> loadAllKeys() {
    return null;
}

NOTE: During the initial loading process, MapStore uses a thread different than the partition threads that
is used by the ExecutorService. After the initialization is completed, the map.get method looks up any inexistent
value from the database in a partition thread or the map.put method looks up the database to return the previously
associated value for a key also in a partition thread.

RELATED INFORMATION
For more MapStore/MapLoader code samples please see here.

Hazelcast supports read-through, write-through, and write-behind persistence modes which are explained in below
subsections.

6.1.5.1 Read-Through

If an entry does not exist in the memory when an application asks for it, Hazelcast asks your loader implementation
to load that entry from the data store. If the entry exists there, the loader implementation gets it, hands it to
Hazelcast, and Hazelcast puts it into the memory. This is read-through persistence mode.
6.1.5.2 Write-Through

MapStore can be configured to be write-through by setting the `write-delay-seconds` property to 0. This means the entries will be put to the data store synchronously.

In this mode, when the `map.put(key, value)` call returns:

- `MapStore.store(key, value)` is successfully called so the entry is persisted.
- In-Memory entry is updated.
- In-Memory backup copies are successfully created on other JVMs (if `backup-count` is greater than 0).

The same behavior goes for a `map.remove(key)` call. The only difference is that `MapStore.delete(key)` is called when the entry will be deleted.

If `MapStore` throws an exception, then the exception will be propagated back to the original `put` or `remove` call in the form of `RuntimeException`.

6.1.5.3 Write-Behind

You can configure MapStore as write-behind by setting the `write-delay-seconds` property to a value bigger than 0. This means the modified entries will be put to the data store asynchronously after a configured delay.

**NOTE:** In write-behind mode, by default Hazelcast coalesces updates on a specific key, i.e. applies only the last update on it. But, you can set `MapStoreConfig#setWriteCoalescing` to `FALSE` and you can store all updates performed on a key to the data store.

**NOTE:** When you set `MapStoreConfig#setWriteCoalescing` to `FALSE`, after you reached per-node maximum write-behind-queue capacity, subsequent put operations will fail with `ReachedMaxSizeException`. This exception will be thrown to prevent uncontrolled grow of write-behind queues. You can set per node maximum capacity with `GroupProperty#MAP_WRITE_BEHIND_QUEUE_CAPACITY`.

In this mode, when the `map.put(key, value)` call returns:

- In-Memory entry is updated.
- In-Memory backup copies are successfully created on other JVMs (if `backup-count` is greater than 0).
- The entry is marked as dirty so that after `write-delay-seconds`, it can be persisted with `MapStore.store(key, value)` call.
- For fault tolerance dirty entries are stored in a queue on the primary member and also on a back-up member.

The same behavior goes for the `map.remove(key)`, the only difference is that `MapStore.delete(key)` is called when the entry will be deleted.

If `MapStore` throws an exception, then Hazelcast tries to store the entry again. If the entry still cannot be stored, a log message is printed and the entry is re-queued.

For batch write operations, which are only allowed in write-behind mode, Hazelcast will call `MapStore.storeAll(map)` and `MapStore.deleteAll(collection)` to do all writes in a single call.

**NOTE:** If a map entry is marked as dirty, i.e. it is waiting to be persisted to the MapStore in a write-behind scenario, the eviction process forces the entry to be stored. By this way, you will have control on the number of entries waiting to be stored, and thus you can prevent a possible OutOfMemory exception.

**NOTE:** `MapStore` or `MapLoader` implementations should not use Hazelcast Map/Queue/MultiMap/List/Set operations. Your implementation should only work with your data store. Otherwise, you may get into deadlock situations.

Here is a sample configuration:
6.1. MAP

```xml
<hazelcast>
  ...
  <map name="default">
    ...
    <map-store enabled="true">
      <class-name>com.hazelcast.examples.DummyStore</class-name>
      <write-delay-seconds>60</write-delay-seconds>
      <write-batch-size>1000</write-batch-size>
      <write-coalescing>true</write-coalescing>
    </map-store>
  </map>
</hazelcast>

RELATED INFORMATION
Please refer to the Map Store section for the full Map Store configuration description.

6.1.5.4 MapStoreFactory And MapLoaderLifecycleSupport Interfaces

A configuration can be applied to more than one map using wildcards (see Using Wildcard), meaning that the configuration is shared among the maps. But MapStore does not know which entries to store when there is one configuration applied to multiple maps. To overcome this, Hazelcast provides the MapStoreFactory interface.

Using the MapStoreFactory interface, MapStores for each map can be created when a wildcard configuration is used. Sample code is shown below.

```java
Config config = new Config();
MapConfig mapConfig = config.getMapConfig( "*" );
MapStoreConfig mapStoreConfig = mapConfig.getMapStoreConfig();
mapStoreConfig.setFactoryImplementation(
    new MapStoreFactory<Object, Object>() {
        @Override
        public MapLoader<Object, Object> newMapStore( String mapName, Properties properties ) {
            return null;
        }
    });
```

If the configuration implements the MapLoaderLifecycleSupport interface, then the user can initialize the MapLoader implementation with the given map name, configuration properties, and the Hazelcast instance. See the following example code.

```java
public interface MapLoaderLifecycleSupport {

    /**
     * Initializes this MapLoader implementation. Hazelcast will call
     * this method when the map is first used on the
     * HazelcastInstance. Implementation can
     * initialize required resources for the implementing
     * mapLoader such as reading a config file and/or creating
     * database connection.
     */
    void init( HazelcastInstance hazelcastInstance, Properties properties, String mapName );

    /**
     * Hazelcast will call this method before shutting down.
     * This method can be overridden to cleanup the resources
     * held by this map loader implementation, such as closing the
     * database connections etc.
     */
```
6.1.5.5 Initialization On Startup

You can use the `MapLoader.loadAllKeys` API to pre-populate the in-memory map when the map is first touched/used. If `MapLoader.loadAllKeys` returns NULL then nothing will be loaded. Your `MapLoader.loadAllKeys` implementation can return all or some of the keys. For example, you may select and return only the hot keys. `MapLoader.loadAllKeys` is the fastest way of pre-populating the map since Hazelcast will optimize the loading process by having each node loading its owned portion of the entries.

The `InitialLoadMode` configuration parameter in the class `MapStoreConfig` has two values: LAZY and EAGER. If `InitialLoadMode` is set to LAZY, data is not loaded during the map creation. If it is set to EAGER, the whole data is loaded while the map is created and everything becomes ready to use. Also, if you add indices to your map with the `MapIndexConfig` class or the `addIndex` method, then `InitialLoadMode` is overridden and `MapStoreConfig` behaves as if EAGER mode is on.

Here is the `MapLoader` initialization flow:

1. When `getMap()` is first called from any node, initialization will start depending on the value of `InitialLoadMode`. If it is set to EAGER, initialization starts. If it is set to LAZY, initialization does not start but data is loaded each time a partition loading completes.
2. Hazelcast will call `MapLoader.loadAllKeys()` to get all your keys on one of the nodes.
3. That node will distribute keys to all other nodes in batches.
4. Each node will load values of all its owned keys by calling `MapLoader.loadAll(keys)`.
5. Each node puts its owned entries into the map by calling `IMap.putTransient(key, value)`.

**NOTE:** If the load mode is LAZY and when the `clear()` method is called (which triggers `MapStore.deleteAll()`), Hazelcast will remove ONLY the loaded entries from your map and datastore. Since the whole data is not loaded for this case (LAZY mode), please note that there may be still entries in your datastore.

**NOTE:** The return type of `loadAllKeys()` is changed from `Set` to `Iterable` with the release of Hazelcast 3.5. `MapLoader` implementations from previous releases are also supported and do not need to be adapted.

#### Incremental Key Loading

If the number of keys to load is large, it is more efficient to load them incrementally than loading them all at once. To support incremental loading, `MapLoader.loadAllKeys()` returns an `Iterable` which can be lazily populated with results of a database query. Hazelcast iterates over the iterable and, while doing so, sends out the keys to their respective owner nodes. The `Iterator` obtained from `MapLoader.loadAllKeys()` may also implement the `Closeable` interface in which case it is closed once the iteration is over. This is intended for releasing resources such as closing a JDBC result set.

6.1.5.6 Forcing All Keys To Be Loaded

The method `loadAll` loads some or all keys into a data store in order to optimize the multiple load operations. The method has two signatures (i.e. the same method can take two different parameter lists). One signature loads the given keys and the other loads all keys. Please see the sample code below.

```java
public class LoadAll {

    public static void main(String[] args) {
        final int numberOfEntriesToAdd = 1000;
        final String mapName = LoadAll.class.getCanonicalName();
        final Config config = createNewConfig(mapName);
        final HazelcastInstance node = Hazelcast.newHazelcastInstance(config);

        void destroy();
    }
```
final IMap<Integer, Integer> map = node.getMap(mapName);

populateMap(map, numberOfEntriesToAdd);
System.out.printf("# Map store has %d elements\n", numberOfEntriesToAdd);

map.evictAll();
System.out.printf("# After evictAll map size\t: %d\n", map.size());

map.loadAll(true);
System.out.printf("# After loadAll map size\t: %d\n", map.size());
}

6.1.5.7 Post Processing Map Store

In some scenarios, you may need to modify the object after storing it into the map store. For example, you can get an ID or version auto generated by your database and then you need to modify your object stored in the distributed map but not to break the sync between database and data grid. You can do that by implementing the PostProcessingMapStore interface to put the modified object into the distributed map. That will cause an extra step of Serialization, so use it only when needed. (This explanation is only valid when using the write-through map store configuration.)

Here is an example of post processing map store:

class ProcessingStore implements MapStore<Integer, Employee>, PostProcessingMapStore {

@Override
    public void store( Integer key, Employee employee ) {
        EmployeeId id = saveEmployee();
        employee.setId( id.getId() );
    }
}

** Note: ** Please be warned that if you are using a post processing map store in combination with entry processors, post-processed values will not be carried to backups.

6.1.6 Near Cache

Map entries in Hazelcast are partitioned across the cluster. Imagine that you are reading the key k so many times and k is owned by another member in your cluster. Each map.get(k) will be a remote operation, meaning lots of network trips. If you have a map that is read-mostly, then you should consider creating a near cache for the map so that reads can be much faster and consume less network traffic. All these benefits do not come free. When using near cache, you should consider the following issues:

- JVM will have to hold extra cached data so it will increase the memory consumption.
- If invalidation is turned on and entries are updated frequently, then invalidations will be costly.
- Near cache breaks the strong consistency guarantees; you might be reading stale data.

Near cache is highly recommended for the maps that are read-mostly. Here is a near cache configuration for a map:

<hazelcast>
...
<map name="my-read-mostly-map">
...
<near-cache>
<!--
Maximum size of the near cache. When max size is reached,
cache is evicted based on the policy defined.
Any integer between 0 and Integer.MAX_VALUE. 0 means
Integer.MAX_VALUE. Default is 0.
-->
<max-size>5000</max-size>

<!--
Maximum number of seconds for each entry to stay in the near cache. Entries that are
older than <time-to-live-seconds> will get automatically evicted from the near cache.
Any integer between 0 and Integer.MAX_VALUE. 0 means
Integer.MAX_VALUE. Default is 0.
-->
<time-to-live-seconds>0</time-to-live-seconds>

<!--
Maximum number of seconds each entry can stay in the near cache as untouched (not-read).
Entries that are not read (touched) more than <max-idle-seconds> value will get removed
from the near cache.
Any integer between 0 and Integer.MAX_VALUE. 0 means
Integer.MAX_VALUE. Default is 0.
-->
<max-idle-seconds>60</max-idle-seconds>

<!--
Valid values are:
NONE (no extra eviction, <time-to-live-seconds> may still apply),
LRU (Least Recently Used),
LFU (Least Frequently Used).
NONE is the default.
Regardless of the eviction policy used, <time-to-live-seconds> will still apply.
-->
<eviction-policy>LRU</eviction-policy>

<!--
Should the cached entries get evicted if the entries are changed (updated or removed).
true or false. Default is true.
-->
<invalidate-on-change>true</invalidate-on-change>

<!--
You may want also local entries to be cached.
This is useful when in memory format for near cache is different than the map’s one.
By default it is disabled.
-->
<cache-local-entries>false</cache-local-entries>

</near-cache>
</map>
</hazelcast>

**NOTE:** Programmatically, near cache configuration is done by using the class NearCacheConfig. And this
class is used both in the nodes and clients. In a client/server system, you must enable the near cache separately
on the client, without needing to configure it on the server. For information on how to create a near cache on a
client (native Java client), please see the Client Near Cache Configuration section. Please note that near cache
configuration is specific to the node or client itself; a map in a node may not have near cache configured while the
same map in a client may have.
6.1. MAP

**NOTE:** If you are using near cache, you should take into account that your hits to the keys in near cache are not reflected as hits to the original keys on the remote nodes; this has an impact on IMap’s maximum idle seconds or time-to-live seconds expiration. Therefore, even there is a hit on a key in near cache, your original key on the remote node may expire.

**NOTE:** Near cache works only when you access data via `map.get(k)` methods. Data returned using a predicate is not stored in the near cache.

### 6.1.7 Map Locks

Hazelcast Distributed Map (IMap) is thread-safe to meet your thread safety requirements. When these requirements increase or you want to have more control on the concurrency, consider the following Hazelcast features and solutions.

Let’s work on a sample case as shown below.

```java
public class RacyUpdateMember {
    public static void main( String[] args ) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IMap<String, Value> map = hz.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.amount++;
            map.put( key, value );
        }
        System.out.println( "Finished! Result = " + map.get(key).amount );
    }

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

If the above code is run by more than one cluster member simultaneously, there will be likely a race condition. You can solve this with Hazelcast.

#### 6.1.7.1 Pessimistic Locking

One way to solve the race issue is the lock mechanism provided by Hazelcast distributed map, i.e. the `map.lock` and `map.unlock` methods. You simply lock the entry until you are finished with it. See the below sample code.

```java
public class PessimisticUpdateMember {
    public static void main( String[] args ) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IMap<String, Value> map = hz.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.amount++;
    map.put( key, value );
} finally {
    map.unlock( key );
}
}
System.out.println( "Finished! Result = " + map.get( key ).amount );

static class Value implements Serializable {
    public int amount;
}
}

The IMap lock will automatically be collected by the garbage collector when the lock is released and no other
waiting conditions exist on the lock.

The IMap lock is reentrant, but it does not support fairness.

Another way to solve the race issue can be acquiring a predictable Lock object from Hazelcast. This way, every
value in the map can be given a lock or you can create a stripe of locks.

6.1.7.2 Optimistic Locking

The Hazelcast way of optimistic locking is to use the map.replace method. See the below sample code.

public class OptimisticMember {
    public static void main( String[] args ) throws Exception {
        HazelcastInstance hz = Hazelcast.newHazelcastInstance();
        IMap<String, Value> map = hz.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.amount++;
                if ( map.replace( key, oldValue, newValue ) )
                    break;
            }
        }
        System.out.println( "Finished! Result = " + map.get( key ).amount );
    }
}

static class Value implements Serializable {
    public int amount;

    public Value() {
    }

    public Value( Value that ) {
        this.amount = that.amount;
    }
}
6.1. MAP

```java
public boolean equals( Object o ) {
    if ( o == this ) return true;
    if ( !( o instanceof Value ) ) return false;
    Value that = ( Value ) o;
    return that.amount == this.amount;
}
```

NOTE: Above sample code is intentionally broken.

6.1.7.3 Pessimistic vs. Optimistic Locking

Depending on the locking requirements, one locking strategy can be picked.

Optimistic locking is better for mostly read only systems. It has a performance boost over pessimistic locking.

Pessimistic locking is good if there are lots of updates on the same key. It is more robust than optimistic locking from the perspective of data consistency. In Hazelcast, use `IExecutorService` to submit a task to a key owner, or to a member or members. This is the recommended way to perform task executions that use pessimistic or optimistic locking techniques. `IExecutorService` will have less network hops and less data over wire, and tasks will be executed very near to the data. Please refer to the Data Affinity section.

6.1.7.4 ABA Problem

The ABA problem occurs in environments when a shared resource is open to change by multiple threads. Even if one thread sees the same value for a particular key in consecutive reads, it does not mean nothing has changed between the reads. Another thread may come and change the value, do work, and change the value back, but the first thread can think that nothing has changed.

To prevent these kind of problems, one solution is to use a version number and to check it before any write to be sure that nothing has changed between consecutive reads. Although all the other fields will be equal, the version field will prevent objects from being seen as equal. This is the optimistic locking strategy, and it is used in environments which do not expect intensive concurrent changes on a specific key.

In Hazelcast, you can apply optimistic locking strategy with the map `replace` method. This method compares values in object or data forms depending on the in-memory format configuration. If the values are equal, it replaces the old value with the new one. If you want to use your defined `equals` method, in-memory format should be `Object`. Otherwise, Hazelcast serializes objects to binary forms and compares them.

6.1.8 Entry Statistics

Hazelcast keeps extra information about each map entry, such as creation time, last update time, last access time, number of hits, and version. This information is exposed to the developer via a `IMap.getEntryView(key)` call. Here is an example:

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.EntryView;

HazelcastInstance hz = Hazelcast.newHazelcastInstance();
EntryView entry = hz.getMap( "quotes" ).getEntryView( "1" );
System.out.println ( "size in memory : " + entry.getCost() );
System.out.println ( "creationTime : " + entry.getCreationTime() );
System.out.println ( "expirationTime : " + entry.getExpirationTime() );
```
System.out.println ( "number of hits : " + entry.getHits() );
System.out.println ( "lastAccessedTime: " + entry.getLastAccessTime() );
System.out.println ( "lastUpdateTime : " + entry.getLastUpdateTime() );
System.out.println ( "version : " + entry.getVersion() );
System.out.println ( "key : " + entry.getKey() );
System.out.println ( "value : " + entry.getValue() );

6.1.9 Map Listener

You can listen to map-wide or entry-based events by implementing a MapListener sub-interface. A map-wide event is fired as a result of a map-wide operation: for example, IMap#clear or IMap#evictAll. An entry-based event is fired after the operations that affect a specific entry: for example, IMap#remove or IMap#evict.

Let’s take a look at the following code sample. To catch an event, you should explicitly implement a corresponding sub-interface of a MapListener, such as EntryAddedListener or MapClearedListener.

NOTE: EntryListener interface still can be implemented, we kept that as is due to backward compatibility reasons. However, if you need to listen to a different event which is not available in the EntryListener interface, you should also implement a relevant MapListener sub-interface.

```java
public class Listen {
    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 EntryAddedListener<String, String>,
        EntryRemovedListener<String, String>,
        EntryUpdatedListener<String, String>,
        EntryEvictedListener<String, String>,
        MapEvictedListener,
        MapClearedListener {

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

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

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

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

    @Override
    public void mapEvicted( MapEvent event ) {
        System.out.println( "Map Evicted:" + event );
    }
}
```
Now, let's perform some modifications on the map entries using the following example code.

```java
public class Modify {

    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);
    }
}
```

If you execute the `Listen` class and then the `Modify` class, you get the following output produced by the `Listen` class.

```
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
```

```java
public class MyEntryListener implements EntryListener{
    private Executor executor = Executors.newFixedThreadPool(5);

    @Override
    public void entryAdded(EntryEvent event) {
        executor.execute(new DoSomethingWithEvent(event));
    }
}
```

A map listener runs on the event threads that are also used by the other listeners: for example, the collection listeners and pub/sub message listeners. This means that the entry listeners can access other partitions. Consider this when you run long tasks, since listening to those tasks may cause the other map/event listeners to starve.

### 6.1.9.1 MapPartitionLostListener

You can listen to `MapPartitionLostEvent` instances by registering an implementation of `MapPartitionLostListener`, which is also a sub-interface of `MapListener`.

Let's consider the following example code:
public static void main(String[] args) {
    Config config = new Config();
    config.getMapConfig("map").setBackupCount(1); // might lose data if any node crashes

    HazelcastInstance instance = HazelcastInstanceFactory.newHazelcastInstance(config);

    IMap<Object, Object> map = instance.getMap("map");
    map.put(0, 0);

    map.addPartitionLostListener(new MapPartitionLostListener() {
        @Override
        public void partitionLost(MapPartitionLostEvent event) {
            System.out.println(event);
        }
    });
}

Within this example code, a `MapPartitionLostListener` implementation is registered to a map that is configured with 1 backup. For this particular map and any of the partitions in the system, if the partition owner node and its first backup node crash simultaneously, the given `MapPartitionLostListener` receives a corresponding `MapPartitionLostEvent`. If only a single node crashes in the cluster, there will be no `MapPartitionLostEvent` fired for this map since backups for the partitions owned by the crashed node are kept on other nodes.

Please refer to the Partition Lost Listener section for more information about partition lost detection and partition lost events.

### 6.1.10 Interceptors

You can add intercept operations and then execute your own business logic synchronously blocking the operations. You can change the returned value from a `get` operation, change the value to be `put` or `cancel` operations by throwing an exception.

Interceptor actions are different from listeners. With listeners, you take an action after the operation has been completed. Interceptor actions are synchronous and you can alter the behavior of operation, change the values, or totally cancel it.

Map interceptors are chained, so adding the same interceptor multiple times to the same map can result in duplicate effects. This can easily happen when the interceptor is added to the map at node initialization, so that each node adds the same interceptor. When adding the interceptor in this way, be sure that the `hashCode()` method is implemented to return the same value for every instance of the interceptor. It is not strictly necessary, but it is a good idea to also implement `equals()` as this will ensure that the map interceptor can be removed reliably.

IMap API has two methods for adding and removing an interceptor to the map, `addInterceptor` and `removeInterceptor`:

```java
/**
 * Adds an interceptor for this map. Added interceptor will intercept operations
 * and execute user defined methods and will cancel operations if user defined method throw exception.
 *
 * @param interceptor map interceptor
 * @return id of registered interceptor
 */
String addInterceptor(MapInterceptor interceptor);
```

```java
/**
 * Removes the given interceptor for this map. So it will not intercept operations anymore.
 *
 */
```

Here is the MapInterceptor interface:

```java
public interface MapInterceptor extends Serializable {

    /**
     * Intercept the get operation before it returns a value.
     * Return another object to change the return value of get(..).
     * Returning null will cause the get(..) operation to return the original value,
     * namely return null if you do not want to change anything.
     *
     * @param value the original value to be returned as the result of get(..) operation
     * @return the new value that will be returned by get(..) operation
     */
    Object interceptGet( Object value );

    /**
     * Called after get(..) operation is completed.
     *
     * @param value the value returned as the result of get(..) operation
     */
    void afterGet( Object value );

    /**
     * Intercept put operation before modifying map data.
     * Return the object to be put into the map.
     * Returning null will cause the put(..) operation to operate as expected,
     * namely no interception. Throwing an exception will cancel the put operation.
     *
     * @param oldValue the value currently in map
     * @param newValue the new value to be put
     * @return new value after intercept operation
     */
    Object interceptPut( Object oldValue, Object newValue );

    /**
     * Called after put(..) operation is completed.
     *
     * @param value the value returned as the result of put(..) operation
     */
    void afterPut( Object value );

    /**
     * Intercept remove operation before removing the data.
     * Return the object to be returned as the result of remove operation.
     * Throwing an exception will cancel the remove operation.
     *
     * @param removedValue the existing value to be removed
     * @return the value to be returned as the result of remove operation
     */
}
```
Object interceptRemove( Object removedValue );

/**
 * Called after remove(..) operation is completed.
 *
 * @param value the value returned as the result of remove(..) operation
 */
void afterRemove( Object value );

Example Usage:

public class InterceptorTest {

@Test
public void testMapInterceptor() throws InterruptedException {
    HazelcastInstance hazelcastInstance1 = Hazelcast.newHazelcastInstance();
    HazelcastInstance hazelcastInstance2 = Hazelcast.newHazelcastInstance();
    IMap<Object, Object> map = hazelcastInstance1.getMap( "testMapInterceptor" );
    SimpleInterceptor interceptor = new SimpleInterceptor();
    map.addInterceptor( interceptor );
    map.put( 1, "New York" );
    map.put( 2, "Istanbul" );
    map.put( 3, "Tokyo" );
    map.put( 4, "London" );
    map.put( 5, "Paris" );
    map.put( 6, "Cairo" );
    map.put( 7, "Hong Kong" );

    try {
        map.remove( 1 );
    } catch ( Exception ignore ) {
    }
    try {
        map.remove( 2 );
    } catch ( Exception ignore ) {
    }
    assertEquals( map.size(), 6 ) ;
    assertEquals( map.get( 1 ), null );
    assertEquals( map.get( 2 ), "ISTANBUL:" );
    assertEquals( map.get( 3 ), "TOKYO:" );
    assertEquals( map.get( 4 ), "LONDON:" );
    assertEquals( map.get( 5 ), "PARIS:" );
    assertEquals( map.get( 6 ), "CAIRO:" );
    assertEquals( map.get( 7 ), "HONG KONG:" );

    map.removeInterceptor( interceptor );
    map.put( 8, "Moscow" );

    assertEquals( map.get( 8 ), "Moscow" );
    assertEquals( map.get( 1 ), null );
    assertEquals( map.get( 2 ), "ISTANBUL" );
    assertEquals( map.get( 3 ), "TOKYO" );
    assertEquals( map.get( 4 ), "LONDON" );
    assertEquals( map.get( 5 ), "PARIS" );
}
```java
assertEquals( map.get( 6 ), "CAIRO" );
assertEquals( map.get( 7 ), "HONG KONG" );
}

static class SimpleInterceptor implements MapInterceptor, Serializable {
    @Override
    public Object interceptGet( Object value ) {
        if (value == null)
            return null;
        return value + ":";
    }
    
    @Override
    public void afterGet( Object value ) {
    }
    
    @Override
    public Object interceptPut( Object oldValue, Object newValue ) {
        return newValue.toString().toUpperCase();
    }
    
    @Override
    public void afterPut( Object value ) {
    }
    
    @Override
    public Object interceptRemove( Object removedValue ) {
        if (removedValue.equals( "ISTANBUL" ))
            throw new RuntimeException( "you can not remove this" );
        return removedValue;
    }
    
    @Override
    public void afterRemove( Object value ) {
        // do something
    }
}

6.1.11 Preventing Out of Memory Exceptions

It is very easy to trigger an out of memory exception (OOME) with query based map methods, especially with large clusters or heap sizes. For example, on a 5 node cluster with 10 GB of data and 25 GB heap size per node, a single call of `IMap.entrySet()` fetches 50 GB of data and crashes the calling instance.

A call of `IMap.values()` may return too much data for a single node. This can also happen with a real query and an unlucky choice of predicates, especially when the parameters are chosen by a user of your application.

To prevent this, you can configure a maximum result size limit for query based operations. This is not a limit like `SELECT * FROM map LIMIT 100`, which you can achieve by a Paging Predicate. A maximum result size limit for query based operations is meant to be a last line of defense to prevent your nodes from retrieving more data than they can handle.

The Hazelcast component which calculates this limit is the `QueryResultSizeLimiter`.
6.1.11.1 Setting Query Result Size Limit

If the QueryResultSizeLimiter is activated, it calculates a result size limit per partition. Each QueryOperation runs on all partitions of a node, so it collects result entries as long as the node limit is not exceeded. If that happens, a QueryResultSizeExceededException is thrown and propagated to the calling instance.

This feature depends on an equal distribution of the data on the cluster nodes to calculate the result size limit per node. Therefore, there is a minimum value defined in QueryResultSizeLimiter.MINIMUM_MAX_RESULT_LIMIT. Configured values below the minimum will be increased to the minimum.

6.1.11.1.1 Local Pre-check

In addition to the distributed result size check in the QueryOperations, there is a local pre-check on the calling instance. If you call the method from a client, the pre-check is executed on the member which invokes the QueryOperations.

Since the local pre-check can increase the latency of a QueryOperation you can configure how many local partitions should be considered for the pre-check or you can deactivate the feature completely.

6.1.11.1.2 Scope of Result Size Limit

Besides the designated query operations, there are other operations which use predicates internally. Those method calls will throw the QueryResultSizeExceededException as well. Please see the following matrix to see the methods that are covered by the query result size limit.

<table>
<thead>
<tr>
<th>Method</th>
<th>MapProxyImpl</th>
<th>ClientMapProxyImpl</th>
<th>TransactionalMapProxy</th>
<th>ClientTxnMapProxy</th>
</tr>
</thead>
<tbody>
<tr>
<td>values()</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>keySet()</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>entrySet()</td>
<td>✓</td>
<td>x</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>values(predicate)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>keySet(predicate)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>entrySet(predicate)</td>
<td>✓</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>localKeySet()</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>localKeySet(predicate)</td>
<td>✓</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Interfaces: IMap TransactionalMap

6.1.11.1.3 Configuring Query Result Size

The query result size limit is configured via the following system properties.

- hazelcast.query.result.size.limit
- hazelcast.query.max.local.partition.limit.for.precheck

Please refer to the System Properties section for explanations of these properties.

6.2 Queue

6.2.1 Queue Overview

Hazelcast distributed queue is an implementation of java.util.concurrent.BlockingQueue. Being distributed, it enables all cluster members to interact with it. Using Hazelcast distributed queue, you can add an item in one machine and remove it from another one.
import com.hazelcast.core.Hazelcast;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
BlockingQueue<MyTask> queue = hazelcastInstance.getQueue( "tasks" );
queue.put( new MyTask() );
MyTask task = queue.take();

boolean offered = queue.offer( new MyTask(), 10, TimeUnit.SECONDS );
task = queue.poll( 5, TimeUnit.SECONDS );
if ( task != null ) {
    //process task
}

FIFO ordering will apply to all queue operations across the cluster. User objects (such as MyTask in the example above) that are enqueued or dequeued have to be Serializable.

Hazelcast distributed queue performs no batching while iterating over the queue. All items will be copied locally and iteration will occur locally.

### 6.2.2 Sample Queue Code

The following sample code illustrates a producer and consumer connected by a distributed queue.

Let’s put one integer on the queue every second, 100 integers total.

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.IQueue;
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!" );
    }
}
```

Producer puts a -1 on the queue to show that the put’s are finished. Now, let’s create a Consumer class that take a message from this queue, as shown below.

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.IQueue;
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!");
}

As seen in the above sample code, Consumer waits 5 seconds before it consumes the next message. It stops once it receives -1. Also note that Consumer puts -1 back on the queue before the loop is ended.

When you first start Producer and then start Consumer, items produced on the queue will be consumed from the same queue.

From the above sample code, you can see that an item is produced every second, and consumed every 5 seconds. Therefore, the consumer keeps growing. To balance the produce/consume operation, let’s start another consumer. By this way, consumption is distributed to these two consumers, as seen in the sample outputs below.

The second consumer is started. After a while, here is the first consumer output:

...  
Consumed 13
Consumed 15
Consumer 17
...

Here is the second consumer output:

...  
Consumed 14
Consumed 16
Consumer 18
...

In the case of a lot of producers and consumers for the queue, using a list of queues may solve the queue bottlenecks. In this case, be aware that the order of the messages being sent to different queues is not guaranteed. Since in most cases strict ordering is not important, a list of queues is a good solution.

**NOTE:** The items are taken from the queue in the same order they were put on the queue. However, if there is more than one consumer, this order is not guaranteed.

### 6.2.3 Bounded Queue

A bounded queue is a queue with a limited capacity. When the bounded queue is full, no more items can be put into the queue until some items are taken out.

A Hazelcast distributed queue can be turned into a bounded queue by setting the capacity limit using the max-size property.

Queue capacity can be set using the max-size property in the configuration, as shown below. max-size specifies the maximum size of the queue. Once the queue size reaches this value, put operations will be blocked until the queue size goes below max-size, that happens when a consumer removes items from the queue.

Let’s set 10 as the maximum size of our sample queue in the Sample Queue Code.
When the producer is started, 10 items are put into the queue and then the queue will not allow more put operations. When the consumer is started, it will remove items from the queue. This means that the producer can put more items into the queue until there are 10 items in the queue again, at which point put operation again become blocked.

But in this sample code, the producer is 5 times faster than the consumer. It will effectively always be waiting for the consumer to remove items before it can put more on the queue. For this sample code, if maximum throughput was the goal, it would be a good option to start multiple consumers to prevent the queue from filling up.

### 6.2.4 Queue Persistence

Hazelcast allows you to load and store the distributed queue items from/to a persistent datastore using the interface `QueueStore`. If queue store is enabled, each item added to the queue will also be stored at the configured queue store. When the number of items in the queue exceeds the memory limit, the subsequent items are persisted in the queue store, they are not stored in the queue memory.

`QueueStore` interface enables you to store, load, and delete items with methods like `store`, `storeAll`, `load` and `delete`. The following example class includes all of the `QueueStore` methods.

```java
public class TheQueueStore implements QueueStore<Item> {
    @Override
    public void delete(Long key) {
        System.out.println("delete");
    }

    @Override
    public void store(Long key, Item value) {
        System.out.println("store");
    }

    @Override
    public void storeAll(Map<Long, Item> map) {
        System.out.println("store all");
    }

    @Override
    public void deleteAll(Collection<Long> keys) {
        System.out.println("deleteAll");
    }

    @Override
    public Item load(Long key) {
        System.out.println("load");
        return null;
    }

    @Override
    public Map<Long, Item> loadAll(Collection<Long> keys) {
        System.out.println("loadAll");
        return null;
    }
}
```
@Override
public Set<Long> loadAllKeys() {
    System.out.println("loadAllKeys");
    return null;
}

Item must be serializable. Following is an example queue store configuration.

<queue-store>
    <class-name>com.hazelcast.QueueStoreImpl</class-name>
    <properties>
        <property name="binary">false</property>
        <property name="memory-limit">1000</property>
        <property name="bulk-load">500</property>
    </properties>
</queue-store>

Let's explain the properties.

- **Binary**: By default, Hazelcast stores the queue items in serialized form in memory. Before it inserts the queue items into datastore, it deserializes them. But if you will not reach the queue store from an external application, you might prefer that the items be inserted in binary form. You can get rid of the de-serialization step; this would be a performance optimization. The binary feature is disabled by default.

- **Memory Limit**: This is the number of items after which Hazelcast will store items only to datastore. For example, if the memory limit is 1000, then the 1001st item will be put only to datastore. This feature is useful when you want to avoid out-of-memory conditions. The default number for memory-limit is 1000. If you want to always use memory, you can set it to Integer.MAX_VALUE.

- **Bulk Load**: When the queue is initialized, items are loaded from QueueStore in bulks. Bulk load is the size of these bulks. By default, bulk-load is 250.

6.2.5 Configuring Queue

An example declarative configuration is shown below.

<hazelcast>
    ...
    <queue name="tasks">
        <max-size>10</max-size>
        <backup-count>1</backup-count>
        <async-backup-count>1</async-backup-count>
        <empty-queue-ttl>10</empty-queue-ttl>
    </queue>
</hazelcast>

Hazelcast distributed queue has one synchronous backup by default. By having this backup, when a cluster member with a queue goes down, another member having the backups will continue. Therefore, no items are lost. You can define the count of synchronous backups using the backup-count element in the declarative configuration. A queue can also have asynchronous backups, you can define the count using the async-backup-count element.

The max-size element defines the maximum size of the queue. You can use the empty-queue-ttl element when you want to purge unused or empty queues after a period of time. If you define a value (time in seconds) for this element, then your queue will be destroyed if it stays empty or unused for the time you give.

**RELATED INFORMATION**

Please refer to the Queue Configuration section for a full description of Hazelcast Distributed Queue configuration.
6.3 MultiMap

Hazelcast MultiMap is a specialized map where you can store multiple values under a single key. Just like any other distributed data structure implementation in Hazelcast, MultiMap is distributed and thread-safe.

Hazelcast MultiMap is not an implementation of java.util.Map due to the difference in method signatures. It supports most features of Hazelcast Map except for indexing, predicates and MapLoader/MapStore. Yet, like Hazelcast Map, entries are almost evenly distributed onto all cluster members. When a new member joins the cluster, the same ownership logic used in the distributed map applies.

6.3.1 Sample MultiMap Code

Let’s write code that puts data into a MultiMap.

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

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

Now let’s print the entries in this MultiMap.

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

After you run the first code sample, run the PrintMember sample. You will see the key a has two values, as shown below.

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

6.3.2 Configuring MultiMap

When using MultiMap, the collection type of the values can be either Set or List. You configure the collection type with the valueCollectionType parameter. If you choose Set, duplicate and null values are not allowed in your collection and ordering is irrelevant. If you choose List, ordering is relevant and your collection can include duplicate and null values.

You can also enable statistics for your MultiMap with the statisticsEnabled parameter. If you enable statisticsEnabled, statistics can be retrieved with getLocalMultiMapStats() method.

**NOTE:** Currently, eviction is not supported for the MultiMap data structure.
6.4 Set

Hazelcast Set is a distributed and concurrent implementation of `java.util.Set`.

- Hazelcast Set does not allow duplicate elements.
- Hazelcast Set does not preserve the order of elements.
- Hazelcast Set is a non-partitioned data structure: all the data that belongs to a set will live on one single partition in that node.
- Hazelcast Set cannot be scaled beyond the capacity of a single machine. Since the whole set lives on a single partition, storing large amount of data on a single set may cause memory pressure. Therefore, you should use multiple sets to store large amount of data; this way all the sets will be spread across the cluster, hence sharing the load.
- A backup of Hazelcast Set is stored on a partition of another node in the cluster so that data is not lost in the event of a primary node failure.
- All items are copied to the local node and iteration occurs locally.
- The equals method implemented in Hazelcast Set uses a serialized byte version of objects, as opposed to `java.util.HashSet`.

6.4.1 Sample Set Code

```java
import com.hazelcast.core.Hazelcast;
import java.util.Set;
import java.util.Iterator;

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
Set<Price> set = hazelcastInstance.getSet("IBM-Quote-History");
set.add(new Price(10, time1));
set.add(new Price(11, time2));
set.add(new Price(12, time3));
set.add(new Price(11, time4));
//....
Iterator<Price> iterator = set.iterator();
while (iterator.hasNext()) {
    Price price = iterator.next();
    //analyze
}
```

6.4.2 Event Registration and Configuration for Set

Hazelcast Set uses `ItemListener` to listen to events which occur when items are added and removed.

```java
import java.util.Queue;
import java.util.Map;
import java.util.Set;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.ItemListener;
import com.hazelcast.core.EntryListener;
import com.hazelcast.core.EntryEvent;
```
public class Sample implements ItemListener {

public static void main( String[] args ) {
    Sample sample = new Sample();
    HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
    ISet<Price> set = hazelcastInstance.getSet( "default" );
    set.addItemListener( sample, true );

    Price price = new Price( 10, time1 )
    set.add( price );
    set.remove( price );
}

public void itemAdded( Object item ) {
    System.out.println( "Item added = " + item );
}

public void itemRemoved( Object item ) {
    System.out.println( "Item removed = " + item );
}
}

RELATED INFORMATION
To learn more about the configuration of listeners please refer to the Listener Configurations section.

RELATED INFORMATION
Please refer to the Set Configuration section for a full description of Hazelcast Distributed Set configuration.

6.5 List

Hazelcast List is similar to Hazelcast Set, but Hazelcast List also allows duplicate elements.

- Besides allowing duplicate elements, Hazelcast List preserves the order of elements.
- Hazelcast List is a non-partitioned data structure where values and each backup are represented by their own single partition.
- Hazelcast List cannot be scaled beyond the capacity of a single machine.
- All items are copied to local and iteration occurs locally.

6.5.1 Sample List Code

```java
import com.hazelcast.core.Hazelcast;
import java.util.List;
import java.util.Iterator;

HazelcastInstance hz = Hazelcast.newHazelcastInstance();

List<Price> list = hz.getList( "IBM-Quote-Frequency" );
list.add( new Price( 10 ) );
list.add( new Price( 11 ) );
list.add( new Price( 12 ) );
list.add( new Price( 11 ) );
list.add( new Price( 12 ) );
```
6.5.2 Event Registration and Configuration for List

Hazelcast List uses ItemListener to listen to events which occur when items are added and removed.

```java
import java.util.Queue;
import java.util.Map;
import java.util.Set;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.ItemListener;
import com.hazelcast.core.EntryListener;
import com.hazelcast.core.EntryEvent;
public class Sample implements ItemListener{

    public static void main( String[] args ) {
        Sample sample = new Sample();
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        IList<Price> list = hazelcastInstance.getList( "default" );
        list.addItemListener( sample, true);
        Price price = new Price( 10, time1 )
        list.add( price );
        list.remove( price );
    }

    public void itemAdded( Object item ) {
        System.out.println( "Item added = " + item );
    }

    public void itemRemoved( Object item ) {
        System.out.println( "Item removed = " + item );
    }
}
```

**RELATED INFORMATION**

To learn more about the configuration of listeners please refer to the Listener Configurations section.

**RELATED INFORMATION**

Please refer to the List Configuration section for a full description of Hazelcast Distributed List configuration.

6.6 Ringbuffer

Hazelcast Ringbuffer is a distributed data structure where the data is stored in a ring-like structure. You can think of it as a circular array with a certain capacity. Each Ringbuffer has a tail and a head. The tail is where the items are added and the head is where the items are overwritten or expired. You can reach each element in a Ringbuffer using a sequence ID, which is mapped to the elements between the head and tail (inclusive) of the Ringbuffer.
Reading from Ringbuffer is very simple. Just get the current head and start reading. The method `readOne` returns the item at the given sequence or blocks if no item is available. To read the next item, the sequence is incremented by one.

```java
Ringbuffer<String> ringbuffer = hz.getRingbuffer("rb");
long sequence = ringbuffer.headSequence();
while(true){
    String item = ringbuffer.readOne(sequence);
    sequence++;
    ... process item
}
```

By exposing the sequence, you can now move the item from Ringbuffer as long as the item is still available. If it is not available any longer, `StaleSequenceException` is thrown.

Adding an item to Ringbuffer is also very easy:

```java
Ringbuffer<String> ringbuffer = hz.getRingbuffer("rb");
ringbuffer.add("someitem")
```

The method `add` returns the sequence of the inserted item and this value will always be unique. This can sometimes be used as a very cheap way of generating unique IDs if you are already using Ringbuffer.

### 6.6.1 IQueue vs. Ringbuffer

Hazelcast Ringbuffer can sometimes be a better alternative than an Hazelcast IQueue. Unlike IQueue, Ringbuffer does not remove the items, it only reads items using a certain position. There are many advantages using this approach:

- The same item can be read multiple times by the same thread; this is useful for realizing semantics of read-at-least-once or read-at-most-once.
- The same item can be read by multiple threads. Normally you could use an IQueue per thread for the same semantic, but this is less efficient because of the increased remoting. A take from an IQueue is destructive, so the change needs to be applied for backup also, which is why a `queue.take()` is more expensive than a `ringBuffer.read(...)`. 
- Reads are extremely cheap since there is no change in the Ringbuffer, therefore no replication is required.
- Reads and writes can be batched to speed up performance. Batching can dramatically improve the performance of Ringbuffer.

### 6.6.2 Capacity

By default, a Ringbuffer is configured with a capacity of 10000 items. Internally, an array is created with exactly that size. If a time-to-live is configured, then an array of longs is also created that stores the expiration time for every item. In a lot of cases, you may want to change this number to something that fits your needs better.

Below is a declarative configuration example of a Ringbuffer with a capacity of 2000 items.

```xml
<ringbuffer name="rb">
    <capacity>2000</capacity>
</ringbuffer>
```

Hazelcast Ringbuffer is not a partitioned data structure in its current state; its data is stored in a single partition and the replicas are stored in another partition. Therefore, create a Ringbuffer that can safely fit in a single cluster member.
6.6.3 Synchronous and Asynchronous Backups

Hazelcast Ringbuffer has a single synchronous backup by default. This can be controlled just like most of the other Hazelcast distributed data structures by setting the sync and async backups. In the example below, a Ringbuffer is configured with 0 sync backups and 1 async backup:

```xml
<ringbuffer name="rb">
  <backup-count>0</backup-count>
  <async-backup-count>1</async-backup-count>
</ringbuffer>
```

An async backup will probably give you better performance. However, there is a chance that the item added is lost when the member owning the primary crashes before the replication could complete. You may want to consider batching methods if you need high performance but do not want to give up on consistency.

6.6.4 Time to live

Hazelcast Ringbuffer can be configured with a time to live seconds. Using this setting, you can control how long the items remain in the Ringbuffer before they are expired. By default, the time to live is set to 0, meaning that unless the item is overwritten, it will remain in the Ringbuffer indefinitely. If a time to live is set and an item is added, then depending on the Overflow Policy, either the oldest item is overwritten, or the call is rejected.

In the example below, a Ringbuffer is configured with a time to live of 180 seconds.

```xml
<ringbuffer name="rb">
  <time-to-live-seconds>180</time-to-live-seconds>
</ringbuffer>
```

6.6.5 Overflow Policy

Using the overflow policy, you can determine what to do if the oldest item in the Ringbuffer is not old enough to expire when more items than the configured RingBuffer capacity are being added. There are currently below options available:

- **OverflowPolicy.OVERWRITE**: The oldest item is overwritten.
- **OverflowPolicy.FAIL**: The call is aborted. The methods that make use of the OverflowPolicy return -1 to indicate that adding the item has failed.

Overflow policy gives fine control on what to do if the Ringbuffer is full. The policy can also be used to apply a back pressure mechanism. The following example code shows the usage of an exponential backoff.

```java
long sleepMs = 100;
for (; ; ) {
    long result = ringbuffer.addAsync(item, OverflowPolicy.FAIL).get();
    if (result != -1) {
        break;
    }
    TimeUnit.MILLISECONDS.sleep(sleepMs);
    sleepMs = min(5000, sleepMs * 2);
}
```
6.6. In-Memory Format

Hazelcast Ringbuffer can also be configured with an in-memory format which controls the format of stored items. By default, **BINARY** is used, meaning that the object is stored in a serialized form. You can select the **OBJECT** in-memory format, which is useful when filtering is applied or when the **OBJECT** in-memory format has a smaller memory footprint than **BINARY**.

In the declarative configuration example below, a Ringbuffer is configured with **OBJECT** in-memory format:

```xml
<ringbuffer name="rb">
  <in-memory-format>BINARY</in-memory-format>
</ringbuffer>
```

6.6.7 Adding Batched Items

In the previous examples, the method `ringBuffer.add()` is used to add an item to the Ringbuffer. The problem with this method is that it always overwrites and that it does not support batching. Batching can have a huge impact on the performance. That is why the method `addAllAsync` is available.

Please see the following example code.

```java
List<String> items = Arrays.asList("1","2","3");
ICompletableFuture<Long> f = rb.addAllAsync(items, OverflowPolicy.OVERWRITE);
f.get()
```

In the above case, three strings are added to the Ringbuffer using the policy `OverflowPolicy.OVERWRITE`. Please see the [Overflow Policy section](#) for more information.

6.6.8 Reading Batched Items

In the previous example the `readOne` was being used. It is simple but not very efficient for the following reasons:

- It does not make use of batching.
- It cannot filter items at the source; they need to be retrieved before being filtered.

That is why the method `readManyAsync` is available.

Please see the following example code.

```java
ICompletableFuture<ReadResultSet<E>> readManyAsync(
    long startSequence,
    int minCount,
    int maxCount,
    IFunction<E, Boolean> filter);
```

This call can read a batch of items and can filter items at the source. The meaning of the arguments are given below.

- **startSequence**: Sequence of the first item to read.
- **minCount**: Minimum number of items to read. If you do not want to block, set it to 0. If you do want to block for at least one item, set it to 1.
- **maxCount**: Maximum number of the items to retrieve. Its value cannot exceed 1000.
- **filter**: A function that accepts an item and checks if it should be returned. If no filtering should be applied, set it to null.

A full example is given below.
long sequence = rb.headSequence();
for(;;) {
    ICompletableFuture<ReadResultSet<String>> f = rb.readManyAsync(sequence, 1, 10, null);
    ReadResultSet<String> rs = f.get();
    for (String s : rs) {
        System.out.println(s);
    }
    sequence+=rs.readCount();
}

Please take a careful look at how the sequence is being incremented. You cannot always rely on the number of items being returned if the items are filtered out.

### 6.6.9 Async Methods

Hazelcast Ringbuffer provides asynchronous methods for more powerful operations like batched reading with filtering or batched writing. To make these methods synchronous, just call the method `get()` on the returned future.

Please see the following example code.

ICompletableFuture f = ringbuffer.addAsync(item, OverflowPolicy.FAIL);
f.get();

However, the `ICompletableFuture` can also be used to get notified when the operation has completed. Please see the example code when you want to get notified when a batch of reads has completed.

ICompletableFuture<ReadResultSet<String>> f = rb.readManyAsync(sequence, min, max, someFilter);
f.andThen(new ExecutionCallback<ReadResultSet<String>>() {
    @Override
    public void onResponse(ReadResultSet<String> response) {
        for (String s : response) {
            System.out.println("Received:" + s);
        }
    }
    @Override
    public void onFailure(Throwable t) {
        t.printStackTrace();
    }
});

The advantage of this approach: The thread that is used for the call is not blocked till the response is returned.

### 6.6.10 Full Configuration examples

The following shows the declarative configuration of a Ringbuffer called `rb`. The configuration is modeled after Ringbuffer defaults.

```
<ringbuffer name="rb">
    <capacity>10000</capacity>
    <backup-count>1</backup-count>
    <async-backup-count>0</async-backup-count>
    <time-to-live-seconds>0</time-to-live-seconds>
    <in-memory-format>BINARY</in-memory-format>
</ringbuffer>
```
You can also configure a Ringbuffer programmatically. The following is programmatic version of the above declarative configuration.

```java
RingbufferConfig rbConfig = new RingbufferConfig("rb")
    .setCapacity(10000)
    .setBackupCount(1)
    .setAsyncBackupCount(0)
    .setTimeToLiveSeconds(0)
    .setInMemoryFormat(InMemoryFormat.BINARY);
Config config = new Config();
config.addRingbufferConfig(rbConfig);
```

**RELATED INFORMATION**

Please refer to the Ringbuffer Configuration section for more information on configuring the Ringbuffer.

### 6.7 Topic

Hazelcast provides a distribution mechanism for publishing messages that are delivered to multiple subscribers. This is also known as a publish/subscribe (pub/sub) messaging model. Publishing and subscribing operations are cluster wide. When a member subscribes to a topic, it is actually registering for messages published by any member in the cluster, including the new members that joined after you add the listener.

**NOTE:** Publish operation is async. It does not wait for operations to run in remote nodes, it works as fire and forget.

#### 6.7.1 Sample Topic Code

```java
import com.hazelcast.core.Topic;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.MessageListener;

public class Sample implements MessageListener<MyEvent> {

    public static void main( String[] args ) {
        Sample sample = new Sample();
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        ITopic topic = hazelcastInstance.getTopic( "default" );
        topic.addMessageListener( sample );
        topic.publish( new MyEvent() );
    }

    public void onMessage( Message<MyEvent> message ) {
        MyEvent myEvent = message.getMessageObject();
        System.out.println( "Message received = " + myEvent.toString() );
        if ( myEvent.isHeavyweight() ) {
            messageExecutor.execute( new Runnable() {
                public void run() {
                    doHeavyweightStuff( myEvent );
                }
            } );
        }
    }

    // ...
```
private final Executor messageExecutor = Executors.newSingleThreadExecutor();
}

6.7.2 Statistics

Topic has two statistic variables that you can query. These values are incremental and local to the member.

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
ITopic<Object> myTopic = hazelcastInstance.getTopic( "myTopicName" );

myTopic.getLocalTopicStats().getPublishOperationCount();
myTopic.getLocalTopicStats().getReceiveOperationCount();

getPublishOperationCount and getReceiveOperationCount returns the total number of published and received messages since the start of this node, respectively. Please note that these values are not backed up, so if the node goes down, these values will be lost.

You can disable this feature with topic configuration. Please see the Topic Configuration section.

**NOTE:** These statistics values can be also viewed in Management Center. Please see the Topics section.

6.7.3 Internals

Each node has a list of all registrations in the cluster. When a new node is registered for a topic, it sends a registration message to all members in the cluster. Also, when a new node joins the cluster, it will receive all registrations made so far in the cluster.

The behavior of a topic varies depending on the value of the configuration parameter `globalOrderEnabled`.

- If `globalOrderEnabled` is disabled:

  Messages are ordered, i.e. listeners (subscribers) process the messages in the order that the messages are published. If cluster member M publishes messages $m_1, m_2, m_3, \ldots, mn$ to a topic $T$, then Hazelcast makes sure that all of the subscribers of topic $T$ will receive and process $m_1, m_2, m_3, \ldots, mn$ in the given order.

  Here is how it works. Let’s say that we have three nodes (`node1`, `node2` and `node3`) and that `node1` and `node2` are registered to a topic named `news`. Note that all three nodes know that `node1` and `node2` are registered to `news`.

  In this example, `node1` publishes two messages: $a_1$ and $a_2$, and `node3` publishes two messages: $c_1$ and $c_2$. When `node1` and `node3` publish a message, they will check their local list for registered nodes, and they will discover that `node1` and `node2` are in their lists, then they will fire messages to those nodes. One possible order of the messages received can be the following.

  `node1` -> `c1`, `b1`, `a2`, `c2`
  `node2` -> `c1`, `c2`, `a1`, `a2`

- If `globalOrderEnabled` is enabled:

  When enabled, `globalOrderEnabled` guarantees that all nodes listening to the same topic will get its messages in the same order.

  Here is how it works. Let’s say that we have three nodes (`node1`, `node2` and `node3`) and that `node1` and `node2` are registered to a topic named `news`. Note that all three nodes know that `node1` and `node2` are registered to `news`.

  In this example, `node1` publishes two messages: $a_1$ and $a_2$, and `node3` publishes two messages: $c_1$ and $c_2$. When a node publishes messages over the topic `news`, it first calculates which partition the `news` ID corresponds to. Then
it sends an operation to the owner of the partition for that node to publish messages. Let’s assume that `news` corresponds to a partition that `node2` owns. `node1` and `node3` first sends all messages to `node2`. Assume that the messages are published in the following order:

`node1` -> `a1, c1, a2, c2`

`node2` then publishes these messages by looking at registrations in its local list. It sends these messages to `node1` and `node2` (it makes a local dispatch for itself).

`node1` -> `a1, c1, a2, c2`

`node2` -> `a1, c1, a2, c2`

This way, we guarantee that all nodes will see the events in the same order.

In both cases, there is a `StripedExecutor` in EventService that is responsible for dispatching the received message. For all events in Hazelcast, the order that events are generated and the order they are published to the user are guaranteed to be the same via this `StripedExecutor`.

In `StripedExecutor`, there are as many threads as are specified in the property `hazelcast.event.thread.count` (default is 5). For a specific event source (for a particular topic name), `hash of that source’s name % 5` gives the ID of the responsible thread. Note that there can be another event source (entry listener of a map, item listener of a collection, etc.) corresponding to the same thread. In order not to make other messages to block, heavy processing should not be done in this thread. If there is time consuming work that needs to be done, the work should be handed over to another thread. Please see the Sample Topic Code section.

### 6.7.4 Configuring Topic

**Declarative Configuration:**

```xml
<hazelcast>
  ...
  <topic name="yourTopicName">
    <global-ordering-enabled>true</global-ordering-enabled>
    <statistics-enabled>true</statistics-enabled>
    <message-listeners>
      <message-listener>MessageListenerImpl</message-listener>
    </message-listeners>
  </topic>
  ...
</hazelcast>
```

**Programmatic Configuration:**

```java
TopicConfig topicConfig = new TopicConfig();
topicConfig.setGlobalOrderingEnabled(true);
topicConfig.setStatisticsEnabled(true);
topicConfig.setName("yourTopicName");
MessageListener<String> implementation = new MessageListener<String>() {
  @Override
  public void onMessage(Message<String> message) {
    // process the message
  }
};
topicConfig.addMessageListenerConfig(new ListenerConfig(implementation));
HazelcastInstance instance = Hazelcast.newHazelcastInstance();
```

Default values are:

- `global-ordering` is `false`, meaning that by default, there is no guarantee of global order.
• statistics is true, meaning that by default, statistics are calculated.

Topic related but not topic specific configuration parameters:

- ‘hazelcast.event.queue.capacity’: default value is 1,000,000
- ‘hazelcast.event.queue.timeout.millis’: default value is 250
- ‘hazelcast.event.thread.count’: default value is 5

RELATED INFORMATION
For description of these parameters, please see the Global Event Configuration section.

RELATED INFORMATION
Please refer to the Topic Configuration section for a full description of Hazelcast Distributed Topic configuration.

6.8 Reliable Topic

Reliable Topic data structure has been introduced with the release of Hazelcast 3.5. The Reliable Topic makes use of the same ITopic interface as a regular topic. The main difference is that it is backed up by the RingBuffer (also introduced with Hazelcast 3.5) data structure. The following are the advantages of this approach:

• Events are not lost since the RingBuffer is configured with 1 synchronous backup by default.
• Each Reliable ITopic gets its own RingBuffer; if there is a topic with a very fast producer, it will not lead to problems at the topic that runs at a slower pace.
• Since the event system behind a regular ITopic is shared with other data structures (e.g. collection listeners), you can run into isolation problems. This does not happen with the Reliable ITopic.

6.8.1 Sample Reliable ITopic Code

```java
import com.hazelcast.core.Topic;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.MessageListener;

public class Sample implements MessageListener<MyEvent> {

    public static void main( String[] args ) {
        Sample sample = new Sample();
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        ITopic topic = hazelcastInstance.getReliableTopic( "default" );
        topic.addMessageListener( sample );
        topic.publish( new MyEvent() );
    }

    public void onMessage( Message<MyEvent> message ) {
        MyEvent myEvent = message.getMessageObject();
        System.out.println( "Message received = " + myEvent.toString() );
    }
}
```

The Reliable ITopic can be configured using its RingBuffer. If there is a Reliable Topic with name Foo, then this topic can be configured by adding a ReliableTopicConfig for a RingBuffer with the name Foo. By default, a RingBuffer does not have any TTL (time to live) and it has a limited capacity; you may want to change the configuration.
By default, the Reliable ITopic uses a shared thread pool. If you need a better isolation, you can configure a custom executor on the ReliableTopicConfig.

Because the reads on a RingBuffer are not destructive, it is easy to apply batching. ITopic uses read batching and reads 10 items at a time (if available) by default.

### 6.8.2 Slow Consumers

The Reliable ITopic provides control and a way to deal with slow consumers. It is unwise to keep events for a slow consumer in memory indefinitely since you do not know when it is going to catch up. The size of the RingBuffer can be controlled using its capacity. For the cases when a RingBuffer runs out of its capacity, you can specify the following policies for the TopicOverloadPolicy configuration:

- **DISCARD_OLDEST**: Overwrite the oldest item, no matter if a TTL is set. In this case the fast producer supersedes a slow consumer
- **DISCARD_NEWEST**: Discard the newest item.
- **BLOCK**: Wait until the items are expired in the RingBuffer.
- **FAIL**: Immediately throw TopicOverloadException if there is no space in the RingBuffer.

### 6.9 Lock

ILock is the distributed implementation of java.util.concurrent.locks.Lock. Meaning if you lock using an ILock, the critical section that it guards is guaranteed to be executed by only one thread in the entire cluster. Even though locks are great for synchronization, they can lead to problems if not used properly. Also note that Hazelcast Lock does not support fairness.

A few warnings when using locks:

- Always use locks with try-catch blocks. It will ensure that locks will be released if an exception is thrown from the code in a critical section. Also note that the lock method is outside the try-catch block, because we do not want to unlock if the lock operation itself fails.

```java
import com.hazelcast.core.Hazelcast;
import java.util.concurrent.locks.Lock;

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
Lock lock = hazelcastInstance.getLock( "myLock" );
lock.lock();
try {
    // do something here
} finally {
    lock.unlock();
}
```

- If a lock is not released in the cluster, another thread that is trying to get the lock can wait forever. To avoid this, use tryLock with a timeout value. You can set a high value (normally it should not take that long) for tryLock. You can check the return value of tryLock as follows:

```java
if ( lock.tryLock ( 10, TimeUnit.SECONDS ) ) {
    try {
        // do some stuff here..
    } finally {
        lock.unlock();
    }
} else {
    // warning
}
```
• You can also avoid indefinitely waiting threads by using lock with lease time: the lock will be released in the
given lease time. Lock can be safely unlocked before the lease time expires. Note that the unlock operation
can throw an `IllegalMonitorStateException` if lock is released because the lease time expires. If that is
the case, critical section guarantee is broken.

Please see the below example.

```java
lock.lock( 5, TimeUnit.SECONDS )
try {
    // do some stuff here..
} finally {
    try {
        lock.unlock();
    } catch ( IllegalMonitorStateException ex ){
        // WARNING Critical section guarantee can be broken
    }
}
```

• Locks are fail-safe. If a member holds a lock and some other members go down, the cluster will keep your
locks safe and available. Moreover, when a member leaves the cluster, all the locks acquired by that dead
member will be removed so that those locks are immediately available for live members.

• Locks are re-entrant: the same thread can lock multiple times on the same lock. Note that for other threads
to be able to require this lock, the owner of the lock must call `unlock` as many times as the owner called `lock`.

• In the split-brain scenario, the cluster behaves as if it were two different clusters. Since two separate clusters
are not aware of each other, two nodes from different clusters can acquire the same lock. For more information
on places where split brain syndrome can be handled, please see split brain syndrome.

• Locks are not automatically removed. If a lock is not used anymore, Hazelcast will not automatically garbage
collect the lock. This can lead to an `OutOfMemoryError`. If you create locks on the fly, make sure they are
destroyed.

• Hazelcast IMap also provides locking support on the entry level with the method `IMap.lock(key)`. Although
the same infrastructure is used, `IMap.lock(key)` is not an ILock and it is not possible to expose it directly.

6.9.1 ICondition

ICondition is the distributed implementation of the `notify`, `notifyAll` and `wait` operations on the Java object.
You can use it to synchronize threads across the cluster. More specifically, you use ICondition when a thread’s
work depends on another thread’s output. A good example can be producer/consumer methodology.

Please see the below code snippets for a sample producer/consumer implementation.

Producer thread:

```java
HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
Lock lock = hazelcastInstance.getLock( "myLockId" );
ICondition condition = lock.newCondition( "myConditionId" );

lock.lock();
try {
    while ( !shouldProduce() ) {
        condition.await(); // frees the lock and waits for signal
        // when it wakes up it re-acquires the lock
        // if available or waits for it to become
        // available
    }
    produce();
    condition.signalAll();
```
Consumer thread:

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
Lock lock = hazelcastInstance.getLock("myLockId");
ICondition condition = lock.newCondition("myConditionId");

lock.lock();
try {
    while (!canConsume()) {
        condition.await(); // frees the lock and waits for signal
        // when it wakes up it re-acquires the lock if
        // available or waits for it to become
        // available
    }
    consume();
    condition.signalAll();
} finally {
    lock.unlock();
}

6.10 IAtomicLong

Hazelcast IAtomicLong is the distributed implementation of java.util.concurrent.atomic.AtomicLong. It offers most of AtomicLong’s operations such as get, set, getAndSet, compareAndSet and incrementAndGet. Since IAtomicLong is a distributed implementation, these operations involve remote calls and hence their performances differ from AtomicLong.

The following sample code creates an instance, increments it by a million, and prints the count.

```java
public class Member {
    public static void main( String[] args ) {
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        IAtomicLong counter = hazelcastInstance.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());
    }
}
```

When you start other instances with the code above, you will see the count as member count times a million.

You can send functions to an IAtomicLong. Function is a Hazelcast owned, single method interface. The following sample Function implementation doubles the original value.

```java
private static class Add2Function implements Function<Long, Long> {
    @Override
    public Long apply( Long input ) {
        return input + 2;
    }
}
```
You can use the following methods to execute functions on `IAtomicLong`.

- **apply**: It applies the function to the value in `IAtomicLong` without changing the actual value and returning the result.
- **alter**: It alters the value stored in the `IAtomicLong` by applying the function. It will not send back a result.
- **alterAndGet**: It alters the value stored in the `IAtomicLong` by applying the function, storing the result in the `IAtomicLong` and returning the result.
- **getAndAlter**: It alters the value stored in the `IAtomicLong` by applying the function and returning the original value.

The following sample code includes these methods.

```java
public class Member {
    public static void main( String[] args ) {
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        IAtomicLong atomicLong = hazelcastInstance.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() );
    }
}
```

The reason for using a function instead of a simple code line like `atomicLong.set(atomicLong.get() + 2)` is that the `IAtomicLong` read and write operations are not atomic. Since `IAtomicLong` is a distributed implementation, those operations can be remote ones, which may lead to race problems. By using functions, the data is not pulled into the code, but the code is sent to the data. This makes it more scalable.

**NOTE**: `IAtomicLong` has 1 synchronous backup and no asynchronous backups. Its backup count is not configurable.

### 6.11 ISemaphore

Hazelcast ISemaphore is the distributed implementation of `java.util.concurrent.Semaphore`. Semaphores offer **permits** to control the thread counts in the case of performing concurrent activities. To execute a concurrent activity, a thread grants a permit or waits until a permit becomes available. When the execution is completed, the permit is released.

**NOTE**: Semaphore with a single permit may be considered as a lock. But unlike the locks, when semaphores are used, any thread can release the permit and semaphores can have multiple permits.
NOTE: Hazelcast Semaphore does not support fairness.

When a permit is acquired on ISemaphore:

- if there are permits, the number of permits in the semaphore is decreased by one and the calling thread performs its activity. If there is contention, the longest waiting thread will acquire the permit before all other threads.
- if no permits are available, the calling thread blocks until a permit becomes available. When a timeout happens during this block, the thread is interrupted. In the case where the semaphore is destroyed, an InstanceDestroyedException is thrown.

The following sample code uses an IAtomicLong resource 1000 times, increments the resource when a thread starts to use it, and decrements it when the thread completes.

```java
public class SemaphoreMember {
    public static void main( String[] args ) throws Exception{
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        ISemaphore semaphore = hazelcastInstance.getSemaphore( "semaphore" );
        IAtomicLong resource = hazelcastInstance.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");
    }
}
```

Let’s limit the concurrent access to this resource by allowing at most 3 threads. You can configure it declaratively by setting the initial-permits property, as shown below.

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

NOTE: If there is a shortage of permits while the semaphore is being created, value of this property can be set to a negative number.

If you execute the above SemaphoreMember class 5 times, the output will be similar to the following:

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

As can be seen, the maximum count of concurrent threads is equal or smaller than 3. If you remove the semaphore acquire/release statements in SemaphoreMember, you will see that there is no limitation on the number of concurrent usages.
Hazelcast also provides backup support for ISemaphore. When a member goes down, another member can take over the semaphore with the permit information (permits are automatically released when a member goes down). To enable this, configure synchronous or asynchronous backup with the properties backup-count and async-backup-count(by default, synchronous backup is already enabled).

A sample configuration is shown below.

```xml
<semaphore name="semaphore">
  <initial-permits>3</initial-permits>
  <backup-count>1</backup-count>
</semaphore>
```

**NOTE:** If high performance is more important (than not losing the permit information), you can disable the backups by setting backup-count to 0.

**RELATED INFORMATION**

Please refer to the Semaphore Configuration section for a full description of Hazelcast Distributed Semaphore configuration.

### 6.12 IAtomicReference

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.

Here is an IAtomicReference example.

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

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

When you execute the above example, you will see the following output.

```
foo
```

Just like IAtomicLong, IAtomicReference has methods that accept a ‘function’ as an argument, such as alter, alterAndGet, getAndAlter and apply. There are two big advantages of using these methods:

- 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 data and therefore cheaper to send over the line. Also the function only needs to be transferred once to the target machine, and the data needs to be transferred twice.
- You do not need to deal with concurrency control. If you would perform a load, transform, store, you could run into a data race since another thread might have updated the value you are about to overwrite.

There are some issues you need to know, described below.
ICOUNTDOWNLATCH

- IAtomicReference works based on the byte-content and not on the object-reference. If you use the compareAndSet method, do not change to 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.
- IAtomicReference will always have 1 synchronous backup.
- All methods returning an object will return a private copy. You can modify the private copy, 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. The receiving side does not need to have the class definition available, unless it needs to be deserialized on the other side (e.g. 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 with many fields or an object graph, and you only need to calculate some information or need a subset of fields, you can use the apply method. With the apply method, the whole object does not need to be sent over the line, only the information that is relevant.

6.13 ICountDownLatch

Hazelcast ICountDownLatch is the distributed implementation of java.util.concurrent.CountDownLatch. As you may know, CountDownLatch is considered to be a gate keeper for concurrent activities. It enables the threads to wait for other threads to complete their operations.

The following code samples describe the mechanism of ICountDownLatch. Assume that there is a leader process and there are follower processes that will wait until the leader completes. Here is the leader:

```java
public class Leader {
  public static void main( String[] args ) throws Exception {
    HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
    ICountDownLatch latch = hazelcastInstance.getCountDownLatch( "countDownLatch" );
    System.out.println( "Starting" );
    latch.trySetCount( 1 );
    Thread.sleep( 30000 );
    latch.countDown();
    System.out.println( "Leader finished" );
    latch.destroy();
  }
}
```

Since only a single step is needed to be completed as a sample, the above code initializes the latch with 1. Then, the code sleeps for a while to simulate a process and starts the countdown. Finally, it clears up the latch. Let’s write a follower:

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

The follower class above first retrieves ICountDownLatch and then calls the await method to enable the thread to listen for the latch. The method await has a timeout value as a parameter. This is useful when countDown method fails. To see ICountDownLatch in action, start the leader first and then start one or more followers. You will see that the followers will wait until the leader completes.
In a distributed environment, the counting down cluster member may go down. In this case, all listeners are notified immediately and automatically by Hazelcast. The state of the current process just before the failure should be verified and 'how to continue now' should be decided (e.g. restart all process operations, continue with the first failed process operation, throw an exception, etc.).

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

**NOTE:** ICountDownLatch has 1 synchronous backup and no asynchronous backups. Its backup count is not configurable. Also, the count cannot be re-set during an active count, it should be re-set after the countdown is finished.

### 6.14 IdGenerator

Hazelcast IdGenerator is used to generate cluster-wide unique identifiers. Generated identifiers are long type primitive values between 0 and Long.MAX_VALUE.

ID generation occurs almost at the speed of AtomicLong.incrementAndGet(). A group of 1 million identifiers is allocated for each cluster member. In the background, this allocation takes place with an IAtomicLong incremented by 1 million. Once a cluster member generates IDs (allocation is done), IdGenerator increments a local counter. If a cluster member uses all IDs in the group, it will get another 1 million IDs. By this way, only one time of network traffic is needed, meaning that 999,999 identifiers are generated in memory instead of over the network. This is fast.

Let’s write a sample identifier generator.

```java
public class IdGeneratorExample {
    public static void main( String[] args ) throws Exception {
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        IdGenerator idGen = hazelcastInstance.getIdGenerator( "newId" );
        while (true) {
            Long id = idGen.newId();
            System.err.println( "Id: " + id );
            Thread.sleep( 1000 );
        }
    }
}
```

Let’s run the above code two times. The output will be similar to the following.

Members [1] {
    Member [127.0.0.1]:5701 this
}  
Id: 1 
Id: 2 
Id: 3 

Members [2] {
    Member [127.0.0.1]:5701 
    Member [127.0.0.1]:5702 this
}  
Id: 1000001 
Id: 1000002 
Id: 1000003
You can see that the generated IDs are unique and counting upwards. If you see duplicated identifiers, it means your instances could not form a cluster.

**NOTE:** Generated IDs are unique during the life cycle of the cluster. If the entire cluster is restarted, IDs start from 0 again or you can initialize to a value using the `init()` method of `IdGenerator`.

**NOTE:** `IdGenerator` has 1 synchronous backup and no asynchronous backups. Its backup count is not configurable.

6.15 Replicated Map

A replicated map is a weakly consistent, distributed key-value data structure provided by Hazelcast.

All other data structures are partitioned in design. A replicated map does not partition data (it does not spread data to different cluster members); instead, it replicates the data to all nodes.

This leads to higher memory consumption. However, a replicated map has faster read and write access since the data are available on all nodes and writes take place on local nodes, eventually being replicated to all other nodes.

Weak consistency compared to eventually consistency means that replication is done on a best efforts basis. Lost or missing updates are neither tracked nor resent. This kind of data structure is suitable for immutable objects, catalogue data, or idempotent calculable data (like HTML pages).

Replicated map nearly fully implements the `java.util.Map` interface, but it lacks the methods from `java.util.concurrent.ConcurrentMap` since there are no atomic guarantees to writes or reads.

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import java.util.Collection;
import java.util.Map;

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
Map<String, Customer> customers = hazelcastInstance.getReplicatedMap("customers");
customers.put("1", new Customer("Joe", "Smith"));
customers.put("2", new Customer("Ali", "Selam"));
customers.put("3", new Customer("Avi", "Noyan"));

Collection<Customer> colCustomers = customers.values();
for (Customer customer : colCustomers) {
    // process customer
}
```

`HazelcastInstance::getReplicatedMap` returns `com.hazelcast.core.ReplicatedMap` which, as stated above, extends the `java.util.Map` interface.

The `com.hazelcast.core.ReplicatedMap` interface has some additional methods for registering entry listeners or retrieving values in an expected order.

**NOTE:** Replicated Map is in the beta stage.

6.15.1 For Consideration

A replicated map **does not** support ordered writes! In case of a conflict caused by two nodes simultaneously written to the same key, a vector clock algorithm resolves and decides on one of the values.

Due to the weakly consistent nature and the previously mentioned behaviors of replicated map, there is a chance of reading stale data at any time. There is no read guarantee like there is for repeatable reads.
6.15.2 Breakage of the Map-Contract

Replicated Map offers a distributed java.util.Map::clear implementation, but due to the asynchronous nature and the weakly consistency of this implementation, there is no point in time where you can say the map is empty. Every node applies that to its local dataset in “a near point in time”. If you need a definite point in time to empty the map, you may want to consider using a lock around the clear operation.

You can simulate the clear method by locking your user codebase and executing a remote operation that uses DistributedObject::destroy to destroy the node’s own proxy and storage of the Replicated Map. A new proxy instance and storage will be created on the next retrieval of the Replicated Map using HazelcastInstance::getReplicatedMap. You will have to reallocate the Replicated Map in your code. Afterwards, just release the lock when finished.

6.15.3 Technical Design

There are several technical design decisions for configurable behavior.

Initial provisioning

If a new member joins, there are two ways you can handle the initial provisioning that is executed to replicate all existing values to the new member.

First, you can have an async fill up, which does not block reads while the fill up operation is underway. That way, you have immediate access on the new member, but it will take time until all values are eventually accessible. Not yet replicated values are returned as non-existing (null). Write operations to already existing keys during this async phase can be lost, since the vector clock for an entry might not be initialized by another member yet, and it might be seen as an old update by other members.

Second, you can perform a synchronous initial fill up, which blocks every read or write access to the map until the fill up operation is finished. Use this way with caution since it might block your application from operating.

Replication delay

By default, the replication of values is delayed by 100 milliseconds when no current waiting replication is found. This collects multiple updates and minimizes the operations overhead on replication. A hard limit of 1000 replications is built into the system to prevent OutOfMemory situations where you put lots of data into the replicated map in a very short time. The delay is configurable. A value of “0” means immediate replication. You can configure the trade off between replication overhead and the time for the value to be replicated.

Concurrency Level

The concurrency level configuration defines the number of mutexes and segments inside the replicated map storage. A mutex/segment is chosen by calculating the hashCode of the key and using the module by the concurrency level. If multiple keys fall into the same mutex, they will wait for other mutex holders on the same mutex to finish their operation.

For a high amount of values, or for high contention on the mutexes, this value can be changed.

6.15.4 Replicated Map Configuration

Replicated Map can be configured using the following two ways (as with most other features in Hazelcast):

- Programmatic: the typical Hazelcast way, using the Config API seen above.
- Declarative: using hazelcast.xml.

6.15.4.1 Replicated Map Declarative Configuration

You can declare your Replicated Map configuration in the Hazelcast configuration file hazelcast.xml. You can use the configuration to tune the behavior of the internal replication algorithm, such as the replication delay which batches up the replication for better network utilization. See the following example declarative configuration.
6.15. REPLICATED MAP

- **in-memory-format**: Defines the internal storage format. Please see the In-Memory Format section. The default value is BINARY.
- **concurrency-level**: Number of parallel mutexes to minimize the contention on the keys. The default value is 32, which is a good number for lots of applications. If higher contention is seen on writes to values inside the replicated map, this value can be adjusted according to the needs.
- **replication-delay-millis**: Defines the period in milliseconds after a put is executed that the put value is replicated to other nodes. During this time, multiple puts can be operated and the values are cached up to be sent all at once. This increases the latency for eventual consistency, but it lowers the I/O operations. The default value is 100ms before a replication is operated. If replication-delay-millis is set to 0, no delay is used (not cached) and all values are replicated one by one.
- **async-fillup**: Defines if the replicated map is available for reads before the initial replication is completed. The default value is true. If set to false (i.e. synchronous initial fill up), no exception will be thrown when the replicated map is not yet ready, but the call will block until it is finished.
- **statistics-enabled**: If set to true, the statistics such as cache hits and misses are collected. The default value is false.
- **entry-listener**: The value of this element is the full canonical classname of the EntryListener implementation.
  - **entry-listener#include-value**: This attribute defines if the event will include the value or not. Sometimes the key is enough to react on an event. In those situations, setting this value to false will save a deserialization cycle. The default value is true.
  - **entry-listener#local**: This attribute is not used for Replicated Map since listeners are always local.

6.15.4.2 Replicated Map Programmatic Configuration

You can use the Config API for programmatic configuration, as you can for all other data structures in Hazelcast. You must create the configuration upfront, when you instantiate the HazelcastInstance.

A basic example on how to configure the Replicated Map using the programmatic approach is shown in the following snippet.

```java
Config config = new Config();

ReplicatedMapConfig replicatedMapConfig =
    config.getReplicatedMapConfig( "default" );

replicatedMapConfig.setInMemoryFormat( InMemoryFormat.BINARY );
replicatedMapConfig.setConcurrentLevel( 32 );
```

All properties that can be configured using the declarative configuration are also available using programmatic configuration by transforming the tag names into getter or setter names.
6.15.4.3 In-Memory Format on Replicated Map

Currently, two in-memory-format values are usable with the Replicated Map.

- **OBJECT** (default): The data will be stored in deserialized form. This configuration is the default choice since the data replication is mostly used for high speed access. Please be aware that changing the values without a Map::put is not reflected on the other nodes but is visible on the changing nodes for later value accesses.

- **BINARY**: The data is stored in serialized binary format and has to be deserialized on every request. This option offers higher encapsulation since changes to values are always discarded as long as the newly changed object is not explicitly Map::put into the map again.

6.15.5 EntryListener on Replicated Map

A com.hazelcast.core.EntryListener used on a Replicated Map serves the same purpose as it would on other data structures in Hazelcast. You can use it to react on add, update, and remove operations. Replicated maps do not yet support eviction.

The fundamental difference in Replicated Map behavior, compared to the other data structures, is that an EntryListener only reflects changes on local data. Since replication is asynchronous, all listener events are fired only when an operation is finished on a local node. Events can fire at different times on different nodes.

```java
import com.hazelcast.core.EntryEvent;
import com.hazelcast.core.EntryListener;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.ReplicatedMap;

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
ReplicatedMap<String, Customer> customers =
    hazelcastInstance.getReplicatedMap( "customers" );

customers.addEntryListener( new EntryListener<String, Customer>() {
    @Override
    public void entryAdded( EntryEvent<String, Customer> event ) {
        log( "Entry added: " + event );
    }

    @Override
    public void entryUpdated( EntryEvent<String, Customer> event ) {
        log( "Entry updated: " + event );
    }

    @Override
    public void entryRemoved( EntryEvent<String, Customer> event ) {
        log( "Entry removed: " + event );
    }

    @Override
    public void entryEvicted( EntryEvent<String, Customer> event ) {
        // Currently not supported, will never fire
    }
});

customers.put( "1", new Customer( "Joe", "Smith" ) ); // add event
customers.put( "1", new Customer( "Ali", "Selam" ) ); // update event
customers.remove( "1" ); // remove event
```
Chapter 7

Distributed Events

You can register for Hazelcast entry events so you will be notified when those events occur. Event Listeners are cluster-wide: when a listener is registered in one member of cluster, it is actually registered for events that originated at any member in the cluster. When a new member joins, events originated at the new member will also be delivered.

An Event is created only if you registered an event listener. If no listener is registered, then no event will be created. If you provided a predicate when you registered the event listener, pass the predicate before sending the event to the listener (node/client).

As a rule of thumb, your event listener should not implement heavy processes in its event methods which block the thread for a long time. If needed, you can use ExecutorService to transfer long running processes to another thread and thus offload the current listener thread.

7.1 Event Listeners for Hazelcast Nodes

Hazelcast offers the following event listeners:

- **Membership Listener** for cluster membership events.
- **Distributed Object Listener** for distributed object creation and destroy events.
- **Migration Listener** for partition migration start and complete events.
- **Partition Lost Listener** for partition lost events.
- **Lifecycle Listener** for HazelcastInstance lifecycle events.
- **Entry Listener** for IMap and MultiMap entry events (please refer to the Map Listener section).
- **Item Listener** for IQueue, ISet and IList item events (please refer to the Event Registration and Configuration parts of the sections Set and List).
- **Message Listener** for ITopic message events.
- **Client Listener** for client connection events.

7.1.1 Membership Listener

The Membership Listener allows to get notified for the following events.

- A new member is added to the cluster.
- An existing member leaves the cluster.
- An attribute of a member is changed. Please refer to the Member Attributes section to learn about member attributes.

The following is an example Membership Listener class.
public class ClusterMembershipListener  
    implements MembershipListener {

    public void memberAdded(MembershipEvent membershipEvent) {
        System.err.println("Added: " + membershipEvent);
    }

    public void memberRemoved(MembershipEvent membershipEvent) {
        System.err.println("Removed: " + membershipEvent);
    }

    public void memberAttributeChanged(MemberAttributeEvent memberAttributeEvent) {
        System.err.println("Member attribute changed: " + memberAttributeEvent);
    }

}

When a respective event is fired, the membership listener outputs the addresses of the members that joined and left, and also which attribute changed on which member.

7.1.2 Distributed Object Listener

The Distributed Object Listener notifies when a distributed object is created or destroyed throughout the cluster.
The following is an example Distributed Object Listener class.

public class Sample implements DistributedObjectListener {
    public static void main(String[] args) {
        Sample sample = new Sample();

        Config config = new Config();
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance(config);
        hazelcastInstance.addDistributedObjectListener(sample);

        Collection<DistributedObject> distributedObjects = hazelcastInstance.getDistributedObjects();
        for (DistributedObject distributedObject : distributedObjects) {
            System.out.println(distributedObject.getName() + "," + distributedObject.getId());
        }
    }

    @Override
    public void distributedObjectCreated(DistributedObjectEvent event) {
        DistributedObject instance = event.getDistributedObject();
        System.out.println("Created " + instance.getName() + "," + instance.getId());
    }

    @Override
    public void distributedObjectDestroyed(DistributedObjectEvent event) {
        DistributedObject instance = event.getDistributedObject();
        System.out.println("Destroyed " + instance.getName() + "," + instance.getId());
    }
}

When a respective event is fired, the distributed object listener outputs the event type, and the name, service (for example, if a Map service provides the distributed object, than it is a Map object), and ID of the object.
7.1. EVENT LISTENERS FOR HAZELCAST NODES

7.1.3 Migration Listener

The Migration Listener notifies for the following events:

- A partition migration is started.
- A partition migration is completed.
- A partition migration is failed.

The following is an example Migration Listener class.

```java
public class ClusterMigrationListener implements MigrationListener {
    @Override
    public void migrationStarted(MigrationEvent migrationEvent) {
        System.err.println("Started: " + migrationEvent);
    }
    @Override
    public void migrationCompleted(MigrationEvent migrationEvent) {
        System.err.println("Completed: " + migrationEvent);
    }
    @Override
    public void migrationFailed(MigrationEvent migrationEvent) {
        System.err.println("Failed: " + migrationEvent);
    }
}
```

When a respective event is fired, the migration listener outputs the partition ID, status of the migration, the old member and the new member. The following is an example output.

```
Started: MigrationEvent{partitionId=98, oldOwner=Member [127.0.0.1]:5701, newOwner=Member [127.0.0.1]:5702 this}
```

7.1.4 Partition Lost Listener

Hazelcast provides fault-tolerance by keeping multiple copies of your data. For each partition, one of your nodes become owner and some of the other nodes become replica nodes based on your configuration. Nevertheless, data loss may occur if a few nodes crash simultaneously.

Let’s consider the following example with three nodes: N1, N2, N3 for a given partition-0. N1 is owner of partition-0, N2 and N3 are the first and second replicas respectively. If N1 and N2 crash simultaneously, partition-0 loses its data that is configured with less than 2 backups. For instance, if we configure a map with 1 backup, that map loses its data in partition-0 since both owner and first replica of partition-0 have crashed. However, if we configure our map with 2 backups, it does not lose any data since a copy of partition-0’s data for the given map also resides in N3.

The Partition Lost Listener notifies for possible data loss occurrences with the information of how many replicas are lost for a partition. It listens to PartitionLostEvent instances. Partition lost events are dispatched per partition.

Partition loss detection is done after a node crash is detected by the other nodes and the crashed node is removed from the cluster. Please note that false-positive PartitionLostEvent instances may be fired on partial network split errors.

The following is an example of Partition Lost Listener.

```java
public class ConsoleLoggingPartitionLostListener implements PartitionLostListener {
    @Override
    public void partitionLost(PartitionLostEvent event) {
        System.out.println(event);
    }
}
```
When a PartitionLostEvent is fired, the partition lost listener given above outputs the partition ID, the replica index that is lost and the node that has detected the partition loss. The following is an example output.

com.hazelcast.partition.PartitionLostEvent{partitionId=242, lostBackupCount=0, eventSource=Address[192.168.2.49]:5701}

7.1.5 Lifecycle Listener

The Lifecycle Listener notifies for the following events:

- A member is starting.
- A member started.
- A member is shutting down.
- A member’s shutdown has completed.
- A member is merging with the cluster.
- A member’s merge operation has completed.
- A Hazelcast Client connected to the cluster.
- A Hazelcast Client disconnected from the cluster.

The following is an example Lifecycle Listener class.

```java
public class NodeLifecycleListener implements LifecycleListener {
    @Override
    public void stateChanged(LifecycleEvent event) {
        System.err.println(event);
    }
}
```

This listener is local to an individual node. It notifies the application that uses Hazelcast about the events mentioned above for a particular node.

7.1.6 Item Listener

The Item Listener is used by the Hazelcast IQueue, ISet and IList interfaces. It notifies when an item is added or removed.

The following is an example Item Listener class.

```java
public class Sample implements ItemListener {

    public static void main( String[] args ) {
        Sample sample = new Sample();
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        ISet<Price> set = hazelcastInstance.getSet( "default" );
        set.addItemListener( sample, true );

        Price price = new Price( 10, time1 )
        set.add( price );
        set.remove( price );
    }

    public void itemAdded( Object item ) {
        System.out.println( "Item added = " + item );
    }
}
```
7.2. EVENT LISTENERS FOR HAZELCAST CLIENTS

```java
public void itemRemoved( Object item ) {
    System.out.println( "Item removed = " + item );
}
```

### 7.1.7 Message Listener

The Message Listener is used by the ITopic interface. It notifies when a message is received for the registered topic. The following is an example Message Listener class.

```java
public class Sample implements MessageListener<MyEvent> {

    public static void main( String[] args ) {
        Sample sample = new Sample();
        HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
        ITopic topic = hazelcastInstance.getTopic( "default" );
        topic.addMessageListener( sample );
        topic.publish( new MyEvent() );
    }

    public void onMessage( Message<MyEvent> message ) {
        MyEvent myEvent = message.getMessageObject();
        System.out.println( "Message received = " + myEvent.toString() );
        if ( myEvent.isHeavyweight() ) {
            messageExecutor.execute( new Runnable() {
                public void run() {
                    doHeavyweightStuff( myEvent );
                }
            } );
        }
    }
}
```

### 7.1.8 Client Listener

The Client Listener is used by the Hazelcast nodes. It notifies the nodes when a client is connected to or disconnected from the cluster.

⚠️ **NOTE:** You can also add event listeners to a Hazelcast client. Please refer to Client Listenerconfig for the related information.

### 7.2 Event Listeners for Hazelcast Clients

You can add event listeners to a Hazelcast Java client. You can configure the following listeners to listen to the events on the client side. Please see the respective sections under the Event Listeners for Hazelcast Nodes section for example code.

- **Lifecycle Listener:** Notifies when the client is starting, started, shutting down and shutdown.
- **Membership Listener:** Notifies when a node joins to/leaves the cluster to which the client is connected, or when an attribute is changed in a node.
- **DistributedObject Listener:** Notifies when a distributed object is created or destroyed throughout the cluster to which the client is connected.
7.3 Global Event Configuration

- `hazelcast.event.queue.capacity`: default value is 1000000
- `hazelcast.event.queue.timeout.millis`: default value is 250
- `hazelcast.event.thread.count`: default value is 5

A striped executor in each node controls and dispatches the received events. This striped executor also guarantees the event order. For all events in Hazelcast, the order in which events are generated and the order in which they are published are guaranteed for given keys. For map and multimap, the order is preserved for the operations on the same key of the entry. For list, set, topic and queue, the order is preserved for events on that instance of the distributed data structure.

You achieve the order guarantee by making only one thread responsible for a particular set of events (entry events of a key in a map, item events of a collection, etc.) in `StripedExecutor`.

If the event queue reaches its capacity (`hazelcast.event.queue.capacity`) and the last item cannot be put into the event queue for the period specified in `hazelcast.event.queue.timeout.millis`, these events will be dropped with a warning message, such as “EventQueue overloaded”.

If event listeners perform a computation that takes a long time, the event queue can reach its maximum capacity and lose events. For map and multimap, you can configure `hazelcast.event.thread.count` to a higher value so that fewer collisions occur for keys, and therefore worker threads will not block each other in `StripedExecutor`. For list, set, topic and queue, you should offload heavy work to another thread. To preserve order guarantee, you should implement similar logic with `StripedExecutor` in the offloaded thread pool.

Please refer to the `Listener Configurations section` on how to configure each listener.
Chapter 8

Distributed Computing

From Wikipedia: Distributed computing refers to the use of distributed systems to solve computational problems. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers.

8.1 Executor Service

One of the coolest features of Java 1.5 is the Executor framework, which allows you to asynchronously execute your tasks (logical units of work), such as database query, complex calculation, and image rendering.

8.1.1 Executor Overview

The default implementation of this framework (ThreadPoolExecutor) is designed to run within a single JVM. In distributed systems, this implementation is not desired since you may want a task submitted in one JVM and processed in another one. Hazelcast offers IExecutorService for you to use in distributed environments: it implements java.util.concurrent.ExecutorService to serve the applications requiring computational and data processing power.

With IExecutorService, you can execute tasks asynchronously and perform other useful tasks. If your task execution takes longer than expected, you can cancel the task execution. Tasks should be Serializable since they will be distributed.

In the Java Executor framework, you implement tasks two ways: Callable or Runnable.

- Callable: If you need to return a value and submit to Executor, implement the task as java.util.concurrent.Callable.
- Runnable: If you do not need to return a value, implement the task as java.util.concurrent.Runnable.

8.1.1.1 Callable

In Hazelcast, when you implement a task as java.util.concurrent.Callable (a task that returns a value), you implement Callable and Serializable.

Below is an example of a Callable.

```java
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.HazelcastInstanceAware;
import com.hazelcast.core.IMap;

import java.io.Serializable;
import java.util.concurrent.Callable;

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

private transient HazelcastInstance hazelcastInstance;

public void setHazelcastInstance( HazelcastInstance hazelcastInstance ) {
    this.hazelcastInstance = hazelcastInstance;
}

public Integer call() throws Exception {
    IMap<String, Integer> map = hazelcastInstance.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;
}
}

Another example is the Echo callable below. In its call() method, it returns the local member and the input passed in. Remember that instance.getCluster().getLocalMember().toString() returns the member's address (IP + port) in String form, just to see which member actually executed the code for our example. Of course, the call() method can do and return anything you like.

import java.util.concurrent.Callable;
import java.io.Serializable;

public class Echo implements Callable<String>, Serializable {
    String input = null;

    public Echo() {
    }

    public Echo(String input) {
        this.input = input;
    }

    public String call() {
        Config cfg = new Config();
        HazelcastInstance instance = Hazelcast.newHazelcastInstance(cfg);
        return instance.getCluster().getLocalMember().toString() + ":" + input;
    }
}

To execute a task with the executor framework:

- Obtain an ExecutorService instance (generally via Executors).
- Submit a task which returns a Future.
- After executing the task, you do not have to wait for the execution to complete, you can process other things.
- When ready, use the Future object to retrieve the result as shown in the code example below.

Below, the Echo task is executed.

ExecutorService executorService = Executors.newSingleThreadExecutor();
Future<String> future = executorService.submit( new Echo( "myinput" ) );
while it is executing, do some useful stuff
when ready, get the result of your execution
String result = future.get();

Please note that the Echo callable in the above code sample also implements a Serializable interface, since it may be sent to another JVM to be processed.

**NOTE:** When a task is deserialized, HazelcastInstance needs to be accessed. To do this, the task should implement HazelcastInstanceAware interface. Please see the HazelcastInstanceAware Interface section for more information.

### 8.1.1.2 Runnable

In Hazelcast, when you implement a task as java.util.concurrent.runnable (a task that does not return a value), you implement Runnable and Serializable.

Below is Runnable example code. It is a task that waits for some time and echoes a message.

```java
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 );
    }
}
```

To execute the task:
* Retrieve the Executor from HazelcastInstance.
* Submit the tasks to the Executor.

Now let’s write a class that submits and executes these echo messages. Executor is retrieved from HazelcastInstance and 1000 echo tasks are submitted.

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

### 8.1.1.3 Executor Thread Configuration

By default, Executor is configured to have 8 threads in the pool. You can change that with the pool-size property in the declarative configuration (hazelcast.xml). An example is shown below (using the above Executor).

```xml
<impl-ref>
    <impl>hazelcast.core.HazelcastInstanceImpl</impl>
    <factory>executor</factory>
    <properties>
        <property>threadPool-size</property>
        <value>16</value>
    </properties>
</impl-ref>
```
8.1.1.4 Scaling

You can scale the Executor service both vertically (scale up) and horizontally (scale out).

To scale up, you should improve the processing capacity of the JVM. You can do this by increasing the `pool-size` property mentioned in the Executor Thread Configuration section (i.e., increasing the thread count). However, please be aware of your JVM’s capacity. If you think it cannot handle such an additional load caused by increasing the thread count, you may want to consider improving the JVM’s resources (CPU, memory, etc.). As an example, set the `pool-size` to 5 and run the above `MasterMember`. You will see that `EchoTask` is run as soon as it is produced.

To scale out, more JVMs should be added instead of increasing only one JVM’s capacity. In reality, you may want to expand your cluster by adding more physical or virtual machines. For example, in the `EchoTask` example in the Runnable section, you can create another Hazelcast instance. That instance will automatically get involved in the executions started in `MasterMember` and start processing.

8.1.2 Execution

The distributed executor service is a distributed implementation of `java.util.concurrent.ExecutorService`. It allows you to execute your code in the cluster. In this section, the code examples are based on the `Echo` class above (please note that the `Echo` class is `Serializable`). The code examples show how Hazelcast can execute your code (Runnable, Callable):

- on a specific cluster member you choose,
- on the member owning the key you choose,
- on the member Hazelcast will pick, and
- on all or subset of the cluster members.

```java
import com.hazelcast.core.Member;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.IExecutorService;
import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import java.util.Set;
public void echoOnTheMember( String input, Member member ) throws Exception {
    Callable<String> task = new Echo( input );
    HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
    IExecutorService executorService = hazelcastInstance.getExecutorService( "default" );
    Future<String> future = executorService.submitToMember( task, member );
    String echoResult = future.get();
}

public void echoOnTheMemberOwningTheKey( String input, Object key ) throws Exception {
    Callable<String> task = new Echo( input );
    HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
```
8.1. EXECUTOR SERVICE

```java
IExecutorService executorService =
    hazelcastInstance.getExecutorService("default");

Future<String> future = executorService.submitToKeyOwner(task, key);
String echoResult = future.get();
}

public void echoOnSomewhere(String input) throws Exception {
    HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
    IExecutorService executorService =
        hazelcastInstance.getExecutorService("default");

    Future<String> future = executorService.submit(new Echo(input));
    String echoResult = future.get();
}

public void echoOnMembers(String input, Set<Member> members) throws Exception {
    HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
    IExecutorService executorService =
        hazelcastInstance.getExecutorService("default");

    Map<Member, Future<String>> futures = executorService
        .submitToMembers(new Echo(input), members);

    for (Future<String> future : futures.values()) {
        String echoResult = future.get();
        // ...
    }
}
```

**NOTE:** You can obtain the set of cluster members via `HazelcastInstance#getCluster().getMembers()` call.

8.1.3 Execution Cancellation

A task in the code you execute in a cluster might take longer than expected. If you cannot stop/cancel that task, it will keep eating your resources.

To cancel a task, you can use the standard Java executor framework’s `cancel()` API. This framework encourages us to code and design for cancellations, a highly ignored part of software development.

8.1.3.1 Example Task to Cancel

The Fibonacci callable class below calculates the Fibonacci number for a given number. In the `calculate` method, we check if the current thread is interrupted so that the code can respond to cancellations once the execution is started.

```java
public class Fibonacci<Long> implements Callable<Long>, Serializable {
    int input = 0;

    public Fibonacci() {
    }

    public Fibonacci(int input) {
        this.input = input;
    }
```
public Long call() {
    return calculate( input );
}

private long calculate( int n ) {
    if ( Thread.currentThread().isInterrupted() ) {
        return 0;
    }
    if ( n <= 1 ) {
        return n;
    } else {
        return calculate( n - 1 ) + calculate( n - 2 );
    }
}

8.1.3.2 Example Method to Execute and Cancel the Task

The fib() method below submits the Fibonacci calculation task above for number ‘n’ and waits a maximum of 3 seconds for the result. If the execution does not completed in 3 seconds, future.get() will throw a TimeoutException and upon catching it, we cancel the execution, saving some CPU cycles.

long fib( int n ) throws Exception {
    HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
    IExecutorService es = hazelcastInstance.getExecutorService();
    Future future = es.submit( new Fibonacci( n ) );
    try {
        return future.get( 3, TimeUnit.SECONDS );
    } catch ( TimeoutException e ) {
        future.cancel( true );
    }
    return -1;
}

fib(20) will probably take less than 3 seconds. However, fib(50) will take much longer. (This is not an example for writing better Fibonacci calculation code, but for showing how to cancel a running execution that takes too long.) The method future.cancel(false) can only cancel execution before it is running (executing), but future.cancel(true) can interrupt running executions if your code is able to handle the interruption. If you are willing to cancel an already running task, then your task should be designed to handle interruptions. If the calculate (int n) method did not have the (Thread.currentThread().isInterrupted()) line, then you would not be able to cancel the execution after it is started.

8.1.4 Execution Callback

You can use the ExecutionCallback offered by Hazelcast to asynchronously be notified when the execution is done.

8.1.4.1 Example Task to Callback

Let’s use the Fibonacci series to explain this. The example code below is the calculation that will be executed. Note that it is Callable and Serializable.

public class Fibonacci<Long> implements Callable<Long>, Serializable {
    int input = 0;
8.1. EXECUTOR SERVICE

```java
public Fibonacci() {
}

public Fibonacci( int input ) {
    this.input = input;
}

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

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

8.1.4.2 Example Method to Callback the Task

The example code below submits the Fibonacci calculation to `ExecutionCallback` and prints the result asynchronously. `ExecutionCallback` has the methods `onResponse` and `onFailure`. In this example code, `onResponse` is called upon a valid response and prints the calculation result, whereas `onFailure` is called upon a failure and prints the stacktrace.

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.ExecutionCallback;
import com.hazelcast.core.IExecutorService;
import java.util.concurrent.Future;

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
IExecutorService es = hazelcastInstance.getExecutorService();
Callable<Long> task = new Fibonacci( 10 );

es.submit(task, new ExecutionCallback<Long>() {
    @Override
    public void onResponse( Long response ) {
        System.out.println( "Fibonacci calculation result = " + response );
    }

    @Override
    public void onFailure( Throwable t ) {
        t.printStackTrace();
    }
});
```

8.1.5 Execution Member Selection

As previously mentioned, it is possible to indicate where in the Hazelcast cluster the `Runnable` or `Callable` is executed. Usually, you will execute these in the cluster based on the location of a key, set of keys or just allow Hazelcast to select a member.

If you want more control over where your code runs, you can use the `MemberSelector` interface. For example, you may want certain tasks to run only on certain members, or you may wish to implement some form of custom
load balancing regime. The MemberSelector is an interface that you can implement and then provide to the IExecutorService when you submit or execute.

The select(Member) method is called for every available member in the cluster and it is up to the implementation to decide if the member is going to be used or not.

In a simple example shown below, we select the cluster members based on the presence of an attribute.

```java
public class MyMemberSelector implements MemberSelector {
    public boolean select(Member member) {
        return Boolean.TRUE.equals(member.getAttribute("my.special.executor"));
    }
}
```

8.2 Entry Processor

Hazelcast supports entry processing. An entry processor is a function that executes your code on a map entry in an atomic way.

An entry processor is a good option if you perform bulk processing on an IMap. Usually, you perform a loop of keys: executing IMap.get(key), mutating the value, and finally putting the entry back in the map using IMap.put(key, value). If you perform this process from a client or from a member where the keys do not exist, you effectively perform 2 network hops for each update: the first to retrieve the data and the second to update the mutated value.

If you are doing the above, you should consider using entry processors. An entry processor executes a read and updates upon the member where the data resides. This eliminates the costly network hops described previously.

8.2.1 Performing Fast In-Memory Map Operations

An entry processor enables fast in-memory operations on your map without you having to worry about locks or concurrency issues. It can be applied to a single map entry or to all map entries. It supports choosing target entries using predicates. You do not need any explicit lock on entry: Hazelcast locks the entry, runs the EntryProcessor, and then unlocks the entry.

Hazelcast sends the entry processor to each cluster member and these members apply it to map entries. Therefore, if you add more members, your processing is completed faster.

8.2.1.1 Using OBJECT In-Memory Format

If entry processing is the major operation for a map and if the map consists of complex objects, you should use OBJECT as the in-memory-format to minimize serialization cost. By default, the entry value is stored as a byte array (BINARY format). When it is stored as an object (OBJECT format), then the entry processor is applied directly on the object. In that case, no serialization or deserialization is performed. But if there is a defined event listener, a new entry value will be serialized when passing to the event publisher service.

⚠️ **NOTE:** When in-memory-format is OBJECT, old value of the updated entry will be null.

8.2.1.2 Entry Processing with IMap Methods

The methods below are in the IMap interface for entry processing.

- `executeOnKey` processes an entry mapped by a key.
- `executeOnKeys` processes entries mapped by a collection of keys.
- `submitToKey` processes an entry mapped by a key while listening to event status.
8.2. ENTRY PROCESSOR

- **executeOnEntries** processes all entries in a map.
- **executeOnEntries** can also process all entries in a map with a defined predicate.

```java
/**
 * Applies the user defined EntryProcessor to the entry mapped by the key.
 * Returns the object which is the result of the process() method of EntryProcessor.
 */
Object executeOnKey( K key, EntryProcessor entryProcessor );
```

```java
/**
 * Applies the user defined EntryProcessor to the entries mapped by the key with
 * specified ExecutionCallback to listen to event status and return immediately.
 */
void submitToKey( K key, EntryProcessor entryProcessor, ExecutionCallback callback );
```

```java
/**
 * Applies the user defined EntryProcessor to all entries in the map.
 * Returns the results mapped by each key in the map.
 */
Map<K, Object> executeOnEntries( EntryProcessor entryProcessor );
```

```java
/**
 * Applies the user defined EntryProcessor to the entries in the map which satisfies
 * provided predicate.
 * Returns the results mapped by each key in the map.
 */
Map<K, Object> executeOnEntries( EntryProcessor entryProcessor, Predicate predicate );
```

**NOTE:** Entry Processors run via Operation Threads that are dedicated to specific partitions. Therefore, with long running Entry Processor executions, other partition operations such as `map.put(key)` cannot be processed. With this in mind, it is a good practice to make your Entry Processor executions as quick as possible.

8.2.1.3 EntryProcessor Interface

The following is the `EntryProcessor` interface:

```java
public interface EntryProcessor<K, V> extends Serializable {
    Object process( Map.Entry<K, V> entry );
    EntryBackupProcessor<K, V> getBackupProcessor();
}
```

**NOTE:** If you want to execute a task on a single key, you can also use `executeOnKeyOwner` provided by Executor Service. But, in this case, you need to perform a lock and serialization.

When using `executeOnEntries` method, if the number of entries is high and you do need the results, then returning null in `process()` method is a good practice. By this way, results of the processing is not stored in the map and hence out of memory errors are eliminated.
8.2.1.4 Processing Backup Entries

If your code modifies the data, then you should also provide a processor for backup entries. This is required to prevent the primary map entries from having different values than the backups; it causes the entry processor to be applied both on the primary and backup entries.

```java
public interface EntryBackupProcessor<K, V> extends Serializable {
    void processBackup( Map.Entry<K, V> entry );
}
```

**NOTE:** There is a possibility that an Entry Processor can see that a key exists but its backup processor may not find it at the runtime due to an unsent backup of a previous operation (e.g., a previous put operation). In those situations, Hazelcast internally eventually will synchronize those owner and backup partitions so you will not lose any data. When coding an EntryBackupProcessor, you should take that case into account, otherwise NullPointerException can be seen since Map.Entry.getValue() may return null.

8.2.2 Creating an Entry Processor

The EntryProcessorTest class has the following methods.

- **testMapEntryProcessor** puts one map entry and calls executeOnKey to process that map entry.
- **testMapEntryProcessorAllKeys** puts all the entries in a map and calls executeOnEntries to process all the entries.

The static class IncrementingEntryProcessor creates an entry processor to process the map entries in the EntryProcessorTest class. It creates the entry processor class by:

- implementing the map interfaces EntryProcessor and EntryBackupProcessor.
- implementing the java.io.Serializable interface.
- implementing the EntryProcessor methods process and getBackupProcessor.
- implementing the EntryBackupProcessor method processBackup.

```java
public class EntryProcessorTest {

    @Test
    public void testMapEntryProcessor() throws InterruptedException {
        Config config = new Config().getMapConfig( "default" )
            .setInMemoryFormat( MapConfig.InMemoryFormat.OBJECT );

        HazelcastInstance hazelcastInstance1 = Hazelcast.newHazelcastInstance( config );
        HazelcastInstance hazelcastInstance2 = Hazelcast.newHazelcastInstance( config );
        IMap<Integer, Integer> map = hazelcastInstance1.getMap( "mapEntryProcessor" );
        map.put( 1, 1 );
        EntryProcessor entryProcessor = new IncrementingEntryProcessor();
        map.executeOnKey( 1, entryProcessor );
        assertEquals( map.get( 1 ), (Object) 2 );
        hazelcastInstance1.getLifecycleService().shutdown();
        hazelcastInstance2.getLifecycleService().shutdown();
    }

    @Test
    public void testMapEntryProcessorAllKeys() throws InterruptedException {
        StaticNodeFactory factory = new StaticNodeFactory( 2 );
        Config config = new Config().getMapConfig( "default" )
            .setInMemoryFormat( MapConfig.InMemoryFormat.OBJECT );
```
HazelcastInstance hazelcastInstance1 = factory.newHazelcastInstance(config);
HazelcastInstance hazelcastInstance2 = factory.newHazelcastInstance(config);
IMap<Integer, Integer> map = hazelcastInstance1
    .getMap("mapEntryProcessorAllKeys");

int size = 100;
for (int i = 0; i < size; i++) {
    map.put(i, i);
}
EntryProcessor entryProcessor = new IncrementingEntryProcessor();
Map<Integer, Object> res = map.executeOnEntries(entryProcessor);
for (int i = 0; i < size; i++) {
    assertEquals(map.get(i), (Object)(i + 1));
}
for (int i = 0; i < size; i++) {
    assertEquals(map.get(i) + 1, res.get(i));
}
hazelcastInstance1.getLifecycleService().shutdown();
hazelcastInstance2.getLifecycleService().shutdown();

static class IncrementingEntryProcessor
    implements EntryProcessor, EntryBackupProcessor, Serializable {

    public Object process(Map.Entry entry) {
        Integer value = (Integer) entry.getValue();
        entry.setValue(value + 1);
        return value + 1;
    }

    public EntryBackupProcessor getBackupProcessor() {
        return IncrementingEntryProcessor.this;
    }

    public void processBackup(Map.Entry entry) {
        entry.setValue((Integer) entry.getValue() + 1);
    }
}

NOTE: You should explicitly call setValue method of Map.Entry when modifying data in Entry Processor.
Otherwise, Entry Processor will be accepted as read-only.

NOTE: An Entry Processor instance is not thread safe. If you are storing partition specific state between
invocations, be sure to register this in a thread-local. An Entry Processor instance can be used by multiple partition
threads.

8.2.3 Abstract Entry Processor

You can use the AbstractEntryProcessor class when the same processing will be performed both on the primary
and backup map entries (i.e. the same logic applies to them). If you use Entry Processor, you need to apply
the same logic to the backup entries separately. The AbstractEntryProcessor class makes this primary/backup
processing easier.
The code below shows the Hazelcast `AbstractEntryProcessor` class. You can use it to create your own Abstract Entry Processor.

```java
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);
        }
    }
}
```

In the above code, the method `getBackupProcessor` returns an `EntryBackupProcessor` instance. This means the same processing will be applied to both the primary and backup entries. If you want to apply the processing only upon the primary entries, then make the `getBackupProcessor` method return null.

**NOTE:** Beware of the null issue described at the note in the Processing Backup Entries section. Due to a yet unsent backup from a previous operation, an `EntryBackupProcessor` may temporarily receive null from `Map.Entry.getValue()` even though the value actually exists in the map. If you decide to use `AbstractEntryProcessor`, make sure your code logic is not sensitive to null values, or you may encounter `NullPointerException` during runtime.
Chapter 9

Distributed Query

Distributed queries access data from multiple data sources stored on either the same or different computers.

9.1 Query Overview

Hazelcast partitions your data and spreads it across cluster of servers. You can iterate over the map entries and look for certain entries (specified by predicates) you are interested in. However, this is not very efficient because you will have to bring the entire entry set and iterate locally. Instead, Hazelcast allows you to run distributed queries on your distributed map.

9.1.1 How It Works

1. The requested predicate is sent to each member in the cluster.
2. Each member looks at its own local entries and filters them according to the predicate. At this stage, key/value pairs of the entries are deserialized and then passed to the predicate.
3. The predicate requester merges all the results coming from each member into a single set.

If you add new members to the cluster, the partition count for each member is reduced and hence the time spent by each member on iterating its entries is reduced. Therefore, the above querying approach is highly scalable. Another reason it is highly scalable is the pool of partition threads that evaluates the entries concurrently in each member. The network traffic is also reduced since only filtered data is sent to the requester.

Hazelcast offers the following APIs for distributed query purposes:

- Criteria API
- Distributed SQL Query

9.1.2 Employee Map Query Example

Assume that you have an “employee” map containing values of Employee objects, as coded below.

```java
import java.io.Serializable;

public class Employee implements Serializable {
    private String name;
    private int age;
    private boolean active;
    private double salary;
```
public Employee(String name, int age, boolean live, double price) {
    this.name = name;
    this.age = age;
    this.active = live;
    this.salary = price;
}

public Employee() {
}

public String getName() {
    return name;
}

public int getAge() {
    return age;
}

public double getSalary() {
    return salary;
}

public boolean isActive() {
    return active;
}

Now, let's look for the employees who are active and have an age less than 30 using the aforementioned APIs (Criteria API and Distributed SQL Query). The following subsections describe each query mechanism for this example.

![NOTE: When using Portable objects, if one field of an object exists on one node but does not exist on another one, Hazelcast does not throw an unknown field exception. Instead, Hazelcast treats that predicate, which tries to perform a query on an unknown field, as an always false predicate.]

### 9.1.3 Criteria API

Criteria API is a programming interface offered by Hazelcast that is similar to the Java Persistence Query Language (JPQL). Below is the code for the above example query.

```java
import com.hazelcast.core.IMap;
import com.hazelcast.query.Predicate;
import com.hazelcast.query.PredicateBuilder;
import com.hazelcast.query.EntryObject;
import com.hazelcast.config.Config;

IMap<String, Employee> map = hazelcastInstance.getMap( "employee" );

Predicate predicate = new PredicateBuilder().getEntryObject().is( "active" ).and( e -> e.get("age") ).lessThan( 30 )

Set<Employee> employees = map.values( predicate );
```

In the above example code, `predicate` verifies whether the entry is active and its `age` value is less than 30. This `predicate` is applied to the `employee` map using the `map.values(predicate)` method. This method sends the
predicate to all cluster members and merges the results coming from them. Since the predicate is communicated between the members, it needs to be serializable.

**NOTE:** Predicates can also be applied to **keySet**, **entrySet** and **localKeySet** of Hazelcast distributed map.

### 9.1.3.1 Predicates Class

The **Predicates** class offered by Hazelcast includes many operators for your query requirements. Some of them are explained below.

- **equal**: checks if the result of an expression is equal to a given value.
- **notEqual**: checks if the result of an expression is not equal to a given 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 placeholder for many characters, (underscore) is 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 2 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.

**RELATED INFORMATION**

Please see the **Predicates class** for all predicates provided.

### 9.1.3.2 Joining Predicates with AND, OR, NOT

Predicates can be joined using the **and**, **or** and **not** operators, as shown in the below examples.

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

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

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

You can simplify predicate usage with the PredicateBuilder class, which offers simpler predicate building. Please see the below example code which selects all people with a certain name and age.

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

9.1.4 Distributed SQL Query

com.hazelcast.query.SqlPredicate takes the regular SQL where clause. Here is an example:

```java
IMap<Employee> map = hazelcastInstance.getMap("employee");
Set<Employee> employees = map.values(new SqlPredicate("active AND age < 30"));
```

9.1.4.1 Supported SQL Syntax

**AND/OR:** <expression> AND <expression> AND <expression>...

- active AND age>30
- active=false OR age = 45 OR name = 'Joe'
- active AND ( age > 20 OR salary < 60000 )

**Equality:** =, !=, <, <=, >, >=

- <expression> = value
- age <= 30
- name = "Joe"
- salary != 50000

**BETWEEN:** <attribute> [NOT] BETWEEN <value1> AND <value2>

- age BETWEEN 20 AND 33 ( same as age >= 20 AND age <= 33 )
- age NOT BETWEEN 30 AND 40 ( same as age < 30 OR age > 40 )

**LIKE:** <attribute> [NOT] LIKE 'expression'

The % (percentage sign) is placeholder for multiple characters, an _ (underscore) is placeholder for only one character.

- 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:** <attribute> [NOT] IN (val1, val2, ...)

- age IN ( 20, 30, 40 )
- age NOT IN ( 60, 70 )
- active AND ( salary >= 50000 OR ( age NOT BETWEEN 20 AND 30 ) )
- age IN ( 20, 30, 40 ) AND salary BETWEEN ( 50000, 80000 )
9.1. Query Overview

9.1.5 Paging Predicate

Hazelcast provides paging for defined predicates. With its `PagingPredicate` class, you can get a collection of keys, values, or entries page by page by filtering them with predicates and giving the size of the pages. Also, you can sort the entries by specifying comparators.

In the example code below, the `greaterEqual` predicate gets values from the “students” map. This predicate has a filter to retrieve the objects with a “age” greater than or equal to 18. Then a `PagingPredicate` is constructed in which the page size is 5, so there will be 5 objects in each page.

The first time the values are called creates the first page. You can get the subsequent pages by using the `nextPage()` method of `PagingPredicate` and querying the map again with the updated `PagingPredicate`.

```java
IMap<Integer, Student> map = hazelcastInstance.getMap( "students" );
Predicate greaterEqual = Predicates.greaterEqual( "age", 18 );
PagingPredicate pagingPredicate = new PagingPredicate( greaterEqual, 5 );
// Retrieve the first page
Collection<Student> values = map.values( pagingPredicate );
...
// Set up next page
pagingPredicate.nextPage();
// Retrieve next page
values = map.values( pagingPredicate );
...
```

If a comparator is not specified for `PagingPredicate`, but you want to get a collection of keys or values page by page, this collection must be an instance of `Comparable` (i.e. it must implement `java.lang.Comparable`). Otherwise, the `java.lang.IllegalArgumentException` exception is thrown.

Paging Predicate, also known as Order & Limit, is not supported in Transactional Context.

⚠️ **NOTE:** Currently, random page accessing is not supported.

**RELATED INFORMATION**

Please refer to the Javadoc for all predicates.

9.1.6 Indexing

Hazelcast distributed queries will run on each member in parallel and only results will return the conn. When a query runs on a member, Hazelcast will iterate through the entire owned entries and find the matching ones. This can be made faster by indexing the mostly queried fields, just like you would do for your database. Indexing will add overhead for each write operation but queries will be a lot faster. If you query your map a lot, make sure to add indexes for the most frequently queried fields. For example, if your `active` and `age < 30` query, make sure you add index for `active` and `age` fields. Here is how to do it.

```java
IMap map = hazelcastInstance.getMap( "employees" );
// ordered, since we have ranged queries for this field
map.addIndex( "age", true );
// not ordered, because boolean field cannot have range
map.addIndex( "active", false );
```

`IMap.addIndex(fieldName, ordered)` is used for adding index. For each indexed field, if you have ranged queries such as `age>30`, `age BETWEEN 40 AND 60`, then you should set the `ordered` parameter to `true`. Otherwise, set it to `false`.

Also, you can define `IMap` indexes in configuration. An example is shown below.
You can also define IMap indexes using programmatic configuration, as in the example below.

```java
mapConfig.addMapIndexConfig( new MapIndexConfig( "name", false ) );
mapConfig.addMapIndexConfig( new MapIndexConfig( "age", true ) );
```

The following is the Spring declarative configuration for the same sample.

```xml
<hz:map name="default">
    <hz:index attribute="name"/>
    <hz:index attribute="age" ordered="true"/>
</hz:map>
```

**NOTE:** Non-primitive types to be indexed should implement `Comparable`.

### 9.1.7 Query Thread Configuration

You can change the size of the thread pool dedicated to query operations using the `pool-size` property. Below is an example of that declarative configuration.

```xml
<executor-service name="hz:query">
    <pool-size>100</pool-size>
</executor-service>
```

Below is an example of the equivalent programmatic configuration.

```java
Config cfg = new Config();
cfg.getExecutorConfig("hz:query").setPoolSize(100);
```

### 9.2 MapReduce

You have likely heard about MapReduce ever since Google released its research white paper on this concept. With Hadoop as the most common and well known implementation, MapReduce gained a broad audience and made it into all kinds of business applications dominated by data warehouses.

MapReduce is a software framework for processing large amounts of data in a distributed way. Therefore, the processing is normally spread over several machines. The basic idea behind MapReduce is to map your source data into a collection of key-value pairs and reducing those pairs, grouped by key, in a second step towards the final result.

The main idea can be summarized with the following steps.

1. Read the source data.
2. Map the data to one or multiple key-value pairs.
3. Reduce all pairs with the same key.
9.2. MAPREDUCE

Use Cases

The best known examples for MapReduce algorithms are text processing tools, such as counting the word frequency in large texts or websites. Apart from that, there are more interesting examples of use cases listed below.

- Log Analysis
- Data Querying
- Aggregation and summing
- Distributed Sort
- ETL (Extract Transform Load)
- Credit and Risk management
- Fraud detection
- and more...

9.2.1 MapReduce Essentials

This section will give a deeper insight on the MapReduce pattern and helps you understand the semantics behind the different MapReduce phases and how they are implemented in Hazelcast.

In addition to this, the following sections compare Hadoop and Hazelcast MapReduce implementations to help adopters with Hadoop backgrounds to quickly get familiar with Hazelcast MapReduce.

9.2.1.1 MapReduce Workflow Example

The flowchart below demonstrates the basic workflow of the word count example (distributed occurrences analysis) mentioned in the MapReduce section. From left to right, it iterates over all the entries of a data structure (in this case an IMap). In the mapping phase, it splits the sentence into single words and emits a key-value pair per word: the word is the key, 1 is the value. In the next phase, values are collected (grouped) and transported to their corresponding reducers, where they are eventually reduced to a single key-value pair, the value being the number of occurrences of the word. At the last step, the different reducer results are grouped up to the final result and returned to the requester.

In pseudo code, the corresponding map and reduce function would look like the following. A Hazelcast code example will be shown in the next section.

```java
map( key:String, document:String ):Void ->
    for each w:word in document:
        emit( w, 1 )

reduce( word:String, counts:List[Int] ):Int ->
    return sum( counts )
```

9.2.1.2 MapReduce Phases

As seen in the workflow example, a MapReduce process consists of multiple phases. The original MapReduce pattern describes two phases (map, reduce) and one optional phase (combine). In Hazelcast, these phases are either only existing virtually to explain the data flow or are executed in parallel during the real operation while the general idea is still persisting.

\[(K \times V)^* \rightarrow (L \times W)^*\]
\[[(k_1, v_1), \ldots, (k_n, v_n)] \rightarrow [(l_1, w_1), \ldots, (l_m, w_m)]\]

Mapping Phase

The mapping phase iterates all key-value pairs of any kind of legal input source. The mapper then analyzes the input pairs and emits zero or more new key-value pairs.

\[K \times V \rightarrow (L \times W)^*\]
(k, v) -> [(l₁, w₁), ..., (lₙ, wₙ)]

**Combine Phase**

In the combine phase, multiple key-value pairs with the same key are collected and combined to an intermediate result before being send to the reducers. **Combine phase is also optional in Hazelcast, but is highly recommended to lower the traffic.**

In terms of the word count example, this can be explained using the sentences “Saturn is a planet but the Earth is a planet, too”. As shown above, we would send two key-value pairs (planet, 1). The registered combiner now collects those two pairs and combines them into an intermediate result of (planet, 2). Instead of two key-value pairs sent through the wire, there is now only one for the key “planet”.

The pseudo code for a combiner is similar to the reducer.

```java
combine( word: String, counts: List[Int] ): Void ->
  emit( word, sum( counts ) )
```

**Grouping / Shuffling Phase**

The grouping or shuffling phase only exists virtually in Hazelcast since it is not a real phase; emitted key-value pairs with the same key are always transferred to the same reducer in the same job. They are grouped together, which is equivalent to the shuffling phase.

**Reducing Phase**

In the reducing phase, the collected intermediate key-value pairs are reduced by their keys to build the final by-key result. This value can be a sum of all the emitted values of the same key, an average value, or something completely different, depending on the use case.

Here is a reduced representation of this phase.

L x W* -> X*  
(l, [w₁, ..., wn]) -> [x₁, ..., xn]
9.2. MAPREDUCE

Producing the Final Result

This is not a real MapReduce phase, but it is the final step in Hazelcast after all reducers are notified that reducing has finished. The original job initiator then requests all reduced results and builds the final result.

9.2.1.3 Additional MapReduce Resources

The Internet is full of useful resources to find deeper information on MapReduce. Below is a short collection of more introduction material. In addition, there are books written about all kinds of MapReduce patterns and how to write a MapReduce function for your use case. To name them all is out of scope of this documentation.

- http://research.google.com/archive/mapreduce.html
- http://en.wikipedia.org/wiki/MapReduce
- http://ksat.me/map-reduce-a-really-simple-introduction-kloudo/
- http://www.slideshare.net/franebandov/an-introduction-to-mapreduce-6789635

9.2.2 Introduction to MapReduce API

This section explains the basics of the Hazelcast MapReduce framework. While walking through the different API classes, we will build the word count example that was discussed earlier and create it step by step.

The Hazelcast API for MapReduce operations consists of a fluent DSL-like configuration syntax to build and submit jobs. JobTracker is the basic entry point to all MapReduce operations and is retrieved from com.hazelcast.core.HazelcastInstance by calling getJobTracker and supplying the name of the required JobTracker configuration. The configuration for JobTrackers will be discussed later, for now we focus on the API itself. In addition, the complete submission part of the API is built to support a fully reactive way of programming.

To give an easy introduction to people used to Hadoop, we created the class names to be as familiar as possible to their counterparts on Hadoop. That means while most users will recognize a lot of similar sounding classes, the way to configure the jobs is more fluent due to the DSL-like styled API.

While building the example, we will go through as many options as possible, e.g. we create a specialized JobTracker configuration (at the end). Special JobTracker configuration is not required, because for all other Hazelcast features you can use “default” as the configuration name. However, special configurations offer better options to predict behavior of the framework execution.

The full example is available here as a ready to run Maven project.

9.2.2.1 JobTracker

JobTracker creates Job instances, whereas every instance of com.hazelcast.mapreduce.Job defines a single MapReduce configuration. The same Job can be submitted multiple times, no matter if it is executed in parallel or after the previous execution is finished.

**NOTE:** After retrieving the JobTracker, be aware that it should only be used with data structures derived from the same HazelcastInstance. Otherwise, you can get unexpected behavior.

To retrieve a JobTracker from Hazelcast, we will start by using the “default” configuration for convenience reasons to show the basic way.

```java
import com.hazelcast.mapreduce.*;

JobTracker jobTracker = hazelcastInstance.getJobTracker( "default" );
```

JobTracker is retrieved using the same kind of entry point as most other Hazelcast features. After building the cluster connection, you use the created HazelcastInstance to request the configured (or default) JobTracker from Hazelcast.
The next step will be to create a new Job and configure it to execute our first MapReduce request against cluster data.

### 9.2.2.2 Job

As mentioned in the JobTracker section, a Job is created using the retrieved JobTracker instance. A Job defines exactly one configuration of a MapReduce task. Mapper, combiner and reducers will be defined per job but since the Job instance is only a configuration, it is possible to be submitted multiple times, no matter if executions happening in parallel or one after the other.

A submitted job is always identified using a unique combination of the JobTracker’s name and a jobId generated on submit-time. The way for retrieving the jobId will be shown in one of the later sections.

To create a Job, a second class com.hazelcast.mapreduce.KeyValueSource is necessary. We will have a deeper look at the KeyValueSource class in the next section, for now it is enough to know that it is used to wrap any kind of data or data structure into a well defined set of key-value pairs.

Below example code is a direct follow up of the example of the JobTracker section and reuses the already created HazelcastInstance and JobTracker instances.

We start by retrieving an instance of our data map and create the Job instance afterwards. Implementations used to configure the Job will be discussed while walking further through the API documentation, they are not yet discussed.

⚠️ **NOTE:** Since the Job class is highly dependent upon generics to support type safety, the generics change over time and may not be assignment compatible to old variable types. To make use of the full potential of the fluent API, we recommend you use fluent method chaining as shown in this example to prevent the need for too many variables.

```java
IMap<String, String> map = hazelcastInstance.getMap( "articles" );
KeyValueSource<String, String> source = KeyValueSource.fromMap( map );
Job<String, String> job = jobTracker.newJob( source );

ICompletableFuture<Map<String, Long>> future = job
  .mapper( new TokenizerMapper() )
  .combiner( new WordCountCombinerFactory() )
  .reducer( new WordCountReducerFactory() )
  .submit();

// Attach a callback listener
future.andThen( buildCallback() );

// Wait and retrieve the result
Map<String, Long> result = future.get();
```

As seen above, we create the Job instance and define a mapper, combiner, reducer and eventually submit the request to the cluster. The submit method returns an ICompletableFuture that can be used to attach our callbacks or just to wait for the result to be processed in a blocking fashion.

There are more options available for job configurations such as defining a general chunk size or on what keys the operation will operate. For more information, please refer to the Javadoc matching your Hazelcast version.

### 9.2.2.3 KeyValueSource

KeyValueSource is able to either wrap Hazelcast data structures (like IMap, MultiMap, IList, ISet) into key-value pair input sources, or build your own custom key-value input source. The latter option makes it possible to feed Hazelcast MapReduce with all kinds of data, such as just-in-time downloaded web page contents or data files. People familiar with Hadoop will recognize similarities with the Input class.
You can imagine a `KeyValueSource` as a bigger `java.util.Iterator` implementation. Whereas most methods are required to be implemented, the `getAllKeys` method is optional to implement. If implementation is able to gather all keys upfront, it should be implemented and `isAllKeysSupported` must return `true`. That way, Job configured KeyPredicates are able to evaluate keys upfront before sending them to the cluster. Otherwise, they are serialized and transferred as well, to be evaluated at execution time.

As shown in the example above, the abstract `KeyValueSource` class provides a number of static methods to easily wrap Hazelcast data structures into `KeyValueSource` implementations already provided by Hazelcast. The data structures' generics are inherited into the resulting `KeyValueSource` instance. For data structures like `IList` or `ISet`, the key type is always `String`. While mapping, the key is the data structure's name whereas the value type and value itself are inherited from the `IList` or `ISet` itself.

```java
// KeyValueSource from com.hazelcast.core.IMap
IMap<String, String> map = hazelcastInstance.getMap( "my-map" );
KeyValueSource<String, String> source = KeyValueSource.fromMap( map );

// KeyValueSource from com.hazelcast.core.MultiMap
MultiMap<String, String> multiMap = hazelcastInstance.getMultiMap( "my-multimap" );
KeyValueSource<String, String> source = KeyValueSource.fromMultiMap( multiMap );

// KeyValueSource from com.hazelcast.core.IList
IList<String> list = hazelcastInstance.getList( "my-list" );
KeyValueSource<String, String> source = KeyValueSource.fromList( list );

// KeyValueSource from com.hazelcast.core.ISet
ISet<String> set = hazelcastInstance.getSet( "my-set" );
KeyValueSource<String, String> source = KeyValueSource.fromSet( set );
```

**PartitionIdAware**

The `com.hazelcast.mapreduce.PartitionIdAware` interface can be implemented by the `KeyValueSource` implementation if the underlying data set is aware of the Hazelcast partitioning schema (as it is for all internal data structures). If this interface is implemented, the same `KeyValueSource` instance is reused multiple times for all partitions on the cluster node. As a consequence, the `close` and `open` methods are also executed multiple times but once per partitionId.

### 9.2.2.4 Mapper

Using the `Mapper` interface, you will implement the mapping logic. Mappers can transform, split, calculate, aggregate data from data sources. In Hazelcast, it is also possible to integrate data from more than the `KeyValueSource` data source by implementing `com.hazelcast.core.HazelcastInstanceAware` and requesting additional maps, multimaps, list, sets.

The mappers `map` function is called once per available entry in the data structure. If you work on distributed data structures that operate in a partition based fashion, then multiple mappers work in parallel on the different cluster nodes, on the nodes' assigned partitions. Mappers then prepare and maybe transform the input key-value pair and emit zero or more key-value pairs for reducing phase.

For our word count example, we retrieve an input document (a text document) and we transform it by splitting the text into the available words. After that, as discussed in the pseudo code, we emit every single word with a key-value pair with the word as the key and 1 as the value.

A common implementation of that `Mapper` might look like the following example:

```java
public class TokenizerMapper implements Mapper<String, String, String, Long> {
    private static final Long ONE = Long.valueOf(1L);
    @Override
```
public void map(String key, String document, Context<String, Long> context) {
    StringTokenizer tokenizer = new StringTokenizer(document.toLowerCase());
    while (tokenizer.hasMoreTokens()) {
        context.emit(tokenizer.nextToken(), ONE);
    }
}

The code splits the mapped texts into their tokens, iterates over the tokenizer as long as there are more tokens, and emits a pair per word. Note that we're not yet collecting multiple occurrences of the same word, we just fire every word on its own.

LifecycleMapper / LifecycleMapperAdapter

The LifecycleMapper interface or its adapter class LifecycleMapperAdapter can be used to make the Mapper implementation lifecycle aware. That means it will be notified when mapping of a partition or set of data begins and when the last entry was mapped.

Only special algorithms might need those additional lifecycle events to prepare, clean up, or emit additional values.

9.2.2.5 Combiner / CombinerFactory

As stated in the introduction, a Combiner is used to minimize traffic between the different cluster nodes when transmitting mapped values from mappers to the reducers. It does this by aggregating multiple values for the same emitted key. This is a fully optional operation, but using it is highly recommended.

Combiners can be seen as an intermediate reducer. The calculated value is always assigned back to the key for which the combiner initially was created. Since combiners are created per emitted key, the Combiner implementation itself is not defined in the jobs configuration; instead, a CombinerFactory is created that is able to create the expected Combiner instance.

Because Hazelcast MapReduce is executing mapping and reducing phase in parallel, the Combiner implementation must be able to deal with chunked data. Therefore, you must reset its internal state whenever you call finalizeChunk. Calling that method creates a chunk of intermediate data to be grouped (shuffled) and sent to the reducers.

Combiners can override beginCombine and finalizeCombine to perform preparation or cleanup work.

For our word count example, we are going to have a simple CombinerFactory and Combiner implementation similar to the following example.

public class WordCountCombinerFactory
    implements CombinerFactory<String, Long, Long> {

    @Override
    public Combiner<Long, Long> newCombiner(String key) {
        return new WordCountCombiner();
    }

    private class WordCountCombiner extends Combiner<Long, Long> {
        private long sum = 0;

        @Override
        public void combine(Long value) {
            sum++;
        }

        @Override
        public Long finalizeChunk() {
            return sum;
        }
    }
}
The Combiner must be able to return its current value as a chunk and reset the internal state by setting `sum` back to 0. Since combiners are always called from a single thread, no synchronization or volatility of the variables is necessary.

### 9.2.2.6 Reducer / ReducerFactory

Reducers do the last bit of algorithm work. This can be aggregating values, calculating averages, or any other work that is expected from the algorithm.

Since values arrive in chunks, the `reduce` method is called multiple times for every emitted value of the creation key. This also can happen multiple times per chunk if no Combiner implementation was configured for a job configuration.

In difference of the combiners, a reducers `finalizeReduce` method is only called once per reducer (which means once per key). Therefore, a reducer does not need to reset its internal state at any time.

Reducers can override `beginReduce` to perform preparation work.

For our word count example, the implementation will look similar to the following code example.

```java
public class WordCountReducerFactory implements ReducerFactory<String, Long, Long> {

    @Override
    public Reducer<Long, Long> newReducer( String key ) {
        return new WordCountReducer();
    }

    private class WordCountReducer extends Reducer<Long, Long> {
        private volatile long sum = 0;

        @Override
        public void reduce( Long value ) {
            sum += value.longValue();
        }

        @Override
        public Long finalizeReduce() {
            return sum;
        }
    }
}
```

Different from combiners, reducers tend to switch threads if running out of data to prevent blocking threads from the `JobTracker` configuration. They are rescheduled at a later point when new data to be processed arrives but unlikely to be executed on the same thread as before. As of Hazelcast version 3.3.3 the guarantee for memory visibility on the new thread is ensured by the framework. This means the previous requirement for making fields volatile is dropped.

### 9.2.2.7 Collator

A Collator is an optional operation that is executed on the job emitting node and is able to modify the finally reduced result before returned to the user’s codebase. Only special use cases are likely to use collators.
CHAPTER 9. DISTRIBUTED QUERY

For an imaginary use case, we might want to know how many words were all over in the documents we analyzed. For this case, a Collator implementation can be given to the submit method of the Job instance.

A collator would look like the following snippet:

```java
public class WordCountCollator implements Collator<Map.Entry<String, Long>, Long> {

    @Override
    public Long collate( Iterable<Map.Entry<String, Long>> values ) {
        long sum = 0;
        for ( Map.Entry<String, Long> entry : values ) {
            sum += entry.getValue().longValue();
        }
        return sum;
    }
}
```

The definition of the input type is a bit strange, but because Combiner and Reducer implementations are optional, the input type heavily depends on the state of the data. As stated above, collators are non-typical use cases and the generics of the framework always help in finding the correct signature.

9.2.2.8 KeyPredicate

A KeyPredicate can be used to pre-select whether or not a key should be selected for mapping in the mapping phase. If the KeyValueSource implementation is able to know all keys prior to execution, the keys are filtered before the operations are divided among the different cluster nodes.

A KeyPredicate can also be used to select only a special range of data (e.g. a time-frame) or similar use cases.

A basic KeyPredicate implementation that only maps keys containing the word “hazelcast” might look like the following code example:

```java
public class WordCountKeyPredicate implements KeyPredicate<String> {

    @Override
    public boolean evaluate( String s ) {
        return s != null && s.toLowerCase().contains( "hazelcast" );
    }
}
```

9.2.2.9 TrackableJob and Job Monitoring

You can retrieve a TrackableJob instance after submitting a job. It is requested from the JobTracker using the unique jobId (per JobTracker). It can be used to get runtime statistics of the job. The information available is limited to the number of processed (mapped) records and the processing state of the different partitions or nodes (if KeyValueSource is not PartitionIdAware).

To retrieve the jobId after submission of the job, use `com.hazelcast.mapreduce.JobCompletableFuture` instead of the `com.hazelcast.core.ICompletableFuture` as the variable type for the returned future.

The example code below gives a quick introduction on how to retrieve the instance and the runtime data. For more information, please have a look at the Javadoc corresponding your running Hazelcast version.

```java
IMap<String, String> map = hazelcastInstance.getMap( "articles" );
KeyValueSource<String, String> source = KeyValueSource.fromMap( map );
Job<String, String> job = jobTracker.newJob( source );
```
JobCompletableFuture<Map<String, Long>> future = job
    .mapper( new TokenizerMapper() )
    .combiner( new WordCountCombinerFactory() )
    .reducer( new WordCountReducerFactory() )
    .submit();

String jobId = future.getJobId();
TrackableJob trackableJob = jobTracker.getTrackableJob(jobId);

JobProcessInformation stats = trackableJob.getJobProcessInformation();
int processedRecords = stats.getProcessedRecords();
log( "ProcessedRecords: " + processedRecords );

JobPartitionState[] partitionStates = stats.getPartitionStates();
for ( JobPartitionState partitionState : partitionStates ) {
  log( "PartitionOwner: " + partitionState.getOwner()
      + ", Processing state: " + partitionState.getState().name() );
}

!! NOTE: Caching of the JobProcessInformation does not work on Java native clients since current values are retrieved while retrieving the instance to minimize traffic between executing node and client.

9.2.2.10 JobTracker Configuration

The JobTracker configuration is used to setup behavior of the Hazelcast MapReduce framework.

Every JobTracker is capable of running multiple MapReduce jobs at once; one configuration is meant as a shared resource for all jobs created by the same JobTracker. The configuration gives full control over the expected load behavior and thread counts to be used.

The following snippet shows a typical JobTracker configuration. We will discuss the configuration properties one by one:

<![CDATA[
<jobtracker name="default">
  <max-thread-size>0</max-thread-size>
  <!-- Queue size 0 means number of partitions * 2 -->
  <queue-size>0</queue-size>
  <retry-count>0</retry-count>
  <chunk-size>1000</chunk-size>
  <communicate-stats>true</communicate-stats>
  <topology-changed-strategy>CANCEL_RUNNING_OPERATION</topology-changed-strategy>
</jobtracker>
]]>

- **max-thread-size**: Configures the maximum thread pool size of the JobTracker.
- **queue-size**: Defines the maximum number of tasks that are able to wait to be processed. A value of 0 means an unbounded queue. Very low numbers can prevent successful execution since job might not be correctly scheduled or intermediate chunks might be lost.
- **retry-count**: Currently not used. Reserved for later use where the framework will automatically try to restart / retry operations from an available save point.
- **chunk-size**: Defines the number of emitted values before a chunk is sent to the reducers. If your emitted values are big or you want to better balance your work, you might want to change this to a lower or higher value. A value of 0 means immediate transmission, but remember that low values mean higher traffic costs. A very high value might cause an OutOfMemoryError to occur if the emitted values do not fit into heap memory before being sent to the reducers. To prevent this, you might want to use a combiner to pre-reduce values on mapping nodes.
- **communicate-stats**: Defines if statistics (for example, statistics about processed entries) are transmitted to the job emitter. This can show progress to a user inside of an UI system, but it produces additional traffic. If not needed, you might want to deactivate this.

- **topology-changed-strategy**: Defines how the MapReduce framework will react on topology changes while executing a job. Currently, only CANCEL_RUNNING_OPERATION is fully supported, which throws an exception to the job emitter (will throw a `com.hazelcast.mapreduce.TopologyChangedException`).

**RELATED INFORMATION**

*Please refer to the MapReduce Jobtracker Configuration section for a full description of Hazelcast MapReduce JobTracker configuration (includes an example programmatic configuration).*

### 9.2.3 Hazelcast MapReduce Architecture

This section explains some of the internals of the MapReduce framework. This is more advanced information. If you’re not interested in how it works internally, you might want to skip this section.

#### 9.2.3.1 Node Interoperation Example

To understand the following technical internals, we first have a short look at what happens in terms of an example workflow.

As a simple example, think of an `IMap<String, Integer>` and emitted keys having the same types. Imagine you have a three node cluster and you initiate the MapReduce job on the first node. After you requested the JobTracker from your running / connected Hazelcast, we submit the task and retrieve the `ICompletableFuture` which gives us a chance to wait for the result to be calculated or to add a callback (and being more reactive).

The example expects that the chunk size is 0 or 1, so an emitted value is directly sent to the reducers. Internally, the job is prepared, started, and executed on all nodes as shown below. The first node acts as the job owner (job emitter).

1. **Node1 starts MapReduce job**
2. **Node1 emits key=Foo, value=1**
3. **Node1 does PartitionService::getKeyOwner(Foo) => results in Node3**
4. **Node2 emits key=Foo, value=14**
5. **Node2 asks jobOwner (Node1) for keyOwner of Foo => results in Node3**
6. **Node1 sends chunk for key=Foo to Node3**
7. **Node3 receives chunk for key=Foo and looks if there is already a Reducer, if not creates one for key=Foo**
8. **Node3 processes chunk for key=Foo**
9. **Node2 sends chunk for key=Foo to Node3**
10. **Node3 receives chunk for key=Foo and looks if there is already a Reducer and uses the previous one**
11. **Node3 processes chunk for key=Foo**
12. **Node1 sends LastChunk information to Node3 because processing local values finished**
13. **Node2 emits key=Foo, value=27**
14. **Node2 has cached keyOwner of Foo => results in Node3**
15. **Node2 sends chunk for key=Foo to Node3**
16. **Node3 receives chunk for key=Foo and looks if there is already a Reducer and uses the previous one**
Node3 processes chunk for key=Foo

Node2 send LastChunk information to Node3 because processing local values finished

Node3 finishes reducing for key=Foo

Node1 registers its local partitions are processed
Node2 registers its local partitions are processed

Node1 sees all partitions processed and requests reducing from all nodes

Node1 merges all reduced results together in a final structure and returns it

The flow is quite complex but extremely powerful since everything is executed in parallel. Reducers do not wait until all values are emitted, but they immediately begin to reduce (when first chunk for an emitted key arrives).

9.2.3.2 Internal Architecture

Beginning with the package level, there is one basic package: com.hazelcast.mapreduce. This includes the external API and the impl package which itself contains the internal implementation.

- The impl package contains all the default KeyValueSource implementations and abstract base and support classes for the exposed API.
- The client package contains all classes that are needed on client and server (node) side when a client offers a MapReduce job.
- The notification package contains all “notification” or event classes that notify other members about progress on operations.
- The operation package contains all operations that are used by the workers or job owner to coordinate work and sync partition or reducer processing.
- The task package contains all classes that execute the actual MapReduce operation. It features the supervisor, mapping phase implementation and mapping and reducing tasks.

9.2.3.3 MapReduce Job Walk-Through

And now to the technical walk-through: a MapReduce Job is always retrieved from a named JobTracker, which is implemented in NodeJobTracker (extends AbstractJobTracker) and is configured using the configuration DSL. All of the internal implementation is completely CompletableFuture-driven and mostly non-blocking in design.

On submit, the Job creates a unique UUID which afterwards acts as a jobId and is combined with the JobTracker’s name to be uniquely identifiable inside the cluster. Then, the preparation is sent around the cluster and every member prepares its execution by creating a JobSupervisor, MapCombineTask, and ReducerTask. The job-emitting JobSupervisor gains special capabilities to synchronize and control JobSupervisors on other nodes for the same job.

If preparation is finished on all nodes, the job itself is started by executing a StartProcessingJobOperation on every node. This initiates a MappingPhase implementation (defaults to KeyValueSourceMappingPhase) and starts the actual mapping on the nodes.

The mapping process is currently a single threaded operation per node, but will be extended to run in parallel on multiple partitions (configurable per Job) in future versions. The Mapper is now called on every available value on the partition and eventually emits values. For every emitted value, either a configured CombinerFactory is called to create a Combiner or a cached one is used (or the default CollectingCombinerFactory is used to create Combiners). When the chunk limit is reached on a node, an IntermediateChunkNotification is prepared by collecting emitted keys to their corresponding nodes. This is either done by asking the job owner to assign members or by an already cached assignment. In later versions, a PartitionStrategy might also be configurable.

The IntermediateChunkNotification is then sent to the reducers (containing only values for this node) and is offered to the ReducerTask. On every offer, the ReducerTask checks if it is already running and if not, it submits itself to the configured ExecutorService (from the JobTracker configuration).
If reducer queue runs out of work, the ReducerTask is removed from the ExecutorService to not block threads but eventually will be resubmitted on next chunk of work.

On every phase, the partition state is changed to keep track of the currently running operations. A JobPartitionState can be in one of the following states with self-explanatory titles: [WAITING, MAPPING, REDUCING, PROCESSED, CANCELLED]. If you have a deeper interest of these states, look at the Javadoc.

- Node asks for new partition to process: WAITING => MAPPING
- Node emits first chunk to a reducer: MAPPING => REDUCING
- All nodes signal that they finished mapping phase and reducing is finished, too: REDUCING => PROCESSED

Eventually (or hopefully), all JobPartitionStates reach the state of PROCESSED. Then, the job emitter’s JobSupervisor asks all nodes for their reduced results and executes a potentially offered Collator. With this Collator, the overall result is calculated before it removes itself from the JobTracker, doing some final cleanup and returning the result to the requester (using the internal TrackableJobFuture).

If a job is cancelled while execution, all partitions are immediately set to the CANCELLED state and a CancelJobSupervisorOperation is executed on all nodes to kill the running processes.

While the operation is running in addition to the default operations, some more operations like ProcessStatsUpdateOperation (updates processed records statistics) or NotifyRemoteExceptionOperation (notifies the nodes that the sending node encountered an unrecoverable situation and the Job needs to be cancelled - e.g. NullPointerException inside of a Mapper) are executed against the job owner to keep track of the process.

### 9.3 Aggregators

Based on the Hazelcast MapReduce framework, Aggregators are ready-to-use data aggregations. These are typical operations like sum up values, finding minimum or maximum values, calculating averages, and other operations that you would expect in the relational database world.

Aggregation operations are implemented, as mentioned above, on top of the MapReduce framework and all operations can be achieved using pure MapReduce calls. However, using the Aggregation feature is more convenient for a big set of standard operations.

#### 9.3.1 Aggregations Basics

This section will quickly guide you through the basics of the Aggregations framework and some of its available classes. We also will implement a first base example.

Aggregations are available on both types of map interfaces, `com.hazelcast.core.IMap` and `com.hazelcast.core.MultiMap`, using the aggregate methods. Two overloaded methods are available that customize resource management of the underlying MapReduce framework by supplying a custom configured `com.hazelcast.mapreduce.JobTracker` instance. To find out how to configure the MapReduce framework, please see the JobTracker Configuration section. We will later see another way to configure the automatically used MapReduce framework if no special `JobTracker` is supplied.

To make Aggregations more convenient to use and future proof, the API is heavily optimized for Java 8 and future versions. The API is still fully compatible with any Java version Hazelcast supports (Java 6 and Java 7). The biggest difference is how you work with the Java generics: on Java 6 and 7, the process to resolve generics is not as strong as on Java 8 and upcoming Java versions. In addition, the whole Aggregations API has full Java 8 Project Lambda (or Closure, JSR 335) support.

For illustration of the differences in Java 6 and 7 in comparison to Java 8, we will have a quick look at code examples for both. After that, we will focus on using Java 8 syntax to keep examples short and easy to understand, and we will see some hints as to what the code looks like in Java 6 or 7.

The first example will produce the sum of some `int` values stored in a Hazelcast IMap. This example does not use much of the functionality of the Aggregations framework, but it will show the main difference.
9.3. AGGREGATORS

```
IMap<String, Integer> personAgeMapping = hazelcastInstance.getMap( "person-age" );
for ( int i = 0; i < 1000; i++ ) {
    String lastName = RandomUtil.randomLastName();
    int age = RandomUtil.randomAgeBetween( 20, 80 );
    personAgeMapping.put( lastName, Integer.valueOf( age ) );
}
```

With our demo data prepared, we can see how to produce the sums in different Java versions.

9.3.1.1 Aggregations and Java 6 or Java 7

Since Java 6 and 7 are not as strong on resolving generics as Java 8, you need to be a bit more verbose with the code you write. You might also consider using raw types, but breaking the type safety to ease this process.

For a short introduction on what the following code example means, look at the source code comments. We will later dig deeper into the different options.

```java
// No filter applied, select all entries
Supplier<String, Integer, Integer> supplier = Supplier.all();
// Choose the sum aggregation
Aggregation<String, Integer, Integer> aggregation = Aggregations.integerSum();
// Execute the aggregation
int sum = personAgeMapping.aggregate( supplier, aggregation );
```

9.3.1.2 Aggregations and Java 8

With Java 8, the Aggregations API looks simpler because Java 8 can resolve the generic parameters for us. That means the above lines of Java 6/7 example code will end up in just one easy line on Java 8.

```java
int sum = personAgeMapping.aggregate( Supplier.all(), Aggregations.integerSum() );
```

9.3.1.3 Quick look at the MapReduce Framework

As mentioned before, the Aggregations implementation is based on the Hazelcast MapReduce framework and therefore you might find overlaps in their APIs. One overload of the `aggregate` method can be supplied with a `JobTracker` which is part of the MapReduce framework.

If you implement your own aggregations, you will use a mixture of the Aggregations and the MapReduce API. If you will implement your own aggregation, e.g. to make the life of colleagues easier, please read the Implementing Aggregations section.

For the full MapReduce documentation please see the MapReduce section.

9.3.2 Introduction to Aggregations API

We now look into the possible options of what can be achieved using the Aggregations API. To work on some deeper examples, let’s quickly have a look at the available classes and interfaces and discuss their usage.

9.3.2.1 Supplier

The `com.hazelcast.mapreduce.aggregation.Supplier` provides filtering and data extraction to the aggregation operation. This class already provides a few different static methods to achieve the most common cases. `Supplier.all()` accepts all incoming values and does not apply any data extraction or transformation upon them before supplying them to the aggregation function itself.

For filtering data sets, you have two different options by default. You can either supply a `com.hazelcast.query.Predicate` if you want to filter on values and / or keys, or you can supply a `com.hazelcast.mapreduce.KeyPredicate` if you can decide directly on the data key without the need to deserialize the value.
9.3.2.1.1 Basic Filtering  As mentioned above, all APIs are fully Java 8 and Lambda compatible. Let’s have a look on how we can do basic filtering using those two options.

First, we have a look at a `KeyPredicate` and only accept people whose last name is “Jones”.

```java
Supplier<...> supplier = Supplier.fromKeyPredicate(
    lastName -> "Jones".equalsIgnoreCase(lastName)
);

class JonesKeyPredicate implements KeyPredicate<String> {
    public boolean evaluate(String key) {
        return "Jones".equalsIgnoreCase(key);
    }
}
```

Using the standard Hazelcast `Predicate` interface, you can also filter based on the value of a data entry. In the following example, you can only select values which are divisible by 4 without a remainder.

```java
Supplier<...> supplier = Supplier.fromPredicate(
    entry -> entry.getValue() % 4 == 0
);

class DivisiblePredicate implements Predicate<String, Integer> {
    public boolean apply(Map.Entry<String, Integer> entry) {
        return entry.getValue() % 4 == 0;
    }
}
```

9.3.2.1.2 Extracting and Transforming Data  As well as filtering, `Supplier` can also extract or transform data before providing it to the aggregation operation itself. The following example shows how to transform an input value to a string.

```java
Supplier<String, Integer, String> supplier = Supplier.all(
    value -> Integer.toString(value)
);
```

You can see a Java 6 / 7 example in the Aggregations Examples section.

Apart from the fact we transformed the input value of type `int` (or `Integer`) to a string, we can see that the generic information of the resulting `Supplier` has changed as well. This indicates that we now have an aggregation working on string values.

9.3.2.1.3 Chaining Multiple Filtering Rules  Another feature of `Supplier` is its ability to chain multiple filtering rules. Let’s combine all of the above examples into one rule set:

```java
Supplier<String, Integer, String> supplier =
    Supplier.fromKeyPredicate(
        lastName -> "Jones".equalsIgnoreCase(lastName),
        Supplier.fromPredicate(
            entry -> entry.getValue() % 4 == 0,
            Supplier.all( value -> Integer.toString(value) )
        )
    );
```
9.3. AGGREGATORS

9.3.2.1.4 Implementing Based on Special Requirements

Last but not least, you might prefer to (or need to) implement your Supplier based on special requirements. This is a very basic task. The Supplier abstract class has just one method.

**NOTE:** Due to a limitation of the Java Lambda API, you cannot implement abstract classes using Lambdas. Instead it is recommended that you create a standard named class.

```java
class MyCustomSupplier extends Supplier<String, Integer, String> {
    public String apply( Map.Entry<String, Integer> entry ) {
        Integer value = entry.getValue();
        if (value == null) {
            return null;
        }
        return value % 4 == 0 ? String.valueOf( value ) : null;
    }
}
```

Suppliers are expected to return null from the apply method whenever the input value should not be mapped to the aggregation process. This can be used, as in the example above, to implement filter rules directly. Implementing filters using the KeyPredicate and Predicate interfaces might be more convenient.

To use your own Supplier, just pass it to the aggregate method or use it in combination with other Suppliers.

```java
int sum = personAgeMapping.aggregate( new MyCustomSupplier(), Aggregations.count() );
Supplier<String, Integer, String> supplier =
    Supplier.fromKeyPredicate(
        lastName -> "Jones".equalsIgnoreCase( lastName ),
        new MyCustomSupplier()
    );
int sum = personAgeMapping.aggregate( supplier, Aggregations.count() );
```

9.3.2.2 Aggregation and Aggregations

The com.hazelcast.mapreduce.aggregation.Aggregation interface defines the aggregation operation itself. It contains a set of MapReduce API implementations like Mapper, Combiner, Reducer, and Collator. These implementations are normally unique to the chosen Aggregation. This interface can also be implemented with your aggregation operations based on MapReduce calls. For more information, refer to Implementing Aggregations section.

The com.hazelcast.mapreduce.aggregation.Aggregations class provides a common predefined set of aggregations. This class contains type safe aggregations of the following types:

- Average (Integer, Long, Double, BigInteger, BigDecimal)
- Sum (Integer, Long, Double, BigInteger, BigDecimal)
- Min (Integer, Long, Double, BigInteger, BigDecimal, Comparable)
- Max (Integer, Long, Double, BigInteger, BigDecimal, Comparable)
- DistinctValues
- Count

Those aggregations are similar to their counterparts on relational databases and can be equated to SQL statements as set out below.
9.3.2.2.1 Average  Calculates an average value based on all selected values.

map.aggregate( Supplier.all( person -> person.getAge() ),
  Aggregations.integerAvg() );

SELECT AVG(person.age) FROM person;

9.3.2.2.2 Sum  Calculates a sum based on all selected values.

map.aggregate( Supplier.all( person -> person.getAge() ),
  Aggregations.integerSum() );

SELECT SUM(person.age) FROM person;

9.3.2.2.3 Minimum (Min)  Finds the minimal value over all selected values.

map.aggregate( Supplier.all( person -> person.getAge() ),
  Aggregations.integerMin() );

SELECT MIN(person.age) FROM person;

9.3.2.2.4 Maximum (Max)  Finds the maximal value over all selected values.

map.aggregate( Supplier.all( person -> person.getAge() ),
  Aggregations.integerMax() );

SELECT MAX(person.age) FROM person;

9.3.2.2.5 Distinct Values  Returns a collection of distinct values over the selected values

map.aggregate( Supplier.all( person -> person.getAge() ),
  Aggregations.distinctValues() );

SELECT DISTINCT person.age FROM person;

9.3.2.2.6 Count  Returns the element count over all selected values

map.aggregate( Supplier.all(), Aggregations.count() );

SELECT COUNT(*) FROM person;

9.3.2.3 PropertyExtractor

We used the com.hazelcast.mapreduce.aggregation.PropertyExtractor interface before when we had a look at the example on how to use a Supplier to transform a value to another type. It can also be used to extract attributes from values.
9.3. AGGREGATORS

```java
class Person {
    private String firstName;
    private String lastName;
    private int age;

    // getters and setters
}

PropertyExtractor<Person, Integer> propertyExtractor = (person) -> person.getAge();

class AgeExtractor implements PropertyExtractor<Person, Integer> {
    public Integer extract(Person value) {
        return value.getAge();
    }
}
```

In this example, we extract the value from the person’s age attribute. The value type changes from Person to Integer which is reflected in the generics information to stay type safe.

PropertyExtractors are meant to be used for any kind of transformation of data. You might even want to have multiple transformation steps chained one after another.

9.3.2.4 Aggregation Configuration

As stated before, the easiest way to configure the resources used by the underlying MapReduce framework is to supply a JobTracker to the aggregation call itself by passing it to either IMap::aggregate or MultiMap::aggregate.

There is another way to implicitly configure the underlying used JobTracker. If no specific JobTracker was passed for the aggregation call, internally one will be created using the following naming specifications:

For IMap aggregation calls the naming specification is created as:

- `hz::aggregation-map-` and the concatenated name of the map.

For MultiMap it is very similar:

- `hz::aggregation-multimap-` and the concatenated name of the MultiMap.

Knowing that (the specification of the name), we can configure the JobTracker as expected (as described in the Jobtracker section) using the naming spec we just learned. For more information on configuration of the JobTracker, please see the JobTracker Configuration section.

To finish this section, let’s have a quick example for the above naming specs:

```java
IMap<String, Integer> map = hazelcastInstance.getMap("mymap");
// The internal JobTracker name resolves to 'hz::aggregation-map-mymap'
map.aggregate(...);

MultiMap<String, Integer> multimap = hazelcastInstance.getMultiMap("mymultimap");
// The internal JobTracker name resolves to 'hz::aggregation-multimap-mymultimap'
multimap.aggregate(...);
```
9.3.3 Aggregations Examples

For the final example, imagine you are working for an international company and you have an employee database stored in Hazelcast IMap with all employees worldwide and a MultiMap for assigning employees to their certain locations or offices. In addition, there is another IMap which holds the salary per employee.

Let’s have a look at our data model:

class Employee implements Serializable {
    private String firstName;
    private String lastName;
    private String companyName;
    private String address;
    private String city;
    private String county;
    private String state;
    private int zip;
    private String phone1;
    private String phone2;
    private String email;
    private String web;

    // getters and setters
}

class SalaryMonth implements Serializable {
    private Month month;
    private int salary;

    // getters and setters
}

class SalaryYear implements Serializable {
    private String email;
    private int year;
    private List<SalaryMonth> months;

    // getters and setters

    public int getAnnualSalary() {
        int sum = 0;
        for (SalaryMonth salaryMonth : getMonths()) {
            sum += salaryMonth.getSalary();
        }
        return sum;
    }
}

The two IMaps and the MultiMap are keyed by the string of email. They are defined as follows:

IMap<String, Employee> employees = hz.getMap("employees");
IMap<String, SalaryYear> salaries = hz.getMap("salaries");
MultiMap<String, String> officeAssignment = hz.getMultiMap("office-employee");

So far, we know all the important information to work out some example aggregations. We will look into some deeper implementation details and how we can work around some current limitations that will be eliminated in future versions of the API.

Let’s start with a very basic example. We want to know the average salary of all of our employees. To do this, we need a PropertyExtractor and the average aggregation for type Integer.
9.3. AGGREGATORS

```java
IMap<String, SalaryYear> salaries = hazelcastInstance.getMap("salaries");
PropertyExtractor<SalaryYear, Integer> extractor =
    (salaryYear) -> salaryYear.getAnnualSalary();
int avgSalary = salaries.aggregate(Supplier.all(extractor),
    Aggregations.integerAvg());
```

That’s it. Internally, we created a MapReduce task based on the predefined aggregation and fired it up immediately. Currently, all aggregation calls are blocking operations, so it is not yet possible to execute the aggregation in a reactive way (using `com.hazelcast.core.ICompletableFuture`) but this will be part of an upcoming version.

9.3.3.1 Map Join Example

The following example is a little more complex. We only want to have our US based employees selected into the average salary calculation, so we need to execute some kind of a join operation between the employees and salaries maps.

```java
class USEmployeeFilter implements KeyPredicate<String>, HazelcastInstanceAware {
    private transient HazelcastInstance hazelcastInstance;

    public void setHazelcastInstance(HazelcastInstance hazelcastInstance) {
        this.hazelcastInstance = hazelcastInstance;
    }

    public boolean evaluate(String email) {
        IMap<String, Employee> employees = hazelcastInstance.getMap("employees");
        Employee employee = employees.get(email);
        return "US".equals(employee.getCountry());
    }
}
```

Using the `HazelcastInstanceAware` interface, we get the current instance of Hazelcast injected into our filter and we can perform data joins on other data structures of the cluster. We now only select employees that work as part of our US offices into the aggregation.

```java
IMap<String, SalaryYear> salaries = hazelcastInstance.getMap("salaries");
PropertyExtractor<SalaryYear, Integer> extractor =
    (salaryYear) -> salaryYear.getAnnualSalary();
int avgSalary = salaries.aggregate(Supplier.fromKeyPredicate(
    new USEmployeeFilter(), extractor),
    Aggregations.integerAvg());
```

9.3.3.2 Grouping Example

For our next example, we will do some grouping based on the different worldwide offices. Currently, a group aggregator is not yet available, so we need a small workaround to achieve this goal. (In later versions of the Aggregations API this will not be required because it will be available out of the box in a much more convenient way.)

Again, let’s start with our filter. This time, we want to filter based on an office name and we need to do some data joins to achieve this kind of filtering.

A short tip: to minimize the data transmission on the aggregation we can use Data Affinity rules to influence the partitioning of data. Be aware that this is an expert feature of Hazelcast.

```java
class OfficeEmployeeFilter implements KeyPredicate<String>, HazelcastInstanceAware {
    private transient HazelcastInstance hazelcastInstance;
    private String office;
```
// Deserialization Constructor
public OfficeEmployeeFilter() {
}

public OfficeEmployeeFilter( String office ) {
    this.office = office;
}

public void setHazelcastInstance( HazelcastInstance hazelcastInstance ) {
    this.hazelcastInstance = hazelcastInstance;
}

public boolean evaluate( String email ) {
    MultiMap<String, String> officeAssignment = hazelcastInstance
        .getMultiMap( "office-employee" );

    return officeAssignment.containsEntry( office, email );
}

Now we can execute our aggregations. As mentioned before, we currently need to do the grouping on our own by executing multiple aggregations in a row.

Map<String, Integer> avgSalariesPerOffice = new HashMap<String, Integer>();

IMap<String, SalaryYear> salaries = hazelcastInstance.getMap( "salaries" );
MultiMap<String, String> officeAssignment = hazelcastInstance.getMultiMap( "office-employee" );

PropertyExtractor<SalaryYear, Integer> extractor =
    (salaryYear) -> salaryYear.getAnnualSalary();

for ( String office : officeAssignment.keySet() ) {
    OfficeEmployeeFilter filter = new OfficeEmployeeFilter( office );
    int avgSalary = salaries.aggregate( Supplier.fromKeyPredicate( filter, extractor ),
        Aggregations.integerAvg() );

    avgSalariesPerOffice.put( office, avgSalary );
}

9.3.3.3 Simple Count Example

After the previous example, we want to end this section by executing one final and easy aggregation. We want to know how many employees we currently have on a worldwide basis. Before reading the next lines of example code, you can try to do it on your own to see if you understood how to execute aggregations.

IMap<String, Employee> employees = hazelcastInstance.getMap( "employees" );
int count = employees.size();

Ok, after that quick joke, we look at the real two code lines:

IMap<String, Employee> employees = hazelcastInstance.getMap( "employees" );
int count = employees.aggregate( Supplier.all(), Aggregations.count() );

We now have an overview of how to use aggregations in real life situations. If you want to do your colleagues a favor, you might want to write your own additional set of aggregations. If so, then read the next section, Implementing Aggregations.
9.4. CONTINUOUS QUERY

9.3.4 Implementing Aggregations

This section explains how to implement your own aggregations in your own application. It is meant to be an advanced section, so if you do not intend to implement your own aggregation, you might want to stop reading here and come back later when you need to know how to implement your own aggregation.

The main interface for making your own aggregation is `com.hazelcast.mapreduce.aggregation.Aggregation`. It consists of four methods.

```java
interface Aggregation<Key, Supplied, Result> {
    Mapper getMapper(Supplier<Key, ?, Supplied> supplier);
    CombinerFactory getCombinerFactory();
    ReducerFactory getReducerFactory();
    Collator<Map.Entry, Result> getCollator();
}
```

An Aggregation implementation is just defining a MapReduce task with a small difference: the Mapper is always expected to work on a Supplier that filters and / or transforms the mapped input value to some output value. `getMapper` and `getReducerFactory` are expected to return non-null values. `getCombinerFactory` and `getCollator` are optional operations and do not need to be implemented. If you can decide to implement them depending on the use case you want to achieve.

For more information on how you implement mappers, combiners, reducers, and collators, refer to the MapReduce section.

For best speed and traffic usage, as mentioned in the MapReduce section, you should add a Combiner to your aggregation whenever it is possible to do some kind of pre-reduction step.

Your implementation also should use DataSerializable or IdentifiedDataSerializable for best compatibility and speed / stream-size reasons.

9.4 Continuous Query

Continuous query enables you to listen to the modifications performed on specific map entries. It is an entry listener with predicates. Please see the Map Listener section for information on how to add entry listeners to a map.

As an example, let’s listen to the changes made on an employee with the surname “Smith”. First, let’s create the Employee class.

```java
import java.io.Serializable;

public class Employee implements Serializable {
    private final String surname;

    public Employee(String surname) {
        this.surname = surname;
    }

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

Then, let’s create the continuous query by adding the entry listener with the surname predicate.
import com.hazelcast.core.*;
import com.hazelcast.query.SqlPredicate;

public class ContinuousQuery {

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

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

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

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

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

        @Override
        public void mapEvicted(MapEvent event) {
            System.out.println("Map Evicted:" + event);
        }
    }
}

And now, let’s play with the employee “smith” and see how that employee will be listened to.

import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.IMap;

public class Modify {

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

        map.put("1", new Employee("smith"));
        map.put("2", new Employee("jordan"));
        System.out.println("done");
    }
}
When you first run the class `ContinuousQuery` and then run `Modify`, you will see output similar to the listing below.

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

### 9.5 Continuous Query

**Hazelcast Enterprise**

**NOTE:** This feature is supported for Hazelcast Enterprise 3.5 or higher.

This feature is used to cache the result of a continuous query. After construction of a continuous query cache, all changes on underlying `IMap` is immediately reflected to this cache as a stream of events. Therefore, this cache will be an always up to date view of the `IMap`.

This feature is beneficial when you need to query the distributed `IMap` data in a very frequent and fast way. By using continuous query cache, the result of the query will be always ready and local to the application.

You can access this continuous query cache from the server and client side respectively as shown below.

```
QueryCacheConfig queryCacheConfig = new QueryCacheConfig("cache-name");
queryCacheConfig.getPredicateConfig().setImplementation(new OddKeysPredicate());

MapConfig mapConfig = new MapConfig("map-name");
mapConfig.addQueryCacheConfig(queryCacheConfig);

Config config = new Config();
config.addMapConfig(mapConfig);

HazelcastInstance node = Hazelcast.newHazelcastInstance(config);
IEnterpriseMap<Integer, String> map = (IEnterpriseMap) node.getMap("map-name");
QueryCache<Integer, String> cache = map.getQueryCache("cache-name");
```

```
QueryCacheConfig queryCacheConfig = new QueryCacheConfig("cache-name");
queryCacheConfig.getPredicateConfig().setImplementation(new OddKeysPredicate());

ClientConfig clientConfig = new ClientConfig();
clientConfig.addQueryCacheConfig("map-name", queryCacheConfig);

HazelcastInstance client = HazelcastClient.newHazelcastClient(clientConfig);
IEnterpriseMap<Integer, Integer> clientMap = (IEnterpriseMap) client.getMap("map-name");
QueryCache<Integer, Integer> cache = clientMap.getQueryCache("cache-name");
```

### 9.5.1 Features of Continuous Query Cache

1. Enable/disable initial query run on the existing `IMap` data during construction of continuous query cache according to the supplied predicate via `QueryCacheConfig#setPopulate`.
2. Indexable and queryable.

3. Evictable. Note that continuous query cache has a default maximum capacity of 10000. If you need a not evictable one, you should configure the eviction via `QueryCacheConfig#setEvictionConfig`.

4. Listenable via `QueryCache#addEntryListener`.

5. Events on `IMap` are guaranteed to be reflected to this cache in the happening order. Note that this happening order is a partition order so you can only expect ordered events from the same partition. Any loss of event can be listened via `EventLostListener` and it can be recoverable with `QueryCache#tryRecover` method. If your buffer size on the node side is big enough, you can recover from a possible event loss scenario. At the moment, setting the size of `QueryCacheConfig#setBufferSize` is the only option for recovery because the events which feed continuous query cache have no backups. Below snippet can be used for recovery case.

```java
QueryCache queryCache = map.getQueryCache("cache-name", new SqlPredicate("this > 20"), true);
queryCache.addEntryListener(new EventLostListener() {
    @Override
    public void eventLost(EventLostEvent event) {
        queryCache.tryRecover();
    }
}, false);
```


7. Declarative and programmatic configuration

8. It can be populated with only keys of entries and subsequent values can be retrieved directly via `QueryCache#get` from the underlying `IMap`. This will help to decrease initial population time if the values are very big in size.
Chapter 10

User Defined Services

In the case of special/custom needs, Hazelcast’s SPI (Service Provider Interface) module allows users to develop their own distributed data structures and services.

10.1 Sample Case

Throughout this section, we create a distributed counter that will be the guide to reveal the Hazelcast SPI usage.

Here is our counter.

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

This counter will have the following features: - It will be stored in Hazelcast. - Different cluster members can call it. - It will be scalable, meaning that the capacity for the number of counters scales with the number of cluster members. - It will be highly available, meaning that if a member hosting this counter goes down, a backup will be available on a different member.

All these features will be realized with the steps below. In each step, a new functionality to this counter will be added.

1. Create the class.
2. Enable the class.
3. Add properties.
4. Place a remote call.
5. Create the containers.
6. Enable partition migration.
7. Create the backups.

10.1.1 Creating Class

To have the counter as a functioning distributed object, we need a class. This class (named CounterService in the following sample) will be the gateway between Hazelcast internals and the counter, allowing us to add features to the counter. In the following sample, the class `CounterService` is created. Its lifecycle will be managed by Hazelcast.

`CounterService` should implement the interface `com.hazelcast.spi.ManagedService` as shown below.
import com.hazelcast.spi.ManagedService;
import com.hazelcast.spi.NodeEngine;

import java.util.Properties;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;

public class CounterService implements ManagedService {
    private NodeEngine nodeEngine;

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

    @Override
    public void shutdown( boolean terminate ) {
        System.out.println( "CounterService.shutdown" );
    }

    @Override
    public void reset() {
    }
}

As can be seen from the code, CounterService implements the following methods.

- **init**: This is called when CounterService is initialized. NodeEngine enables access to Hazelcast internals such as HazelcastInstance and PartitionService. Also, the object Properties will provide us with the ability to create our own properties.
- **shutdown**: This is called when CounterService is shutdown. It cleans up the resources.
- **reset**: This is called when cluster members are faced with the Split-Brain issue. 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 CounterService we are going to implement as a no-op.

### 10.1.2 Enabling Class

Now, we need to enable the class CounterService. The declarative way of doing this is shown below.

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

CounterService is declared within the services configuration element.

- Setting the enabled attribute as true enables the service.
10.1. SAMPLE CASE

- The name attribute defines the name of the service. It should be a unique name (CounterService in our case) since it will be looked up when a remote call is made. Note that the value of this attribute will be sent at each request, and that a longer name value means more data (de)serialization. A good practice is to give an understandable name with the shortest possible length.
- class-name is the class name of the service (CounterService in our case). The class should have a no-arg constructor. Otherwise, the object cannot be initialized.

Note that multicast is enabled as the join mechanism. In the later sections for the CounterService example, we will see why.

RELATED INFORMATION
Please refer to the Services Configuration section for a full description of Hazelcast SPI configuration.

10.1.3 Adding Properties

The init method for CounterService takes the Properties object as an argument. This means we can add properties to the service that are passed to the method init. You can add properties declaratively as shown below.

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

If you want to parse a more complex XML, you can use the interface com.hazelcast.spi.ServiceConfigurationParser. It gives you access to the XML DOM tree.

10.1.4 Starting Service

Now, let’s start a HazelcastInstance as shown below, which will start the CounterService.

```java
import com.hazelcast.core.Hazelcast;

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

Once it is started, the CounterService#init method prints the following output.

CounterService.init

Once the HazelcastInstance is shutdown (for example with Ctrl+C), the CounterService#shutdown method prints the following output.

CounterService.shutdown

10.1.5 Placing a Remote Call - Proxy

In the previous sections for the CounterService example, we started CounterService as part of a HazelcastInstance startup.

Now, let’s connect the Counter interface to CounterService and perform a remote call to the cluster member hosting the counter data. Then, we will return a dummy result.
Remote calls are performed via a proxy in Hazelcast. Proxies expose the methods at the client side. Once a method is called, proxy creates an operation object, sends this object to the cluster member responsible from executing that operation, and then sends the result.

10.1.5.1 Making Counter a Distributed Object

First, we need to make the `Counter` interface a distributed object by extending the `DistributedObject` interface, as shown below.

```java
import com.hazelcast.core.DistributedObject;

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

10.1.5.2 Implementing ManagedService and RemoteService

Now, we need to make the `CounterService` class implement not only the `ManagedService` interface, but also the interface `com.hazelcast.spi.RemoteService`. This way, a client will be able to get a handle of a counter proxy.

```java
import com.hazelcast.core.DistributedObject;
import com.hazelcast.core.ManagedService;
import com.hazelcast.core.NodeEngine;
import com.hazelcast.spi.RemoteService;
import java.util.Properties;

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) {
        // for the time being a no-op, but in the later examples this will be implemented
    }

    @Override
    public void init(NodeEngine nodeEngine, Properties properties) {
        this.nodeEngine = nodeEngine;
    }

    @Override
    public void shutdown(boolean terminate) {
    }

    @Override
    public void reset() {
    }
}
```
The CounterProxy returned by the method createDistributedObject is a local representation to (potentially) remote managed data and logic.

**NOTE:** Note that caching and removing the proxy instance are done outside of this service.

### 10.1.5.3 Implementing CounterProxy

Now, it is time to implement the CounterProxy as shown below.

```java
import com.hazelcast.spi.AbstractDistributedObject;
import com.hazelcast.spi.InvocationBuilder;
import com.hazelcast.spi.NodeEngine;
import com.hazelcast.util.ExceptionUtil;
import java.util.concurrent.Future;

public class CounterProxy extends AbstractDistributedObject<CounterService> implements Counter {
    private final String name;

    public CounterProxy(String name, NodeEngine nodeEngine, CounterService counterService) {
        super(nodeEngine, counterService);
        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);
        }
    }
}
```

CounterProxy is a local representation of remote data/functionality. It does not include the counter state. Therefore, the method inc should be invoked on the cluster member hosting the real counter. You can invoke it using Hazelcast SPI; then it will send the operations to the correct member and return the results.

Let’s dig deeper into the method inc.

- First, we create IncOperation with a given name and amount.
Then, we get the partition ID based on the name; by this way, all operations for a given name will result in the same partition ID.

Then, we create an InvocationBuilder where the connection between operation and partition is made.

Finally, we invoke the InvocationBuilder and wait for its result. This waiting is performed with a future.get(). In our case, timeout is not important. However, it is a good practice to use a timeout for a real system since operations should complete in a certain amount of time.

10.1.5.4 Dealing with Exceptions

Hazelcast’s ExceptionUtil is a good solution when it comes to dealing with execution exceptions. When the execution of the operation fails with an exception, an ExecutionException is thrown and handled with the method ExceptionUtil.rethrow(Throwable).

If it is an InterruptedException, we have two options: Either propagating the exception or just using the ExceptionUtil.rethrow for all exceptions. Please see below sample.

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

10.1.5.5 Implementing the PartitionAwareOperation Interface

Now, let’s write the IncOperation. It implements PartitionAwareOperation interface, meaning that it will be executed on the partition that hosts the counter.

```java
import com.hazelcast.nio.ObjectDataInput;
import com.hazelcast.nio.ObjectDataOutput;
import com.hazelcast.spi.AbstractOperation;
import com.hazelcast.spi.PartitionAwareOperation;
import java.io.IOException;

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

```
The method `run` does the actual execution. Since `IncOperation` will return a response, the method `returnsResponse` returns `true`. If your method is asynchronous and does not need to return a response, it is better to return `false` since it will be faster. The actual response is stored in the field `returnValue`; you can retrieve it with the method `getResponse`.

There are two more methods in the above code: `writeInternal` and `readInternal`. Since `IncOperation` needs to be serialized, these two methods should be overwritten, and hence, `objectId` and `amount` will be serialized and available when those operations are executed.

For the deserialization, note that the operation must have a `no-arg` constructor.

### 10.1.5.6 Running the Code

Now, let’s run our code.

```java
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");
    }
}
```
CHAPTER 10. USER DEFINED SERVICES

System.exit(0);
}
}

Once run, you will see the output as below.

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

Note that counters are stored in different cluster members. Also note that increment is not active for now since the value remains as 0.

Until now, we have performed the basics to get this up and running. In the next section, we will make a real counter, cache the proxy instances and deal with proxy instance destruction.

10.1.6 Creating Containers

Let’s create a Container for every partition in the system. This container will contain all counters and proxies.

```java
import java.util.HashMap;
import java.util.Map;

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

    int inc(String id, int amount) {
        Integer counter = values.get(id);
        if (counter == null) {
            counter = 0;
        }
        counter += amount;
        values.put(id, counter);
        return counter;
    }

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

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

    ...
}
```
Hazelcast guarantees that a single thread will be active in a single partition. Therefore, when accessing a container, concurrency control will not be an issue.

The code in our example uses a Container instance per partition approach. With this approach, there will not be any mutable shared state between partitions. This approach also makes operations on partitions simpler since you do not need to filter out data that does not belong to a certain partition.

10.1.6.1 Integrating the Container in the CounterService

Let’s integrate the Container in the CounterService, as shown below.

```java
import com.hazelcast.spi.ManagedService;
import com.hazelcast.spi.NodeEngine;
import com.hazelcast.spi.RemoteService;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

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

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

    @Override
    public void shutdown(boolean terminate) {
    }

    @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);
    }

    @Override
    public void reset() {
    }

    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);
}

We create a container for every partition with the method `init`. Then we create the proxy with the method `createDistributedObject`. And finally, we need to remove the value of the object with the method `destroyDistributedObject`, otherwise we may get an OutOfMemory exception.

**10.1.6.2 Connecting the IncOperation.run Method to the Container**

As the last step in creating a Container, we connect the method `IncOperation.run` to the Container, as shown below.

```java
import com.hazelcast.nio.ObjectDataInput;
import com.hazelcast.nio.ObjectDataOutput;
import com.hazelcast.spi.AbstractOperation;
import com.hazelcast.spi.PartitionAwareOperation;
import java.io.IOException;
import java.util.Map;

class IncOperation extends AbstractOperation implements PartitionAwareOperation {
    private String objectId;
    private int amount, returnValue;

    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: "+ getNodeId().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;
    }

    @Override
    public boolean returnsResponse() {
        return true;
    }
}
```
@Override
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);
    }
}

partitionId has a range between 0 and partitionCount and can be used as an index for the container array. Therefore, you can use partitionId to retrieve the container, and once the container has been retrieved, you can access the value.

10.1.6.3 Running the Sample Code

Let’s run the following sample code.

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

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);
    }
}
```

The output will be as follows. It indicates that we have now a basic distributed counter up and running.
10.1.7 Partition Migration

In the previous section, we created a real distributed counter. Now, we need to make sure that the content of the partition containers is migrated to different cluster members when a member joins or leaves the cluster. To make this happen, first we need to add three new methods (applyMigrationData, toMigrationData and clear) to the Container, as shown below.

```java
import java.util.HashMap;
import java.util.Map;

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

    int inc(String id, int amount) {
        Integer counter = values.get(id);
        if (counter == null) {
            counter = 0;
        }
        counter += amount;
        values.put(id, counter);
        return counter;
    }

    void clear() {
        values.clear();
    }

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

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

    public void init(String objectName) {
        values.put(objectName, 0);
    }
}
```
10.1. SAMPLE CASE

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

- `toMigrationData`: This method is called when Hazelcast wants to start the partition migration from the member owning the partition. The result of the `toMigrationData` method is the partition data in a form that can be serialized to another member.
- `applyMigrationData`: This method is called when `migrationData` (created by the method `toMigrationData`) will be applied to the member that will be the new partition owner.
- `clear`: This method is called when the partition migration is successfully completed and the old partition owner gets rid of all data in the partition. This method is also called when the partition migration operation fails and the to-be-the-new partition owner needs to roll back its changes.

10.1.7.1 Transferring `migrationData`

After you add these three methods to the `Container`, you need to create a `CounterMigrationOperation` class that transfers `migrationData` from one member to another and calls the method `applyMigrationData` on the correct partition of the new partition owner. A sample is shown below.

```java
import com.hazelcast.nio.ObjectDataInput;
import com.hazelcast.nio.ObjectDataOutput;
import com.hazelcast.spi.AbstractOperation;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class CounterMigrationOperation extends AbstractOperation {
    Map<String, Integer> migrationData;

    public CounterMigrationOperation() {
    }

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

    @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());
}

NOTE: During a partition migration, no other operations are executed on the related partition.

10.1.7.2 Letting Hazelcast Know CounterService Can Do Partition Migrations

We need to make our CounterService class implement the MigrationAwareService interface. This will let Hazelcast know that the CounterService can perform partition migration. See the below code.

```java
import com.hazelcast.core.DistributedObject;
import com.hazelcast.partition.MigrationEndpoint;
import com.hazelcast.spi.*;
import java.util.Map;
import java.util.Properties;

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

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

    @Override
    public void shutdown(boolean terminate) {
    }

    @Override
    public DistributedObject 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);
    }

    @Override
    public void beforeMigration(PartitionMigrationEvent e) {
    }
```
With the MigrationAwareService interface, some additional methods are exposed. For example, the method prepareMigrationOperation returns all the data of the partition that is going to be moved.

The method commitMigration commits the data, meaning in this case, it clears the partition container of the old owner.

10.1.7.3 Running the Sample Code

We can run the following code.

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;

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);

And we get the following output.

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

You can see that the counters have moved. 0counter moved from 192.168.1.103:5702 to 192.168.1.103:5705 and it is incremented correctly. Our counters can now move around in the cluster. You will see the counters will be
redistributed once you add or remove a cluster member.

10.1.8 Creating Backups

Finally, we make sure that the data of counter is available on another node when a member goes down. We need to have the `IncOperation` class implement the `BackupAwareOperation` interface contained in the SPI package. See the following code.

```java
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);
    }
}
```

The methods `getAsyncBackupCount` and `getSyncBackupCount` specify the count for asynchronous and synchronous backups. Our sample has one synchronous backup and no asynchronous backups. In the above code, counts of the backups are hard-coded, but they can also be passed to `IncOperation` as parameters.

The method `shouldBackup` specifies whether our Operation needs a backup or not. For our sample, it returns `true`, meaning the Operation will always have a backup even if there are no changes. Of course, in real systems, we want to have backups if there is a change. For `IncOperation` for example, having a backup when `amount` is null would be a good practice.

The method `getBackupOperation` returns the operation (`IncBackupOperation`) that actually performs the backup creation; the backup itself is an operation and will run on the same infrastructure.

If a backup should be made and `getSyncBackupCount` returns 3, then three `IncBackupOperation` instances are created and sent to the three machines containing the backup partition. If fewer machines are available, then backups need to be created. Hazelcast will just send a smaller number of operations.

### 10.1.8.1 Performing the Backup with IncBackupOperation

Now, let’s have a look at the `IncBackupOperation`.

```java
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);
}

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

10.1.8.2 Running the Sample Code

Let’s see the backup functionality in action with the following code.

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);
    }
}

Once it is run, the following output will be seen.

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

As it can be seen, both IncOperation and IncBackupOperation are executed. Notice that these operations have been executed on different cluster members to guarantee high availability.

10.2 WaitNotifyService

WaitNotifyService is an interface offered by SPI for the objects (e.g. Lock, Semaphore) to be used when a thread needs to wait for a lock to be released.

WaitNotifyService keeps a list of waiters. For each notify operation:

- it looks for a waiter,
- it asks the waiter whether it wants to keep waiting,
- if the waiter responds no, the service executes its registered operation (operation itself knows where to send a response),
- it rinses and repeats until a waiter wants to keep waiting.

Each waiter can sit on a wait-notify queue for, at most, its operation’s call timeout. For example, by default, each waiter can wait here for at most 1 minute. There is a continuous task that scans expired/timed-out waiters and invalidates them with CallTimeoutException. Each waiter on the remote side should retry and keep waiting if it still wants to wait. This is a liveness check for remote waiters.

This way, it is possible to distinguish an unresponsive node and a long (~infinite) wait. On the caller side, if the waiting thread does not get a response for either a call timeout or for more than 2 times the call-timeout, it will exit with OperationTimeoutException.

Note that this behavior breaks the fairness. Hazelcast does not support fairness for any of the data structures with blocking operations (i.e. lock and semaphore).
Chapter 11

Transactions

You can use Hazelcast in transactional context.

11.1 Transaction Interface

You create a TransactionContext to begin, commit, and rollback a transaction. You can obtain transaction-aware instances of queues, maps, sets, lists, multimaps via TransactionContext, work with them, and commit/rollback in one shot.

Hazelcast supports two types of transactions: LOCAL (One Phase) and TWO_PHASE. With the type, you have influence over how much guarantee you get when a member crashes while a transaction is committing. The default behavior is TWO_PHASE.

- **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. A problem can happen during the commit phase: if one or more members crash, then the system could be left in an inconsistent state.

- **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 TWO_PHASE commits, it copies the commit log to other members, so in case of member failure, another member can complete the commit.

```java
import java.util.Queue;
import java.util.Map;
import java.util.Set;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.Transaction;

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();

TransactionOptions options = new TransactionOptions()
    .setTransactionType( TransactionType.LOCAL );

TransactionContext context = hazelcastInstance.newTransactionContext( options )
    .beginTransaction();

TransactionalQueue queue = context.getQueue( "myqueue" );
TransactionalMap map = context.getMap( "mymap" );
TransactionalSet set = context.getSet( "myset" );

try {
    Object obj = queue.poll();
    //process obj
}
```

163
map.put( "1", "value1" );
set.add( "value" );
//do other things..
context.commitTransaction();
}
catch ( Throwable t ) {
    context.rollbackTransaction();
}

In a transaction, operations will not be executed immediately. Their changes will be local to the 
TransactionContext until committed. However, they will ensure the changes via locks.

For the above example, when map.put is executed, no data will be put to the map but the key will get locked for 
changes. While committing, operations will be executed, the value will be put to the map, and the key will be 
unlocked.

Isolation level in Hazelcast Transactions is READ_COMMITTED. If you are in a transaction, you can read the data in 
your transaction and the data that is already committed. If you are not in a transaction, you can only read the 
committed data.

**NOTE:** The REPEATABLE_READ isolation level can also be exercised using the method getForUpdate() of 
TransactionalMap.

Implementation is different for queue/set/list and map/multimap. For queue operations (offer, poll), offered and/or 
polled objects are copied to the owner member in order to safely commit/rollback. For map/multimap, Hazelcast first 
acquires the locks for the write operations (put, remove) and holds the differences (what is added/removed/updated) 
locally for each transaction. When the transaction is set to commit, Hazelcast will release the locks and apply the 
differences. When rolling back, Hazelcast will release the locks and discard the differences.

MapStore and QueueStore does not participate in transactions. Hazelcast will suppress exceptions thrown by store 
in a transaction. Please refer to the XA Transactions section for further information.

### 11.1.1 LOCAL versus TWO PHASE

As it can be understood from the above Transaction Interface section, when you choose LOCAL as the transaction 
type, Hazelcast tracks all changes you make locally in a commit log, i.e. list of changes. In this case, all the other 
members are asked to agree that the commit can succeed and once it is agreed, Hazelcast starts to write the changes. 
However, if the member that initiates the commit crashes after it has written to at least one member (but has not 
completed writing to all other members), your system may be left in an inconsistent state.

On the other hand, if you choose TWO_PHASE as the transaction type, the commit log is again tracked locally 
but it is copied to another cluster member. Therefore, when a failure happens (e.g. the member initiating the 
commit crashes), you still have the commit log in another member and that member can complete the commit. 
However, copying the commit log to another member makes the TWO_PHASE approach slow.

Consequently, it is recommended that you choose LOCAL as the transaction type if you want a better performance 
and TWO_PHASE if reliability of your system is more important than the performance.

### 11.2 XA Transactions

XA describes the interface between the global transaction manager and the local resource manager. XA allows 
multiple resources (such as databases, application servers, message queues, transactional caches, etc.) to be accessed 
within the same transaction, thereby preserving the ACID properties across applications. XA uses a two-phase 
commit to ensure that all resources either commit or rollback any particular transaction consistently (all do the 
same).

By implementing the XAResource interface, Hazelcast provides XA transactions. You can obtain the 
HazelcastXAResource instance via HazelcastInstance. Below is example code that uses Atomikos for 
transaction management.
11.3. J2EE INTEGRATION

UserTransactionManager tm = new UserTransactionManager();
tm.setTransactionTimeout(60);
tm.begin();

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
HazelcastXAResource xaResource = hazelcastInstance.getXAResource();

Transaction transaction = tm.getTransaction();
transaction.enlistResource(xaResource);
// other resources (database, app server etc...) can be enlisted

try {
    TransactionContext context = xaResource.getTransactionContext();
    TransactionalMap map = context.getMap("m");
    map.put("key", "value");
    // other resource operations

    transaction.deListResource(xaResource, XAResource.TMSUCCESS);
    tm.commit();
} catch (Exception e) {
    tm.rollback();
}

11.3 J2EE Integration

Hazelcast can be integrated into J2EE containers via the Hazelcast Resource Adapter (hazelcast-jca-rar-version.rar). After proper configuration, Hazelcast can participate in standard J2EE transactions.

<%@page import="javax.resource.ResourceException" %>
<%@page import="javax.transaction.*" %>
<%@page import="javax.naming.*" %>
<%@page import="javax.resource.cci.*" %>
<%@page import="java.util.*" %>
<%@page import="com.hazelcast.core.*" %>
<%@page import="com.hazelcast.jca.*" %>

<% UserTransaction txn = null; HazelcastConnection conn = null; HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();

try {
    Context context = new InitialContext();
    txn = (UserTransaction) context.lookup("java:comp/UserTransaction");
    txn.begin();

    HazelcastConnectionFactory cf = (HazelcastConnectionFactory)
        context.lookup("java:comp/env/HazelcastCF");

    conn = cf.getConnection();

    TransactionalMap<String, String> txMap = conn.getTransactionMap("default");
    txMap.put("key", "value");

    txn.commit();
} catch ( Throwable e ) {

if ( txn != null ) {
    try {
        txn.rollback();
    } catch ( Exception ix ) {
        ix.printStackTrace();
    }
}

e.printStackTrace();

finally {
    if ( conn != null ) {
        try {
            conn.close();
        } catch (Exception ignored) {
        }
    }
}

11.3.1 Sample Code for J2EE Integration

Please see our sample application for J2EE Integration.

11.3.2 Resource Adapter Configuration

Deploying and configuring the Hazelcast resource adapter is no different than configuring any other resource adapter since the Hazelcast resource adapter is a standard JCA one. However, resource adapter installation and configuration is container specific, so please consult your J2EE vendor documentation for details. The most common steps are:

1. Add the hazelcast-version.jar and hazelcast-jca-version.jar to the container's classpath. Usually there is a lib directory that is loaded automatically by the container on startup.
2. Deploy hazelcast-jca-rar-version.rar. Usually there is some kind of a deploy directory. The name of the directory varies by container.
3. Make container specific configurations when/after deploying hazelcast-jca-rar-version.rar. Besides container specific configurations, set the JNDI name for the Hazelcast resource.
4. Configure your application to use the Hazelcast resource. Update web.xml and/or ejb-jar.xml to let the container know that your application will use the Hazelcast resource and define the resource reference.
5. Make the container specific application configuration to specify the JNDI name used for the resource in the application.

11.3.3 Sample Glassfish v3 Web Application Configuration

1. Place the hazelcast-version.jar and hazelcast-jca-version.jar into the GLASSFISH_HOME/glassfish/domains/domain1/lib/ext/ folder.
2. Place the hazelcast-jca-rar-version.rar into GLASSFISH_HOME/glassfish/domains/domain1/autodeploy/ folder.
3. Add the following lines to the web.xml file.

```xml
<resource-ref>
    <res-ref-name>HazelcastCF</res-ref-name>
    <res-type>com.hazelcast.jca.ConnectionFactoryImpl</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

Notice that we did not have to put sun-ra.xml into the RAR file since it already comes with the hazelcast-ra-version.rar file.
If the Hazelcast resource is used from EJBs, you should configure `ejb-jar.xml` for resource reference and JNDI definitions, just like for the `web.xml` file.

### 11.3.4 Sample JBoss AS 5 Web Application Configuration

- Place the `hazelcast-version.jar` and `hazelcast-jca-version.jar` into the JBOSS_HOME/server/deploy/default/lib folder.
- Place the `hazelcast-jca-rar-version.rar` into the JBOSS_HOME/server/deploy/default/deploy folder.
- Create a `hazelcast-ds.xml` file containing the following content in the JBOSS_HOME/server/deploy/default/deploy folder. Make sure to set the `rar-name` element to `hazelcast-ra-version.rar`.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE connection-factories
PUBLIC "-//JBoss//DTD JBOSS JCA Config 1.5//EN"
"http://www.jboss.org/j2ee/dtd/jboss-ds_1_5.dtd">
<connection-factories>
  <tx-connection-factory>
    <local-transaction/>
    <track-connection-by-tx>true</track-connection-by-tx>
    <jndi-name>HazelcastCF</jndi-name>
    <rar-name>hazelcast-jca-rar-version.rar</rar-name>
    <connection-definition>
      javax.resource.cci.ConnectionFactory
    </connection-definition>
  </tx-connection-factory>
</connection-factories>
```

- Add the following lines to the `web.xml` file.

```xml
<resource-ref>
  <res-ref-name>HazelcastCF</res-ref-name>
  <res-type>com.hazelcast.jca.ConnectionFactoryImpl</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

- Add the following lines to the `jboss-web.xml` file.

```xml
<resource-ref>
  <res-ref-name>HazelcastCF</res-ref-name>
  <jndi-name>java:HazelcastCF</jndi-name>
</resource-ref>
```

If the Hazelcast resource is used from EJBs, you should configure `ejb-jar.xml` and `jboss.xml` for resource reference and JNDI definitions.

### 11.3.5 Sample JBoss AS 7 / EAP 6 Web Application Configuration

Deployment on JBoss AS 7 or JBoss EAP 6 is a fairly straightforward process. The steps you perform are shown below. The only non-trivial step is the creation of a new JBoss module with Hazelcast libraries.

- Create the folder `<jboss_home>/modules/system/layers/base/com/hazelcast/main`.
- Place the `hazelcast-<version>.jar` and `hazelcast-jca-<version>.jar` into the folder you created in the previous step.
• Create the file `module.xml` and place it in the same folder. This file should have the following content.

```xml
<module xmlns="urn:jboss:module:1.0" name="com.hazelcast">
<resources>
  <resource-root path=".">
  </resource-root>
  <resource-root path="hazelcast-<version>.jar"/>
  <resource-root path="hazelcast-jca-<version>.jar"/>
</resources>
<dependencies>
  <module name="sun.jdk"/>
  <module name="javax.api"/>
  <module name="javax.resource.api"/>
  <module name="javax.validation.api"/>
  <module name="org.jboss.ironjacamar.api"/>
</dependencies>
</module>
```

At this point, you have a new JBoss module with Hazelcast in it. You can now start JBoss and deploy the `hazelcast-jca-rar-<version>.rar` file via JBoss CLI or Administration Console.

Once the Hazelcast Resource Adapter is deployed, you can start using it. The easiest way is to let a container inject `ConnectionFactory` into your beans.

```java
package com.hazelcast.examples.rar;

import com.hazelcast.core.TransactionalMap;
import com.hazelcast.jca.HazelcastConnection;
import javax.annotation.Resource;
import javax.resource.ResourceException;
import javax.resource.cci.ConnectionFactory;
import java.util.logging.Level;
import java.util.logging.Logger;

@javax.ejb.Stateless
public class ExampleBean implements ExampleInterface {
    private final static Logger log = Logger.getLogger(ExampleBean.class.getName());
    
    @Resource(mappedName = "java:/HazelcastCF")
    protected ConnectionFactory connectionFactory;

    public void insert(String key, String value) {
        HazelcastConnection hzConn = null;
        try {
            hzConn = getConnection();
            TransactionalMap<String,String> txmap = hzConn.getTransactionMap("txmap");
            txmap.put(key, value);
        } finally {
            closeConnection(hzConn);
        }
    }

    private HazelcastConnection getConnection() {
        try {
            return (HazelcastConnection) connectionFactory.getConnection();
        } catch (ResourceException e) {
            throw new RuntimeException("Error while getting Hazelcast connection", e);
        }
    }
}
```
private void closeConnection(HazelcastConnection hzConn) {
    if (hzConn != null) {
        try {
            hzConn.close();
        } catch (ResourceException e) {
            log.log(Level.WARNING, "Error while closing Hazelcast connection.", e);
        }
    }
}

11.3.5.1 Known Issues

- There is a regression in JBoss EAP 6.1.0 causing failure during Hazelcast Resource Adapter deployment. The issue is fixed in JBoss EAP 6.1.1. See this for additional details.
Chapter 12

Hazelcast JCache

This chapter describes the basics of JCache: the standardized Java caching layer API. The JCache caching API is specified by the Java Community Process (JCP) as Java Specification Request (JSR) 107.

Caching keeps data in memory that either are slow to calculate/process or originate from another underlying backend system whereas caching is used to prevent additional request round trips for frequently used data. In both cases, caching could be used to gain performance or decrease application latencies.

12.1 JCache Overview

Starting with Hazelcast release 3.3.1, a specification compliant JCache implementation is offered. To show our commitment to this important specification the Java world was waiting for over a decade, we do not just provide a simple wrapper around our existing APIs but implemented a caching structure from ground up to optimize the behavior to the needs of JCache. As mentioned before, the Hazelcast JCache implementation is 100% TCK (Technology Compatibility Kit) compliant and therefore passes all specification requirements.

In addition to the given specification, we added some features like asynchronous versions of almost all operations to give the user extra power.

This chapter gives a basic understanding of how to configure your application and how to setup Hazelcast to be your JCache provider. It also shows examples of basic JCache usage as well as the additionally offered features that are not part of JSR-107. To gain a full understanding of the JCache functionality and provided guarantees of different operations, read the specification document (which is also the main documentation for functionality) at the specification page of JSR-107:


12.2 Setup and Configuration

This sub-chapter shows what is necessary to provide the JCache API and the Hazelcast JCache implementation for your application. In addition, it demonstrates the different configuration options as well as a description of the configuration properties.

12.2.1 Application Setup

To provide your application with this JCache functionality, your application needs the JCache API inside its classpath. This API is the bridge between the specified JCache standard and the implementation provided by Hazelcast.

The way to integrate the JCache API JAR into the application classpath depends on the build system used. For Maven, Gradle, SBT, Ivy and many other build systems, all using Maven based dependency repositories, perform the integration by adding the Maven coordinates to the build descriptor.
As already mentioned, next to the default Hazelcast coordinates that might be already part of the application, you have to add JCache coordinates.

For Maven users, the coordinates look like the following code:

```xml
<dependency>
  <groupId>javax.cache</groupId>
  <artifactId>cache-api</artifactId>
  <version>1.0.0</version>
</dependency>
```

With other build systems, you might need to describe the coordinates in a different way.

### 12.2.1.1 Activating Hazelcast as JCache Provider

To activate Hazelcast as the JCache provider implementation, add either `hazelcast-all.jar` or `hazelcast.jar` to the classpath (if not already available) by either one of the following Maven snippets.

If you use `hazelcast-all.jar`:

```xml
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast-all</artifactId>
  <version>3.4</version>
</dependency>
```

If you use `hazelcast.jar`:

```xml
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast</artifactId>
  <version>3.4</version>
</dependency>
```

The users of other build systems have to adjust the way of defining the dependency to their needs.

### 12.2.1.2 Connecting Clients to Remote Server

When the users want to use Hazelcast clients to connect to a remote cluster, the `hazelcast-client.jar` dependency is also required on the client side applications. This JAR is already included in `hazelcast-all.jar`. Or, you can add it to the classpath using the following Maven snippet:

```xml
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast</artifactId>
  <version>3.4</version>
</dependency>
```

For other build systems, e.g. ANT, the users have to download these dependencies from either the JSR-107 specification and Hazelcast community website (http://www.hazelcast.org) or from the Maven repository search page (http://search.maven.org).
12.2. Quick Example

Before moving on to configuration, let’s have a look at a basic introductory example. The following code shows how to use the Hazelcast JCache integration inside an application in an easy but typesafe way.

```java
// Retrieve the CachingProvider which is automatically backed by
// the chosen Hazelcast server or client provider
CachingProvider cachingProvider = Caching.getCachingProvider();

// Create a CacheManager
CacheManager cacheManager = cachingProvider.getCacheManager();

// Create a simple but typesafe configuration for the cache
CompleteConfiguration<String, String> config =
    new MutableConfiguration<String, String>()
        .setTypes( String.class, String.class );

// Create and get the cache
Cache<String, String> cache = cacheManager.createCache( "example", config );
// Alternatively to request an already existing cache
// Cache<String, String> cache = cacheManager
//     .getCache( name, String.class, String.class );

// Put a value into the cache
cache.put( "world", "Hello World" );

// Retrieve the value again from the cache
String value = cache.get( "world" );

// Print the value 'Hello World'
System.out.println( value );
```

Although the example is simple, let’s go through the code lines one by one.

12.2.2.1 Getting the Hazelcast JCache Implementation

First of all, we retrieve the `javax.cache.spi.CachingProvider` using the static method from `javax.cache.Caching::getCachingManager` which automatically picks up Hazelcast as the underlying JCache implementation, if available in the classpath. This way the Hazelcast implementation of a `CachingProvider` will automatically start a new Hazelcast node or client (depending on the chosen provider type) and pick up the configuration from either the command line parameter or from the classpath. We will show how to use an existing `HazelcastInstance` later in this chapter, for now we keep it simple.

12.2.2.2 Setting up the JCache Entry Point

In the next line, we ask the `CachingProvider` to return a `javax.cache.CacheManager`. This is the general application’s entry point into JCache. The `CachingProvider` creates and manages named caches.

12.2.2.3 Configuring the Cache Before Creating It

The next few lines create a simple `javax.cache.configuration.MutableConfiguration` to configure the cache before actually creating it. In this case, we only configure the key and value types to make the cache typesafe which is highly recommended and checked on retrieval of the cache.
12.2.2.4 Creating the Cache

To create the cache, we call `javax.cache.CacheManager::createCache` with a name for the cache and the previously created configuration; the call returns the created cache. If you need to retrieve a previously created cache, you can use the corresponding method overload `javax.cache.CacheManager::getCache`. If the cache was created using type parameters, you must retrieve the cache afterward using the type checking version of `getCache`.

12.2.2.5 get, put, and getAndPut

The following lines are simple `put` and `get` calls from the `java.util.Map` interface. The `javax.cache.Cache::put` has a `void` return type and does not return the previously assigned value of the key. To imitate the `java.util.Map::put` method, the JCache cache has a method called `getAndPut`.

12.2.3 JCache Configuration

Hazelcast JCache provides two different ways of cache configuration:

- programmatically: the typical Hazelcast way, using the Config API seen above,
- and declaratively: using `hazelcast.xml` or `hazelcast-client.xml`.

12.2.3.1 JCache Declarative Configuration

You can declare your JCache cache configuration using the `hazelcast.xml` or `hazelcast-client.xml` configuration files. Using this declarative configuration makes the creation of the `javax.cache.Cache` fully transparent and automatically ensures internal thread safety. You do not need a call to `javax.cache.Cache::createCache` in this case: you can retrieve the cache using `javax.cache.Cache::getCache` overloads and by passing in the name defined in the configuration for the cache.

To retrieve the cache defined in the declaration files, you need only perform a simple call (example below) because the cache is created automatically by the implementation.

```java
CachingProvider cachingProvider = Caching.getCachingProvider();
CacheManager cacheManager = cachingProvider.getCacheManager();
Cache<Object, Object> cache = cacheManager.getCache( "default", Object.class, Object.class );
```

Note that this section only describes the JCache provided standard properties. For the Hazelcast specific properties, please see the JCache Configuration section.

```xml
<cache name="default">
  <key-type class-name="java.lang.Object" />
  <value-type class-name="java.lang.Object" />
  <statistics-enabled>false</statistics-enabled>
  <management-enabled>false</management-enabled>
  <read-through>true</read-through>
  <write-through>true</write-through>
  <cache-loader-factory
    class-name="com.example.cache.MyCacheLoaderFactory" />
  <cache-writer-factory
    class-name="com.example.cache.MyCacheWriterFactory" />
  <expiry-policy-factory
    class-name="com.example.cache.MyExpirePolicyFactory" />
  <cache-entry-listeners>
```

```xml```
12.2. SETUP AND CONFIGURATION

```xml
<cache-entry-listener old-value-required="false" synchronous="false">
  <cache-entry-listener-factory
    class-name="com.example.cache.MyEntryListenerFactory" />
  <cache-entry-event-filter-factory
    class-name="com.example.cache.MyEntryEventFilterFactory" />
</cache-entry-listener>
...
</cache-entry-listeners>
</cache>
```

- **key-type##class-name**: The fully qualified class name of the cache key type, defaults to `java.lang.Object`.
- **value-type##class-name**: The fully qualified class name of the cache value type, defaults to `java.lang.Object`.
- **statistics-enabled**: If set to true, statistics like cache hits and misses are collected. Its default value is false.
- **management-enabled**: If set to true, JMX beans are enabled and collected statistics are provided - It doesn't automatically enables statistics collection, defaults to false.
- **read-through**: If set to true, enables read-through behavior of the cache to an underlying configured `javax.cache.integration.CacheLoader` which is also known as lazy-loading, defaults to false.
- **write-through**: If set to true, enables write-through behavior of the cache to an underlying configured `javax.cache.integration.CacheWriter` which passes any changed value to the external backend resource, defaults to false.
- **cache-loader-factory##class-name**: The fully qualified class name of the `javax.cache.configuration.Factory` implementation providing a `javax.cache.integration.CacheLoader` instance to the cache.
- **cache-writer-factory##class-name**: The fully qualified class name of the `javax.cache.configuration.Factory` implementation providing a `javax.cache.integration.CacheWriter` instance to the cache.
- **expiry-policy-factory##class-name**: The fully qualified class name of the `javax.cache.expiry.ExpiryPolicy` instance to the cache.
- **cache-entry-listener**: A set of attributes and elements, explained below, to describe a `javax.cache.event.CacheEntryListener`.
  - **cache-entry-listener##old-value-required**: If set to true, previously assigned values for the affected keys will be sent to the `javax.cache.event.CacheEntryListener` implementation. Setting this attribute to true creates additional traffic, defaults to false.
  - **cache-entry-listener##synchronous**: If set to true, the `javax.cache.event.CacheEntryListener` implementation will be called in a synchronous manner, defaults to false.
  - **cache-entry-listener/entry-listener-factory##class-name**: The fully qualified class name of the `javax.cache.configuration.Factory` implementation providing a `javax.cache.event.CacheEntryListener` instance.
  - **cache-entry-listener/entry-event-filter-factory##class-name**: The fully qualified class name of the `javax.cache.event.CacheEntryEventFilter` instance.

**NOTE**: The JMX MBeans provided by Hazelcast JCache show statistics of the local node only. To show the cluster-wide statistics, the user should collect statistic information from all nodes and accumulate them to the overall statistics.

12.2.3.2 JCache Programmatic Configuration

To configure the JCache programmatically:

- either instantiate `javax.cache.configuration.MutableConfiguration` if you will use only the JCache standard configuration,
- or instantiate `com.hazelcast.config.CacheConfig` for a deeper Hazelcast integration.
com.hazelcast.config.CacheConfig offers additional options that are specific to Hazelcast like asynchronous and synchronous backup counts. Both classes share the same supertype interface `javax.cache.configuration.CompleteConfiguration` which is part of the JCache standard.

**NOTE:** To stay vendor independent, try to keep your code as near as possible to the standard JCache API. We recommend you to use declarative configuration and use the `javax.cache.configuration.Configuration` or `javax.cache.configuration.CompleteConfiguration` interfaces in your code only when you need to pass the configuration instance throughout your code.

If you don’t need to configure Hazelcast specific properties, it is recommended that you instantiate `javax.cache.configuration.MutableConfiguration` and that you use the setters to configure Hazelcast as shown in the example in the Quick Example section. Since the configurable properties are the same as the ones explained in the JCache Declarative Configuration section, they are not mentioned here. For Hazelcast specific properties, please read the ICache Configuration section section.

### 12.3 JCache Providers

Use JCache providers to create caches for a specification compliant implementation. Those providers abstract the platform specific behavior and bindings, and provide the different JCache required features.

Hazelcast has two types of providers. Depending on your application setup and the cluster topology, you can use the Client Provider (used from Hazelcast clients) or the Server Provider (used by cluster nodes).

#### 12.3.1 Provider Configuration

You configure the JCache `javax.cache.spi.CachingProvider` by either specifying the provider at the command line or by declaring the provider inside the Hazelcast configuration XML file. For more information on setting properties in this XML configuration file, please see the JCache Declarative Configuration section.

Hazelcast implements a delegating `CachingProvider` that can automatically be configured for either client or server mode and that delegates to the real underlying implementation based on the user’s choice. It is recommended that you use this `CachingProvider` implementation.

The delegating `CachingProviders` fully qualified class name is:

`com.hazelcast.cache.HazelcastCachingProvider`

To configure the delegating provider at the command line, add the following parameter to the Java startup call, depending on the chosen provider:

```
-Dhazelcast.jcache.provider.type=[client|server]
```

By default, the delegating `CachingProvider` is automatically picked up by the JCache SPI and provided as shown above. In cases where multiple `javax.cache.spi.CachingProvider` implementations reside on the classpath (like in some Application Server scenarios), you can select a `CachingProvider` by explicitly calling `Caching::getCachingProvider` overloads and providing them using the canonical class name of the provider to be used. The class names of server and client providers provided by Hazelcast are mentioned in the following two subsections.

**NOTE:** Hazelcast advises that you use the `Caching::getCachingProvider` overloads to select a `CachingProvider` explicitly. This ensures that uploading to later environments or Application Server versions doesn’t result in unexpected behavior like choosing a wrong `CachingProvider`.

For more information on cluster topologies and Hazelcast clients, please see the Hazelcast Topology section.
12.3.2 JCache Client Provider

For cluster topologies where Hazelcast light clients are used to connect to a remote Hazelcast cluster, use the Client Provider to configure JCache.

The Client Provider provides the same features as the Server Provider. However, it does not hold data on its own but instead delegates requests and calls to the remotely connected cluster.

The Client Provider can connect to multiple clusters at the same time. This can be achieved by scoping the client side `CacheManager` with different Hazelcast configuration files. For more information, please see the Scopes and Namespaces section.

For requesting this `CachingProvider` using `Caching#getCachingProvider(String)` or `Caching#getCachingProvider(String, ClassLoader)`, use the following fully qualified class name:

```
com.hazelcast.client.cache.impl.HazelcastClientCachingProvider
```

12.3.3 JCache Server Provider

If a Hazelcast node is embedded into an application directly and the Hazelcast client is not used, the Server Provider is required. In this case, the node itself becomes a part of the distributed cache and requests and operations are distributed directly across the cluster by its given key.

The Server Provider provides the same features as the Client provider, but it keeps data in the local Hazelcast node and also distributes non-owned keys to other direct cluster members.

Like the Client Provider, the Server Provider is able to connect to multiple clusters at the same time. This can be achieved by scoping the client side `CacheManager` with different Hazelcast configuration files. For more information please see the Scopes and Namespaces section.

To request this `CachingProvider` using `Caching#getCachingProvider(String)` or `Caching#getCachingProvider(String, ClassLoader)`, use the following fully qualified class name:

```
com.hazelcast.cache.impl.HazelcastServerCachingProvider
```

12.4 Introduction to the JCache API

This section explains the JCache API by providing simple examples and use cases. While walking through the examples, we will have a look at a couple of the standard API classes and see how these classes are used.

12.4.1 JCache API Walk-through

The code in this subsection creates a small account application by providing a caching layer over an imagined database abstraction. The database layer will be simulated using single demo data in a simple DAO interface. To show the difference between the “database” access and retrieving values from the cache, a small waiting time is used in the DAO implementation to simulate network and database latency.

12.4.1.1 Basic User Class

Before we implement the JCache caching layer, let’s have a quick look at some basic classes we need for this example.

The `User` class is the representation of a user table in the database. To keep it simple, it has just two properties: `userId` and `username`.
public class User {
    private int userId;
    private String username;

    // Getters and setters
}

12.4.1.2 DAO Interface Example

The DAO interface is also kept easy in this example. It provides a simple method to retrieve (find) a user by its
userId.

public interface UserDAO {
    User findUserById( int userId );
    boolean storeUser( int userId, User user );
    boolean removeUser( int userId );
    Collection<Integer> allUserIds();
}

12.4.1.3 Configuration Example

To show most of the standard features, the configuration example is a little more complex.

```
// Create javax.cache.configuration.CompleteConfiguration subclass
CompleteConfiguration<Integer, User> config =
    new MutableConfiguration<Integer, User>(){
        // Configure the cache to be typesafe
        .setTypes( Integer.class, User.class )
        // Configure to expire entries 30 secs after creation in the cache
        .setExpiryPolicyFactory( FactoryBuilder.factoryOf(
            new AccessedExpiryPolicy( new Duration( TimeUnit.SECONDS, 30 ) )
        )
        // Configure read-through of the underlying store
        .setReadThrough( true )
        // Configure write-through to the underlying store
        .setWriteThrough( true )
        // Configure the javax.cache.integration.CacheLoader
        .setCacheLoaderFactory( FactoryBuilder.factoryOf(
            new UserCacheLoader( userDao )
        )
        // Configure the javax.cache.integration.CacheWriter
        .setCacheWriterFactory( FactoryBuilder.factoryOf(
            new UserCacheWriter( userDao )
        )
        // Configure the javax.cache.event.CacheEntryListener with no
        // javax.cache.event.CacheEntryEventFilter, to include old value
        // and to be executed synchronously
        .addCacheEntryListenerConfiguration(
            new MutableCacheEntryListenerConfiguration<Integer, User>(
                new UserCacheEntryListenerFactory(),
                null, true, true
            )
        );
```

Let’s go through this configuration line by line.
12.4. INTRODUCTION TO THE JCACHE API

12.4.1.4 Setting the Cache Type and Expire Policy

First, we set the expected types for the cache, which is already known from the previous example. On the next line, an `javax.cache.expiry.ExpirePolicy` is configured. Almost all integration `ExpirePolicy` implementations are configured using `javax.cache.configuration.Factory` instances. `Factory` and `FactoryBuilder` are explained later in this chapter.

12.4.1.5 Configuring Read-Through and Write-Through

The next two lines configure the thread that will be read-through and write-through to the underlying backend resource that is configured over the next few lines. The JCache API offers `javax.cache.integration.CacheLoader` and `javax.cache.integration.CacheWriter` to implement adapter classes to any kind of backend resource, e.g. JPA, JDBC, or any other backend technology implementable in Java. The interfaces provide the typical CRUD operations like `create`, `get`, `update`, `delete` and some bulk operation versions of those common operations. We will look into the implementation of those implementations later.

12.4.1.6 Configuring Entry Listeners

The last configuration setting defines entry listeners based on sub-interfaces of `javax.cache.event.CacheEntryListener`. This config does not use a `javax.cache.event.CacheEntryEventFilter` since the listener is meant to be fired on every change that happens on the cache. Again we will look in the implementation of the listener in later in this chapter.

12.4.1.7 Full Example Code

A full running example that is presented in this subsection is available in the code samples repository. The application is built to be a command line app. It offers a small shell to accept different commands. After startup, you can enter `help` to see all available commands and their descriptions.

12.4.2 Roundup of Basics

In the Quick Example section, we have already seen a couple of the base classes and explained how those work. Following are quick descriptions of them.

**javax.cache.Caching:**

The access point into the JCache API. It retrieves the general `CachingProvider` backed by any compliant JCache implementation, such as Hazelcast JCache.

**javax.cache.spi.CachingProvider:**

The SPI that is implemented to bridge between the JCache API and the implementation itself. Hazelcast nodes and clients use different providers chosen as seen in the Provider Configuration section which enable the JCache API to interact with Hazelcast clusters.

When a `javax.cache.spi.CachingProvider::getCacheManager` overload is used that takes a `java.lang.ClassLoader` argument, this classloader will be part of the scope of the created `java.cache.Cache` and it is not possible to retrieve it on other nodes. We advise not to use those overloads, those are not meant to be used in distributed environments!

**javax.cache.CacheManager:**

The `CacheManager` provides the capability to create new and manage existing JCache caches.

![NOTE: A `javax.cache.Cache` instance created with key and value types in the configuration provides a type checking of those types at retrieval of the cache. For that reason, all non-types retrieval methods like `getCache` throw an exception because types cannot be checked.](image-url)

**javax.cache.configuration.Configuration, javax.cache.configuration.MutableConfiguration:**
180  

CHAPTER 12. HAZELCAST JCACHE

These two classes are used to configure a cache prior to retrieving it from a `CacheManager`. The `Configuration` interface, therefore, acts as a common super type for all compatible configuration classes such as `MutableConfiguration`. Hazelcast itself offers a special implementation (`com.hazelcast.config.CacheConfig`) of the `Configuration` interface which offers more options on the specific Hazelcast properties that can be set to configure features like synchronous and asynchronous backups counts or selecting the underlying In Memory Format of the cache. For more information on this configuration class, please see the reference in JCache Programmatic Configuration section.

**javax.cache.Cache:**

This interface represents the cache instance itself. It is comparable to `java.util.Map` but offers special operations dedicated to the caching use case. Therefore, for example `javax.cache.Cache::put`, unlike `java.util.Map::put`, does not return the old value previously assigned to the given key.

⚠️ **NOTE:** Bulk operations on the `Cache` interface guarantee atomicity per entry but not over all given keys in the same bulk operations since no transactional behavior is applied over the whole batch process.

12.4.3 Factory and FactoryBuilder

The `javax.cache.configuration.Factory` implementations are used to configure features like `CacheEntryListener`, `ExpirePolicy` and `CacheLoaders` or `CacheWriters`. These factory implementations are required to distribute the different features to nodes in a cluster environment like Hazelcast. Therefore, these factory implementations have to be serializable.

Factory implementations are easy to do: they follow the default Provider- or Factory-Pattern. The sample class `UserCacheEntryListenerFactory` shown below implements a custom JCache Factory.

```java
public class UserCacheEntryListenerFactory implements Factory<CacheEntryListener<Integer, User>> {

    @Override
    public CacheEntryListener<Integer, User> create() {
        // Just create a new listener instance
        return new UserCacheEntryListener();
    }
}
```

To simplify the process for the users, JCache API offers a set of helper methods collected in `javax.cache.configuration.FactoryBuilder`. In the above configuration example, `FactoryBuilder::factoryOf` is used to create a singleton factory for the given instance.

12.4.4 CacheLoader

`javax.cache.integration.CacheLoader` loads cache entries from any external backend resource. If the cache is configured to be `read-through`, then `CacheLoader::load` is called transparently from the cache when the key or the value is not yet found in the cache. If no value is found for a given key, it returns null.

If the cache is not configured to be `read-through`, nothing is loaded automatically. However, the user code must call `javax.cache.Cache::loadAll` to load data for the given set of keys into the cache.

For the bulk load operation (`loadAll()`), some keys may not be found in the returned result set. In this case, a `javax.cache.integration.CompletionListener` parameter can be used as an asynchronous callback after all the key-value pairs are loaded because loading many key-value pairs can take lots of time.

Let’s look at the `UserCacheLoader` implementation.

```java
public class UserCacheLoader implements CacheLoader<Integer, User>, Serializable {
```
12.4. INTRODUCTION TO THE JCACHE API

```java
private final UserDao userDao;

public UserCacheLoader( UserDao userDao ) {
    // Store the dao instance created externally
    this.userDao = userDao;
}

@Override
public User load( Integer key )
    throws CacheLoaderException {
    // Just call through into the dao
    return userDao.findUserById( key );
}

@Override
public Map<Integer, User> loadAll( Iterable<? extends Integer> keys )
    throws CacheLoaderException {
    // Create the resulting map
    Map<Integer, User> loaded = new HashMap<Integer, User>();
    // For every key in the given set of keys
    for ( Integer key : keys ) {
        // Try to retrieve the user
        User user = userDao.findUserById( key );
        // If user is not found do not add the key to the result set
        if ( user != null ) {
            loaded.put( key, user );
        }
    }
    return loaded;
}
```

The implementation is quite straightforward. An important note is that any kind of exception has to be wrapped into `javax.cache.integration.CacheLoaderException`.

12.4.5 CacheWriter

A `javax.cache.integration.CacheWriter` is used to update an external backend resource. If the cache is configured to be write-through this process is executed transparently to the users code otherwise at the current state there is no way to trigger writing changed entries to the external resource to a user defined point in time.

If bulk operations throw an exception, `java.util.Collection` has to be cleaned of all successfully written keys so the cache implementation can determine what keys are written and can be applied to the cache state.

```java
public class UserCacheWriter
    implements CacheWriter<Integer, User>, Serializable {
    private final UserDao userDao;

    public UserCacheWriter( UserDao userDao ) {
        // Store the dao instance created externally
        this.userDao = userDao;
    }

    @Override
    public void write( Cache.Entry<? extends Integer, ? extends User> entry )
        throws CacheWriterException {
```
// Store the user using the dao
userDao.storeUser( entry.getKey(), entry.getValue() );
}

@Override
public void writeAll( Collection<Cache.Entry<...>> entries )
throws CacheWriterException {

// Retrieve the iterator to clean up the collection from
// written keys in case of an exception
Iterator<Cache.Entry<...>> iterator = entries.iterator();
while ( iterator.hasNext() ) {
    // Write entry using dao
    write( iterator.next() );
    // Remove from collection of keys
    iterator.remove();
}
}

@Override
public void delete( Object key )
throws CacheWriterException {
    // Test for key type
    if ( !( key instanceof Integer ) ) {
        throw new CacheWriterException( "Illegal key type" );
    }
    // Remove user using dao
    userDao.removeUser( ( Integer ) key );
}

@Override
public void deleteAll( Collection<?> keys )
throws CacheWriterException {
    // Retrieve the iterator to clean up the collection from
    // written keys in case of an exception
    Iterator<?> iterator = keys.iterator();
    while ( iterator.hasNext() ) {
        // Write entry using dao
        delete( iterator.next() );
        // Remove from collection of keys
        iterator.remove();
    }
}

Again the implementation is pretty straight forward and also as above all exceptions thrown by the external resource,
like java.sql.SQLException has to be wrapped into a javax.cache.integration.CacheWriterException. Note
this is a different exception from the one thrown by CacheLoader.

12.4.6 JCache EntryProcessor

With javax.cache.processor.EntryProcessor, you can apply an atomic function to a cache entry. In a distributed
environment like Hazelcast, you can move the mutating function to the node that owns the key. If the value object
is big, it might prevent traffic by sending the object to the mutator and sending it back to the owner to update it.

By default, Hazelcast JCache sends the complete changed value to the backup partition. Again, this can cause a lot
of traffic if the object is big. Another option to prevent this is part of the Hazelcast ICache extension. Further
information is available at BackupAwareEntryProcessor.
An arbitrary number of arguments can be passed to the `Cache::invoke` and `Cache::invokeAll` methods. All of those arguments need to be fully serializable because in a distributed environment like Hazelcast, it is very likely that these arguments have to be passed around the cluster.

```java
public class UserUpdateEntryProcessor
    implements EntryProcessor<Integer, User, User> {

    @Override
    public User process(MutableEntry<Integer, User> entry, Object... arguments)
    throws EntryProcessorException {

        // Test arguments length
        if (arguments.length < 1) {
            throw new EntryProcessorException("One argument needed: username");
        }

        // Get first argument and test for String type
        Object argument = arguments[0];
        if ( !(argument instanceof String) ) {
            throw new EntryProcessorException("First argument has wrong type, required java.lang.String");
        }

        // Retrieve the value from the MutableEntry
        User user = entry.getValue();

        // Retrieve the new username from the first argument
        String newUsername = (String) arguments[0];

        // Set the new username
        user.setUsername(newUsername);

        // Set the changed user to mark the entry as dirty
        entry.setValue(user);

        // Return the changed user to return it to the caller
        return user;
    }
}
```

**NOTE:** By executing the bulk `Cache::invokeAll` operation, atomicity is only guaranteed for a single cache entry. No transactional rules are applied to the bulk operation.

**NOTE:** JCache `EntryProcessor` implementations are not allowed to call `javax.cache.Cache` methods; this prevents operations from deadlocking between different calls.

In addition, when using a `Cache::invokeAll` method, a `java.util.Map` is returned that maps the key to its `javax.cache.processor.EntryProcessorResult`, and which itself wraps the actual result or a thrown `javax.cache.processor.EntryProcessorException`.

### 12.4.7 CacheEntryListener

The `javax.cache.event.CacheEntryListener` implementation is straight forward. `CacheEntryListener` is a super-interface which is used as a marker for listener classes in JCache. The specification brings a set of sub-interfaces.
• **CacheEntryCreatedListener**: Fires after a cache entry is added (even on read-through by a CacheLoader) to the cache.
• **CacheEntryUpdatedListener**: Fires after an already existing cache entry was updates.
• **CacheEntryRemovedListener**: Fires after a cache entry was removed (not expired) from the cache.
• **CacheEntryExpiredListener**: Fires after a cache entry has been expired. Expiry does not have to be parallel process, it is only required to be executed on the keys that are requested by Cache::get and some other operations. For a full table of expiry please see the https://www.jcp.org/en/jsr/detail?id=107 point 6.

To configure CacheEntryListener, add a javax.cache.configuration.CacheEntryListenerConfiguration instance to the JCache configuration class, as seen in the above example configuration. In addition listeners can be configured to be executed synchronously (blocking the calling thread) or asynchronously (fully running in parallel).

In this example application, the listener is implemented to print event information on the console. That visualizes what is going on in the cache.

```java
public class UserCacheEntryListener implements CacheEntryCreatedListener<Integer, User>, CacheEntryUpdatedListener<Integer, User>, CacheEntryRemovedListener<Integer, User>, CacheEntryExpiredListener<Integer, User> {

    @Override
    public void onCreated( Iterable<CacheEntryEvent<?>> cacheEntryEvents )
        throws CacheEntryListenerException {
        printEvents( cacheEntryEvents );
    }

    @Override
    public void onUpdated( Iterable<CacheEntryEvent<?>> cacheEntryEvents )
        throws CacheEntryListenerException {
        printEvents( cacheEntryEvents );
    }

    @Override
    public void onRemoved( Iterable<CacheEntryEvent<?>> cacheEntryEvents )
        throws CacheEntryListenerException {
        printEvents( cacheEntryEvents );
    }

    @Override
    public void onExpired( Iterable<CacheEntryEvent<?>> cacheEntryEvents )
        throws CacheEntryListenerException {
        printEvents( cacheEntryEvents );
    }

    private void printEvents( Iterable<CacheEntryEvent<?>> cacheEntryEvents ) {
        Iterator<CacheEntryEvent<?>> iterator = cacheEntryEvents.iterator();
        while ( iterator.hasNext() ) {
            CacheEntryEvent<?> event = iterator.next();
            System.out.println( event.getEventType() );
        }
    }
}
```
12.4.8 ExpirePolicy

In JCache, `javax.cache.expiry.ExpirePolicy` implementations are used to automatically expire cache entries based on different rules.

Expiration timeouts are defined using `javax.cache.expiry.Duration`, which is a pair of `java.util.concurrent.TimeUnit`, which describes a time unit and a long, defining the timeout value. The minimum allowed `TimeUnit` is `TimeUnit.MILLISECONDS`. The long value `durationAmount` must be equal or greater than zero. A value of zero (or `Duration.ZERO`) indicates that the cache entry expires immediately.

By default, JCache delivers a set of predefined expiry strategies in the standard API.

- `AccessedExpiryPolicy`: Expires after a given set of time measured from creation of the cache entry, the expiry timeout is updated on accessing the key.
- `CreatedExpiryPolicy`: Expires after a given set of time measured from creation of the cache entry, the expiry timeout is never updated.
- `EternalExpiryPolicy`: Never expires, this is the default behavior, similar to `ExpiryPolicy` to be set to null.
- `ModifiedExpiryPolicy`: Expires after a given set of time measured from creation of the cache entry, the expiry timeout is updated on updating the key.
- `TouchedExpiryPolicy`: Expires after a given set of time measured from creation of the cache entry, the expiry timeout is updated on accessing or updating the key.

Because `EternalExpiryPolicy` does not expire cache entries, it is still possible to evict values from memory if an underlying `CacheLoader` is defined.

### 12.5 Hazelcast JCache Extension - ICache

Hazelcast provides extension methods to Cache API through the interface `com.hazelcast.cache.ICache`.

It has two sets of extensions:

- Asynchronous version of all cache operations.
- Cache operations with custom `ExpiryPolicy` parameter to apply on that specific operation.

#### 12.5.1 Scopes and Namespaces

As mentioned before, a `CacheManager` can be scoped in the case of client to connect to multiple clusters, or in the case of an embedded node, a `CacheManager` can be scoped to join different clusters at the same time. This process is called scoping. To apply it, request a `CacheManager` by passing a `java.net.URI` instance to `CachingProvider::getCacheManager`. The `java.net.URI` instance must point to either a Hazelcast configuration or to the name of a named `com.hazelcast.core.HazelcastInstance` instance.

#### 12.5.1.1 Configuration Scope

To connect or join different clusters, apply a configuration scope to the `CacheManager`. If the same `URI` is used to request a `CacheManager` that was created previously, those `CacheManagers` share the same underlying `HazelcastInstance`.

**NOTE:** Multiple requests for the same `java.net.URI` result in returning a `CacheManager` instance that shares the same `HazelcastInstance` as the `CacheManager` returned by the previous call.

#### 12.5.1.2 Configuration Scope

To apply a configuration scope, pass in the path of the configuration file using the location property `HazelcastCachingProvider#HAZELCAST_CONFIG_LOCATION` (which resolves to `hazelcast.config.location`) as a mapping inside a `java.util.Properties` instance to the `CachingProvider#getCacheManager(uri, classLoader, properties)` call.

Here is an example of using Configuration Scope.
CachingProvider cachingProvider = Caching.getCachingProvider();

// Create Properties instance pointing to a Hazelcast config file
Properties properties = new Properties();
properties.setProperty( HazelcastCachingProvider.HAZELCAST_CONFIG_LOCATION,
"classpath://my-configs/scoped-hazelcast.xml" );

URI cacheManagerName = new URI("my-cache-manager");
CacheManager cacheManager = cachingProvider
  .getCacheManager( cacheManagerName, null, properties );

Here is an example using HazelcastCachingProvider::propertiesByLocation helper method.

CachingProvider cachingProvider = Caching.getCachingProvider();

// Create Properties instance pointing to a Hazelcast config file
String configFile = "classpath://my-configs/scoped-hazelcast.xml";
Properties properties = HazelcastCachingProvider
  .propertiesByLocation( configFile );

URI cacheManagerName = new URI("my-cache-manager");
CacheManager cacheManager = cachingProvider
  .getCacheManager( cacheManagerName, null, properties );

The retrieved CacheManager is scoped to use the HazelcastInstance that was just created and was configured using the given XML configuration file.

Available protocols for config file URL include classpath:// to point to a classpath location, file:// to point to a filesystem location, http:// an https:// for remote web locations. In addition, everything that does not specify a protocol is recognized as a placeholder that can be configured using a system property.

String configFile = "my-placeholder";
Properties properties = HazelcastCachingProvider
  .propertiesByLocation( configFile );

Can be set on the command line by:

-Dmy-placeholder=classpath://my-configs/scoped-hazelcast.xml

NOTE: No check is performed to prevent creating multiple CacheManagers with the same cluster configuration on different configuration files. If the same cluster is referred from different configuration files, multiple cluster members or clients are created.

NOTE: The configuration file location will not be a part of the resulting identity of the CacheManager. An attempt to create a CacheManager with a different set of properties but an already used name will result in undefined behavior.

12.5.1.2 Named Instance Scope

A CacheManager can be bound to an existing and named HazelcastInstance instance. If the instanceName is specified in com.hazelcast.config.Config, it can be used directly by passing it to CachingProvider implementation. Otherwise (instanceName not set or instance is a client instance) instance name must be get over HazelcastInstance instance via String getName() method to pass the CachingProvider implementation. Please note that instanceName is not configurable for the client side HazelcastInstance instance and is auto-generated
by using group name (if it is specified). In general, `String getName()` method over `HazelcastInstance` is more safe and preferable way for getting name of instance. Multiple `CacheManagers` created using an equal `java.net.URI` will share the same `HazelcastInstance`.

A named scope is applied nearly the same way as the configuration scope: pass in the instance name using the `HazelcastCachingProvider#HAZELCAST_INSTANCE_NAME` (which resolves to `hazelcast.instance.name`) property as a mapping inside a `java.util.Properties` instance to the `CachingProvider#getCacheManager(uri, classLoader, properties)` call.

Here is an example of Named Instance Scope with specified name.

```java
Config config = new Config();
config.setInstanceName( "my-named-hazelcast-instance" );
// Create a named HazelcastInstance
Hazelcast.newHazelcastInstance( config );

CachingProvider cachingProvider = Caching.getCachingProvider();
// Create Properties instance pointing to a named HazelcastInstance
Properties properties = new Properties();
properties.setProperty( HazelcastCachingProvider.HAZELCAST_INSTANCE_NAME,
    "my-named-hazelcast-instance" );

URI cacheManagerName = new URI( "my-cache-manager" );
CacheManager cacheManager = cachingProvider
    .getCacheManager( cacheManagerName, null, properties );

Here is an example of Named Instance Scope with auto-generated name.

```java
Config config = new Config();
// Create a auto-generated named HazelcastInstance
HazelcastInstance instance = Hazelcast.newHazelcastInstance( config );
String instanceName = instance.getName();

CachingProvider cachingProvider = Caching.getCachingProvider();
// Create Properties instance pointing to a named HazelcastInstance
Properties properties = new Properties();
properties.setProperty( HazelcastCachingProvider.HAZELCAST_INSTANCE_NAME,
    instanceName );

URI cacheManagerName = new URI( "my-cache-manager" );
CacheManager cacheManager = cachingProvider
    .getCacheManager( cacheManagerName, null, properties );
```

Here is an example of Named Instance Scope with auto-generated name on client instance.

```java
ClientConfig clientConfig = new ClientConfig();
ClientNetworkConfig networkConfig = clientConfig.getNetworkConfig();
networkConfig.addAddress("127.0.0.1", "127.0.0.2");
// Create a client side HazelcastInstance
HazelcastInstance instance = HazelcastClient.newHazelcastClient( clientConfig );
String instanceName = instance.getName();

CachingProvider cachingProvider = Caching.getCachingProvider();
// Create Properties instance pointing to a named HazelcastInstance
```
Properties properties = new Properties();
properties.setProperty( HazelcastCachingProvider.HAZELCAST_INSTANCE_NAME, instanceName );

URI cacheManagerName = new URI( "my-cache-manager" );
CacheManager cacheManager = cachingProvider .getCacheManager( cacheManagerName, null, properties );

Here is an example using HazelcastCachingProvider::propertiesByInstanceName method.

Config config = new Config();
config.setInstanceName( "my-named-hazelcast-instance" );
// Create a named HazelcastInstance
Hazelcast.newHazelcastInstance( config );

CachingProvider cachingProvider = Caching.getCachingProvider();
// Create Properties instance pointing to a named HazelcastInstance
Properties properties = HazelcastCachingProvider .propertiesByInstanceName( "my-named-hazelcast-instance" );

URI cacheManagerName = new URI( "my-cache-manager" );
CacheManager cacheManager = cachingProvider .getCacheManager( cacheManager, null, properties );

NOTE: The instanceName will not be a part of the resulting identity of the CacheManager. An attempt to create a CacheManager with a different set of properties but an already used name will result in undefined behavior.

12.5.1.3 Namespaces

The java.net.URI s that don’t use the above mentioned Hazelcast specific schemes are recognized as namespacing. Those CacheManagers share the same underlying default HazelcastInstance created (or set) by the CachingProvider, but they cache with the same names but differently namespaces on CacheManager level, and therefore won’t share the same data. This is useful where multiple applications might share the same Hazelcast JCache implementation (e.g. on application or OSGI servers) but are developed by independent teams. To prevent interfering on caches using the same name, every application can use its own namespace when retrieving the CacheManager.

Here is an example of using namespaces.

CachingProvider cachingProvider = Caching.getCachingProvider();

URI nsApp1 = new URI( "application-1" );
CacheManager cacheManagerApp1 = cachingProvider .getCacheManager( nsApp1, null );

URI nsApp2 = new URI( "application-2" );
CacheManager cacheManagerApp2 = cachingProvider .getCacheManager( nsApp2, null );

That way both applications share the same HazelcastInstance instance but not the same caches.

12.5.2 Retrieving an ICache Instance

Besides Scopes and Namespaces, which are implemented using the URI feature of the specification, all other extended operations are required to retrieve the com.hazelcast.cache.ICache interface instance from the JCache javax.cache.Cache instance. For Hazelcast, both interfaces are implemented on the same object instance. It
is recommended that you stay with the specification way to retrieve the ICache version, since ICache might be subject to change without notification.

To retrieve or unwrap the ICache instance, you can execute the following code snippet:

```java
CachingProvider cachingProvider = Caching.getCachingProvider();
CacheManager cacheManager = cachingProvider.getCacheManager();
Cache<Object, Object> cache = cacheManager.getCache(...);

ICache<Object, Object> unwrappedCache = cache.unwrap( ICache.class );
```

After unwrapping the Cache instance into an ICache instance, you have access to all of the following operations, e.g. Async Operations and Additional Methods.

### 12.5.3 ICache Configuration

As mentioned in the JCache Declarative Configuration section, the Hazelcast ICache extension offers additional configuration properties over the default JCache configuration. These additional properties include internal storage format, backup counts and eviction policy.

The declarative configuration for ICache is a superset of the previously discussed JCache configuration:

```xml
<cache>
  <!-- ... default cache configuration goes here ... -->
  <backup-count>1</backup-count>
  <async-backup-count>1</async-backup-count>
  <in-memory-format>BINARY</in-memory-format>
  <eviction size="10000" max-size-policy="ENTRY_COUNT" eviction-policy="LRU" />
</cache>
```

- **backup-count**: The number of synchronous backups. Those backups are executed before the mutating cache operation is finished. The mutating operation is blocked. `backup-count` default value is `1`.
- **async-backup-count**: The number of asynchronous backups. Those backups are executed asynchronously so the mutating operation is not blocked and it will be done immediately. `async-backup-count` default value is `0`.
- **in-memory-format**: Defines the internal storage format. For more information, please see the In Memory Format section. Default is `BINARY`.
- **eviction**: Defines the used eviction strategies and sizes for the cache. For more information on eviction, please see the JCache Eviction.
  - **size**: The maximum number of records or maximum size in bytes depending on the `max-size-policy` property. Size can be any integer between `0` and `Integer.MAX_VALUE`. Default max-size-policy is `ENTRY_COUNT` and default size is `10.000`.
  - **max-size-policy**: The size policy property defines a maximum size. If maximum size is reached, the cache is evicted based on the eviction policy. Default max-size-policy is `ENTRY_COUNT` and default size is `10.000`. The following eviction policies are available:
    - `ENTRY_COUNT`: Maximum number of cache entries in the cache. Available on heap based cache record store only.
    - `USED_NATIVE_MEMORY_SIZE`: Maximum used native memory size in megabytes for each instance. Available on High-Density Memory cache record store only.
    - `USED_NATIVE_MEMORY_PERCENTAGE`: Maximum used native memory size percentage for each instance. Available on High-Density Memory cache record store only.
    - `FREE_NATIVE_MEMORY_SIZE`: Maximum free native memory size in megabytes for each instance. Available on High-Density Memory cache record store only.
    - `FREE_NATIVE_MEMORY_PERCENTAGE`: Maximum free native memory size percentage for each instance. Available on High-Density Memory cache record store only.
# Chapter 12. Hazelcast JCACHE

- **eviction-policy**: The defined eviction policy to compare values with to find the best matching eviction candidate. Default is **LRU**.
  - *LRU*: Less Recently Used - finds the best eviction candidate based on the lastAccessTime.
  - *LFU*: Less Frequently Used - finds the best eviction candidate based on the number of hits.

Since `javax.cache.configuration.MutableConfiguration` misses the above additional configuration properties, Hazelcast ICache extension provides an extended configuration class called `com.hazelcast.config.CacheConfig`. This class is an implementation of `javax.cache.configuration.CompleteConfiguration` and all the properties shown above can be configured using its corresponding setter methods.

**NOTE:** At the client side, ICache can be configured only programmatically.

## 12.5.4 Async Operations

As another addition of Hazelcast ICache over the normal JCache specification, Hazelcast provides asynchronous versions of almost all methods, returning a `com.hazelcast.core.ICompletableFuture`. By using these methods and the returned future objects, you can use JCache in a reactive way by registering zero or more callbacks on the future to prevent blocking the current thread.

Name of the asynchronous versions of the methods append the phrase **Async** to the method name. Sample code is shown below using the method `putAsync()`.

```java
ICache<Integer, String> unwrappedCache = cache.unwrap( ICache.class );
ICompletableFuture<String> future = unwrappedCache.putAsync( 1, "value" );
future.thenAccept( new ExecutionCallback<String>() {
    public void onResponse( String response ) {
        System.out.println( "Previous value: " + response );
    }
    public void onFailure( Throwable t ) {
        t.printStackTrace();
    }
});
```

Following methods are available in asynchronous versions:

- **get(key):**
  - `getAsync(key)`
  - `getAsync(key, expiryPolicy)`
- **put(key, value):**
  - `putAsync(key, value)`
  - `putAsync(key, value, expiryPolicy)`
- **putIfAbsent(key, value):**
  - `putIfAbsentAsync(key, value)`
  - `putIfAbsentAsync(key, value, expiryPolicy)`
- **getAndPut(key, value):**
  - `getAndPutAsync(key, value)`
  - `getAndPutAsync(key, value, expiryPolicy)`
- **remove(key):**
  - `removeAsync(key)`
  - `remove(key, value):`
12.5. HAZELCAST JCACHE EXTENSION - ICACHE

- removeAsync(key, value)
- getAndRemove(key):
  - getAndRemoveAsync(key)
- replace(key, value):
  - replaceAsync(key, value)
  - replaceAsync(key, value, expiryPolicy)
- replace(key, oldValue, newValue):
  - replaceAsync(key, oldValue, newValue)
  - replaceAsync(key, oldValue, newValue, expiryPolicy)
- getAndReplace(key, value):
  - getAndReplaceAsync(key, value)
  - getAndReplaceAsync(key, value, expiryPolicy)

The methods with a given `javax.cache.expiry.ExpiryPolicy` are further discussed in the Custom ExpiryPolicy section.

**NOTE:** Asynchronous versions of the methods are not compatible with synchronous events.

12.5.5 Custom ExpiryPolicy

The JCache specification has an option to configure a single `ExpiryPolicy` per cache. Hazelcast ICache extension offers the possibility to define a custom `ExpiryPolicy` per key by providing a set of method overloads with an `expirePolicy` parameter, as in the list of asynchronous methods in the Async Methods section. This means that custom expiry policies can passed to a cache operation.

Here is how an `ExpirePolicy` is set on JCache configuration:

```java
CompleteConfiguration<String, String> config =
    new MutableConfiguration<String, String>()
    .setExpiryPolicyFactory(
        AccessedExpiryPolicy.factoryOf(Duration.ONE_MINUTE)
    );
```

To pass a custom `ExpirePolicy`, a set of overloads is provided and can be used as shown in the following code snippet:

```java
ICache<Integer, String> unwrappedCache = cache.unwrap(ICache.class);
unwrappedCache.put(1, "value", new AccessedExpiryPolicy(Duration.ONE_DAY));
```

The `ExpirePolicy` instance can be pre-created, cached, and re-used, but only for each cache instance. This is because `ExpirePolicy` implementations can be marked as `java.io.Closeable`. The following list shows the provided method overloads over `javax.cache.Cache` by `com.hazelcast.cache.ICache` featuring the `ExpiryPolicy` parameter:

- get(key):
  - get(key, expiryPolicy)
- getAll(keys):
  - getAll(keys, expiryPolicy)
- put(key, value):
- put(key, value, expirePolicy)

• getAndPut(key, value):
  - getAndPut(key, value, expirePolicy)

• putAll(map):
  - putAll(map, expirePolicy)

• putIfAbsent(key, value):
  - putIfAbsent(key, value, expirePolicy)

• replace(key, value):
  - replace(key, value, expirePolicy)

• replace(key, oldValue, newValue):
  - replace(key, oldValue, newValue, expirePolicy)

• getAndReplace(key, value):
  - getAndReplace(key, value, expirePolicy)

Asynchronous method overloads are not listed here. Please see the Async Operations section for the list of asynchronous method overloads.

12.5.6 JCache Eviction

Growing to an infinite size is in general not the expected behavior of caches. Implementing an expiry policy is one way to prevent the infinite growth but sometimes it is hard to define a meaningful expiration timeout. Therefore, Hazelcast JCache provides the eviction feature. Eviction offers the possibility to remove entries based on the cache size or amount of used memory (Hazelcast Enterprise Only) and not based on timeouts.

12.5.6.1 General Information

Since a cache is designed for high throughput and fast reads, a lot of effort went into designing the eviction system as predictable as possible. All built-in implementations provide an amortized O(1) runtime. The default operation runtime is rendered as O(1) but can be faster than the normal runtime cost if the algorithm finds an expired entry while sampling.

Most importantly, in typical production system two common types of caches are found:

- **Reference Caches**: Caches for reference data are normally small and are used to speed up the de-referencing as a lookup table. Those caches are commonly tend to be small and contain a previously known, fixed number of elements (e.g. states of the USA or abbreviations of elements).

- **Active DataSet Caches**: The other type of caches normally caches an active data set. These caches run to their maximum size and evict the oldest or not frequently used entries to keep in memory bounds. They sit in front of a database or HTML generators to cache the latest requested data.

Hazelcast JCache eviction supports both types of caches using a slightly different approach based on the configured maximum size of the cache. For detailed information, please see the Eviction Algorithm section.
12.5.6.2 Eviction Policies

Hazelcast JCache provides two commonly known eviction policies, LRU and LFU, but loosens the rules for predictable runtime behavior. LRU, normally recognized as Least Recently Used, is implemented as Less Recently Used, and LFU known as Least Frequently Used is implemented as Less Frequently Used. The details about this difference is explained in the Eviction Algorithm section.

Eviction Policies are configured by providing the corresponding abbreviation to the configuration as shown in the ICache Configuration section. As already mentioned, two built-in policies are available:

To configure the use of the LRU (Less Recently Used) policy:

```xml
<eviction size="10000" max-size-policy="ENTRY_COUNT" eviction-policy="LRU" />
```

And to configure the use of the LFU (Less Frequently Used) policy:

```xml
<eviction size="10000" max-size-policy="ENTRY_COUNT" eviction-policy="LFU" />
```

The default eviction policy is LRU. Therefore, Hazelcast JCache does not offer the possibility to perform no eviction.

12.5.6.3 Eviction Strategy

Eviction strategies implement the logic of selecting one or more eviction candidates from the underlying storage implementation and passing them to the eviction policies. Hazelcast JCache provides an amortized O(1) cost implementation for this strategy to select a fixed number of samples from the current partition that it is executed against.

The default implementation is `com.hazelcast.cache.impl.eviction.impl.strategy.sampling.SamplingBasedEvictionStrategy` which, as mentioned, samples random 15 elements. A detailed description of the algorithm will be explained in the next section.

12.5.6.4 Eviction Algorithm

The Hazelcast JCache eviction algorithm is specially designed for the use case of high performance caches and with predictability in mind. The built-in implementations provide an amortized O(1) runtime and therefore provide a highly predictable runtime behavior which does not rely on any kind of background threads to handle the eviction. Therefore, the algorithm takes some assumptions into account to prevent network operations and concurrent accesses.

As an explanation of how the algorithm works, let’s examine the following flowchart step by step.

1. A new cache is created. Without any special settings, the eviction is configured to kick in when the cache exceeds 10,000 elements and an LRU (Less Recently Used) policy is set up.
2. The user puts in a new entry (e.g. a key-value pair).
3. For every put, the eviction strategy evaluates the current cache size and decides if an eviction is necessary or not. If not the entry is stored in step 10.
4. If eviction is required, a new sampling is started. The built-in sampler is implemented as a lazy iterator.
5. The sampling algorithm selects a random sample from the underlying data storage.
6. The eviction strategy tests the sampled entry to already be expired (lazy expiration). If expired, the sampling stops and the entry is removed in step 9.
7. If not yet expired, the entry (eviction candidate) is compared to the last best matching candidate (based on the eviction policy) and the new best matching candidate is remembered.
8. The sampling is repeated for 15 times and then the best matching eviction candidate is returned to the eviction strategy.
9. The expired or best matching eviction candidate is removed from the underlying data storage.
10. The new put entry is stored.
11. The put operation returns to the user.
As seen by the flowchart, the general eviction operation is easy. As long as the cache does not reach its maximum capacity or you execute updates (put/replace), no eviction is executed.

To prevent network operations and concurrent access, as mentioned earlier, the cache size is estimated based on the size of the currently handled partition. Due to the imbalanced partitions, the single partitions might start to evict earlier than the other partitions.

As mentioned in the General Information section, typically two types of caches are found in the production systems. For small caches, referred to as Reference Caches, the eviction algorithm has a special set of rules depending on the maximum configured cache size. Please see the Reference Caches section for details. The other type of cache is referred to as Active DataSet Cache, which in most cases makes heavy use of the eviction to keep the most active data set in the memory. Those kinds of caches using a very simple but efficient way to estimate the cluster-wide cache size.

All of the following calculations have a well known set of fixed variables:

- **GlobalCapacity**: The user defined maximum cache size (cluster-wide).
- **PartitionCount**: The number of partitions in the cluster (defaults to 271).
- **BalancedPartitionSize**: The number of elements in a balanced partition state, \( \text{BalancedPartitionSize} := \frac{\text{GlobalCapacity}}{\text{PartitionCount}} \).
- **Deviation**: An approximated standard deviation (tests proofed it to be pretty near), \( \text{Deviation} := \sqrt{\text{BalancedPartitionSize}} \).

### 12.5.6.4.1 Reference Caches

A Reference Cache is typically small and the number of elements to store in the reference caches is normally known prior to creating the cache. Typical examples of reference caches are lookup tables for abbreviations or the states of a country. They tend to have a fixed but small element number and the eviction is an unlikely event and rather undesirable behavior.

Since an imbalanced partition is the worst problem in the small and mid-sized caches than for the caches with millions of entries, the normal estimation rule (as discussed in a bit) is not applied to these kinds of caches. To
12.5. HAZELCAST JCACHE EXTENSION - ICACHE

prevent unwanted eviction on the small and mid-sized caches, Hazelcast implements a special set of rules to estimate the cluster size.

To adjust the imbalance of partitions as found in the typical runtime, the actual calculated maximum cache size (as known as the eviction threshold) is slightly higher than the user defined size. That means more elements can be stored into the cache than expected by the user. This needs to be taken into account especially for large objects, since those can easily exceed the expected memory consumption!

**Small caches:**

If a cache is configured with no more than 4,000 element, this cache is considered to be a small cache. The actual partition size is derived from the number of elements (GlobalCapacity) and the deviation using the following formula:

\[
\text{MaxPartitionSize} := \text{Deviation} \times 5 + \text{BalancedPartitionSize}
\]

This formula ends up with big partition sizes which summed up exceed the expected maximum cache size (set by the user), but since the small caches typically have a well known maximum number of elements, this is not a big issue. Only if the small caches are used for a use case other than using it as a reference cache, this needs to be taken into account.

**Mid-sized caches**

A mid-sized cache is defined as a cache with a maximum number of elements that is bigger than 4,000 but not bigger than 1,000,000 elements. The calculation of mid-sized caches is similar to that of the small caches but with a different multiplier. To calculate the maximum number of elements per partition, the following formula is used:

\[
\text{MaxPartitionSize} := \text{Deviation} \times 3 + \text{BalancedPartitionSize}
\]

12.5.6.4.2 Active DataSet Caches  For large caches, where the maximum cache size is bigger than 1,000,000 elements, there is no additional calculation needed. The maximum partition size is considered to be equal to BalancedPartitionSize since statistically big partitions are expected to almost balance themselves. Therefore, the formula is as easy as the following:

\[
\text{MaxPartitionSize} := \text{BalancedPartitionSize}
\]

12.5.6.4.3 Cache Size Estimation  As mentioned earlier, Hazelcast JCache provides an estimation algorithm to prevent cluster-wide network operations, concurrent access to other partitions and background tasks. It also offers a highly predictable operation runtime when the eviction is necessary.

The estimation algorithm is based on the previously calculated maximum partition size (please see the Reference Caches section and Active DataSet Caches section) and is calculated against the current partition only.

The algorithm to reckon the number of stored entries in the cache (cluster-wide) and if the eviction is necessary is shown in the following pseudo-code example:

\[
\text{RequiresEviction[Boolean]} := \text{CurrentPartitionSize} \geq \text{MaxPartitionSize}
\]

12.5.7 JCache Near Cache

Cache entries in Hazelcast are stored as partitioned across the cluster. When you try to read a record with the key k, if the current node is not the owner of that key (i.e. not the owner of partition that the key belongs to), Hazelcast sends a remote operation to the owner node. Each remote operation means lots of network trips. If your cache is used for mostly read operations, it is advised to use a near cache storage in front of the cache itself to read cache records faster and consume less network traffic. **NOTE:** Near cache for JCache is only available for clients NOT servers.

However, using near cache comes with some trade-off for some cases:
CHAPTER 12. HAZELCAST JCACHE

- There will be extra memory consumption for storing near cache records at local.
- If invalidation is enabled and entries are updated frequently, there will be many invalidation events across the cluster.
- Near cache does not give strong consistency but gives eventual consistency guarantees. It is possible to read stale data.

12.5.7.1 JCache Near Cache Invalidation

Invalidation is the process of removing an entry from the near cache since the entry is not valid anymore (its value is updated or it is removed from actual cache). Near cache invalidation happens asynchronously at the cluster level, but synchronously in real-time at the current node. This means when an entry is updated (explicitly or via entry processor) or removed (deleted explicitly or via entry processor, evicted, expired), it is invalidated from all near caches asynchronously within the whole cluster but updated/removed at/from the current node synchronously. Generally, whenever the state of an entry changes in the record store by updating its value or removing it, the invalidation event is sent for that entry.

Invalidation events can be sent either individually or in batches. If there are lots of mutating operations such as put/remove on the cache, sending the events in batches is advised. This reduces the network traffic and keeps the eventing system less busy.

You can use the following system properties to configure the sending of invalidation events in batches:

- `hazelcast.cache.invalidation.batch.enabled`: Specifies whether the cache invalidation event batch sending is enabled or not. The default value is `true`.
- `hazelcast.cache.invalidation.batch.size`: Defines the maximum number of cache invalidation events to be drained and sent to the event listeners in a batch. The default value is `100`.
- `hazelcast.cache.invalidation.batchfrequency.seconds`: Defines cache invalidation event batch sending frequency in seconds. When event size does not reach to `hazelcast.cache.invalidation.batch.size` in the given time period, those events are gathered into a batch and sent to the target. The default value is `5` seconds.

So if there are so many clients or so many mutating operations, batching should remain enabled and the batch size should be configured with the `hazelcast.cache.invalidation.batch.size` system property to a suitable value.

12.5.7.2 JCache Near Cache Expiration

Expiration means the eviction of expired records. A record is expired: - if it is not touched (accessed/read) for `<max-idle-seconds>`, - `<time-to-live-seconds>` passed since it is put to near-cache.

Expiration is performed in two cases:

- When a record is accessed, it is checked about if it is expired or not. If it is expired, it is evicted and returns `null` to caller.
- In the background, there is an expiration task that periodically (currently 5 seconds) scans records and evicts the expired records.

12.5.7.3 JCache Near Cache Eviction

In the scope of near cache, eviction means evicting (clearing) the entries selected according to the given `eviction-policy` when the specified `max-size-policy` has been reached. Eviction is handled with `max-size policy` and `eviction-policy` elements. Please see the JCache Near Cache Configuration section.
12.5.7.3.1 max-size-policy  This element defines the state when near cache is full and whether the eviction should be triggered. The following policies for maximum cache size are supported by the near cache eviction:

- **ENTRY_COUNT**: Maximum size based on the entry count in the near cache. Available only for **BINARY** and **OBJECT** in-memory formats.
- **USED_NATIVE_MEMORY_SIZE**: Maximum used native memory size of the specified near cache in MB to trigger the eviction. If the used native memory size exceeds this threshold, the eviction is triggered. Available only for **NATIVE** in-memory format. This is supported only by Hazelcast Enterprise.
- **USED_NATIVE_MEMORY_PERCENTAGE**: Maximum used native memory percentage of the specified near cache to trigger the eviction. If the native memory usage percentage (relative to maximum native memory size) exceeds this threshold, the eviction is triggered. Available only for **NATIVE** in-memory format. This is supported only by Hazelcast Enterprise.
- **FREE_NATIVE_MEMORY_SIZE**: Minimum free native memory size of the specified near cache in MB to trigger the eviction. If free native memory size goes down below this threshold, eviction is triggered. Available only for **NATIVE** in-memory format. This is supported only by Hazelcast Enterprise.
- **FREE_NATIVE_MEMORY_PERCENTAGE**: Minimum free native memory percentage of the specified near cache to trigger eviction. If free native memory percentage (relative to maximum native memory size) goes down below this threshold, eviction is triggered. Available only for **NATIVE** in-memory format. This is supported only by Hazelcast Enterprise.

12.5.7.3.2 eviction-policy  Once a near cache is full (reached to its maximum size as specified with the max-size-policy element), an eviction policy determines which, if any, entries must be evicted. Currently, the following eviction policies are supported by near cache eviction:

- LRU (Least Recently Used)
- LFU (Least Frequently Used)

12.5.7.4 JCache Near Cache Configuration

The following are the example configurations.

Declarative:

```xml
<hazelcast-client>
  ...
  <near-cache name="myCache">
    <in-memory-format>BINARY</in-memory-format>
    <invalidate-on-change>true</invalidate-on-change>
    <cache-local-entries>false</cache-local-entries>
    <time-to-live-seconds>3600000</time-to-live-seconds>
    <max-idle-seconds>600000</max-idle-seconds>
    <eviction size="1000" max-size-policy="ENTRY_COUNT" eviction-policy="LFU"/>
  </near-cache>
  ...
</hazelcast-client>
```

Programmatic:

```java
EvictionConfig evictionConfig = new EvictionConfig();
evictionConfig.setMaxSizePolicy(MaxSizePolicy.ENTRY_COUNT);
evictionConfig.setEvictionPolicy(EvictionPolicy.LFU);
evictionConfig.setSize(10000);

NearCacheConfig nearCacheConfig = new NearCacheConfig()
  .setName("myCache")
```
.setInMemoryFormat(InMemoryFormat.BINARY)
.setInvalidateOnChange(true)
.setCacheLocalEntries(false)
.setTimeToLiveSeconds(60 * 60 * 1000) // 1 hour TTL
.setMaxIdleSeconds(10 * 60 * 1000) // 10 minutes max idle seconds
.setEvictionConfig(evictionConfig);
...

clientConfig.addNearCacheConfig(nearCacheConfig);

The following are the definitions of the configuration elements and attributes.

- **in-memory-format**: Storage type of near cache entries. Available values are BINARY, OBJECT and NATIVE_MEMORY. NATIVE_MEMORY is available only for Hazelcast Enterprise. Default value is BINARY.
- **invalidate-on-change**: Specifies whether the cached entries are evicted when the entries are changed (updated or removed) on the local and global. Available values are true and false. Default value is true.
- **cache-local-entries**: Specifies whether the local cache entries are stored eagerly (immediately) to near cache when a put operation from the local is performed on the cache. Available values are true and false. Default value is false.
- **time-to-live-seconds**: Maximum number of seconds for each entry to stay in the near cache. Entries that are older than <time-to-live-seconds> will be automatically evicted from the near cache. It can be any integer between 0 and Integer.MAX_VALUE. 0 means infinite. Default value is 0.
- **max-idle-seconds**: Maximum number of seconds each entry can stay in the near cache as untouched (not-read). Entries that are not read (touched) more than <max-idle-seconds> value will be removed from the near cache. It can be any integer between 0 and Integer.MAX_VALUE. 0 means Integer.MAX_VALUE. Default is 0.
- **eviction**: Specifies when the eviction is triggered (max-size policy) and which eviction policy (LRU or LFU) is used for the entries to be evicted. The default value for max-size-policy is ENTRY_COUNT, default size is 10000 and default eviction-policy is LRU. For High-Density Memory Store near cache, since ENTRY_COUNT eviction policy is not supported yet, eviction must be explicitly configured with one of the supported policies:
  - USED_NATIVE_MEMORY_SIZE
  - USED_NATIVE_MEMORY_PERCENTAGE
  - FREE_NATIVE_MEMORY_SIZE
  - FREE_NATIVE_MEMORY_PERCENTAGE.

Near cache can be configured only at the client side.

### 12.5.7.5 Notes About Client Near Cache Configuration

Near cache configuration can be defined at the client side (using hazelcast-client.xml or ClientConfig) as independent configuration (independent from the CacheConfig). Near cache configuration lookup is handled as described below:

- Look for near cache configuration with the name of the cache given in the client configuration.
- If a defined near cache configuration is found, use this near cache configuration defined at the client.
- Otherwise:
  - If there is a defined default near cache configuration is found, use this default near cache configuration.
  - If there is no default near cache configuration, it means there is no near cache configuration for cache.

### 12.5.8 Additional Methods

In addition to the operations explained in the Async Operations section and Custom ExpiryPolicy section, Hazelcast ICache also provides a set of convenience methods. These methods are not part of the JCache specification.
12.5. HAZELCAST JCACHE EXTENSION - ICACHE

- `size()`: Returns the estimated size of the distributed cache.
- `destroy()`: Destroys the cache and removes the data from memory. This is different from the method `javax.cache.Cache::close`.
- `getLocalCacheStatistics()`: Returns a `com.hazelcast.cache.CacheStatistics` instance providing the same statistics data as the JMX beans. This method is not available yet on Hazelcast clients: the exception `java.lang.UnsupportedOperationException` is thrown when you use this method on a Hazelcast client.

12.5.9 BackupAwareEntryProcessor

Another feature, especially interesting for distributed environments like Hazelcast, is the JCache specified `javax.cache.processor.EntryProcessor`. For more general information, please see the JCache EntryProcessor section.

Since Hazelcast provides backups of cached entries on other nodes, the default way to backup an object changed by an `EntryProcessor` is to serialize the complete object and send it to the backup partition. This can be a huge network overhead for big objects.

Hazelcast offers a sub-interface for `EntryProcessor` called `com.hazelcast.cache.BackupAwareEntryProcessor`. This allows the user to create or pass another `EntryProcessor` to run on backup partitions and apply delta changes to the backup entries.

The backup partition `EntryProcessor` can either be the currently running processor (by returning `this`) or it can be a specialized `EntryProcessor` implementation (other from the currently running one) which does different operations or leaves out operations, e.g. sending emails.

If we again take the `EntryProcessor` example from the demonstration application provided in the JCache EntryProcessor section, the changed code will look like the following snippet.

```java
public class UserUpdateEntryProcessor
    implements BackupAwareEntryProcessor<Integer, User, User> {

    @Override
    public User process( MutableEntry<Integer, User> entry, Object... arguments )
        throws EntryProcessorException {

        // Test arguments length
        if ( arguments.length < 1 ) {
            throw new EntryProcessorException( "One argument needed: username" );
        }

        // Get first argument and test for String type
        Object argument = arguments[0];
        if ( !( argument instanceof String ) ) {
            throw new EntryProcessorException( "First argument has wrong type, required java.lang.String" );
        }

        // Retrieve the value from the MutableEntry
        User user = entry.getValue();

        // Retrieve the new username from the first argument
        String newUsername = ( String ) arguments[0];

        // Set the new username
        user.setUsername( newUsername );

        // Set the changed user to mark the entry as dirty
        entry.setValue( user );
    }
}
```
// Return the changed user to return it to the caller
return user;
}

public EntryProcessor<K, V, T> createBackupEntryProcessor() {
    return this;
}
}

You can use the additional method BackupAwareEntryProcessor::createBackupEntryProcessor to create or return the EntryProcessor implementation to run on the backup partition (in the example above, the same processor again).

⚠️ **NOTE:** For the backup runs, the returned value from the backup processor is ignored and not returned to the user.

## 12.6 JCache Specification Compliance

Hazelcast JCache is fully compliant with the JSR 107 TCK (Technology Compatibility Kit), therefore it is officially a JCache implementation. This is tested by running the TCK against the Hazelcast implementation.

You can test Hazelcast JCache for compliance by executing the TCK. Just perform the instructions below:

2. Change the properties in tck-parent/pom.xml as shown below.
3. Run the TCK by `mvn clean install`.

```xml
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <CacheInvocationContextImpl>
        javax.cache.annotation.impl.cdi.CdiCacheKeyInvocationContextImpl
    </CacheInvocationContextImpl>
    <domain-lib-dir>${project.build.directory}/domainlib</domain-lib-dir>
    <domain-jar>domain.jar</domain-jar>
</properties>

<!-- Change the following properties on the command line to override with the coordinates for your implementation-->
<implementation-groupId>com.hazelcast</implementation-groupId>
<implementation-artifactId>hazelcast</implementation-artifactId>
<implementation-version>3.4</implementation-version>

<!-- Change the following properties to your CacheManager and Cache implementation. Used by the unwrap tests. -->
<CacheManagerImpl>
    com.hazelcast.client.cache.impl.HazelcastClientCacheManager
</CacheManagerImpl>
<CacheImpl>com.hazelcast.cache.ICache</CacheImpl>
<CacheEntryImpl>
    com.hazelcast.cache.impl.CacheEntry
</CacheEntryImpl>
```
This will run the tests using an embedded Hazelcast Member.
Chapter 13

Integrated Clustering

In this chapter, we show you how Hazelcast is integrated with Hibernate 2nd level cache and Spring, and how Hazelcast helps with your Filter, Tomcat and Jetty based web session replications.

The Hibernate Second Level Cache section tells how you should configure Hazelcast and Hibernate to integrate them. It explains the modes of Hazelcast that can be used by Hibernate and also provides how to perform advanced settings like accessing the underlying HazelcastInstance used by Hibernate.

The Web Session Replication section provides information on how to cluster user HTTP sessions automatically. Also, you can learn how to enable session replication for JEE web applications with Tomcat and Jetty containers. Please note that Tomcat and Jetty based web session replications are Hazelcast Enterprise only modules.

The Spring Integration section tells how you can integrate Hazelcast into a Spring project by explaining the Hazelcast instance and client configurations with the `hazelcast` namespace. It also lists the supported Spring bean attributes.

13.1 Hibernate Second Level Cache

Hazelcast provides distributed second level cache for your Hibernate entities, collections and queries.

13.1.1 Sample Code for Hibernate

Please see our sample application for Hibernate Second Level Cache.

13.1.2 Supported Hibernate Versions

- hibernate 3.3.x+
- hibernate 4.x

13.1.3 Hibernate Configuration

To configure for Hibernate, add `hazelcast-hibernate3-<hazelcastversion>.jar` or `hazelcast-hibernate4-<hazelcastversion>.jar` into your classpath depending on your Hibernate version.

Then add the following properties into your Hibernate configuration file (e.g. `hibernate.cfg.xml`).

**Enabling the use of second level cache**

```xml
<property name="hibernate.cache.use_second_level_cache">true</property>
```

Hazelcast RegionFactory

203
• **HazelcastCacheRegionFactory**

HazelcastCacheRegionFactory uses Hazelcast Distributed Map to cache the data, so all cache operations go through the wire.

```xml
<property name="hibernate.cache.region.factory_class">
    com.hazelcast.hibernate.HazelcastCacheRegionFactory
</property>
```

• **HazelcastLocalCacheRegionFactory**

You can use HazelcastLocalCacheRegionFactory which stores data in a local node and sends invalidation messages when an entry is updated/deleted locally.

```xml
<property name="hibernate.cache.region.factory_class">
    com.hazelcast.hibernate.HazelcastLocalCacheRegionFactory
</property>
```

**Optional Settings**

• To enable use of query cache:

```xml
<property name="hibernate.cache.use_query_cache">true</property>
```

• To force minimal puts into query cache:

```xml
<property name="hibernate.cache.use_minimal_puts">true</property>
```

**NOTE:** QueryCache is always LOCAL to the node and never distributed across Hazelcast Cluster.

### 13.1.4 Hazelcast Configuration for Hibernate

• To configure Hazelcast for Hibernate, put the configuration file named hazelcast.xml into the root of your classpath. If Hazelcast cannot find hazelcast.xml, then it will use the default configuration from hazelcast.jar.

• You can define a custom-named Hazelcast configuration XML file with one of these Hibernate configuration properties.

```xml
<property name="hibernate.cache.provider_configuration_file_resource_path">
    hazelcast-custom-config.xml
</property>
```

```xml
<property name="hibernate.cache.hazelcast.configuration_file_path">
    hazelcast-custom-config.xml
</property>
```

Hazelcast creates a separate distributed map for each Hibernate cache region. You can easily configure these regions via Hazelcast map configuration. You can define backup, eviction, TTL and Near Cache properties.

• Backup Configuration

• Eviction And TTL Configuration

• Near Cache Configuration
13.1.5 RegionFactory Options

13.1.5.0.1 HazelcastCacheRegionFactory HazelcastCacheRegionFactory uses standard Hazelcast Distributed Maps. All operations like get, put, and remove will be performed using the Distributed Map logic. The only downside of using HazelcastCacheRegionFactory may be the lower performance compared to HazelcastLocalCacheRegionFactory since operations are handled as distributed calls.

**NOTE:** If you use HazelcastCacheRegionFactory, you can see your maps on Management Center.

With HazelcastCacheRegionFactory, all of the following caches are distributed across Hazelcast Cluster.

- Entity Cache
- Collection Cache
- Timestamp Cache

13.1.5.0.2 HazelcastLocalCacheRegionFactory With HazelcastLocalCacheRegionFactory, each cluster member has a local map and each of them is registered to a Hazelcast Topic (ITopic). Whenever a put or remove operation is performed on a member, an invalidation message is generated on the ITopic and sent to the other members. Those other members remove the related key-value pair on their local maps as soon as they get these invalidation messages. The new value is only updated on this member when a get operation runs on that key. In the case of get operations, invalidation messages are not generated and reads are performed on the local map.

An illustration of the above logic is shown below.

If your operations are mostly reads, then this option gives better performance.

**NOTE:** If you use HazelcastLocalCacheRegionFactory, you cannot see your maps on Management Center.

With HazelcastLocalCacheRegionFactory, all of the following caches are not distributed and are kept locally in the Hazelcast Node.

- Entity Cache
• Collection Cache
• Timestamp Cache

Entity and Collection are invalidated on update. When they are updated on a node, an invalidation message is sent to all other nodes in order to remove the entity from their local cache. When needed, each node reads that data from the underlying DB.

Timestamp cache is replicated. On every update, a replication message is sent to all the other nodes.

Eviction support is limited to maximum size of the map (defined by max-size configuration element) and TTL only. When maximum size is hit, 20% of the entries will be evicted automatically.

### 13.1.6 Hazelcast Modes for Hibernate Usage

Hibernate 2nd Level Cache can use Hazelcast in two modes: Peer-to-Peer and Client/Server.

#### 13.1.6.1 P2P (Peer-to-Peer)

With P2P mode, each Hibernate deployment launches its own Hazelcast Instance. You can also configure Hibernate to use an existing instance, so instead of creating a new HazelcastInstance for each SessionFactory, you can use an existing instance by setting the hibernate.cache.hazelcast.instance_name Hibernate property to the HazelcastInstance’s name. For more information, please see the Named HazelcastInstance section.

**Disabling shutdown during SessionFactory.close()**

Shutting down HazelcastInstance can be disabled during SessionFactory.close(). To achieve this set the Hibernate property hibernate.cache.hazelcast.shutdown_on_session_factory_close to false. *(In this case Hazelcast property hazelcast.shutdownhook.enabled should not be set to false.)* Default value is true.

#### 13.1.6.2 Client/Server

- You can set up Hazelcast to connect to the cluster as Native Client. Native client is not a member; it connects to one of the cluster members and delegates all cluster wide operations to it. When the relied cluster member dies, client will transparently switch to another live member.

```xml
<property name="hibernate.cache.hazelcast.use_native_client">true</property>
```

To set up Native Client, add the Hazelcast group-name, group-password and cluster member address properties. Native Client will connect to the defined member and will get the addresses of all members in the cluster. If the connected member dies or leaves the cluster, the client will automatically switch to another member in the cluster.

```xml
<property name="hibernate.cache.hazelcast.native_client_address">10.34.22.15</property>
<property name="hibernate.cache.hazelcast.native_client_group">dev</property>
<property name="hibernate.cache.hazelcast.native_client_password">dev-pass</property>
```

**NOTE:** To use Native Client, add hazelcast-client-<version>.jar into your classpath. Refer to Clients for more information.

### 13.1.7 Hibernate Concurrency Strategies

Hibernate has four cache concurrency strategies: read-only, read-write, nonstrict-read-write and transactional. Hibernate does not force cache providers to support all those strategies. Hazelcast supports the first three: read-only, read-write, and nonstrict-read-write. It has no support for transactional strategy yet.
13.2 WEB SESSION REPLICATION

- If you are using XML based class configurations, add a cache element into your configuration with the usage attribute set to one of the read-only, read-write, or nonstrict-read-write strategies.

```xml
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>

<class name="eg.Cat" >
    <cache usage="read-write"/>
    ....
    <set name="kittens" >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

- If you are using Hibernate-Annotations, then you can add a class-cache or collection-cache element into your Hibernate configuration file with the usage attribute set to read-only, read/write, or nonstrict read/write.

```xml
<class-cache usage="read-only" class="eg.Immutable"/>
<class-cache usage="read-write" class="eg.Cat"/>
<collection-cache collection="eg.Cat.kittens" usage="read-write"/>
```

- Or alternatively, you can put Hibernate Annotation’s @Cache annotation on your entities and collections.

```java
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Cat implements Serializable {
    ...
}
```

13.1.8 Advanced Settings

Accessing underlying HazelcastInstance

Using com.hazelcast.hibernate.instance.HazelcastAccessor, you can access the underlying HazelcastInstance used by Hibernate SessionFactory.

```java
SessionFactory sessionFactory = ...;
HazelcastInstance hazelcastInstance = HazelcastAccessor
    .getHazelcastInstance(sessionFactory);
```

Changing/setting lock timeout value of read-write strategy

You can set a lock timeout value using the hibernate.cache.hazelcast.lock_timeout_in_seconds Hibernate property. The value should be in seconds. The default value is 300 seconds.

13.2 Web Session Replication

If you are using Tomcat as your web container, please see the Tomcat based Web Session Replication section.
13.2.1 Filter Based Web Session Replication

**Sample Code:** Please see our sample application for Filter Based Web Session Replication.

Assume that you have more than one web server (A, B, C) with a load balancer in front of it. If server A goes down, your users on that server will be directed to one of the live servers (B or C), but their sessions will be lost.

We need to have all these sessions backed up somewhere if we do not want to lose the sessions upon server crashes. Hazelcast Web Manager (WM) allows you to cluster user HTTP sessions automatically. The following are required before enabling Hazelcast Session Clustering:

- Target application or web server should support Java 1.6 or higher.
- Target application or web server should support Servlet 3.0 or higher spec.
- Session objects that need to be clustered have to be Serializable.
- In the client/server architecture, session classes does not have to be present in the server classpath.

Here are the steps to setup Hazelcast Session Clustering:

- Put the `hazelcast` and `hazelcast-wm` jars in your `WEB-INF/lib` directory. Optionally, if you wish to connect to a cluster as a client, add `hazelcast-client` as well.
- Put the following XML into `web.xml` file. Make sure Hazelcast filter is placed before all the other filters if any; for example, you can put it at the top.

```xml
<filter>
  <filter-name>hazelcast-filter</filter-name>
  <filter-class>com.hazelcast.web.WebFilter</filter-class>

  <!--
  Name of the distributed map storing your web session objects
  -->
  <init-param>
    <param-name>map-name</param-name>
    <param-value>my-sessions</param-value>
  </init-param>

  <!--
  TTL value of the distributed map storing your web session objects.
  Any integer between 0 and Integer.MAX_VALUE.
  Default is 1800 which is 30 minutes.
  -->
  <init-param>
    <param-name>session-ttl-seconds</param-name>
    <param-value>10</param-value>
  </init-param>

  <!--
  How is your load-balancer configured?
  sticky-session means all requests of a session is routed to the node where the session is first created.
  This is excellent for performance.
  If sticky-session is set to false, when a session is updated on a node, entry for this session on all other nodes is invalidated.
  You have to know how your load-balancer is configured before setting this parameter. Default is true.
  -->
  <init-param>
    <param-name>sticky-session</param-name>
  </init-param>
</filter>
```
<param-value>true</param-value>
</init-param>
 <!--
 Name of session id cookie
 -->
<init-param>
  <param-name>cookie-name</param-name>
  <param-value>hazelcast.sessionId</param-value>
</init-param>
 <!--
 Domain of session id cookie. Default is based on incoming request.
 -->
<init-param>
  <param-name>cookie-domain</param-name>
  <param-value>.mywebsite.com</param-value>
</init-param>
 <!--
 Should cookie only be sent using a secure protocol? Default is false.
 -->
<init-param>
  <param-name>cookie-secure</param-name>
  <param-value>false</param-value>
</init-param>
 <!--
 Should HttpOnly attribute be set on cookie ? Default is false.
 -->
<init-param>
  <param-name>cookie-http-only</param-name>
  <param-value>false</param-value>
</init-param>
 <!--
 Are you debugging? Default is false.
 -->
<init-param>
  <param-name>debug</param-name>
  <param-value>true</param-value>
</init-param>
 <!--
 Configuration xml location;
 * as servlet resource OR
 * as classpath resource OR
 * as URL
 Default is one of hazelcast-default.xml
 or hazelcast.xml in classpath.
 -->
<init-param>
  <param-name>config-location</param-name>
  <param-value>/WEB-INF/hazelcast.xml</param-value>
</init-param>
 <!--
 Do you want to use an existing HazelcastInstance? Default is null.
 -->
<init-param>
  <param-name>instance-name</param-name>
  <param-value>default</param-value>
</init-param>
 <!--
Do you want to connect as a client to an existing cluster?
Default is false.

```xml
<init-param>
  <param-name>use-client</param-name>
  <param-value>false</param-value>
</init-param>

Client configuration location;
  * as servlet resource OR
  * as classpath resource OR
  * as URL
Default is null.

```xml
<init-param>
  <param-name>client-config-location</param-name>
  <param-value>/WEB-INF/hazelcast-client.properties</param-value>
</init-param>

Do you want to shutdown HazelcastInstance during
web application undeploy process?
Default is true.

```xml
<init-param>
  <param-name>shutdown-on-destroy</param-name>
  <param-value>true</param-value>
</init-param>

Do you want to cache sessions locally in each instance?
Default is false.

```xml
<init-param>
  <param-name>deferred-write</param-name>
  <param-value>false</param-value>
</init-param>

```

**Filter**

```xml
<filter>
  <filter-name>hazelcast-filter</filter-name>
  <url-pattern>/**</url-pattern>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>REQUEST</dispatcher>
</filter>

<listener>
  <listener-class>com.hazelcast.web.SessionListener</listener-class>
</listener>

• Package and deploy your war file as you would normally do.

It is that easy. All HTTP requests will go through Hazelcast WebFilter and it will put the session objects into Hazelcast distributed map if needed.

### 13.2.2 Spring Security Support

*Sample Code: Please see our sample application for Spring Security Support.*
If Spring-based security is used for your application, you should use `com.hazelcast.web.spring.SpringAwareWebFilter` instead of `com.hazelcast.web.WebFilter` in your filter definition.

```xml
<filter>
  <filter-name>hazelcast-filter</filter-name>
  <filter-class>com.hazelcast.web.spring.SpringAwareWebFilter</filter-class>
</filter>
```

`SpringAwareWebFilter` notifies Spring by publishing events to Spring context. These events are used by the `org.springframework.security.core.session.SessionRegistry` instance.

As before, you must also define `com.hazelcast.web.SessionListener` in your `web.xml`. However, you do not need to define `org.springframework.security.web.session.HttpSessionEventPublisher` in your `web.xml` as before, since `SpringAwareWebFilter` already informs Spring about session based events like `create` or `destroy`.

### 13.2.2.1 Client Mode vs. P2P Mode

Hazelcast Session Replication works as P2P by default. To switch to Client/Server architecture, you need to set the `use-client` parameter to `true`. P2P mode is more flexible and requires no configuration in advance; in Client/Server architecture, clients need to connect to an existing Hazelcast Cluster. In case of connection problems, clients will try to reconnect to the cluster. The default retry count is 3. In the client/server architecture, if servers go down, Hazelcast web manager will keep the updates in the local and after servers come back, the clients will update the distributed map.

Note that, in the client/server mode of session replication, `session-ttl-seconds` configuration does not have any effect. The reason is that the filter based session replication uses IMap and a Hazelcast client cannot change the configuration of a distributed map. Instead, you should configure the `max-idle-seconds` element in your `hazelcast.xml` on the server side.

```xml
... <map name="my-sessions"> <!-- How much seconds do you want your session attributes to be stored on server? Default is 0. --> <max-idle-seconds>20</max-idle-seconds> </map> ...
```

Also make sure that name of the distributed map is same as the `map-name` parameter defined in your `web.xml` configuration file.

### 13.2.2.2 Caching Locally with `deferred-write`

If the value for `deferred-write` is set as `true`, Hazelcast will cache the session locally and will update the local session when an attribute is set or deleted. At the end of the request, it will update the distributed map with all the updates. It will not update the distributed map upon each attribute update, but will only call it once at the end of the request. It will also cache it, i.e. whenever there is a read for the attribute, it will read it from the cache.

**Important note about `deferred-write=false` setting:**

If `deferred-write` is `false`, any update (i.e. `setAttribute`) on the session will directly be available in the cluster. One exception to this behavior is the changes to the session attribute objects. To update an attribute cluster-wide, `setAttribute` must be called after changes are made to the attribute object.

The following example explains how to update an attribute in the case of `deferred-write=false` setting:

```java
session.setAttribute("myKey", new ArrayList());
List list1 = session.getAttribute("myKey");
list1.add("myValue");
session.setAttribute("myKey", list1); // changes updated in the cluster
```
13.2.2.3 SessionId Generation

SessionId generation is done by the Hazelcast Web Session Module if session replication is configured in the web application. The default cookie name for the sessionId is `hazelcast.sessionId`. This name is configurable with a `cookie-name` parameter in the `web.xml` file of the application. `hazelcast.sessionId` is just a UUID prefixed with “HZ” character and without “-” character, e.g. HZ6F2D036789E4404893E99C05D8CA70C7.

When called by the target application, the value of `HttpSession.getId()` is the same as the value of `hazelcast.sessionId`.

13.2.2.4 Session Expiry

Hazelcast automatically removes sessions from the cluster if the sessions are expired on the Web Container. This removal is done by `com.hazelcast.web.SessionListener`, which is an implementation of `javax.servlet.http.HttpSessionListener`.

Default session expiration configuration depends on the Servlet Container that is being used. You can also define it in your `web.xml`.

```
<session-config>
  <session-timeout>60</session-timeout>
</session-config>
```

13.2.2.5 sticky-session

Hazelcast holds whole session attributes in a distributed map and in a local HTTP session. Local session is required for fast access to data and distributed map is needed for fail-safety.

- If `sticky-session` is not used, whenever a session attribute is updated in a node (in both node local session and clustered cache), that attribute should be invalidated in all other nodes’ local sessions, because now they have dirty values. Therefore, when a request arrives at one of those other nodes, that attribute value is fetched from clustered cache.

- To overcome the performance penalty of sending invalidation messages during updates, you can use sticky sessions. If Hazelcast knows sessions are sticky, invalidation will not be sent because Hazelcast assumes there is no other local session at the moment. When a server is down, requests belonging to a session held in that server will routed to other server, and that server will fetch session data from clustered cache. That means, using sticky sessions, one will not suffer the performance penalty of accessing clustered data and can benefit recover from a server failure.

13.2.2.6 transient-attributes

If you have some attributes that you do not want them to be distributed, you can mark those attributes as transient. Transient attributes are kept in and when the server is shutdown, you lost the attribute values. You can set the transient attributes in your `web.xml` file. Here is an example:

```
...  
  <init-param>
    <param-name>transient-attributes</param-name>
    <param-value>key1,key2,key3</param-value>
  </init-param>
  ...
```
13.2. WEB SESSION REPLICATION

13.2.2.7 Client/Server Configuration

In client/server mode of session replication `session-ttl-seconds` config does not have any effect, because filter based session replication uses IMap and hazelcast client cannot change configuration of a distributed map. You should configure `max-idle-seconds` parameter in your `hazelcast.xml` on the server side.

```xml
... <map name="my-sessions"> <!-- How much seconds you want your session attributes to stored on server? Default is 0. --> <max-idle-seconds>20</max-idle-seconds> </map> ...
```

Also make sure that name of the distributed map is same as the `map-name` parameter defined in your `web.xml` config file.

13.2.3 Tomcat Based Web Session Replication

Hazelcast Enterprise

⚠️ **NOTE:** This feature is supported for Hazelcast Enterprise 3.3 or higher.

**Sample Code:** Please see our sample application for Tomcat Based Web Session Replication.

13.2.3.1 Overview

Session Replication with Hazelcast Enterprise is a container specific module that enables session replication for JEE Web Applications without requiring changes to the application.

**Features**

1. Seamless Tomcat 6, 7 & 8 integration (Tomcat 8 is supported for Hazelcast Enterprise 3.5 or higher.)
2. Support for sticky and non-sticky sessions
3. Tomcat failover
4. Deferred write for performance boost

**Supported Containers**

Tomcat Web Session Replication Module has been tested against the following containers.

- Tomcat 6.0.x - It can be downloaded here.
- Tomcat 7.0.x - It can be downloaded here.
- Tomcat 8.0.x - It can be downloaded here.

The latest tested versions are 6.0.39, 7.0.40 and 8.0.20.

**Requirements**

- Tomcat instance must be running with Java 1.6 or higher.
- Session objects that need to be clustered have to be Serializable.

13.2.3.2 How Tomcat Session Replication works

Tomcat Session Replication in Hazelcast Enterprise is a Hazelcast Module where each created `HttpSession` Object is kept in the Hazelcast Distributed Map. If configured with Sticky Sessions, each Tomcat Instance has its own local copy of the session for performance boost.

Since the sessions are in Hazelcast Distributed Map, you can use all the available features offered by Hazelcast Distributed Map implementation, such as MapStore and WAN Replication.

Tomcat Web Sessions run in two different modes:

- **P2P**: all Tomcat instances launch its own Hazelcast Instance and join to the Hazelcast Cluster and,
- **Client/Server**: all Tomcat instances put/retrieve the session data to/from an existing Hazelcast Cluster.
13.2.3.3 P2P (Peer-to-Peer) Deployment

P2P deployment launches an embedded Hazelcast Node in each server instance.

Features

This type of deployment is simple: just configure your Tomcat and launch. There is no need for an external Hazelcast cluster.

Sample P2P Configuration to use Hazelcast Session Replication

- Go to `hazelcast.com` and download the latest Hazelcast Enterprise.
- Unzip the Hazelcast Enterprise zip file into the folder `$HAZELCAST_ENTERPRISE_ROOT`.
- Update `$HAZELCAST_ENTERPRISE_ROOT/bin/hazelcast.xml` with the provided Hazelcast Enterprise License Key.
- Put a `<Listener>` element into the file `$CATALINA_HOME/conf/server.xml` as shown below.

```xml
<Server>
  ...
  <Listener className="com.hazelcast.session.P2PLifecycleListener"/>
  ...
</Server>
```

- Put a `<Manager>` element into the file `$CATALINA_HOME/conf/context.xml` as shown below.

```xml
<Context>
  ...
  <Manager className="com.hazelcast.session.HazelcastSessionManager"/>
  ...
</Context>
```

- Start Tomcat instances with a configured load balancer and deploy the web application.

Optional Attributes for Listener Element

- Optionally, you can add `configLocation` attribute into the `<Listener>` element. If not provided, `hazelcast.xml` in the classpath is used by default. URL or full filesystem path as a `configLocation` value is supported.

13.2.3.4 Client/Server Deployment

In this deployment type, Tomcat instances work as clients on an existing Hazelcast Cluster.

Features

- The existing Hazelcast cluster is used as the Session Replication Cluster.
- Offloading Session Cache from Tomcat to the Hazelcast Cluster.
- The architecture is completely independent. Complete reboot of Tomcat instances.

Sample Client/Server Configuration to use Hazelcast Session Replication

```xml
<Server>
  ...
  <Listener className="com.hazelcast.session.P2PLifecycleListener"/>
  ...
</Server>
```
13.2. WEB SESSION REPLICATION

- Go to hazelcast.com and download the latest Hazelcast Enterprise.
- Unzip the Hazelcast Enterprise zip file into the folder $HAZELCAST_ENTERPRISE_ROOT.
- Put a <Listener> element into the $CATALINA_HOME/conf/server.xml as shown below.

```
<Listener className="com.hazelcast.session.ClientServerLifecycleListener"/>
```

- Update the <Manager> element in the $CATALINA_HOME/conf/context.xml as shown below.

```
<Manager className="com.hazelcast.session.HazelcastSessionManager" clientOnly="true"/>
```

- Launch a Hazelcast Instance using $HAZELCAST.Enterprise_ROOT/bin/server.sh or $HAZELCAST.Enterprise_ROOT/bin/server.bat.
- Start Tomcat instances with a configured load balancer and deploy the web application.

Optional Attributes for Listener Element

- Optionally, you can add configLocation attribute into the <Listener> element. If not provided, hazelcast-client-default.xml in hazelcast-client-<version>.jar file is used by default. Any client XML file in the classpath, URL or full filesystem path as a configLocation value is also supported.

13.2.3.5 Optional Attributes for Manager Element

<Manager> element is used both in P2P and Client/Server mode. You can use the following attributes to configure Tomcat Session Replication Module to better serve your needs.

- Add mapName attribute into <Manager> element. Its default value is default Hazelcast Distributed Map. Use this attribute if you have a specially configured map for special cases like WAN Replication, Eviction, MapStore, etc.
- Add sticky attribute into <Manager> element. Its default value is true.
- Add processExpiresFrequency attribute into <Manager> element. It specifies the frequency of session validity check, in seconds. Its default value is 6 and the minimum value that you can set is 1.
- Add deferredWrite attribute into <Manager> element. Its default value is true.

13.2.3.6 Session Caching and deferredWrite parameter

Tomcat Web Session Replication Module has its own nature of caching. Attribute changes during the HTTP Request/HTTP Response cycle is cached by default. Distributing those changes to the Hazelcast Cluster is costly. Because of that, Session Replication is only done at the end of each request for updated and deleted attributes. The risk in this approach is losing data if a Tomcat crash happens in the middle of the HTTP Request operation.

You can change that behavior by setting deferredWrite=false in your <Manager> element. By disabling it, all updates that are done on session objects are directly distributed into Hazelcast Cluster.
13.2.3.7 Session Expiry

Based on Tomcat configuration or sessionTimeout setting in web.xml, sessions are expired over time. This requires a cleanup on the Hazelcast Cluster since there is no need to keep expired sessions in the cluster.

processExpiresFrequency, which is defined in <Manager>, is the only setting that controls the behavior of session expiry policy in the Tomcat Web Session Replication Module. By setting this, you can set the frequency of the session expiration checks in the Tomcat Instance.

13.2.3.8 Enabling Session Replication in Multi-App environment

Tomcat can be configured in two ways to enable Session Replication for deployed applications.

- Server Context.xml Configuration
- Application Context.xml Configuration

Server Context.xml Configuration

By configuring $CATALINA_HOME$/conf/context.xml, you can enable session replication for all applications deployed in the Tomcat Instance.

Application Context.xml Configuration

By configuring $CATALINA_HOME/conf/[enginename]/[hostname]/[applicationName].xml, you can enable Session Replication per deployed application.

13.2.3.9 Session Affinity

Sticky Sessions (default)

Sticky Sessions are used to improve the performance since the sessions do not move around the cluster.

Request goes always to the same instance where the session was firstly created. By using a sticky session, you eliminate session replication problems mostly, except for the failover cases. In case of failovers, Hazelcast helps you not lose existing sessions.

Non-Sticky Sessions

Non-Sticky Sessions are not good for performance because you need to move session data all over the cluster every time a new request comes in.

However, load balancing might be super easy with Non-Sticky caches. In case of heavy load, you can distribute the request to the least used Tomcat instance. Hazelcast supports Non-Sticky Sessions as well.

13.2.3.10 Tomcat Failover and jvmRoute Parameter

Each HTTP Request is redirected to the same Tomcat instance if sticky sessions are enabled. The parameter jvmRoute is added to the end of session ID as a suffix, to make Load Balancer aware of the target Tomcat instance.

When Tomcat Failure happens and Load Balancer cannot redirect the request to the owning instance, it sends a request to one of the available Tomcat instances. Since the jvmRoute parameter of session ID is different than that of the target Tomcat instance, Hazelcast Session Replication Module updates the session ID of the session with the new jvmRoute parameter. That means that the Session is moved to another Tomcat instance and Load Balancer will redirect all subsequent HTTP Requests to the new Tomcat Instance.

NOTE: If stickySession is enabled, jvmRoute parameter must be set in $CATALINA_HOME$/conf/server.xml and unique among Tomcat instances in the cluster.

<Engine name="Catalina" defaultHost="localhost" jvmRoute="tomcat-8080">
13.2.4 Jetty Based Web Session Replication

Hazelcast Enterprise

⚠️ NOTE: This feature is supported for Hazelcast Enterprise 3.4 or higher.

Sample Code: Please see our sample application for Jetty Based Web Session Replication.

13.2.4.1 Overview

Jetty Web Session Replication with Hazelcast Enterprise is a container specific module that enables session replication for JEE Web Applications without requiring changes to the application.

Features

1. Jetty 7 & 8 & 9 support
2. Support for sticky and non-sticky sessions
3. Jetty failover
4. Deferred write for performance boost
5. Client/Server and P2P modes
6. Declarative and programmatic configuration

Supported Containers

Jetty Web Session Replication Module has been tested against the following containers.

- Jetty 7 - It can be downloaded here.
- Jetty 8 - It can be downloaded here.
- Jetty 9 - It can be downloaded here.

Latest tested versions are 7.6.16.v20140903, 8.1.16.v20140903 and 9.2.3.v20140905

Requirements

- Jetty instance must be running with Java 1.6 or higher.
- Session objects that need to be clustered have to be Serializable.
- Hazelcast Jetty-based Web Session Replication is built on top of the jetty-nosql module. This module (jetty-nosql-<jettyversion>.jar) needs to be added to $JETTY_HOME/lib/ext. This module can be found here.

13.2.4.2 How Jetty Session Replication Works

Jetty Session Replication in Hazelcast Enterprise is a Hazelcast Module where each created HttpSession Object’s state is kept in Hazelcast Distributed Map.

Since the session data are in Hazelcast Distributed Map, you can use all the available features offered by Hazelcast Distributed Map implementation, such as MapStore and WAN Replication.

Jetty Web Session Replication runs in two different modes:

- **P2P**: all Jetty instances launch its own Hazelcast Instance and join to the Hazelcast Cluster and,
- **Client/Server**: all Jetty instances put/retrieve the session data to/from an existing Hazelcast Cluster.
13.2.4.3 P2P (Peer-to-Peer) Deployment

P2P deployment launches embedded Hazelcast Node in each server instance.

**Features**

This type of deployment is simple: just configure your Jetty and launch. There is no need for an external Hazelcast cluster.

**Sample P2P Configuration to use Hazelcast Session Replication**

- Go to hazelcast.com and download the latest Hazelcast Enterprise.
- Unzip the Hazelcast Enterprise zip file into the folder $HAZELCAST_ENTERPRISE_ROOT.
- Update $HAZELCAST_ENTERPRISE_ROOT/bin/hazelcast.xml with the provided Hazelcast Enterprise License Key.
- Put hazelcast.xml in the folder $JETTY_HOME/etc.
- Configure Session ID Manager and Session Manager. Please see the following explanations for configuring these managers.

**Configuring the HazelcastSessionIdManager**

You need to configure a com.hazelcast.session.HazelcastSessionIdManager instance in jetty.xml. Add the following lines to your jetty.xml.

```xml
<Set name="sessionIdManager">
    <New id="hazelcastIdMgr" class="com.hazelcast.session.HazelcastSessionIdManager">
        <Arg><Ref id="Server"/></Arg>
        <Set name="configLocation">etc/hazelcast.xml</Set>
    </New>
</Set>
```

**Configuring the HazelcastSessionManager**

HazelcastSessionManager can be configured from a context.xml file. Each application has a context file in the $CATALINA_HOMES/contexts folder. You need to create this context file if it does not exist. The context filename must be the same as the application name, e.g. example.war should have a context file named example.xml.

The file context.xml should have the following content.

```xml
<Ref name="Server" id="Server">
    <Call id="hazelcastIdMgr" name="getSessionIdManager"/>
</Ref>
<Set name="sessionHandler">
    <New class="org.eclipse.jetty.server.session.SessionHandler">
        <Arg>
            <New id="hazelcastMgr" class="com.hazelcast.session.HazelcastSessionManager">
                <Set name="idManager">
                    <Ref id="hazelcastIdMgr"/>
                </Set>
            </New>
        </Arg>
    </New>
</Set>
```

- Start Jetty instances with a configured load balancer and deploy the web application.
![NOTE: In Jetty 9, there is no folder with the name contexts. You have to put the file context.xml under the webapps directory. And you need to add the following lines to context.xml.]

```xml
<Ref name="Server" id="Server">
  <Call id="hazelcastIdMgr" name="getSessionIdManager"/>
</Ref>
<Set name="sessionIdManager">
  <New class="org.eclipse.jetty.server.session.SessionHandler">
    <Arg>
      <New id="hazelcastMgr" class="com.hazelcast.session.HazelcastSessionManager">
        <Set name="sessionIdManager">
          <Ref id="hazelcastIdMgr"/>
        </Set>
      </New>
    </Arg>
  </New>
</Set>
```

### 13.2.4.4 Client/Server Deployment

In client/server deployment type, Jetty instances work as clients to an existing Hazelcast Cluster.

**Features**

- Existing Hazelcast cluster is used as the Session Replication Cluster.
- The architecture is completely independent. Complete reboot of Jetty instances without losing data.

**Sample Client/Server Configuration to use Hazelcast Session Replication**

- Go to hazelcast.com and download the latest Hazelcast Enterprise.
- Unzip the Hazelcast Enterprise zip file into the folder $HAZELCAST_ENTERPRISE_ROOT.
- Update $HAZELCAST_ENTERPRISE_ROOT/bin/hazelcast.xml with the provided Hazelcast Enterprise License Key.
- Put hazelcast.xml in the folder $JETTY_HOME/etc.
- Configure Session ID Manager and Session Manager. Please see below explanations for configuring these managers.

**Configuring the HazelcastSessionIdManager**

You need to configure a com.hazelcast.session.HazelcastSessionIdManager instance in jetty.xml. Add the following lines to your jetty.xml.

```xml
<Set name="sessionIdManager">
  <New id="hazelcastIdMgr" class="com.hazelcast.session.HazelcastSessionIdManager">
    <Arg><Ref id="Server"/></Arg>
    <Set name="configLocation">etc/hazelcast.xml</Set>
    <Set name="clientOnly">true</Set>
  </New>
</Set>
```

**Configuring the HazelcastSessionManager**

HazelcastSessionManager can be configured from a context.xml file. Each application has a context file under the $CATALINA_HOME/contexts folder. You need to create this context file if it does not exist. The context filename must be the same as the application name, e.g. example.war should have a context file named example.xml.
NOTE: In Jetty 9, there is no folder with name contexts. You have to put the file context.xml file under webapps directory. And you need to add below lines to context.xml.

13.2.4.5 Optional HazelcastSessionIdManager Parameters

HazelcastSessionIdManager is used both in P2P and Client/Server mode. Use the following parameters to configure the Jetty Session Replication Module to better serve your needs.

- **workerName**: Set this attribute to a unique value for each Jetty instance to enable session affinity with a sticky-session configured load balancer.
- **cleanUpPeriod**: Defines the working period of session clean-up task in milliseconds.
- **configLocation**: specifies the location of hazelcast.xml.

13.2.4.6 Optional HazelcastSessionManager Parameters

HazelcastSessionManager is used both in P2P and Client/Server mode. Use the following parameters to configure Jetty Session Replication Module to better serve your needs.
13.3. SPRING INTEGRATION

- **savePeriod**: Sets the interval of saving session data to the Hazelcast cluster. Jetty Web Session Replication Module has its own nature of caching. Attribute changes during the HTTP Request/HTTP Response cycle are cached by default. Distributing those changes to the Hazelcast Cluster is costly, so Session Replication is only done at the end of each request for updated and deleted attributes. The risk with this approach is losing data if a Jetty crash happens in the middle of the HTTP Request operation. You can change that behavior by setting the `savePeriod` attribute.

Notes:

- If `savePeriod` is set to -2, `HazelcastSessionManager.save` method is called for every `doPutOrRemove` operation.
- If it is set to -1, the same method is never called if Jetty is not shut down.
- If it is set to 0 (the default value), the same method is called at the end of request.
- If it is set to 1, the same method is called at the end of request if session is dirty.

13.2.4.7 Session Expiry

Based on Tomcat configuration or `sessionTimeout` setting in `web.xml`, the sessions are expired over time. This requires a cleanup on Hazelcast Cluster, since there is no need to keep expired sessions in it.

`cleanUpPeriod`, which is defined in `HazelcastSessionIdManager`, is the only setting that controls the behavior of session expiry policy in Jetty Web Session Replication Module. By setting this, you can set the frequency of the session expiration checks in the Jetty Instance.

13.2.4.8 Session Affinity

`HazelcastSessionIdManager` can work in sticky and non-sticky setups.

The clustered session mechanism works in conjunction with a load balancer that supports stickiness. Stickiness can be based on various data items, such as source IP address, or characteristics of the session ID, or a load-balancer specific mechanism. For those load balancers that examine the session ID, `HazelcastSessionIdManager` appends a node ID to the session ID, which can be used for routing. You must configure the `HazelcastSessionIdManager` with a `workerName` that is unique across the cluster. Typically the name relates to the physical node on which the instance is executed. If this name is not unique, your load balancer might fail to distribute your sessions correctly. If sticky sessions are enabled, the `workerName` parameter has to be set, as shown below.

```xml
<Set name="sessionIdManager">
  <New id="hazelcastIdMgr" class="com.hazelcast.session.HazelcastSessionIdManager">
    <Arg><Ref id="Server"/></Arg>
    <Set name="configLocation">etc/hazelcast.xml</Set>
    <Set name="workerName">unique-worker-1</Set>
  </New>
</Set>
```

13.3 Spring Integration

You can integrate Hazelcast with Spring and this chapter explains the configuration of Hazelcast within Spring context.

13.3.1 Supported Versions

- Spring 2.5+
13.3.2 Spring Configuration

Sample Code: Please see our sample application for Spring Configuration.

13.3.2.1 Bean Declaration by Spring beans Namespace

Classpath Configuration
This configuration requires the following jar file in the classpath:

- hazelcast-<version>.jar

Bean Declaration
You can declare Hazelcast Objects using the default Spring beans namespace. You can find an example usage of Hazelcast Instance declaration as follows:

```xml
<bean id="instance" class="com.hazelcast.core.Hazelcast" factory-method="newHazelcastInstance">
    <constructor-arg>
        <bean class="com.hazelcast.config.Config">
            <property name="groupConfig">
                <bean class="com.hazelcast.config.GroupConfig">
                    <property name="name" value="dev"/>
                    <property name="password" value="pwd"/>
                </bean>
            </property>
        </bean>
    </constructor-arg>
</bean>

<bean id="map" factory-bean="instance" factory-method="getMap">
    <constructor-arg value="map"/>
</bean>
```

13.3.2.2 Bean Declaration by hazelcast Namespace

Classpath Configuration
Hazelcast-Spring integration requires the following JAR files in the classpath:

- hazelcast-spring-<version>.jar
- hazelcast-<version>.jar

or

- hazelcast-all-<version>.jar

Bean Declaration
Hazelcast has its own namespace hazelcast for bean definitions. You can easily add the namespace declaration xmlns:hz="http://www.hazelcast.com/schema/spring" to the beans element in the context file so that hz namespace shortcut can be used as a bean declaration.

Here is an example schema definition for Hazelcast 3.3.x:
13.3. SPRING INTEGRATION

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:hz="http://www.hazelcast.com/schema/spring"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.hazelcast.com/schema/spring
                           http://www.hazelcast.com/schema/spring/hazelcast-spring.xsd">

13.3.2.3 Supported Configurations with hazelcast Namespace

- Hazelcast Instance Configuration

  <hz:hazelcast id="instance">
    <hz:config>
      <hz:group name="dev" password="password"/>
      <hz:network port="5701" port-auto-increment="false">
        <hz:join>
          <hz:multicast enabled="false"
                         multicast-group="224.2.2.3"
                         multicast-port="54327"/>
          <hz:tcp-ip enabled="true">
            <hz:members>10.10.1.2, 10.10.1.3</hz:members>
          </hz:tcp-ip>
        </hz:join>
      </hz:network>
      <hz:map name="map"
               backup-count="2"
               max-size="0"
               eviction-percentage="30"
               read-backup-data="true"
               eviction-policy="NONE"
               merge-policy="com.hazelcast.map.merge.PassThroughMergePolicy"/>
    </hz:config>
  </hz:hazelcast>

- Hazelcast Client Configuration

  <hz:client id="client">
    <hz:group name="${cluster.group.name}" password="${cluster.group.password}"/>
    <hz:network connection-attempt-limit="3"
                connection-attempt-period="3000"
                connection-timeout="1000"
                redo-operation="true"
                smart-routing="true">
      <hz:member>10.10.1.2:5701</hz:member>
      <hz:member>10.10.1.3:5701</hz:member>
    </hz:network>
  </hz:client>

- Hazelcast Supported Type Configurations and Examples

  - map
  - multiMap
  - replicatedmap
  - queue
  - topic
- set
- list
- executorService
- idGenerator
- atomicLong
- atomicReference
- semaphore
- countDownLatch
- lock

```xml
<hz:map id="map" instance-ref="client" name="map" lazy-init="true" />
<hz:multiMap id="multiMap" instance-ref="instance" name="multiMap"
  lazy-init="false" />
<hz:replicatedmap id="replicatedmap" instance-ref="instance"
  name="replicatedmap" lazy-init="false" />
<hz:queue id="queue" instance-ref="client" name="queue"
  lazy-init="true" depends-on="instance"/>
<hz:topic id="topic" instance-ref="instance" name="topic"
  depends-on="instance, client"/>
<hz:set id="set" instance-ref="instance" name="set" />
<hz:list id="list" instance-ref="instance" name="list"/>
<hz:executorService id="executorService" instance-ref="client"
  name="executorService"/>
<hz:idGenerator id="idGenerator" instance-ref="instance"
  name="idGenerator"/>
<hz:atomicLong id="atomicLong" instance-ref="instance"
  name="atomicLong"/>
<hz:atomicReference id="atomicReference" instance-ref="instance"
  name="atomicReference"/>
<hz:semaphore id="semaphore" instance-ref="instance" name="semaphore"/>
<hz:countDownLatch id="countDownLatch" instance-ref="instance"
  name="countDownLatch"/>
<hz:lock id="lock" instance-ref="instance" name="lock"/>
```

- Supported Spring Bean Attributes

Hazelcast also supports lazy-init, scope and depends-on bean attributes.

```xml
<hz:hazelcast id="instance" lazy-init="true" scope="singleton">
  ...
</hz:hazelcast>
<hz:client id="client" scope="prototype" depends-on="instance">
  ...
</hz:client>
```

- MapStore and NearCache Configuration

For map-store, you should set either the class-name or the implementation attribute.

```xml
<hz:config>
  <hz:map name="map1">
    <hz:near-cache time-to-live-seconds="0" max-idle-seconds="60"
      eviction-policy="LRU" max-size="5000" invalidate-on-change="true"/>
    <hz:map-store enabled="true" class-name="com.foo.DummyStore"
      write-delay-seconds="0"/>
  </hz:map>
</hz:config>
```
13.3. SPRING INTEGRATION

13.3.3 Spring Managed Context with @SpringAware

Hazelcast Distributed Objects could be marked with @SpringAware if the object wants:

- to apply bean properties,
- to apply factory callbacks such as ApplicationContextAware, BeanNameAware,
- to apply bean post-processing annotations such as InitializingBean, @PostConstruct.

Hazelcast Distributed ExecutorService, or more generally any Hazelcast managed object, can benefit from these features. To enable SpringAware objects, you must first configure HazelcastInstance using hazelcast namespace as explained in the Spring Configuration section and add <hz:spring-aware /> tag.

13.3.3.1 SpringAware Examples

- Configure a Hazelcast Instance (3.3.x) via Spring Configuration and define someBean as Spring Bean.
- Add <hz:spring-aware /> to Hazelcast configuration to enable @SpringAware.

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:hz="http://www.hazelcast.com/schema/spring"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd
                           http://www.hazelcast.com/schema/spring
                           http://www.hazelcast.com/schema/spring/hazelcast-spring.xsd">

  <context:annotation-config />

  <hz:hazelcast id="instance">
    <hz:config>
      <hz:spring-aware />
      <hz:group name="dev" password="password"/>
      <hz:network port="5701" port-auto-increment="false">
        <hz:join>
          <hz:multicast enabled="false"/>
          <hz:tcp-ip enabled="true">
            <hz:members>10.10.1.2, 10.10.1.3</hz:members>
          </hz:tcp-ip>
        </hz:join>
      </hz:network>
    </hz:config>
  </hz:hazelcast>

  <bean id="someBean" class="com.hazelcast.examples.spring.SomeBean"/>
```

```xml
<hz:map name="map2">
  <hz:map-store enabled="true" implementation="dummyMapStore"
               write-delay-seconds="0"/>
</hz:map>

<bean id="dummyMapStore" class="com.foo.DummyStore" />
</hz:config>
```
Distributed Map Example:

- Create a class called `SomeValue` which contains Spring Bean definitions like `ApplicationContext` and `SomeBean`.

```java
@SpringAware
@Component("someValue")
@Scope("prototype")
public class SomeValue implements Serializable, ApplicationContextAware {

    private transient ApplicationContext context;
    private transient SomeBean someBean;
    private transient boolean init = false;

    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        context = applicationContext;
    }

    @Autowired
    public void setSomeBean(SomeBean someBean) {
        this.someBean = someBean;
    }

    @PostConstruct
    public void init() {
        someBean.doSomethingUseful();
        init = true;
    }

    ...
}
```

- Get `SomeValue` Object from Context and put it into Hazelcast Distributed Map on Node-1.

```java
HazelcastInstance hazelcastInstance = (HazelcastInstance) context.getBean( "hazelcast" );
SomeValue value = (SomeValue) context.getBean( "someValue" )
IMap<String, SomeValue> map = hazelcastInstance.getMap( "values" );
map.put( "key", value );
```

- Read `SomeValue` Object from Hazelcast Distributed Map and assert that `init` method is called since it is annotated with `@PostConstruct`.

```java
HazelcastInstance hazelcastInstance = (HazelcastInstance) context.getBean( "hazelcast" );
IMap<String, SomeValue> map = hazelcastInstance.getMap( "values" );
SomeValue value = map.get( "key" );
Assert.assertTrue( value.init );
```

ExecutorService Example:
13.3. SPRING INTEGRATION

- Create a Callable Class called SomeTask which contains Spring Bean definitions like ApplicationContext, SomeBean.

```java
@SpringAware
class SomeTask
    implements Callable<Long>, ApplicationContextAware, Serializable {

    private transient ApplicationContext context;

    private transient SomeBean someBean;

    public Long call() throws Exception {
        return someBean.value;
    }

    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        context = applicationContext;
    }

    @Autowired
    public void setSomeBean(SomeBean someBean) {
        this.someBean = someBean;
    }
}
```

- Submit SomeTask to two Hazelcast Members and assert that someBean is autowired.

```java
HazelcastInstance hazelcastInstance = (HazelcastInstance) context.getBean("hazelcast");
SomeBean bean = (SomeBean) context.getBean("someBean");
Future<Long> f = hazelcastInstance.getExecutorService().submit(new SomeTask());
Assert.assertEquals(bean.value, f.get().longValue());

// choose a member
Member member = hazelcastInstance.getCluster().getMembers().iterator().next();
Future<Long> f2 = (Future<Long>) hazelcast.getExecutorService().
    .submitToMember(new SomeTask(), member);
Assert.assertEquals(bean.value, f2.get().longValue());
```

**NOTE:** Spring managed properties/fields are marked as transient.

13.3.4 Spring Cache

**Sample Code:** Please see our sample application for Spring Cache.

As of version 3.1, Spring Framework provides support for adding caching into an existing Spring application.

13.3.4.1 Declarative Spring Cache Configuration

```xml
<cache:annotation-driven cache-manager="cacheManager" />

<hz:hazelcast id="hazelcast"/>
```
13.3.4.2 Annotation Based Spring Cache Configuration

Annotation Based Configuration does not require any XML definition.

- Implement a `CachingConfiguration` class with related Annotations.

```java
@Configuration
@EnableCaching
public class CachingConfiguration implements CachingConfigurer{
    @Bean
    public CacheManager cacheManager() {
        ClientConfig config = new ClientConfig();
        HazelcastInstance client = HazelcastClient.newHazelcastClient(config);
        return new HazelcastCacheManager(client);
    }
    @Bean
    public KeyGenerator keyGenerator() {
        return null;
    }
}
```

- Launch Application Context and register `CachingConfiguration`.

```java
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
context.register(CachingConfiguration.class);
context.refresh();
```

For more information about Spring Cache, please see Spring Cache Abstraction.

13.3.5 Hibernate 2nd Level Cache Config

**Sample Code**: Please see our sample application for Hibernate 2nd Level Cache Config.

If you are using Hibernate with Hazelcast as 2nd level cache provider, you can easily create `RegionFactory` instances within Spring configuration (by Spring version 3.1). That way, you can use the same `HazelcastInstance` as Hibernate L2 cache instance.

```xml
<hz:hibernate-region-factory id="regionFactory" instance-ref="instance"
    mode="LOCAL" />
...
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
    scope="singleton">
    <property name="dataSource" ref="dataSource"/>
    <property name="cacheRegionFactory" ref="regionFactory" />
    ...
</bean>
```

Hibernate RegionFactory Modes
13.3. **SPRING INTEGRATION**

- LOCAL
- DISTRIBUTED

Please refer to the Hibernate RegionFactory Options section for more information.

### 13.3.6 Best Practices

Spring tries to create a new `Map/Collection` instance and fill the new instance by iterating and converting values of the original `Map/Collection` (`IMap`, `IQueue`, etc.) to required types when generic type parameters of the original `Map/Collection` and the target property/attribute do not match.

Since Hazelcast `Maps/Collections` are designed to hold very large data which a single machine cannot carry, iterating through whole values can cause out of memory errors.

To avoid this issue, the target property/attribute can be declared as un-typed `Map/Collection` as shown below.

```java
public class SomeBean {
    @Autowired
    IMap map; // instead of IMap<K, V> map

    @Autowired
    IQueue queue; // instead of IQueue<E> queue

    ...
}
```

Or, parameters of injection methods (constructor, setter) can be un-typed as shown below.

```java
public class SomeBean {

    IMap<K, V> map;
    IQueue<E> queue;

    // Instead of IMap<K, V> map
    public SomeBean(IMap map) {
        this.map = map;
    }

    ...

    // Instead of IQueue<E> queue
    public void setQueue(IQueue queue) {
        this.queue = queue;
    }
}
```

**RELATED INFORMATION**

For more information please see *Spring issue-3407*. 
Chapter 14

Storage

This chapter describes Hazelcast’s High-Density Memory Store and its configuration and provides information on the High-Density Memory First Generation, also known as Hazelcast Elastic Memory. It also gives recommendations on the storage sizing.

14.1 High-Density Memory Store

Hazelcast Enterprise

Hazelcast High-Density Memory Store, the successor to Hazelcast Elastic Memory, is Hazelcast’s new enterprise grade backend storage solution. This solution is used with the Hazelcast JCache implementation.

By default, Hazelcast offers a production ready, low garbage collection (GC) pressure, storage backend. Serialized keys and values are still stored in the standard Java map, such as data structures on the heap. The data structures are stored in serialized form for the highest data compaction, and are still subject to Java Garbage Collection.

In Hazelcast Enterprise, the High-Density Memory Store is built around a pluggable memory manager which enables multiple memory stores. These memory stores are all accessible using a common access layer that scales up to Terabytes of main memory on a single JVM. At the same time, by further minimizing the GC pressure, High-Density Memory Store enables predictable application scaling and boosts performance and latency while minimizing pauses for Java Garbage Collection.

This foundation includes, but is not limited to, storing keys and values next to the heap in a native memory region.

RELATED INFORMATION

Please refer to the Hazelcast JCache chapter for the details of Hazelcast JCache implementation. As mentioned, High-Density Memory Store is used with Hazelcast JCache implementation.

14.1.1 Configuring Hi-Density Memory Store

To use the Hi-Density memory storage, the native memory usage must be enabled using the programmatic or declarative configuration. Also, you can configure its size, memory allocator type, minimum block size, page size and metadata space percentage.

- **size**: Size of the total native memory to allocate. Default value is 512 MB.
- **allocator type**: Type of the memory allocator. Available values are:
  - STANDARD: allocate/free memory using default OS memory manager.
  - POOLED: manage memory blocks in thread local pools.

Default value is **POOLED**.
- **minimum block size**: Minimum size of the blocks in bytes to split and fragment a page block to assign to an allocation request. It is used only by the **POOLED** memory allocator. Default value is **16**.
- **page size**: Size of the page in bytes to allocate memory as a block. It is used only by the POOLED memory allocator. Default value is $1 \ll 22 = 4194304$ Bytes, about 4 MB.
- **metadata space percentage**: Defines the percentage of the allocated native memory that is used for the metadata such as indexes, offsets, etc. It is used only by the POOLED memory allocator. Default value is 12.5.

The following is the programmatic configuration example.

```java
MemorySize memorySize = new MemorySize(512, MemoryUnit.MEGABYTES);
NativeMemoryConfig nativeMemoryConfig =
    new NativeMemoryConfig()
        .setAllocatorType(NativeMemoryConfig.MemoryAllocatorType.POOLED)
        .setSize(memorySize)
        .setEnabled(true)
        .setMinBlockSize(16)
        .setPageSize(1 << 20);
```

The following is the declarative configuration example.

```xml
<native-memory enabled="true" allocator-type="POOLED">
    <size value="512" unit="MEGABYTES"/>
</native-memory>
```

## 14.2 Elastic Memory (High-Density Memory First Generation)

By default, Hazelcast stores your distributed data (map entries, queue items) into Java heap which is subject to garbage collection (GC). As your heap gets bigger, garbage collection might cause your application to pause tens of seconds, badly effecting your application performance and response times. Elastic Memory (High-Density Memory First Generation) is Hazelcast with off-heap memory storage to avoid GC pauses. Even if you have terabytes of cache in-memory with lots of updates, GC will have almost no effect; resulting in more predictable latency and throughput.

Here are the steps to enable Elastic Memory:

- Set the maximum direct memory JVM can allocate, e.g. `java -XX:MaxDirectMemorySize=60G`.
- Enable Elastic Memory by setting the `hazelcast.elastic.memory.enabled` property to true.
- Set the total direct memory size for HazelcastInstance by setting the `hazelcast.elastic.memory.total.size` property. Size can be in MB or GB and abbreviation can be used, such as 60G and 500M.
- Set the chunk size by setting the `hazelcast.elastic.memory.chunk.size` property. Hazelcast will partition the entire off-heap memory into chunks. Default chunk size is 1K.
- You can enable `sun.misc.Unsafe` based off-heap storage implementation instead of `java.nio.DirectByteBuffer` based one, by setting the `hazelcast.elastic.memory.unsafe.enabled` property to true. Default value is false.
- Configure maps that will use Elastic Memory by setting `InMemoryFormat` to `NATIVE`. Default value is `BINARY`.

Below is the declarative configuration.

```xml
<hazelcast>
    ...
    <map name="default">
        ...
        <in-memory-format>NATIVE</in-memory-format>
    </map>
</hazelcast>
```
And, the programmatic configuration:

```java
MapConfig mapConfig = new MapConfig();
mapConfig.setInMemoryFormat( InMemoryFormat.NATIVE );
```

And, the following are the High-Density Memory First Generation related system properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hazelcast.elastic.memory.enabled</td>
<td>false</td>
<td>bool</td>
</tr>
<tr>
<td>hazelcast.elastic.memory.total.size</td>
<td>128</td>
<td>int</td>
</tr>
<tr>
<td>hazelcast.elastic.memory.chunk.size</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>hazelcast.elastic.memory.shared.storage</td>
<td>false</td>
<td>bool</td>
</tr>
<tr>
<td>hazelcast.elastic.memory.unsafe.enabled</td>
<td>false</td>
<td>bool</td>
</tr>
</tbody>
</table>

### 14.3 Sizing Practices

Data in Hazelcast is both active data and backup data for high availability, so the total memory footprint is the size of active data plus the size of backup data. If you use a single backup, it means the total memory footprint is two times the active data (active data + backup data). If you use, for example, two backups, then the total memory footprint is three times the active data (active data + backup data + backup data).

If you use only heap memory, each Hazelcast node with a 4 GB heap should accommodate a maximum of 3.5 GB of total data (active and backup). If you use the High-Density Memory Store, up to 75% of your physical memory footprint may be used for active and backup data, with headroom of 25% for normal fragmentation. In both cases, however, you should also keep some memory headroom available to handle any node failure or explicit node shutdown. When a node leaves the cluster, the data previously owned by the newly offline node will be distributed among the remaining servers. For this reason, we recommend that you plan to use only 60% of available memory, with 40% headroom to handle node failure or shutdown.
Chapter 15

Hazelcast Java Client

There are currently three ways to connect to a running Hazelcast cluster:

- Native Clients (Java, C++, .NET)
- Memcache Client
- REST Client

Native Clients enable you to perform almost all Hazelcast operations without being a member of the cluster. It connects to one of the cluster members and delegates all cluster wide operations to it (dummy client), or it connects to all of them and delegates operations smartly (smart client). When the relied cluster member dies, the client will transparently switch to another live member.

There can be hundreds, even thousands of clients connected to the cluster. By default, there are core count * 10 threads on the server side that will handle all the requests (e.g. if the server has 4 cores, it will be 40).

Imagine a trading application where all the trading data are stored and managed in a Hazelcast cluster with tens of nodes. Swing/Web applications at the traders’ desktops can use Native Clients to access and modify the data in the Hazelcast cluster.

Currently, Hazelcast has Native Java, C++ and .NET Clients available. This chapter describes the Java Client.

**IMPORTANT:** Starting with the Hazelcast 3.5. release, a new client library is introduced in the release package: hazelcast-client-new-<version>.jar. This new Java native client library has the support for different versions of clients in a Hazelcast cluster. This support is not valid for the releases before 3.5.

15.1 Hazelcast Clients Feature Comparison

Before detailing the Java Client, this section provides the below comparison matrix to show which features are supported by the Hazelcast clients.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Java Client</th>
<th>.NET Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Queue</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Set</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>List</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>MultiMap</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Replicated Map</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Topic</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Feature</td>
<td>Java Client</td>
<td>.NET Client</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>MapReduce</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Lock</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AtomicLong</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AtomicReference</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IdGenerator</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CountDownLatch</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Transactional Map</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Transactional MultiMap</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Transactional Queue</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Transactional List</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Transactional Set</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JCache</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Ringbuffer</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Reliable Topic</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Client Configuration Import</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Hazelcast Client Protocol</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fail Fast on Invalid Configuration</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Sub-Listener Interfaces for Map ListenerMap</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Continuous Query Caching</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Distributed Executor Service</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Query</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Near Cache</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Heartbeat</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Declarative Configuration</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Programmatic Configuration</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SSL Support</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>XA Transactions</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Smart Client</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Dummy Client</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lifecycle Service</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Event Listeners</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DataSerializable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IdentifiedDataSerializable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Portable</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
15.2 Java Client Overview

The Java client is the most full featured client. It is offered both with Hazelcast and Hazelcast Enterprise. The main idea behind the Java client is to provide the same Hazelcast functionality by proxying each operation through a Hazelcast node. It can access and change distributed data, and it can listen to distributed events of an already established Hazelcast cluster from another Java application.

15.2.1 Java Client Dependencies

You should include two dependencies in your classpath to start using the Hazelcast client: hazelcast.jar and hazelcast-client.jar.

After adding these dependencies, you can start using the Hazelcast client as if you are using the Hazelcast API. The differences are discussed in the below sections.

If you prefer to use maven, add the following lines to your pom.xml.

```xml
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast-client</artifactId>
  <version>$LATEST_VERSION$</version>
</dependency>
<dependency>
  <groupId>com.hazelcast</groupId>
  <artifactId>hazelcast</artifactId>
  <version>$LATEST_VERSION$</version>
</dependency>
```

15.2.2 Getting Started with Client API

The first step is configuration. You can configure the Java client declaratively or programatically. We will use the programmatic approach throughout this tutorial. Please refer to the Java Client Declarative Configuration section for details.

```java
ClientConfig clientConfig = new ClientConfig();
clientConfig.getGroupConfig().setName("dev").setPassword("dev-pass");
clientConfig.getNetworkConfig().addAddress("10.90.0.1", "10.90.0.2:5702");
```

The second step is to initialize the HazelcastInstance to be connected to the cluster.

```java
HazelcastInstance client = HazelcastClient.newHazelcastClient(clientConfig);
```

This client interface is your gateway to access all Hazelcast distributed objects.

Let's create a map and populate it with some data.

```java
IMap<String, Customer> mapCustomers = client.getMap("customers"); //creates the map proxy

mapCustomers.put("1", new Customer("Joe", "Smith"));
mapCustomers.put("2", new Customer("Ali", "Selam"));
mapCustomers.put("3", new Customer("Avi", "Noyan"));
```

As a final step, if you are done with your client, you can shut it down as shown below. This will release all the used resources and will close connections to the cluster.

```java
client.shutdown();
```
15.2.3 Java Client Operation modes

The client has two operation modes because of the distributed nature of the data and cluster.

**Smart Client**: In smart mode, clients connect to each cluster node. Since each data partition uses the well known and consistent hashing algorithm, each client can send an operation to the relevant cluster node, which increases the overall throughput and efficiency. Smart mode is the default mode.

**Dummy Client**: For some cases, the clients can be required to connect to a single node instead of to each node in the cluster. Firewalls, security, or some custom networking issues can be the reason for these cases.

In dummy client mode, the client will only connect to one of the configured addresses. This single node will behave as a gateway to the other nodes. For any operation requested from the client, it will redirect the request to the relevant node and return the response back to the client returned from this node.

15.2.4 Failure Handling

There are two main failure cases you should be aware of, and configurations you can perform to achieve proper behavior.

15.2.4.1 Client Connection Failure

While the client is trying to connect initially to one of the members in the `ClientNetworkConfig.addressList`, all the members might be not available. Instead of giving up, throwing an exception and stopping the client, the client will retry as many as `connectionAttemptLimit` times. Please see the Connection Attempt Limit section.

The client executes each operation through the already established connection to the cluster. If this connection(s) disconnects or drops, the client will try to reconnect as configured.

15.2.4.2 Retry-able Operation Failure

While sending the requests to related nodes, operation can fail due to various reasons. Read-only operations are retried by default. If you want to enable this for the other operations, set the `redoOperation` to `true`. Please see the Redo Operation section.

The number of retries is given with the property `hazelcast.client.request.retry.count` in `ClientProperties`. The client will resend the request as many as RETRY-COUNT, then it will throw an exception. Please see the Client System Properties section.

15.2.5 Supported Distributed Data Structures

Most of the Distributed Data Structures are supported by the client. Please check for the exceptions for the clients in other languages.

As a general rule, you configure these data structures on the server side and access them through a proxy on the client side.

**Map**

You can use any Distributed Map object with the client, as shown below.

```java
Imap<Integer, String> map = client.getMap("myMap");
map.put(1, "Ali");
String value = map.get(1);
map.remove(1);
```
Locality is ambiguous for the client, so `addEntryListener` and `localKeySet` are not supported. Please see the Distributed Map section for more information.

**MultiMap:**

A MultiMap usage example is shown below.

```java
MultiMap<Integer, String> multiMap = client.getMultiMap("myMultiMap");
multiMap.put(1,"ali");
multiMap.put(1,"veli");
Collection<String> values = multiMap.get(1);
```

`addEntryListener`, `localKeySet` and `getLocalMultiMapStats` are not supported because locality is ambiguous for the client. Please see the Distributed MultiMap section for more information.

**Queue:**

A sample usage is shown below.

```java
IQueue<String> myQueue = client.getQueue("theQueue");
myQueue.offer("ali")
```

`getLocalQueueStats` is not supported because locality is ambiguous for the client. Please see the Distributed Queue section for more information.

**Topic:**

`getLocalTopicStats` is not supported because locality is ambiguous for the client.

**Other Supported Distributed Structures:**

The distributed data structures listed below are also supported by the client. Since their logic is the same in both the node side and client side, you can refer to their sections as listed below.

- Replicated Map
- MapReduce
- List
- Set
- IAtomicLong
- IAtomicReference
- ICountDownLatch
- ISemaphore
- IdGenerator
- Lock

### 15.2.6 Client Services

Below services are provided for some common functionalities on the client side.

**Distributed Executor Service:**

The distributed executor service is for distributed computing. It can be used to execute tasks on the cluster on a designated partition or on all the partitions. It can also be used to process entries. Please see the Distributed Executor Service section for more information.

```java
IExecutorService executorService = client.getExecutorService("default");
```
After getting an instance of `IExecutorService`, you can use the instance as the interface with the one provided on the server side. Please see the Distributed Computing chapter for detailed usage.

**NOTE:** This service is only supported by the Java client.

**Client Service:**

If you need to track clients and you want to listen to their connection events, you can use the `clientConnected` and `clientDisconnected` methods of the `ClientService` class. This class must be run on the node side. The following is an example code.

```java
final ClientService clientService = hazelcastInstance.getClientService();
final Collection<Client> connectedClients = clientService.getConnectedClients();

clientService.addClientListener(new ClientListener() {
    @Override
    public void clientConnected(Client client) {
        //Handle client connected event
    }

    @Override
    public void clientDisconnected(Client client) {
        //Handle client disconnected event
    }
});
```

**Partition Service:**

You use partition service to find the partition of a key. It will return all partitions. See the example code below.

```java
PartitionService partitionService = client.getPartitionService();

//partition of a key
Partition partition = partitionService.getPartition(key);

//all partitions
Set<Partition> partitions = partitionService.getPartitions();
```

**Lifecycle Service:**

Lifecycle handling performs the following:

- checks to see if the client is running,
- shuts down the client gracefully,
- terminates the client ungracefully (forced shutdown), and
- adds/removes lifecycle listeners.

```java
LifecycleService lifecycleService = client.getLifecycleService();

if(lifecycleService.isRunning()){
    //it is running
}

//shutdown client gracefully
lifecycleService.shutdown();
```
15.2.7 Client Listeners

You can configure listeners to listen to various event types on the client side. You can configure global events not relating to any distributed object through Client ListenerConfig. You should configure distributed object listeners like map entry listeners or list item listeners through their proxies. You can refer to the related sections under each distributed data structure in this reference manual.

15.2.8 Client Transactions

Transactional distributed objects are supported on the client side. Please see the Transactions chapter on how to use them.

15.3 Java Client Configuration

Hazelcast Java Client can be configured declaratively (XML) or programmatically (API).

For declarative configuration, the Hazelcast client looks into the following places for the client configuration file:

- **System property**: The client first checks if `hazelcast.client.config` system property is set to a file path, e.g. `-Dhazelcast.client.config=C:/myhazelcast.xml`.
- **Classpath**: If config file is not set as a system property, the client checks the classpath for `hazelcast-client.xml` file.

If the client does not find any configuration file, it starts with the default configuration (`hazelcast-client-default.xml`) located in the `hazelcast-client.jar` library. Before configuring the client, please try to work with the default configuration to see if it works for you. Default should be just fine for most of the users. If not, then consider custom configuration for your environment.

If you want to specify your own configuration file to create a Config object, the Hazelcast client supports the following.

- Config `cfg = new XmlClientConfigBuilder(xmlFileName).build();`
- Config `cfg = new XmlClientConfigBuilder(inputStream).build();`

For programmatic configuration of the Hazelcast Java Client, just instantiate a ClientConfig object and configure the desired aspects, a sample of which is shown below.

```java
ClientConfig clientConfig = new ClientConfig();
clientConfig.setGroupConfig(new GroupConfig("dev","dev-pass");
clientConfig.setLoadBalancer(yourLoadBalancer);
... ...
```

15.3.1 Client Network Configuration

All network related configuration of Hazelcast Java Client is performed via the network element in the declarative configuration file or the class ClientNetworkConfig when using programmatic configuration. Let's first give the examples for these two approaches. Then we will look at its sub-elements and attributes.

Declarative:
...<network>
  <cluster-members>
    <address>127.0.0.1</address>
    <address>127.0.0.2</address>
  </cluster-members>
  <smart-routing>true</smart-routing>
  <redo-operation>true</redo-operation>
  <socket-interceptor enabled="true">
    <class-name>com.hazelcast.XYZ</class-name>
    <properties>
      <property name="kerberos-host">kerb-host-name</property>
      <property name="kerberos-config-file">kerb.conf</property>
    </properties>
  </socket-interceptor>
  <aws enabled="true" connection-timeout-seconds="11">
    <inside-aws>false</inside-aws>
    <access-key>my-access-key</access-key>
    <secret-key>my-secret-key</secret-key>
    <iam-role>s3access</iam-role>
    <region>us-west-1</region>
    <host-header>ec2.amazonaws.com</host-header>
    <security-group-name>hazelcast-sg</security-group-name>
    <tag-key>type</tag-key>
    <tag-value>hz-nodes</tag-value>
  </aws>
</network>

Programmatic:

```java
ClientConfig clientConfig = new ClientConfig();
ClientNetworkConfig networkConfig = clientConfig.getNetworkConfig();
```

### 15.3.1.1 Address List

Address List is the initial list of cluster addresses to which the client will connect. The client uses this list to find an alive node. Although it may be enough to give only one address of a node in the cluster (since all nodes communicate with each other), it is recommended that you give all the nodes’ addresses.

Declarative:

```xml
<hazelcast-client>
  ...
  <network>
    <cluster-members>
      <address>10.1.1.21</address>
      <address>10.1.1.22:5703</address>
    </cluster-members>
  ...
  </network>
  ...
</hazelcast-client>
```

Programmatic:
ClientConfig clientConfig = new ClientConfig();
ClientNetworkConfig networkConfig = clientConfig.getNetworkConfig();
networkConfig().addAddress("10.1.1.21", "10.1.1.22:5703");

If the port part is omitted, then 5701, 5702, and 5703 will be tried in random order.
You can provide multiple addresses with ports provided or not as seen above. The provided list is shuffled and tried in random order. Default value is localhost.

15.3.1.2 Smart Routing

It defines whether the client mode is smart or dummy. The following are the example configurations.

Declarative:

```xml
...<network>...
  <smart-routing>true</smart-routing>
...</network>
```

Programmatic:

```java
ClientConfig clientConfig = new ClientConfig();
ClientNetworkConfig networkConfig = clientConfig.getNetworkConfig();
networkConfig().setSmartRouting(true);
```

The default is smart client mode.

15.3.1.3 Redo Operation

It enables/disables redo-able operations as described in Retry-able Operation Failure. The following are the example configurations.

Declarative:

```xml
...<network>...
  <redo-operation>true</redo-operation>
...</network>
```

Programmatic:

```java
ClientConfig clientConfig = new ClientConfig();
ClientNetworkConfig networkConfig = clientConfig.getNetworkConfig();
networkConfig().setRedoOperation(true);
```

Default is disabled.
15.3.1.4 Connection Timeout

It is the timeout value in milliseconds for nodes to accept client connection requests. The following are the example configurations.

**Declarative:**

```xml
...<network>
  ...<connection-timeout>5000</connection-timeout>
...</network>
```

**Programmatic:**

```java
ClientConfig clientConfig = new ClientConfig();
clientConfig.getNetworkConfig().setConnectionTimeout(5000);
```

The default value is 5000 milliseconds.

15.3.1.5 Connection Attempt Limit

While the client is trying to connect initially to one of the members in the `ClientNetworkConfig.addressList`, all members might be not available. Instead of giving up, throwing an exception and stopping the client, the client will retry as many as `ClientNetworkConfig.connectionAttemptLimit` times. The following are the example configurations.

**Declarative:**

```xml
...<network>
  ...<connection-attempt-limit>5</connection-attempt-limit>
...</network>
```

**Programmatic:**

```java
ClientConfig clientConfig = new ClientConfig();
clientConfig.getNetworkConfig().setConnectionAttemptLimit(5);
```

Default value is 2.

15.3.1.6 Connection Attempt Period

It is the duration in milliseconds between the connection attempts defined by `ClientNetworkConfig.connectionAttemptPeriod`. The following are the example configurations.

**Declarative:**

```xml
...<network>
  ...<connection-attempt-period>5000</connection-attempt-period>
...</network>
```
15.3. JAVA CLIENT CONFIGURATION

Programmatic:

ClientConfig clientConfig = new ClientConfig();
clientConfig.getNetworkConfig().setConnectionAttemptPeriod(5000);

Default value is 3000.

15.3.1.7 Client Socket Interceptor

Following is a client configuration to set a socket interceptor. Any class implementing `com.hazelcast.nio.SocketInterceptor` is a socket Interceptor.

```java
public interface SocketInterceptor {
    void init(Properties properties);
    void onConnect(Socket connectedSocket) throws IOException;
}
```

SocketInterceptor has two steps. First, it will be initialized by the configured properties. Second, it will be informed just after the socket is connected using `onConnect`.

```java
SocketInterceptorConfig socketInterceptorConfig = clientConfig
    .getNetworkConfig().getSocketInterceptorConfig();
MyClientSocketInterceptor myClientSocketInterceptor = new MyClientSocketInterceptor();
socketInterceptorConfig.setEnabled(true);
socketInterceptorConfig.setImplementation(myClientSocketInterceptor);
```

If you want to configure the socket connector with a class name instead of an instance, see the example below.

```java
SocketInterceptorConfig socketInterceptorConfig = clientConfig
    .getNetworkConfig().getSocketInterceptorConfig();
MyClientSocketInterceptor myClientSocketInterceptor = new MyClientSocketInterceptor();
socketInterceptorConfig.setEnabled(true);
socketInterceptorConfig.setProperty("kerberos-host", "kerb-host-name");
socketInterceptorConfig.setProperty("kerberos-config-file", "kerb.conf");
```

RELATED INFORMATION

Please see the Socket Interceptor section for more information.

15.3.1.8 Client Socket Options

You can configure the network socket options using `SocketOptions`. It has the following methods.

- `socketOptions.setKeepAlive(x)`: Enables/disables the `SO_KEEPALIVE` socket option. The default value is `true`. 
• socketOptions.setTcpNoDelay(x): Enables/disables the TCP_NODELAY socket option. The default value is true.

• socketOptions.setReuseAddress(x): Enables/disables the SO_REUSEADDR socket option. The default value is true.

• socketOptions.setLingerSeconds(x): Enables/disables SO_LINGER with the specified linger time in seconds. The default value is 3.

• socketOptions.setBufferSize(x): Sets the SO_SNDBUF and SO_RCVBUF options to the specified value in KB for this Socket. The default value is 32.

SocketOptions socketOptions = clientConfig.getNetworkConfig().getSocketOptions();
socketOptions.setBufferSize(32);
socketOptions.setKeepAlive(true);
socketOptions.setTcpNoDelay(true);
socketOptions.setReuseAddress(true);
socketOptions.setLingerSeconds(3);

15.3.1.9 Client SSL

You can use SSL to secure the connection between the client and the nodes. If you want SSL enabled for the client-cluster connection, you should set SSLConfig. Once set, the connection (socket) is established out of an SSL factory defined either by a factory class name or factory implementation. Please see the SSLConfig class in the com.hazelcast.config package at the JavaDocs page of the Hazelcast Documentation web site.

15.3.1.10 Client Configuration for AWS

The example declarative and programmatic configurations below show how to configure a Java client for connecting to a Hazelcast cluster in AWS.

Declarative:

...<network>
  <aws enabled="true">
    <inside-aws>false</inside-aws>
    <access-key>my-access-key</access-key>
    <secret-key>my-secret-key</secret-key>
    <iam-role>s3access</iam-role>
    <region>us-west-1</region>
    <host-header>ec2.amazonaws.com</host-header>
    <security-group-name>hazelcast-sg</security-group-name>
    <tag-key>type</tag-key>
    <tag-value>hz-nodes</tag-value>
  </aws>
...<network>

Programmatic:

ClientConfig clientConfig = new ClientConfig();
ClientAwsConfig clientAwsConfig = new ClientAwsConfig();
clientAwsConfig.setInsideAws( false )
15.3. JAVA CLIENT CONFIGURATION

```java
.setAccessKey( "my-access-key" )
.setSecretKey( "my-secret-key" )
.setRegion( "us-west-1" )
.setHostHeader( "ec2.amazonaws.com" )
.setSecurityGroupName( "hazelcast-sg" )
.setTagKey( "type" )
.setTagValue( "hz-nodes" )
.setIamRole( "s3access" );
```

```java
clientConfig.getNetworkConfig().setAwsConfig( clientAwsConfig );
```

```
HazelcastInstance client = HazelcastClient.newHazelcastClient( clientConfig );
```

**NOTE:** If the `inside-aws*` parameter is not set, the private addresses of nodes will always be converted to public addresses. Also, the client will use public addresses to connect to the nodes. In order to use private addresses, set the `inside-aws` parameter to `true`. Also note that, when connecting outside from AWS, setting the `inside-aws` parameter to `true` will cause the client to not be able to reach the nodes.*

IAM roles are used to make secure requests from your clients. You can provide the name of your IAM role that you created previously on your AWS console using the `iam-role` or `setIamRole()` method.

### 15.3.2 Client Load Balancer Configuration

`LoadBalancer` allows you to send operations to one of a number of endpoints (Members). Its main purpose is to determine the next `Member` if queried. It is up to your implementation to use different load balancing policies. You should implement the interface `com.hazelcast.client.LoadBalancer` for that purpose.

If the client is configured in smart mode, only the operations that are not key-based will be routed to the endpoint that is returned by the `LoadBalancer`. If the client is not a smart client, `LoadBalancer` will be ignored.

The following are the example configurations.

**Declarative:**

```xml
<hazelcast-client>
...
<load-balancer type="random">
  yourLoadBalancer
</load-balancer>
...
</hazelcast-client>
```

**Programmatic:**

```java
ClientConfig clientConfig = new ClientConfig();
clientConfig.setLoadBalancer(yourLoadBalancer);
```

### 15.3.3 Client Near Cache Configuration

Hazelcast distributed map has a Near Cache feature to reduce network latencies. Since the client always requests data from the cluster nodes, it can be helpful for some of your use cases to configure a near cache on the client side. The client supports the same Near Cache that is used in Hazelcast distributed map.

You can create Near Cache on the client side by providing a configuration per map name, as shown below.

```java
ClientConfig clientConfig = new ClientConfig();
CacheConfig nearCacheConfig = new NearCacheConfig();
nearCacheConfig.setName("mapName");
clientConfig.addNearCacheConfig(nearCacheConfig);
```
You can use wildcards for the map name, as shown below.

```java
nearCacheConfig.setName("map*");
nearCacheConfig.setName("*map");
```

And, the following is an example declarative configuration for Near Cache.

```xml
<hazelcast-client>
  ...
  ...
  <near-cache name="MENU">
    <max-size>2000</max-size>
    <time-to-live-seconds>0</time-to-live-seconds>
    <max-idle-seconds>0</max-idle-seconds>
    <eviction-policy>LFU</eviction-policy>
    <invalidate-on-change>true</invalidate-on-change>
    <in-memory-format>OBJECT</in-memory-format>
  </near-cache>
  ...
</hazelcast-client>
```

Name of Near Cache on client side must be the same as the name of IMap on server for which this Near Cache is being created.

Near Cache can have its own `in-memory-format` which is independent of the `in-memory-format` of the servers.

### 15.3.4 Client Group Configuration

Clients should provide a group name and password in order to connect to the cluster. You can configure them using `GroupConfig`, as shown below.

```java
clientConfig.setGroupConfig(new GroupConfig("dev","dev-pass");
```

### 15.3.5 Client Security Configuration

In the cases where the security established with `GroupConfig` is not enough and you want your clients connecting securely to the cluster, you can use `ClientSecurityConfig`. This configuration has a `credentials` parameter to set the IP address and UID. Please see `ClientSecurityConfig.java` in our code.

### 15.3.6 Client Serialization Configuration

For the client side serialization, use Hazelcast configuration. Please refer to the Serialization chapter.

### 15.3.7 Client Listener Configuration

You can configure global event listeners using `ListenerConfig` as shown below.

```java
ClientConfig clientConfig = new ClientConfig();
ListenerConfig listenerConfig = new ListenerConfig(LifecycleListenerImpl);
clientConfig.addListenerConfig(listenerConfig);
```

```java
ClientConfig clientConfig = new ClientConfig();
ListenerConfig listenerConfig = new ListenerConfig("com.hazelcast.example.MembershipListenerImpl");
clientConfig.addListenerConfig(listenerConfig);
```
15.5. SAMPLE CODES FOR CLIENT

You can add three types of event listeners.

- LifecycleListener
- MembershipListener
- DistributedObjectListener

**RELATED INFORMATION**

Please refer to Hazelcast JavaDocs and see LifecycleListener, MembershipListener and DistributedObjectListener in the `com.hazelcast.core` package.

15.3.8 ExecutorPoolSize

Hazelcast has an internal executor service (different from the data structure Executor Service) that has threads and queues to perform internal operations such as handling responses. This parameter specifies the size of the pool of threads which perform these operations laying in the executor’s queue. If not configured, this parameter has the value as `5 * core size of the client` (i.e. it is 20 for a machine that has 4 cores).

15.3.9 ClassLoader

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.

15.4 Client System Properties

There are some advanced client configuration properties to tune some aspects of Hazelcast Client. You can set them as property name and value pairs through declarative configuration, programmatic configuration, or JVM system property. Please see the System Properties section to learn how to set these properties.

The table below lists the client configuration properties with their descriptions.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hazelcast.client.event.queue.capacity</code></td>
<td>1000000</td>
<td>string</td>
<td>The default value of the capacity of executor</td>
</tr>
<tr>
<td><code>hazelcast.client.event.thread.count</code></td>
<td>5</td>
<td>string</td>
<td>The thread count for handling incoming event packets</td>
</tr>
<tr>
<td><code>hazelcast.client.heartbeat.interval</code></td>
<td>10000</td>
<td>string</td>
<td>The frequency of heartbeat messages sent by the client</td>
</tr>
<tr>
<td><code>hazelcast.client.heartbeat.timeout</code></td>
<td>300000</td>
<td>string</td>
<td>Timeout for the heartbeat messages sent by the client</td>
</tr>
<tr>
<td><code>hazelcast.client.invocation.timeout.seconds</code></td>
<td>120</td>
<td>string</td>
<td>Time to give up the invocation when a member is</td>
</tr>
<tr>
<td><code>hazelcast.client.shuffle.member.list</code></td>
<td>true</td>
<td>string</td>
<td>The client shuffles the given member list to prevent</td>
</tr>
</tbody>
</table>

15.5 Sample Codes for Client

Please refer to Client Code Samples.
Chapter 16

Other Client Implementations

This chapter describes the clients other than the Hazelcast Java Client.

16.1 C++ Client

You can use Native C++ Client to connect to Hazelcast nodes and perform almost all operations that a node can perform. Clients differ from nodes in that clients do not hold data. The C++ Client is by default a smart client, i.e. it knows where the data is and asks directly for the correct node. You can disable this feature (using the `ClientConfig::setSmart` method) if you do not want the clients to connect to every node.

The features of C++ Clients are:

- Access to distributed data structures (IMap, IQueue, MultiMap, ITopic, etc.).
- Access to transactional distributed data structures (TransactionalMap, TransactionalQueue, etc.).
- Ability to add cluster listeners to a cluster and entry/item listeners to distributed data structures.
- Distributed synchronization mechanisms with ILock, ISemaphore and ICountDownLatch.

16.1.1 How to Setup

Hazelcast C++ Client is shipped with 32/64 bit, shared and static libraries. You only need to include the boost `shared_ptr.hpp` header in your compilation since the API makes use of the boost `shared_ptr`.

The downloaded release folder consists of:

- Mac_64/
- Windows_32/
- Windows_64/
- Linux_32/
- Linux_64/
- docs/ *(HTML Doxygen documents are here)*

Each of the folders above contains the following:

- examples/
  - testApp.exe => example command line client tool to connect hazelcast servers.
  - TestApp.cpp => code of the example command line tool.
- hazelcast/
  - lib/ => Contains both shared and static library of hazelcast.
CHAPTER 16. OTHER CLIENT IMPLEMENTATIONS

- include/ => Contains headers of client.
- external/
  - include/ => Contains headers of dependencies. (boost::shared_ptr)

16.1.2 Platform Specific Installation Guides

The C++ Client is tested on Linux 32/64-bit, Mac 64-bit and Windows 32/64-bit machines. For each of the headers above, it is assumed that you are in the correct folder for your platform. Folders are Mac_64, Windows_32, Windows_64, Linux_32 or Linux_64.

16.1.2.1 Linux

For Linux, there are two distributions: 32 bit and 64 bit.

Here is an example script to build with static library:

```
g++ main.cpp -pthread -I./external/include -I./hazelcast/include ./hazelcast/lib/static/libHazelcastClientStatic_64.a
```

Here is an example script to build with shared library:

```
g++ main.cpp -lpthread -Wl,–no-as-needed -lrt -I./external/include -I./hazelcast/include -L./hazelcast/lib/shared -lHazelcastClientShared_64
```

16.1.2.2 Mac

For Mac, there is one distribution: 64 bit.

Here is an example script to build with static library:

```
g++ main.cpp -I./external/include -I./hazelcast/include ./hazelcast/lib/static/libHazelcastClientStatic_64.a
```

Here is an example script to build with shared library:

```
g++ main.cpp -I./external/include -I./hazelcast/include -L./hazelcast/lib/shared -lHazelcastClientShared_64
```

16.1.2.3 Windows

For Windows, there are two distributions; 32 bit and 64 bit.

16.1.3 Code Examples

A Hazelcast node should be running to make the code examples work.

**NOTE:** The license key should be provided in the configuration as `config->getGroupConfig().setLicenseKey(PROVIDED_ENTERPRISE_KEY);`

16.1.3.1 Map Example

```
#include <hazelcast/client/HazelcastAll.h>
#include <iostream>

using namespace hazelcast::client;

int main() {
    ClientConfig clientConfig;
    clientConfig->getGroupConfig().setLicenseKey(PROVIDED_ENTERPRISE_KEY);
    Address address("localhost", 5701);
```
clientConfig.addAddress( address );

HazelcastClient hazelcastClient( clientConfig );

IMap<int,int> myMap = hazelcastClient.getMap<int,int>( "myIntMap" );
myMap.put( 1,3 );
boost::shared_ptr<int> value = myMap.get( 1 );
if( value.get() != NULL ) {
    //process the item
}
return 0;

16.1.3.2 Queue Example

#include <hazelcast/client/HazelcastAll.h>
#include <iostream>
#include <string>
using namespace hazelcast::client;

int main() {
    ClientConfig clientConfig;
    clientConfig->getGroupConfig().setLicenseKey(PROVIDED.Enterprise_KEY);
    Address address( "localhost", 5701 );
    clientConfig.addAddress( address );

    HazelcastClient hazelcastClient( clientConfig );

    IQueue<std::string> queue = hazelcastClient.getQueue<std::string>( "q" );
    queue.offer( "sample" );
    boost::shared_ptr<std::string> value = queue.poll();
    if( value.get() != NULL ) {
        //process the item
    }
    return 0;
}

16.1.3.3 Entry Listener Example

#include "hazelcast/client/ClientConfig.h"
#include "hazelcast/client/EntryEvent.h"
#include "hazelcast/client/IMap.h"
#include "hazelcast/client/Address.h"
#include "hazelcast/client/HazelcastClient.h"
#include <iostream>
#include <string>
using namespace hazelcast::client;

class SampleEntryListener {

public:

    void entryAdded( EntryEvent<std::string, std::string> &event ) {
        std::cout << "entry added " << event.getKey() << " " << event.getValue() << std::endl;
    }

};
void entryRemoved( EntryEvent<std::string, std::string> &event ) {
    std::cout << "entry added " << event.getKey() << " "
            << event.getValue() << std::endl;
}

void entryUpdated( EntryEvent<std::string, std::string> &event ) {
    std::cout << "entry added " << event.getKey() << " "
            << event.getValue() << std::endl;
}

void entryEvicted( EntryEvent<std::string, std::string> &event ) {
    std::cout << "entry added " << event.getKey() << " "
            << event.getValue() << std::endl;
}

int main( int argc, char **argv ) {
    ClientConfig clientConfig;
    Address address( "localhost", 5701 );
    clientConfig.addAddress( address );
    HazelcastClient hazelcastClient( clientConfig );
    IMap<std::string, std::string> myMap = hazelcastClient
                             .getMap<std::string ,std::string>( "myIntMap" );
    SampleEntryListener * listener = new SampleEntryListener();
    std::string id = myMap.addEntryListener( *listener, true );
    // Prints entryAdded
    myMap.put( "key1", "value1" );
    // Prints updated
    myMap.put( "key1", "value2" );
    // Prints entryRemoved
    myMap.remove( "key1" );
    // Prints entryEvicted after 1 second
    myMap.put( "key2", "value2", 1000 );

    // WARNING: deleting listener before removing it from hazelcast leads to crashes.
    myMap.removeEntryListener( id );
    // Delete listener after remove it from hazelcast.
    delete listener;
    return 0;
};

16.1.3.4 Serialization Example

Assume that you have the following two classes in Java and you want to use them with a C++ client.

```java
class Foo implements Serializable {
    private int age;
    private String name;
}

class Bar implements Serializable {
```
16.1. C++ CLIENT

```cpp
private float x;
private float y;
}
```

First, let them implement Portable or IdentifiedDataSerializable as shown below.

```cpp
class Foo implements Portable {
    private int age;
    private String name;

    public int getFactoryId() {
        // a positive id that you choose
        return 123;
    }

    public int getClassId() {
        // a positive id that you choose
        return 2;
    }

    public void writePortable(PortableWriter writer) throws IOException {
        writer.writeUTF("n", name);
        writer.writeInt("a", age);
    }

    public void readPortable(PortableReader reader) throws IOException {
        name = reader.readUTF("n");
        age = reader.readInt("a");
    }
}

class Bar implements IdentifiedDataSerializable {
    private float x;
    private float y;

    public int getFactoryId() {
        // a positive id that you choose
        return 4;
    }

    public int getId() {
        // a positive id that you choose
        return 5;
    }

    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeFloat(x);
        out.writeFloat(y);
    }

    public void readData(ObjectDataInput in) throws IOException {
        x = in.readFloat();
        y = in.readFloat();
    }
}
```

Then, implement the corresponding classes in C++ with same factory and class ID as shown below.
class Foo : public Portable {
    public:
    int getFactoryId() const {
        return 123;
    }
    int getClassId() const {
        return 2;
    }
    void writePortable( serialization::PortableWriter &writer ) const {
        writer.writeUTF( "n", name );
        writer.writeInt( "a", age );
    }
    void readPortable( serialization::PortableReader &reader ) {
        name = reader.readUTF( "n" );
        age = reader.readInt( "a" );
    }
    private:
    int age;
    std::string name;
};

class Bar : public IdentifiedDataSerializable {
    public:
    int getFactoryId() const {
        return 4;
    }
    int getClassId() const {
        return 2;
    }
    void writeData( serialization::ObjectDataOutput& out ) const {
        out.writeFloat(x);
        out.writeFloat(y);
    }
    void readData( serialization::ObjectDataInput& in ) {
        x = in.readFloat();
        y = in.readFloat();
    }
    private:
    float x;
    float y;
};

Now, you can use the classes Foo and Bar in distributed structures. For example, use as Key or Value of IMap or as an Item in IQueue.

16.2 .NET Client

You can use the native .NET client to connect to Hazelcast nodes. All you need is to add HazelcastClient3x.dll into your .NET project references. The API is very similar to the Java native client.
.NET Client has the following distributed objects.

- `IMap<K, V>`
- `IMultiMap<K, V>`
- `IQueue<E>`
- `ITopic<E>`
- `ILHList<E>`
- `ILHSet<E>`
- `IIdGenerator`
- `ILock`
- `ISemaphore`
- `ICountDownLatch`
- `IAtomicLong`
- `ITransactionContext`

`ITransactionContext` can be used to obtain:

- `ITransactionalMap<K, V>`,
- `ITransactionalMultiMap<K, V>`,
- `ITransactionalList<E>`, and
- `ITransactionalSet<E>`.

At present the following features are not available as in the Java Client:

- Distributed Executor Service
- Replicated Map
- JCache

A code example is shown below.

```csharp
using Hazelcast.Client;
using System.Collections.Generic;

namespace Hazelcast.Client.Example
{
    public class SimpleExample
    {
        public static void Test()
        {
            var clientConfig = new ClientConfig();
            clientConfig.GetNetworkConfig().AddAddress( "10.0.0.1" );
            clientConfig.GetNetworkConfig().AddAddress( "10.0.0.2:5702" );

            IHazelcastInstance client = HazelcastClient.NewHazelcastClient( clientConfig );
            IMap<string, Customer> mapCustomers = client.GetMap<string, Customer>( "customers" );
        }
    }
}
```
mapCustomers.Put( "1", new Customer( "Joe", "Smith" ) );
mapCustomers.Put( "2", new Customer( "Ali", "Selam" ) );
mapCustomers.Put( "3", new Customer( "Avi", "Noyan" ) );

ICollection<Customer> customers = mapCustomers.Values();
foreach (var customer in customers)
{
    // process customer
}

public class MyPortableFactory : IPortableFactory
{
    public const int FactoryId = 1;

    public IPortable Create( int classId )
    {
        if ( Customer.Id == classId )
            return new Customer();
        else
            return null;
    }
}

public class Customer : IPortable
{
    private string name;
    private string surname;

    public const int Id = 5;

    public Customer( string name, string surname )
    {
        this.name = name;
        this.surname = surname;
    }

    public Customer() {}

    public int GetFactoryId()
    {
        return MyPortableFactory.FactoryId;
    }

    public int GetClassId()
    {
        return Id;
    }

    public void WritePortable( IPortableWriter writer )
    {
        writer.WriteUTF( "n", name );
        writer.WriteUTF( "s", surname );
    }

    public void ReadPortable( IPortableReader reader )
    {
        name = reader.ReadUTF( "n" );
    }
16.3. REST CLIENT

Hazelcast provides a REST interface, i.e. it provides an HTTP service in each node so that you can access your map and queue using HTTP protocol. Assuming mapName and queueName are already configured in your Hazelcast, its structure is shown below.

http://node IP address:port/hazelcast/rest/maps/mapName/key
http://node IP address:port/hazelcast/rest/queues/queueName

For the operations to be performed, standard REST conventions for HTTP calls are used.

Assume that your cluster members are as shown below.

Members [5] {
    Member [10.20.17.1:5701]
    Member [10.20.17.2:5701]
NOTE: All of the requests below can return one of the following responses in case of a failure.

- If the HTTP request syntax is not known, the following response will be returned.

HTTP/1.1 400 Bad Request
Content-Length: 0

- In case of an unexpected exception, the following response will be returned.

< HTTP/1.1 500 Internal Server Error
< Content-Length: 0

Creating/Updating Entries in a Map

You can put a new key1/value1 entry into a map by using POST call to http://10.20.17.1:5701/hazelcast/rest/maps/mapName/key1 URL. This call’s content body should contain the value of the key. Also, if the call contains the MIME type, Hazelcast stores this information, too.

A sample POST call is shown below.

$ curl -v -X POST -H "Content-Type: text/plain" -d "bar" http://10.20.17.1:5701/hazelcast/rest/maps/mapName/foo

It will return the following response if successful:

< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 0

Retrieving Entries from a Map

If you want to retrieve an entry, you can use a GET call to http://10.20.17.1:5701/hazelcast/rest/maps/mapName/key1. You can also retrieve this entry from another member of your cluster, such as http://10.20.17.3:5701/hazelcast/rest/maps/mapName/key1.

An example of a GET call is shown below.

$ curl -X GET http://10.20.17.3:5701/hazelcast/rest/maps/mapName/foo

It will return the following response if there is a corresponding value:

< HTTP/1.1 200 OK
< Content-Type: text/plain
< Content-Length: 3
  bar
This GET call returned a value, its length, and also the MIME type (text/plain) since the POST call example shown above included the MIME type.

It will return the following if there is no mapping for the given key:

```http
HTTP/1.1 204 No Content
Content-Length: 0
```

Removing Entries from a Map

You can use a DELETE call to remove an entry. A sample DELETE call is shown below with its response.

```
$ curl -v -X DELETE http://10.20.17.1:5701/hazelcast/rest/maps/mapName/foo
```

```http
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 0
```

If you leave the key empty as follows, DELETE will delete all entries from the map.

```
$ curl -v -X DELETE http://10.20.17.1:5701/hazelcast/rest/maps/mapName
```

```http
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 0
```

Offering Items on a Queue

You can use a POST call to create an item on the queue. A sample is shown below.

```
$ curl -v -X POST -H "Content-Type: text/plain" -d "foo"
http://10.20.17.1:5701/hazelcast/rest/queues/myEvents
```

The above call is equivalent to HazelcastInstance#getQueue("myEvents").offer("foo");.

It will return the following if successful:

```http
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 0
```

It will return the following if the queue is full and the item is not able to be offered to the queue:

```http
HTTP/1.1 503 Service Unavailable
Content-Length: 0
```

Retrieving Items from a Queue

You can use a DELETE call for retrieving items from a queue. Note that you should state the poll timeout while polling for queue events by an extra path parameter.

An example is shown below (10 being the timeout value).

```
$ curl -v -X DELETE http://10.20.17.1:5701/hazelcast/rest/queues/myEvents/10
```

The above call is equivalent to HazelcastInstance#getQueue("myEvents").poll(10, SECONDS);. Below is the response.
When the timeout is reached, the response will be No Content success, i.e. there is no item on the queue to be returned.

Getting the size of the queue

$ curl -v -X GET \http://10.20.17.1:5701/hazelcast/rest/queues/myEvents/size

The above call is equivalent to HazelcastInstance#getQueue("myEvents").size(); Below is a sample response.

RESTful access is provided through any member of your cluster. You can even put an HTTP load-balancer in front of your cluster members for load balancing and fault tolerance.

**NOTE**: You need to handle the failures on REST polls as there is no transactional guarantee.

### 16.4 Memcache Client

**NOTE**: Hazelcast Memcache Client only supports ASCII protocol. Binary Protocol is not supported.

A Memcache client written in any language can talk directly to a Hazelcast cluster. No additional configuration is required.

Assume that your cluster members are as shown below.

Members [5] {
 Member [10.20.17.1:5701]
 Member [10.20.17.2:5701]
 Member [10.20.17.4:5701]
 Member [10.20.17.3:5701]
 Member [10.20.17.5:5701]
}

Assume that you have a PHP application that uses PHP Memcache client to cache things in Hazelcast. All you need to do is have your PHP Memcache client connect to one of these members. It does not matter which member the client connects to because the Hazelcast cluster looks like one giant machine (Single System Image). Here is a PHP client code example.

```php
<?php
$memcache = new Memcache;
$memcache->connect( '10.20.17.1', 5701 ) or die ( "Could not connect" );
$memcache->set( 'key1', 'value1', 0, 3600 );
$get_result = $memcache->get( 'key1' ); // retrieve your data
var_dump( $get_result ); // show it
?>
```
Notice that Memcache client connects to 10.20.17.1 and uses port 5701. Here is a Java client code example with SpyMemcached client:

```java
MemcachedClient client = new MemcachedClient(
    AddrUtil.getAddresses( "10.20.17.1:5701 10.20.17.2:5701" ) );
client.set( "key1", 3600, "value1" );
System.out.println( client.get( "key1" ) );
```

If you want your data to be stored in different maps (e.g. to utilize per map configuration), you can do that with a map name prefix as in the following example code.

```java
MemcachedClient client = new MemcachedClient(
    AddrUtil.getAddresses( "10.20.17.1:5701 10.20.17.2:5701" ) );
client.set( "map1:key1", 3600, "value1" ); // store to hz_memcache_map1
client.set( "map2:key1", 3600, "value1" ); // store to hz_memcache_map2
System.out.println( client.get( "key1" ) ); // get from hz_memcache_map1
System.out.println( client.get( "key2" ) ); // get from hz_memcache_map2
```

`hz_memcache_prefix_` separates Memcache maps from Hazelcast maps. If no map name is given, it will be stored in a default map named `hz_memcache_default`.

An entry written with a Memcache client can be read by another Memcache client written in another language.

### 16.4.1 Unsupported Operations

- CAS operations are not supported. In operations that get CAS parameters, such as append, CAS values are ignored.
- Only a subset of statistics are supported. Below is the list of supported statistic values.

- `cmd_set`
- `cmd_get`
- `incr_hits`
- `incr_misses`
- `decr_hits`
- `decr_misses`
Chapter 17

Serialization

17.1 Serialization Overview

You need to serialize the Java objects that you put into Hazelcast because Hazelcast is a distributed system. The data and its replicas are stored in different partitions on multiple nodes. The data you need may not be present on the local machine, and in that case, Hazelcast retrieves that data from another machine. This requires serialization.

Hazelcast serializes all your objects into an instance of `com.hazelcast.nio.serialization.Data`. Data is the binary representation of an object.

Serialization is used when:

- key/value objects are added to a map,
- items are put in a queue/set/list,
- a runnable is sent using an executor service,
- an entry processing is performed within a map,
- an object is locked, and
- a message is sent to a topic.

Hazelcast optimizes the serialization for the below types. You cannot override this behavior.

![Figure 17.1: image](image)

Hazelcast also optimizes the following types. However, you can override these types by creating a custom serializer and registering it. See Custom Serialization for more information.

Hazelcast optimizes all of the above object types. You do not need to worry about their (de)serializations.

17.2 Serialization Interfaces

For complex objects, the following interfaces are used for serialization and deserialization.
Serializability and Externalizability

A class often needs to implement the `java.io.Serializable` interface; native Java serialization is the easiest way to do serialization. Let’s take a look at the example code below.

```java
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;
    private String surname;
    ...
}
```

## Serialization Interface

<table>
<thead>
<tr>
<th>Serialization Interface</th>
<th>Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serializable</td>
<td>- A standard and basic Java interface - Requires no implementation</td>
</tr>
<tr>
<td>Externalizable</td>
<td>- A standard Java interface - More CPU and memory usage efficient than Serializable</td>
</tr>
<tr>
<td>DataSerializable</td>
<td>- More CPU and memory usage efficient than Serializable</td>
</tr>
<tr>
<td>IdentifiedDataSerializable</td>
<td>- More CPU and memory usage efficient than Serializable - Reflection is not used during deserialization</td>
</tr>
<tr>
<td>Portable</td>
<td>- More CPU and memory usage efficient than Serializable - Reflection is not used during deserialization</td>
</tr>
<tr>
<td>Custom Serialization</td>
<td>- Does not require class to implement an interface - Convenient and flexible - Can be based on</td>
</tr>
</tbody>
</table>

Let’s dig into the details of the above serialization mechanisms in the following sections.
Here, the fields that are non-static and non-transient are automatically serialized. To eliminate class compatibility issues, it is recommended that you add a `serialVersionUID`, as shown above. Also, when you are using methods that perform byte-content comparisons (e.g. `IMap.replace()`) and if byte-content of equal objects is different, you may face unexpected behaviors. Therefore, if the class relies on, for example, a hash map, `replace` method may fail. The reason for this is the hash map is a serialized data structure with unreliable byte-content.

Hazelcast also supports `java.io.Externalizable`. This interface offers more control on the way fields are serialized or deserialized. Compared to native Java serialization, it also can have a positive effect on performance. With `java.io.Externalizable`, there is no need to add `serialVersionUID`.

Let’s take a look at the example code below.

```java
public class Employee implements Externalizable {
    private String surname;
    public Employee(String surname) {
        this.surname = surname;
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        this.surname = in.readUTF();
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeUTF(surname);
    }
}
```

Writing and reading of fields are performed explicitly. Note that reading should be performed in the same order as writing.

### 17.5 DataSerializable

As mentioned in the Serializable & Externalizable section, Java serialization is an easy mechanism. However, we do not have a control on how fields are serialized or deserialized. Moreover, this mechanism can lead to excessive CPU loads since it keeps track of objects to handle the cycles and streams class descriptors. These are performance decreasing factors; thus, serialized data may not have an optimal size.

The `DataSerializable` interface of Hazelcast overcomes these issues. Here is an example of a class implementing the `com.hazelcast.nio.serialization.DataSerializable` interface.

```java
public class Address implements DataSerializable {
    private String street;
    private int zipCode;
    private String city;
    private String state;

    public Address() {} {
        //getters setters...
    }
}
```
public void writeData( ObjectDataOutput out ) throws IOException {
    out.writeUTF(street);
    out.writeInt(zipCode);
    out.writeUTF(city);
    out.writeUTF(state);
}

public void readData( ObjectDataInput in ) throws IOException {
    street = in.readUTF();
    zipCode = in.readInt();
    city = in.readUTF();
    state = in.readUTF();
}
}

Let’s take a look at another example which encapsulates a DataSerializable field.

public class Employee implements DataSerializable {
    private String firstName;
    private String lastName;
    private int age;
    private double salary;
    private Address address; // address itself is DataSerializable

    public Employee() {}

    // getters setters...

    public void writeData( ObjectDataOutput out ) throws IOException {
        out.writeUTF(firstName);
        out.writeUTF(lastName);
        out.writeInt(age);
        out.writeDouble(salary);
        address.writeData(out);
    }

    public void readData( ObjectDataInput in ) throws IOException {
        firstName = in.readUTF();
        lastName = in.readUTF();
        age = in.readInt();
        salary = in.readDouble();
        address = new Address(); // since Address is DataSerializable let it read its own internal state
        address.readData(in);
    }
}

As you can see, since address field itself is DataSerializable, it is calling address.writeData(out) when writing and address.readData(in) when reading. Also note that, the order of writing and reading fields should be the same. While Hazelcast serializes a DataSerializable, it writes the className first. When Hazelcast de-serializes it, className is used to instantiate the object using reflection.

NOTE: Since Hazelcast needs to create an instance during deserialization, DataSerializable class has a no-arg constructor.

NOTE: DataSerializable is a good option if serialization is only needed for in-cluster communication.
17.5. DATASERIALIZABLE

17.5.1 IdentifiedDataSerializable

For a faster serialization of objects, avoiding reflection and long class names, Hazelcast recommends you implement `com.hazelcast.nio.serialization.IdentifiedDataSerializable` which is a slightly better version of `DataSerializable`.

`DataSerializable` uses reflection to create a class instance, as mentioned in the `DataSerializable` section. But, `IdentifiedDataSerializable` uses a factory for this purpose and it is faster during deserialization which requires new instance creations.

`IdentifiedDataSerializable` extends `DataSerializable` and introduces two new methods.

- int getId();
- int getFactoryId();

`IdentifiedDataSerializable` uses `getId()` instead of class name, and it uses `getFactoryId()` to load the class when given the Id. To complete the implementation, `com.hazelcast.nio.serialization.DataSerializableFactory` should also be implemented and registered into `SerializationConfig` which can be accessed from `Config.getSerializationConfig()`. Factory’s responsibility is to return an instance of the right `IdentifiedDataSerializable` object, given the Id. So far this is the most efficient way of Serialization that Hazelcast supports off the shelf.

Let’s take a look at the example code below and configuration to see `IdentifiedDataSerializable` in action.

```java
public class Employee  
  implements IdentifiedDataSerializable {

  private String surname;

  public Employee() {}  

  public Employee( String surname ) {  
    this.surname = surname;
  }

  @Override  
  public void readData( ObjectDataInput in )  
    throws IOException {
    this.surname = in.readUTF();
  }

  @Override  
  public void writeData( ObjectDataOutput out )  
    throws IOException {
    out.writeUTF( surname );
  }

  @Override  
  public int getFactoryId() {
    return EmployeeDataSerializableFactory.FACTORY_ID;
  }

  @Override  
  public int getId() {
    return EmployeeDataSerializableFactory.EMPLOYEE_TYPE;
  }

  @Override  
  public String toString() {
```
The methods `getId` and `getFactoryId` return a unique positive number within the `EmployeeDataSerializableFactory`. Now, let’s create an instance of this `EmployeeDataSerializableFactory`.

```java
public class EmployeeDataSerializableFactory
    implements DataSerializableFactory{

    public static final int FACTORY_ID = 1;
    public static final int EMPLOYEE_TYPE = 1;

    @Override
    public IdentifiedDataSerializable create(int typeId) {
        if ( typeId == EMPLOYEE_TYPE ) {
            return new Employee();
        } else {
            return null;
        }
    }
}
```

The only method that should be implemented is `create`, as seen in the above example. It is recommended that you use a `switch-case` statement instead of multiple `if-else` blocks if you have a lot of subclasses. Hazelcast throws an exception if null is returned for `typeId`.

As the last step, you need to register `EmployeeDataSerializableFactory` declaratively (declare in the configuration file `hazelcast.xml`) as shown below. Note that `factory-id` has the same value of `FACTORY_ID` in the above code. This is crucial to enable Hazelcast to find the correct factory.

```xml
<hazelcast>
    ...
    <serialization>
        <data-serializable-factories>
            <data-serializable-factory
                factory-id="1">
                EmployeeDataSerializableFactory
            </data-serializable-factory>
        </data-serializable-factories>
    </serialization>
    ...
</hazelcast>
```

**RELATED INFORMATION**

Please refer to the Serialization Configuration section for a full description of Hazelcast Serialization configuration.

## 17.6 Portable

As an alternative to the existing serialization methods, Hazelcast offers a language/platform independent Portable serialization that has the following advantages:

- Supports multi-version of the same object type.
- Fetches individual fields without having to rely on reflection.
- Queries and indexing support without de-serialization and/or reflection.
In order to support these features, a serializedPortable object contains meta information like the version and the 
concrete location of the each field in the binary data. This way, Hazelcast navigates in the byte[] and de-serializes 
only the required field without actually de-serializing the whole object. This improves the Query performance.

With multi-version support, you can have two nodes where each of them have different versions of the same object. 
Hazelcast will store both meta information and use the correct one to serialize and de-serialize Portable objects 
depending on the node. This is very helpful when you are doing a rolling upgrade without shutting down the 
cluster.

Portable serialization is totally language independent and is used as the binary protocol between Hazelcast server 
and clients.

A sample Portable implementation of a Foo class would look like the following.

```java
public class Foo implements Portable{
    final static int ID = 5;

    private String foo;

    public String getFoo() {
        return foo;
    }

    public void setFoo( String foo ) {
        this.foo = foo;
    }

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

    @Override
    public int getClassId() {
        return ID;
    }

    @Override
    public void writePortable( PortableWriter writer )
        throws IOException {
        writer.writeUTF( "foo", foo );
    }

    @Override
    public void readPortable( PortableReader reader )
        throws IOException {
        foo = reader.readUTF( "foo" );
    }
}
```

Similar to IdentifiedDataSerializable, a Portable Class must provide classId and factoryId. The Factory 
object will create the Portable object given the classId.

An example Factory could be implemented as following:

```java
public class MyPortableFactory implements PortableFactory {

    @Override
    public Portable create( int classId ) {
        if ( Foo.ID == classId )
            return new Foo();
        else
```
The last step is to register the Factory to the SerializationConfig. Below are the programmatic and declarative configurations for this step.

```java
Config config = new Config();
cfg.getSerializationConfig().addPortableFactory( 1, new MyPortableFactory() );
```

```xml
<hazelcast>
  <serialization>
    <portable-version>0</portable-version>
    <portable-factories>
      <portable-factory factory-id="1">com.hazelcast.nio.serialization.MyPortableFactory</portable-factory>
    </portable-factories>
  </serialization>
</hazelcast>
```

Note that the id that is passed to the SerializationConfig is the same as the factoryId that the Foo class returns.

### 17.6.1 Versions

More than one version of the same class may need to be serialized and deserialized. For example, a client may have an older version of a class, and the node to which it is connected can have a newer version of the same class.

Portable serialization supports versioning. You can declare Version in the configuration file `hazelcast.xml` using the `portable-version` element, as shown below.

```xml
<serialization>
  <portable-version>1</portable-version>
  <portable-factories>
    <portable-factory factory-id="1">PortableFactoryImpl</portable-factory>
  </portable-factories>
</serialization>
```

You should consider the following when you perform versioning:

- It is important to change the version whenever an update is performed in the serialized fields of a class (e.g. increment the version).
- If a client performs a Portable deserialization on a field, and then that Portable is updated by removing that field on the cluster side, this may lead to a problem.
- Portable serialization does not use reflection and hence, fields in the class and in the serialized content are not automatically mapped. Field renaming is a simpler process. Also, since the class ID is stored, renaming the Portable does not lead to problems.
- Types of fields need to be updated carefully. Hazelcast performs basic type upgradings (e.g. `int` to `float`).
17.6.2 Null Portable Serialization

Be careful when serializing null portables. Hazelcast lazily creates a class definition of portable internally when the user first serializes. This class definition is stored and used later for deserializing that portable class. When the user tries to serialize a null portable when there is no class definition at the moment, Hazelcast throws an exception saying that `com.hazelcast.nio.serialization.HazelcastSerializationException: Cannot write null portable without explicitly registering class definition!`

There are two solutions to get rid of this exception. Either put a non-null portable class of the same type before any other operation, or manually register a class definition in serialization configuration as shown below.

```java
Config config = new Config();
final ClassDefinition classDefinition = new ClassDefinitionBuilder(Foo.factoryId, Foo.getClassId) .addUTFField("foo").build();
config.getSerializationConfig().addClassDefinition(classDefinition);
Hazelcast.newHazelcastInstance(config);
```

17.6.3 DistributedObject Serialization

Putting a DistributedObject (e.g. Hazelcast Semaphore, Queue, etc.) in a machine and getting it from another one is not a straightforward operation. Passing the ID and type of the DistributedObject can be a solution. For deserialization, you can get the object from HazelcastInstance. For instance, if your distributed object is an instance of IQueue, you can either use `HazelcastInstance.getQueue(id)` or `Hazelcast.getDistributedObject`.

You can use the `HazelcastInstanceAware` interface in the case of a deserialization of a Portable DistributedObject if it gets an ID to be looked up. HazelcastInstance is set after deserialization, so you first need to store the ID and then retrieve the DistributedObject using the `setHazelcastInstance` method.

**RELATED INFORMATION**

Please refer to the Serialization Configuration section for a full description of Hazelcast Serialization configuration.

17.7 Custom Serialization

Hazelcast lets you plug a custom serializer for serializing objects. You can use `StreamSerializer` and `ByteArraySerializer` interfaces for this purpose.

17.7.1 StreamSerializer

You can use a stream to serialize and deserialize data by using `StreamSerializer`. This is a good option for your own implementations. It can also be adapted to external serialization libraries like Kryo, JSON, and protocol buffers.

17.7.1.1 StreamSerializer Example 1

First, let’s create a simple object.

```java
public class Employee {
    private String surname;

    public Employee( String surname ) {
        this.surname = surname;
    }
}
```

Now, let’s implement StreamSerializer for `Employee` class.
In practice, classes may have many fields. Just make sure the fields are read in the same order as they are written. The type ID must be unique and greater than or equal to 1. Uniqueness of the type ID enables Hazelcast to determine which serializer will be used during deserialization.

As the last step, let’s register the `EmployeeStreamSerializer` in the configuration file `hazelcast.xml`, as shown below.

```xml
<serialization>
  <serializers>
    <serializer type-class="Employee" class-name="EmployeeStreamSerializer" />
  </serializers>
</serialization>
```

**NOTE:** `StreamSerializer` cannot be created for well-known types (e.g. `Long`, `String`) and primitive arrays. Hazelcast already registers these types.

### 17.7.1.2 StreamSerializer Example 2

Let’s take a look at another example implementing `StreamSerializer`.

```java
public class Foo {
    private String foo;

    public String getFoo() {
        return foo;
    }

    public void setFoo(String foo) {
        this.foo = foo;
    }
}
```
Assume that our custom serialization will serialize Foo into XML. First we need to implement a `com.hazelcast.nio.serialization.StreamSerializer`. A very simple one that uses `XMLEncoder` and `XMLDecoder` could look like the following:

```java
class FooXmlSerializer implements StreamSerializer<Foo> {
    @Override
    public int getTypeId() {
        return 10;
    }

    @Override
    public void write( ObjectDataOutput out, Foo object ) throws IOException {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        XMLEncoder encoder = new XMLEncoder( bos );
        encoder.writeObject( object );
        encoder.close();
        out.write( bos.toByteArray() );
    }

    @Override
    public Foo read( ObjectDataInput in ) throws IOException {
        InputStream inputStream = (InputStream) in;
        XMLDecoder decoder = new XMLDecoder( inputStream );
        return (Foo) decoder.readObject();
    }

    @Override
    public void destroy() {
    }
}
```

Note that `typeId` must be unique because Hazelcast will use it to look up the `StreamSerializer` while it deserializes the object. The last required step is to register the `StreamSerializer` to the `Configuration`. Below are the programmatic and declarative configurations for this step.

```java
SerializerConfig sc = new SerializerConfig()
    .setImplementation(new FooXmlSerializer())
    .setTypeClass(Foo.class);
Config config = new Config();
config.getSerializationConfig().addSerializerConfig(sc);
```

```xml
<hazelcast>
    <serialization>
        <serializers>
        </serializers>
    </serialization>
</hazelcast>
```

From now on, Hazelcast will use `FooXmlSerializer` to serialize Foo objects. This way one can write an adapter (StreamSerializer) for any Serialization framework and plug it into Hazelcast.

**RELATED INFORMATION**

Please refer to the Serialization Configuration section for a full description of Hazelcast Serialization configuration.
17.7.2 ByteArraySerializer

ByteArraySerializer exposes the raw ByteArray used internally by Hazelcast. It is a good option if the serialization library you are using deals with ByteArrays instead of streams.

Let’s implement ByteArraySerializer for the Employee class mentioned in the StreamSerializer section.

```java
public class EmployeeByteArraySerializer implements ByteArraySerializer<Employee> {

    @Override
    public void destroy () {
    }

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

    @Override
    public byte[] write( Employee object )
        throws IOException {
        return object.getName().getBytes();
    }

    @Override
    public Employee read( byte[] buffer )
        throws IOException {
        String surname = new String( buffer );
        return new Employee( surname );
    }
}
```

As usual, let’s register the EmployeeByteArraySerializer in the configuration file hazelcast.xml, as shown below.

```xml
<serialization>
    <serializers>
        <serializer type-class="Employee">EmployeeByteArraySerializer</serializer>
    </serializers>
</serialization>
```

**RELATED INFORMATION**

Please refer to the Serialization Configuration section for a full description of Hazelcast Serialization configuration.

17.8 HazelcastInstanceAware Interface

You can implement the HazelcastInstanceAware interface to access distributed objects for cases where an object is deserialized and needs access to HazelcastInstance.

Let’s implement it for the Employee class mentioned in the Custom Serialization section.

```java
public class Employee
    implements Serializable, HazelcastInstanceAware {

    private static final long serialVersionUID = 1L;
```
private String surname;
private transient HazelcastInstance hazelcastInstance;

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

@Override
public void setHazelcastInstance(HazelcastInstance hazelcastInstance) {
    this.hazelcastInstance = hazelcastInstance;
    System.out.println("HazelcastInstance set");
}

@Override
public String toString() {
    return String.format("Person(surname=%s)", surname);
}
}

After deserialization, the object is checked if it implements HazelcastInstanceAware and the method setHazelcastInstance is called. Notice the hazelcastInstance is transient. This is because this field should not be serialized.

It may be a good practice to inject a HazelcastInstance into a domain object (e.g. Employee in the above sample) when used together with Runnable/Callable implementations. These runnables/callables are executed by IExecutorService which sends them to another machine. And after a task is deserialized, run/call method implementations need to access HazelcastInstance.

We recommend you only to set the HazelcastInstance field while using setHazelcastInstance method and not to execute operations on the HazelcastInstance. Because, when HazelcastInstance is injected for a HazelcastInstanceAware implementation, it may not be up and running at the injection time.
Chapter 18

Management

This chapter provides information on managing and monitoring your Hazelcast cluster. It gives detailed instructions related to gathering statistics, monitoring via JMX protocol and managing the cluster with useful utilities. It also includes the usage explanations of Hazelcast Management Center.

18.1 Statistics API per Node

You can gather various statistics from your distributed data structures via Statistics API. Since the data structures are distributed in the cluster, the Statistics API provides statistics for the local portion (1/Number of Nodes) of data on each node.

18.1.1 Map Statistics

The IMap interface has a `getLocalMapStats()` method which returns a `LocalMapStats` object that holds local map statistics.

```java
HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
IMap<String, Customer> customers = hazelcastInstance.getMap( "customers" );
LocalMapStats mapStatistics = customers.getLocalMapStats();
System.out.println( "number of entries owned on this node = 
    + mapStatistics.getOwnedEntryCount() );
```

Below is the list of metrics that you can access via the `LocalMapStats` object.

```java
/**
 * Returns the number of entries owned by this member.
 */
long getOwnedEntryCount();

/**
 * Returns the number of backup entries hold by this member.
 */
long getBackupEntryCount();

/**
 * Returns the number of backups per entry.
 */
int getBackupCount();
```
* Returns memory cost (number of bytes) of owned entries in this member.
*/
long getOwnedEntryMemoryCost();

/**
 * Returns memory cost (number of bytes) of backup entries in this member.
 */
long getBackupEntryMemoryCost();

/**
 * Returns the creation time of this map on this member.
 */
long getCreationTime();

/**
 * Returns the last access (read) time of the locally owned entries.
 */
long getLastAccessTime();

/**
 * Returns the last update time of the locally owned entries.
 */
long getLastUpdateTime();

/**
 * Returns the number of hits (reads) of the locally owned entries.
 */
long getHits();

/**
 * Returns the number of currently locked locally owned keys.
 */
long getLockedEntryCount();

/**
 * Returns the number of entries that the member owns and are dirty (updated
 * but not persisted yet).
 * dirty entry count is meaningful when there is a persistence defined.
 */
long getDirtyEntryCount();

/**
 * Returns the number of put operations
 */
long getPutOperationCount();

/**
 * Returns the number of get operations
 */
long getGetOperationCount();

/**
 * Returns the number of Remove operations
 */
long getRemoveOperationCount();

/**
 * Returns the total latency of put operations. To get the average latency,
18.1. STATISTICS API PER NODE

long getTotalPutLatency();

/**
 * Returns the total latency of get operations. To get the average latency,
 * divide by number of gets
 */
long getTotalGetLatency();

/**
 * Returns the total latency of remove operations. To get the average latency,
 * divide by number of gets
 */
long getTotalRemoveLatency();

/**
 * Returns the maximum latency of put operations. To get the average latency,
 * divide by number of puts
 */
long getMaxPutLatency();

/**
 * Returns the maximum latency of get operations. To get the average latency,
 * divide by number of gets
 */
long getMaxGetLatency();

/**
 * Returns the maximum latency of remove operations. To get the average latency,
 * divide by number of gets
 */
long getMaxRemoveLatency();

/**
 * Returns the number of Events Received
 */
long getEventOperationCount();

/**
 * Returns the total number of Other Operations
 */
long getOtherOperationCount();

/**
 * Returns the total number of total operations
 */
long total();

/**
 * Cost of map & near cache & backup in bytes
 * todo in object mode object size is zero.
 */
long getHeapCost();

/**
 * Returns statistics related to the Near Cache.
 */
NearCacheStats getNearCacheStats();

18.1.1.1 Near Cache Statistics

You can access Near Cache statistics from the LocalMapStats object via the getNearCacheStats() method, which returns a NearCacheStats object.

HazelcastInstance node = Hazelcast.newHazelcastInstance();
IMap<String, Customer> customers = node.getMap("customers");
LocalMapStats mapStatistics = customers.getLocalMapStats();
NearCacheStats nearCacheStatistics = mapStatistics.getNearCacheStats();
System.out.println("near cache hit/miss ratio= " + nearCacheStatistics.getRatio());

Below is the list of metrics that you can access via the NearCacheStats object. This behavior applies to both client and node near caches.

/**< 
 * Returns the creation time of this NearCache on this member 
 */
long getCreationTime();

/**< 
 * Returns the number of entries owned by this member. 
 */
long getOwnedEntryCount();

/**< 
 * Returns memory cost (number of bytes) of entries in this cache. 
 */
long getOwnedEntryMemoryCost();

/**< 
 * Returns the number of hits (reads) of the locally owned entries. 
 */
long getHits();

/**< 
 * Returns the number of misses of the locally owned entries. 
 */
long getMisses();

/**< 
 * Returns the hit/miss ratio of the locally owned entries. 
 */
double getRatio();

18.1.2 Multimap Statistics

The MultiMap interface has a getLocalMultiMapStats() method which returns a LocalMultiMapStats object that holds local MultiMap statistics.

HazelcastInstance node = Hazelcast.newHazelcastInstance();
MultiMap<String, Customer> customers = node.getMultiMap("customers");
LocalMultiMapStats multiMapStatistics = customers.getLocalMultiMapStats();
System.out.println("last update time = " + multiMapStatistics.getLastUpdateTime());
Below is the list of metrics that you can access via the `LocalMultiMapStats` object.

```java
/**
 * Returns the number of entries owned by this member.
 */
long getOwnedEntryCount();

/**
 * Returns the number of backup entries hold by this member.
 */
long getBackupEntryCount();

/**
 * Returns the number of backups per entry.
 */
int getBackupCount();

/**
 * Returns memory cost (number of bytes) of owned entries in this member.
 */
long getOwnedEntryMemoryCost();

/**
 * Returns memory cost (number of bytes) of backup entries in this member.
 */
long getBackupEntryMemoryCost();

/**
 * Returns the creation time of this map on this member.
 */
long getCreationTime();

/**
 * Returns the last access (read) time of the locally owned entries.
 */
long getLastAccessTime();

/**
 * Returns the last update time of the locally owned entries.
 */
long getLastUpdateTime();

/**
 * Returns the number of hits (reads) of the locally owned entries.
 */
long getHits();

/**
 * Returns the number of currently locked locally owned keys.
 */
long getLockedEntryCount();

/**
 * Returns the number of entries that the member owns and are dirty (updated
 * but not persisted yet).
 * Dirty entry count is meaningful when a persistence is defined.
 */
long getDirtyEntryCount();
```
*/
* Returns the number of put operations
*/
long getPutOperationCount();

/**
 * Returns the number of get operations
 */
long getGetOperationCount();

/**
 * Returns the number of Remove operations
 */
long getRemoveOperationCount();

/**
 * Returns the total latency of put operations. To get the average latency,
 * divide by number of puts
 */
long getTotalPutLatency();

/**
 * Returns the total latency of get operations. To get the average latency,
 * divide by number of gets
 */
long getTotalGetLatency();

/**
 * Returns the total latency of remove operations. To get the average latency,
 * divide by number of gets
 */
long getTotalRemoveLatency();

/**
 * Returns the maximum latency of put operations. To get the average latency,
 * divide by number of puts
 */
long getMaxPutLatency();

/**
 * Returns the maximum latency of get operations. To get the average latency,
 * divide by number of gets
 */
long getMaxGetLatency();

/**
 * Returns the maximum latency of remove operations. To get the average latency,
 * divide by number of gets
 */
long getMaxRemoveLatency();

/**
 * Returns the number of Events Received
 */
long getEventOperationCount();

/**
18.1. STATISTICS API PER NODE

```java
// Returns the total number of Other Operations
long getOtherOperationCount();

/**
 * Returns the total number of total operations
 */
long total();

/**
 * Cost of map & near cache & backup in bytes
 * todo in object mode object size is zero.
 */
long getHeapCost();
```

### 18.1.3 Queue Statistics

The `IQueue` interface has a `getLocalQueueStats()` method which returns a `LocalQueueStats` object that holds local queue statistics.

```java
HazelcastInstance node = Hazelcast.newHazelcastInstance();
IQueue<Order> orders = node.getQueue( "orders" );
LocalQueueStats queueStatistics = orders.getLocalQueueStats();
System.out.println( "average age of items = "
   + queueStatistics.getAvgAge() );
```

Below is the list of metrics that you can access via the `LocalQueueStats` object.

```java
/**
 * Returns the number of owned items in this member.
 */
long getOwnedItemCount();

/**
 * Returns the number of backup items in this member.
 */
long getBackupItemCount();

/**
 * Returns the min age of the items in this member.
 */
long getMinAge();

/**
 * Returns the max age of the items in this member.
 */
long getMaxAge();

/**
 * Returns the average age of the items in this member.
 */
long getAvgAge();

/**
 * Returns the number of offer/put/add operations.
 * Offers returning false will be included.
 * #getRejectedOfferOperationCount can be used
 */
```
* to get the rejected offers.
*/
long getOfferOperationCount();

/**
 * Returns the number of rejected offers. Offer can be rejected because of max-size limit
 * on the queue.
 */
long getRejectedOfferOperationCount();

/**
 * Returns the number of poll/take/remove operations. Polls returning null (empty) will be included.
 * #getEmptyPollOperationCount can be used to get the number of polls returned null.
 */
long getPollOperationCount();

/**
 * Returns number of null returning poll operations. Poll operation might return null, if the queue is empty.
 */
long getEmptyPollOperationCount();

/**
 * Returns number of other operations
 */
long getOtherOperationsCount();

/**
 * Returns number of event operations
 */
long getEventOperationCount();

18.1.4 Topic Statistics

The ITopic interface has a getLocalTopicStats() method which returns a LocalTopicStats object that holds local topic statistics.

HazelcastInstance node = Hazelcast.newHazelcastInstance();
ITopic<Object> news = node.getTopic( "news" );
LocalTopicStats topicStatistics = news.getLocalTopicStats();
System.out.println( "number of publish operations = "
    + topicStatistics.getPublishOperationCount() );

Below is the list of metrics that you can access via the LocalTopicStats object.

 /**
 * Returns the creation time of this topic on this member
 */
long getCreationTime();

 /**
 * Returns the total number of published messages of this topic on this member
 */
long getPublishOperationCount();
18.2. JMX API PER NODE

Hazelcast members expose various management beans which include statistics about distributed data structures and the states of Hazelcast node internals.

The metrics are local to the nodes, i.e. they do not reflect cluster wide values.
You can find the JMX API definition below with descriptions and the API methods in parenthesis.

**Atomic Long (IAtomicLong)**

- Name (name)
- Current Value (currentValue)
- Set Value (set(v))
- Add value and Get (addAndGet(v))
- Compare and Set (compareAndSet(e,v))
- Decrement and Get (decrementAndGet())
- Get and Add (getAndAdd(v))
- Get and Increment (getAndIncrement())
- Get and Set (getAndSet(v))
- Increment and Get (incrementAndGet())
- Partition key (partitionKey)

**Atomic Reference (IAtomicReference)**

- Name (name)
- Partition key (partitionKey)

**Countdown Latch (ICountDownLatch)**

- Name (name)
- Current count (count)
- Countdown (countDown())
-Partition key (partitionKey)

**Executor Service (IExecutorService)**

- Local pending operation count (localPendingTaskCount)
- Local started operation count (localStartedTaskCount)
- Local completed operation count (localCompletedTaskCount)
- Local cancelled operation count (localCancelledTaskCount)
- Local total start latency (localTotalStartLatency)
- Local total execution latency (localTotalExecutionLatency)

**List (IList)**

- Name (name)
- Clear list (clear)
- Total added item count (totalAddedItemCount)
- Total removed item count (totalRemovedItemCount)

**Lock (ILock)**

- Name (name)
- Lock Object (lockObject)
- Partition key (partitionKey)

**Map (IMap)**

- Name (name)
18.2. JMX API PER NODE

- Size (size)
- Config (config)
- Owned entry count (localOwnedEntryCount)
- Owned entry memory cost (localOwnedEntryMemoryCost)
- Backup entry count (localBackupEntryCount)
- Backup entry cost (localBackupEntryMemoryCost)
- Backup count (localBackupCount)
- Creation time (localCreationTime)
- Last access time (localLastAccessTime)
- Last update time (localLastUpdateTime)
- Hits (localHits)
- Locked entry count (localLockedEntryCount)
- Dirty entry count (localDirtyEntryCount)
- Put operation count (localPutOperationCount)
- Get operation count (localGetOperationCount)
- Remove operation count (localRemoveOperationCount)
- Total put latency (localTotalPutLatency)
- Total get latency (localTotalGetLatency)
- Total remove latency (localTotalRemoveLatency)
- Max put latency (localMaxPutLatency)
- Max get latency (localMaxGetLatency)
- Max remove latency (localMaxRemoveLatency)
- Event count (localEventOperationCount)
- Other (keySet, entrySet etc..) operation count (localOtherOperationCount)
- Total operation count (localTotal)
- Heap Cost (localHeapCost)
- Total added entry count (totalAddedEntryCount)
- Total removed entry count (totalRemovedEntryCount)
- Total updated entry count (totalUpdatedEntryCount)
- Total evicted entry count (totalEvictedEntryCount)
- Clear (clear())
- Values (values(p))
- Entry Set (entrySet(p))

MultiMap (MultiMap)

- Name (name)
- Size (size)
- Owned entry count (localOwnedEntryCount)
- Owned entry memory cost (localOwnedEntryMemoryCost)
- Backup entry count (localBackupEntryCount)
- Backup entry cost (localBackupEntryMemoryCost)
- Backup count (localBackupCount)
- Creation time (localCreationTime)
- Last access time (localLastAccessTime)
- Last update time (localLastUpdateTime)
- Hits (localHits)
- Locked entry count (localLockedEntryCount)
- Put operation count (localPutOperationCount)
- Get operation count (localGetOperationCount)
- Remove operation count (localRemoveOperationCount)
- Total put latency (localTotalPutLatency)
- Total get latency (localTotalGetLatency)
- Total remove latency (localTotalRemoveLatency)
• Max put latency (localMaxPutLatency)
• Max get latency (localMaxGetLatency)
• Max remove latency (localMaxRemoveLatency)
• Event count (localEventOperationCount)
• Other (keySet, entrySet etc.) operation count (localOtherOperationCount)
• Total operation count (localTotal)
• Clear (clear())

Replicated Map (ReplicatedMap)

• Name (name)
• Size (size)
• Config (config)
• Owned entry count (localOwnedEntryCount)
• Creation time (localCreationTime)
• Last access time (localLastAccessTime)
• Last update time (localLastUpdateTime)
• Hits (localHits)
• Put operation count (localPutOperationCount)
• Get operation count (localGetOperationCount)
• Remove operation count (localRemoveOperationCount)
• Total put latency (localTotalPutLatency)
• Total get latency (localTotalGetLatency)
• Total remove latency (localTotalRemoveLatency)
• Max put latency (localMaxPutLatency)
• Max get latency (localMaxGetLatency)
• Max remove latency (localMaxRemoveLatency)
• Event count (localEventOperationCount)
• Replication event count (localReplicationEventCount)
• Other (keySet, entrySet etc.) operation count (localOtherOperationCount)
• Total operation count (localTotal)
• Total added entry count (totalAddedEntryCount)
• Total removed entry count (totalRemovedEntryCount)
• Total updated entry count (totalUpdatedEntryCount)
• Clear (clear())
• Values (values())
• Entry Set (entrySet())

Queue (IQueue)

• Name (name)
• Config (QueueConfig)
• Partition key (partitionKey)
• Owned item count (localOwnedItemCount)
• Backup item count (localBackupItemCount)
• Minimum age (localMinAge)
• Maximum age (localMaxAge)
• Average age (localAveAge)
• Offer operation count (localOfferOperationCount)
• Rejected offer operation count (localRejectedOfferOperationCount)
• Poll operation count (localPollOperationCount)
• Empty poll operation count (localEmptyPollOperationCount)
• Other operation count (localOtherOperationsCount)
• Event operation count (localEventOperationCount)
18.2. **JMX API PER NODE**

- Total added item count (totalAddedItemCount)
- Total removed item count (totalRemovedItemCount)
- Clear (clear())

**Semaphore (ISemaphore)**

- Name (name)
- Available permits (available)
- Partition key (partitionKey)
- Drain (drain())
- Shrink available permits by given number (reduce(v))
- Release given number of permits (release(v))

**Set (ISet)**

- Name (name)
- Partition key (partitionKey)
- Total added item count (totalAddedItemCount)
- Total removed item count (totalRemovedItemCount)
- Clear (clear())

**Topic (ITopic)**

- Name (name)
- Config (config)
- Creation time (localCreationTime)
- Publish operation count (localPublishOperationCount)
- Receive operation count (localReceiveOperationCount)
- Total message count (totalMessageCount)

**Hazelcast Instance (HazelcastInstance)**

- Name (name)
- Version (version)
- Build (build)
- Configuration (config)
- Configuration source (configSource)
- Group name (groupName)
- Network Port (port)
- Cluster-wide Time (clusterTime)
- Size of the cluster (memberCount)
- List of members (Members)
- Running state (running)
- Shutdown the member (shutdown())
  - Node (HazelcastInstance.Node)
    - Address (address)
    - Master address (masterAddress)
• Event Service (HazelcastInstance.EventService)
  – Event thread count (eventThreadCount)
  – Event queue size (eventQueueSize)
  – Event queue capacity (eventQueueCapacity)

• Operation Service (HazelcastInstance.OperationService)
  – Response queue size (responseQueueSize)
  – Operation executor queue size (operationExecutorQueueSize)
  – Running operation count (runningOperationsCount)
  – Remote operation count (remoteOperationCount)
  – Executed operation count (executedOperationCount)
  – Operation thread count (operationThreadCount)

• Proxy Service (HazelcastInstance.ProxyService)
  – Proxy count (proxyCount)

• Partition Service (HazelcastInstance.PartitionService)
  – Partition count (partitionCount)
  – Active partition count (activePartitionCount)
  – Cluster Safe State (isClusterSafe)
  – LocalMember Safe State (isLocalMemberSafe)

• Connection Manager (HazelcastInstance.ConnectionManager)
  – Client connection count (clientConnectionCount)
  – Active connection count (activeConnectionCount)
  – Connection count (connectionCount)

• Client Engine (HazelcastInstance.ClientEngine)
  – Client endpoint count (clientEndpointCount)

• System Executor (HazelcastInstance.ManagedExecutorService)
  – Name (name)
  – Work queue size (queueSize)
  – Thread count of the pool (poolSize)
  – Maximum thread count of the pool (maximumPoolSize)
  – Remaining capacity of the work queue (remainingQueueCapacity)
  – Is shutdown (isShutdown)
  – Is terminated (isTerminated)
  – Completed task count (completedTaskCount)

• Operation Executor (HazelcastInstance.ManagedExecutorService)
  – Name (name)
  – Work queue size (queueSize)
  – Thread count of the pool (poolSize)
  – Maximum thread count of the pool (maximumPoolSize)
  – Remaining capacity of the work queue (remainingQueueCapacity)
  – Is shutdown (isShutdown)
  – Is terminated (isTerminated)
  – Completed task count (completedTaskCount)

• Async Executor (HazelcastInstance.ManagedExecutorService)
18.3 Monitoring with JMX

You can monitor your Hazelcast members via the JMX protocol.

- Name (name)
- Work queue size (queueSize)
- Thread count of the pool (poolSize)
- Maximum thread count of the pool (maximumPoolSize)
- Remaining capacity of the work queue (remainingQueueCapacity)
- Is shutdown (isShutdown)
- Is terminated (isTerminated)
- Completed task count (completedTaskCount)

- **Scheduled Executor** (HazelcastInstance.ManagedExecutorService)
  - Name (name)
  - Work queue size (queueSize)
  - Thread count of the pool (poolSize)
  - Maximum thread count of the pool (maximumPoolSize)
  - Remaining capacity of the work queue (remainingQueueCapacity)
  - Is shutdown (isShutdown)
  - Is terminated (isTerminated)
  - Completed task count (completedTaskCount)

- **Client Executor** (HazelcastInstance.ManagedExecutorService)
  - Name (name)
  - Work queue size (queueSize)
  - Thread count of the pool (poolSize)
  - Maximum thread count of the pool (maximumPoolSize)
  - Remaining capacity of the work queue (remainingQueueCapacity)
  - Is shutdown (isShutdown)
  - Is terminated (isTerminated)
  - Completed task count (completedTaskCount)

- **Query Executor** (HazelcastInstance.ManagedExecutorService)
  - Name (name)
  - Work queue size (queueSize)
  - Thread count of the pool (poolSize)
  - Maximum thread count of the pool (maximumPoolSize)
  - Remaining capacity of the work queue (remainingQueueCapacity)
  - Is shutdown (isShutdown)
  - Is terminated (isTerminated)
  - Completed task count (completedTaskCount)

- **IO Executor** (HazelcastInstance.ManagedExecutorService)
  - Name (name)
  - Work queue size (queueSize)
  - Thread count of the pool (poolSize)
  - Maximum thread count of the pool (maximumPoolSize)
  - Remaining capacity of the work queue (remainingQueueCapacity)
  - Is shutdown (isShutdown)
  - Is terminated (isTerminated)
  - Completed task count (completedTaskCount)
• 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` (please refer to the System Properties section):
  – using Hazelcast configuration (API, XML, Spring).
  – or by setting the system property `-Dhazelcast.jmx=true`
  – or by programmatic configuration: `config.setProperty(GroupProperties.PROP_ENABLE_JMX, "true")`;

• If you want to use cluster-wide stats (referred as total) in List, Map, ReplicatedMap, Queue, Set and Topic: enable the Hazelcast property `hazelcast.jmx.detailed` (please refer to the System Properties section):
  – using Hazelcast configuration (API, XML, Spring).
  – or by setting the system property `-Dhazelcast.jmx=true`
  – or by programmatic configuration: `config.setProperty(GroupProperties.PROP_ENABLE_JMX_DETAILED, "true")`;

**NOTE:** Note that this might cause some performance overheads since underlying structure adds listeners to get these total statistics.

• Use jconsole, jvisualvm (with mbean plugin) or another JMX compliant monitoring tool.

### 18.4 Cluster Utilities

This section provides information on programmatic utilities you can use to listen to the cluster events, to check whether the cluster and/or members are safe before shutting down a member, and to define the minimum number of cluster members required for the cluster to remain up and running.

#### 18.4.1 Cluster Interface

Hazelcast allows you to register for membership events so you will be notified when members are added or removed. You can also get the set of cluster members.

```java
import com.hazelcast.core.*;

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
Cluster cluster = hazelcastInstance.getCluster();
cluster.addMembershipListener( new MembershipListener() {
    public void memberAdded( MembershipEvent membershipEvent ) {
        System.out.println( "MemberAdded " + membershipEvent );
    }

    public void memberRemoved( MembershipEvent membershipEvent ) {
        System.out.println( "MemberRemoved " + membershipEvent );
    }
} );

Member localMember = cluster.getLocalMember();
System.out.println ( "my inetAddress= " + localMember.getInetAddress() );
```
Set setMembers = cluster.getMembers();
for (Member member : setMembers) {
    System.out.println("isLocalMember " + member.localMember());
    System.out.println("member.inetaddress " + member.getInetAddress());
    System.out.println("member.port " + member.getPort());
}

RELATED INFORMATION
Please refer to the Membership Listener section for more information on membership events.

18.4.2 Member Attributes

You can define various member attributes on your Hazelcast members. You can use these member attributes to tag your members as your business logic requirements.

In order to define member attribute on a member you can either:

- provide MemberAttributeConfig to your Config object,
- or provide member attributes at runtime via attribute setter methods on the Member interface.

For example, you can tag your members with their CPU characteristics and you can route CPU intensive tasks to those CPU-rich members.

MemberAttributeConfig fourCore = new MemberAttributeConfig();
memberAttributeConfig.setIntAttribute("CPU_CORE_COUNT", 4);
MemberAttributeConfig twelveCore = new MemberAttributeConfig();
memberAttributeConfig.setIntAttribute("CPU_CORE_COUNT", 12);
MemberAttributeConfig twentyFourCore = new MemberAttributeConfig();
memberAttributeConfig.setIntAttribute("CPU_CORE_COUNT", 24);

Config member1Config = new Config();
config.setMemberAttributeConfig(fourCore);
Config member2Config = new Config();
config.setMemberAttributeConfig(twelveCore);
Config member3Config = new Config();
config.setMemberAttributeConfig(twentyFourCore);

HazelcastInstance member1 = Hazelcast.newHazelcastInstance(member1Config);
HazelcastInstance member2 = Hazelcast.newHazelcastInstance(member2Config);
HazelcastInstance member3 = Hazelcast.newHazelcastInstance(member3Config);

IExecutorService executorService = member1.getExecutorService("processor");

executorService.execute(new CPUIntensiveTask(), new MemberSelector() {
    @Override
    public boolean select(Member member) {
        int coreCount = (int) member.getIntAttribute("CPU_CORE_COUNT");
        // Task will be executed at either member2 or member3
        if (coreCount > 8) {
            return true;
        }
        return false;
    }
});

HazelcastInstance member4 = Hazelcast.newHazelcastInstance();
// We can also set member attributes at runtime.
member4.setIntAttribute("CPU_CORE_COUNT", 2);
18.4.3 Cluster-Member Safety Check

To prevent data loss when shutting down a node, Hazelcast provides a graceful shutdown feature. You perform this by calling the method `HazelcastInstance.shutdown()`. Once this method is called, it checks the following conditions to ensure the node is safe to shutdown.

- There is no active migration.
- At least one backup of partitions are synced with primary ones.

Even if the above conditions are not met, `HazelcastInstance.shutdown()` will force them to be completed. Eventually, when this method returns, it means the node has been brought to a safe state and it can be shut down without any data loss.

What if you want to be sure that your cluster is in a safe state? What does it mean that cluster is safe to shut down without any data loss?

There may be some use cases like rolling upgrades, development/testing or any logic that require a cluster/member to be safe. To provide this, Hazelcast offers the `PartitionService` interface with the methods `isClusterSafe`, `isMemberSafe`, `isLocalMemberSafe` and `forceLocalMemberToBeSafe`. These methods can be deemed as decoupled pieces from the method `Hazelcast.shutdown`.

```java
public interface PartitionService {
    ...
    boolean isClusterSafe();
    boolean isMemberSafe(Member member);
    boolean isLocalMemberSafe();
    boolean forceLocalMemberToBeSafe(long timeout, TimeUnit unit);
}
```

The method `isClusterSafe` checks whether the cluster is in a safe state. It returns `true` if there are no active partition migrations and there are sufficient backups for each partition. Once it returns `true`, the cluster is safe and a node can be shut down without data loss.

The method `isMemberSafe` checks whether a specific node is in a safe state. This check controls if the first backups of partitions of the given node are synced with the primary ones. Once it returns `true`, the given node is safe and it can be shut down without data loss. Similarly, the method `isLocalMemberSafe` does the same check for the local member. The method `forceLocalMemberToBeSafe` forces the owned and backup partitions to be synchronized, making the local member safe.

⚠️ **NOTE:** These methods are available from Hazelcast 3.3.

18.4.3.1 Sample Codes

```java
PartitionService partitionService = hazelcastInstance.getPartitionService().isClusterSafe();
if (partitionService().isClusterSafe()) {
    hazelcastInstance.shutdown(); // or terminate
}
```

OR

```java
PartitionService partitionService = hazelcastInstance.getPartitionService().isClusterSafe();
if (partitionService().isLocalMemberSafe()) {
    hazelcastInstance.shutdown(); // or terminate
}
```

**RELATED INFORMATION**

For more code samples please refer to `PartitionService Code Samples`. 
18.4.4 Cluster Quorum

Hazelcast Cluster Quorum enables you to define the minimum number of machines required in a cluster for the cluster to remain in an operational state. If the number of machines is below the defined minimum at any time, the operations are rejected and the rejected operations return a QuorumException to their callers.

When a network partitioning happens, by default Hazelcast chooses to be available. With Cluster Quorum, you can tune your Hazelcast instance towards to achieve better consistency, by rejecting updates with a minimum threshold, this reduces the chance that number of concurrent updates to an entry from two partitioned clusters. Note that the consistency defined here is best effort not full or strong consistency.

Hazelcast initiates a quorum when a change happens on the member list.

NOTE: Currently cluster quorum only applies to the Map and Transactional Map, support for other data structures will be added soon. Also lock methods in the IMap interface do not participate in a quorum.

18.4.4.1 Configuration

You can set up Cluster Quorum using either declarative or programmatic configuration.

Assume that you have a 5-node Hazelcast Cluster and you want to set the minimum number of 3 nodes for cluster to continue operating. The following are example configurations for this scenario.

18.4.4.1.1 Declarative Configuration

```xml
<hazelcast>
  ...
  <quorum name="quorumRuleWithThreeNodes" enabled=true>
    <quorum-size>3</quorum-size>
  </quorum>
  <map name="default">
    <quorum-name>quorumRuleWithThreeNodes</quorum-name>
  </map>
  ...
</hazelcast>
```

18.4.4.1.2 Programmatic Configuration

```java
QuorumConfig quorumConfig = new QuorumConfig();
quorumConfig.setName("quorumRuleWithThreeNodes");
quorumConfig.setEnabled(true);
quorumConfig.setSize(3);

MapConfig mapConfig = new MapConfig();
mapConfig.setQuorumName("quorumRuleWithThreeNodes");

Config config = new Config();
config.addQuorumConfig(quorumConfig);
config.addMapConfig(mapConfig);
```

18.4.4.2 Quorum Listeners

You can register quorum listeners to be notified about quorum results. Quorum listeners are local to the node that they are registered, so they receive only events occurred on that local node.

Quorum listeners can be configured via declarative or programmatic configuration. The following are the example configurations.
CHAPTER 18. MANAGEMENT

18.4.4.2.1 Declarative Configuration

```xml
<hazelcast>
...
<quorum name="quorumRuleWithThreeNodes" enabled=true>
  <quorum-size>3</quorum-size>
  <quorum-listeners>
    <quorum-listener>com.company.quorum.ThreeNodeQuorumListe</quorum-listener>
  </quorum-listeners>
</quorum>

<map name="default">
  <quorum-name>quorumRuleWithThreeNodes</quorum-name>
</map>
...
</hazelcast>
```

18.4.4.2.2 Programmatic Configuration

```java
QuorumListenerConfig listenerConfig = new QuorumListenerConfig();
// You can either directly set quorum listener implementation of your own
listenerConfig.setImplementation(new QuorumListener() {
    @Override
    public void onChange(QuorumEvent quorumEvent) {
        if (QuorumResult.PRESENT.equals(quorumEvent.getType())) {
            // handle quorum presence
        } else if (QuorumResult.ABSENT.equals(quorumEvent.getType())) {
            // handle quorum absence
        }
    }
});
// Or you can give the name of the class that implements QuorumListener interface.
listenerConfig.setClassName("com.company.quorum.ThreeNodeQuorumListener");

QuorumConfig quorumConfig = new QuorumConfig();
quorumConfig.setName("quorumRuleWithThreeNodes");
quorumConfig.setEnabled(true);
quorumConfig.setSize(3);
quorumConfig.addListenerConfig(listenerConfig);

MapConfig mapConfig = new MapConfig();
mapConfig.setQuorumName("quorumRuleWithThreeNodes");

Config config = new Config();
config.addQuorumConfig(quorumConfig);
cfg.addMapConfig(mapConfig);
```

18.4.4.3 Quorum Service

Quorum service gives an ability to query quorum results over the Quorum instances.

18.4.4.3.1 Quorum
Quorum instances let you to query quorum result of a particular quorum.

Here is the Quorum interface that you can interact with.
/**
 * @link Quorum} provides access to the current status of a quorum.
 */
public interface Quorum {
    /**
     * Returns true if quorum is present, false if absent.
     * @return boolean presence of the quorum
     */
    boolean isPresent();
}

You can retrieve quorum instance for a particular quorum over the quorum service. An example can be seen below

```java
String quorumName = "at-least-one-storage-member";
QuorumConfig quorumConfig = new QuorumConfig();
quorumConfig.setName(quorumName);
quorumConfig.setEnabled(true);

MapConfig mapConfig = new MapConfig();
mapConfig.setQuorumName(quorumName);

Config config = new Config();
config.addQuorumConfig(quorumConfig);
config.addMapConfig(mapConfig);

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance(config);
QuorumService quorumService = hazelcastInstance.getQuorumService();
Quorum quorum = quorumService.getQuorum(quorumName);

boolean quorumPresence = quorum.isPresent();
```

### 18.5 Management Center

#### 18.5.1 Introduction

Hazelcast Management Center enables you to monitor and manage your nodes running Hazelcast. In addition to monitoring overall state of your clusters, you can also analyze and browse your data structures in detail, update map configurations and take thread dump from nodes. With its scripting and console module, you can run scripts (JavaScript, Groovy, etc.) and commands on your nodes.

#### 18.5.1.1 Installation

You have two options for installing Hazelcast Management Center. You can either deploy the `mancenter-version.war` application into your Java application server/container or you can start Hazelcast Management Center from the command line and then have the Hazelcast nodes communicate with that web application. This means that your Hazelcast nodes should know the URL of the `mancenter` application before they start.

Here are the steps:

- Download the latest Hazelcast ZIP from hazelcast.org. The ZIP contains the `mancenter-version.war` file.
- You can directly start `mancenter-version.war` file from the command line. The following command will start Hazelcast Management Center on port 8080 with context root 'mancenter' (http://localhost:8080/mancenter).

```
java -jar mancenter-*version*.war 8080 mancenter
```
• Or, you can deploy it to your web server (Tomcat, Jetty, etc.). Let us say it is running at http://localhost:8080/mancenter.
• After you perform the above steps, make sure that http://localhost:8080/mancenter is up.
• Configure your Hazelcast nodes by adding the URL of your web application to your hazelcast.xml. Hazelcast nodes will send their states to this URL.

<management-center enabled="true"> http://localhost:8080/mancenter </management-center>

• Start your Hazelcast cluster.
• Browse to http://localhost:8080/mancenter and login. Initial login username/password is admin/admin

The Management Center creates a folder with the name “mancenter” under your “user/home” folder to save data files. You can change the data folder by setting the hazelcast.mancenter.home system property.

RELATED INFORMATION
Please refer to the Management Center Configuration section for a full description of Hazelcast Management Center configuration.

18.5.2 Tool Overview

Once the page is loaded after selecting a cluster, the tool’s home page appears as shown below.

This page provides the fundamental properties of the selected cluster which are explained in the Home Page section. The page has a toolbar on the top and a menu on the left.

18.5.2.1 Toolbar

The toolbar has the following buttons:

• Home: Loads the home page shown above. Please see the Home Page section.
• **Scripting**: Loads the page used to write and execute user’s own scripts on the cluster. Please see the Scripting section.

• **Console**: Loads the page used to execute commands on the cluster. Please see the Console section.

• **Alerts**: Creates alerts by specifying filters. Please see the Alerts section.

• **Documentation**: Opens the Management Center documentation in a window inside the tool. Please see the Documentation section.

• **Administration**: Used by the admin users to manage users in the system. Please see the Administration section.

• **Time Travel**: Sees the cluster’s situation at a time in the past. Please see the Time Travel section.

• **Cluster Selector**: Switches between clusters. When the mouse is moved onto this item, a drop down list of clusters appears.

The user can select any cluster and once selected, the page immediately loads with the selected cluster’s information.

• **Logout**: Closes the current user’s session.

**NOTE:** Some of the above listed toolbar items are not visible to users who are not admin or who have read-only permission. Also, some of the operations explained in the later sections cannot be performed by users with read-only permission. Please see the Administration section for details.

### 18.5.2.2 Menu

The Home page includes a menu on the left which lists the distributed data structures in the cluster and all the cluster members (nodes), as shown below.

**NOTE:** Distributed data structures will be shown there when the proxies are created for them.

You can expand and collapse menu items by clicking on them. Below is the list of menu items with links to their explanations.

- Caches
- Maps
- Queues
- Topics
- MultiMaps
- Executors
- Members
18.5. MANAGEMENT CENTER

18.5.2.3 Tabbed View

Each time you select an item from the toolbar or menu, the item is added to the main view as a tab, as shown below.

In the above example, Home, Scripting, Console, queue1 and map1 windows can be seen as tabs. Windows can be closed using the icon on each tab (except the Home Page; it cannot be closed).

18.5.3 Home Page

This is the first page appearing after logging in. It gives an overview of the connected cluster. The following subsections describe each portion of the page.

18.5.3.1 CPU Utilization

This part of the page provides load and utilization information for the CPUs for each node, as shown below.

The first column lists the nodes with their IPs and ports. The next columns list the loads on each CPU for the last 1, 5 and 15 minutes. The last column (Chart) graphically shows the utilization of CPUs. When you move the mouse cursor on a desired graph, you can see the CPU utilization at the time where the cursor is placed. Graphs under this column shows the CPU utilizations approximately for the last 2 minutes.

18.5.3.2 Memory Utilization

This part of the page provides information related to memory usages for each node, as shown below.

The first column lists the nodes with their IPs and ports. The next columns show the used and free memories out of the total memory reserved for Hazelcast usage, in real-time. The Max column lists the maximum memory capacity of each node and the Percent column lists the percentage value of used memory out of the maximum memory. The last column (Chart) shows the memory usage of nodes graphically. When you move the mouse cursor on a desired graph, you can see the memory usage at the time where the cursor is placed. Graphs under this column shows the memory usages approximately for the last 2 minutes.
18.5.3.3 Memory Distribution

This part of the page graphically provides the cluster wise breakdown of memory, as shown below. The blue area is the memory used by maps, the dark yellow area is the memory used by non-Hazelcast entities, and the green area is the free memory out of the whole cluster’s memory capacity.

In the above example, you can see 0.32% of the total memory is used by Hazelcast maps (it can be seen by placing the mouse cursor on it), 58.75% is used by non-Hazelcast entities and 40.85% of the total memory is free.

18.5.3.4 Map Memory Distribution

This part is the breakdown of the blue area shown in the Memory Distribution graph explained above. It provides the percentage values of the memories used by each map, out of the total cluster memory reserved for all Hazelcast maps.

In the above example, you can see 49.55% of the total map memory is used by map1 and 49.55% is used by map2.

18.5.3.5 Partition Distribution

This pie chart shows what percentage of partitions each node has, as shown below.

You can see each node’s partition percentages by placing the mouse cursor on the chart. In the above example, you can see the node “127.0.0.1:5708” has 5.64% of the total partition count (which is 271 by default and configurable, please see the hazelcast.partition.count property explained in the System Properties section).

18.5.4 Caches

You can monitor your caches’ metrics by clicking the cache name listed on the left panel under Caches menu item. A new tab for monitoring that cache instance is opened on the right, as shown below.
On top of the page, four charts monitor the Gets, Puts, Removals and Evictions in real-time. The X-axis of all the charts show the current system time. To open a chart as a separate dialog, click on the button placed at the top right of each chart.

Under these charts is the Cache Statistics Data Table. From left to right, this table lists the IP addresses and ports of each node, get, put, removal, eviction, hit and miss count per second in real-time.

You can navigate through the pages using the buttons at the bottom right of the table (First, Previous, Next, Last). You can ascend or descend the order of the listings in each column by clicking on column headings.

18.5.5 Maps

Map instances are listed under the Maps menu item on the left. When you click on a map, a new tab for monitoring that map instance opens on the right, as shown below. In this tab, you can monitor metrics and also re-configure the selected map.

The below subsections explain the portions of this window.

18.5.5.1 Map Browser

Map Browser is a tool you can use to retrieve properties of the entries stored in the selected map. To open it, click on the Map Browser button, located at the top right of the window. Once opened, the tool appears as a dialog, as shown below.

Once the key and key’s type is specified and the Browse button is clicked, the key’s properties along with its value are listed.

18.5.5.2 Map Config

By using the Map Config tool, you can set selected map’s attributes like the backup count, TTL, and eviction policy. To open it, click on the Map Config button, located at the top right of the window. Once opened, the tool appears as a dialog, as shown below.

Change any attribute as required and click the Update button to save changes.
18.5.5.3 Map Monitoring

Besides Map Browser and Map Config tools, this page has monitoring options explained below. All of these options perform real-time monitoring.

On top of the page, small charts monitor the size, throughput, memory usage, backup size, etc. of the selected map in real-time. The X-axis of all the charts show the current system time. You can select other small monitoring charts using the button at the top right of each chart. When you click the button, the monitoring options are listed, as shown below.

When you click on a desired monitoring, the chart is loaded with the selected option. To open a chart as a separate dialog, click on the button placed at the top right of each chart. The monitoring charts below are available:

- **Size**: Monitors the size of the map. Y-axis is the entry count (should be multiplied by 1000).
- **Throughput**: Monitors get, put and remove operations performed on the map. Y-axis is the operation count.
- **Memory**: Monitors the memory usage on the map. Y-axis is the memory count.
- **Backups**: It is the chart loaded when “Backup Size” is selected. Monitors the size of the backups in the map. Y-axis is the backup entry count (should be multiplied by 1000).
- **Backup Memory**: It is the chart loaded when “Backup Mem.” is selected. Monitors the memory usage of the backups. Y-axis is the memory count.
- **Hits**: Monitors the hit count of the map.
- **Puts/s, Gets/s, Removes/s**: These three charts monitor the put, get and remove operations (per second) performed on the selected map.

Under these charts are Map Memory and Map Throughput data tables. The Map Memory data table provides memory metrics distributed over nodes, as shown below.

From left to right, this table lists the IP address and port, entry counts, memory used by entries, backup entry counts, memory used by backup entries, events, hits, locks and dirty entries (in the cases where MapStore is enabled, these are the entries that are put to/removed from the map but not written to/removed from a database yet) of each node in the map. You can navigate through the pages using the buttons at the bottom right of the table (First, Previous, Next, Last). You can ascend or descend the order of the listings by clicking on the column headings.

Map Throughput data table provides information about the operations (get, put, remove) performed on each node in the map, as shown below.
From left to right, this table lists the IP address and port of each node, the put, get and remove operations on each node, the average put, get, remove latencies, and the maximum put, get, remove latencies on each node.

You can select the period in the combo box placed at the top right corner of the window, for which the table data will be shown. Available values are Since Beginning, Last Minute, Last 10 Minutes and Last 1 Hour.

You can navigate through the pages using the buttons placed at the bottom right of the table (First, Previous, Next, Last). To ascend or descent the order of the listings, click on the column headings.

18.5.6 Queues

Using the menu item Queues, you can monitor your queues data structure. When you expand this menu item and click on a queue, a new tab for monitoring that queue instance is opened on the right, as shown below.

On top of the page, small charts monitor the size, offers and polls of the selected queue in real-time. The X-axis of all the charts shows the current system time. To open a chart as a separate dialog, click on the button placed at the top right of each chart. The monitoring charts below are available:

- **Size**: Monitors the size of the queue. Y-axis is the entry count (should be multiplied by 1000).
- **Offers**: Monitors the offers sent to the selected queue. Y-axis is the offer count.
- **Polls**: Monitors the polls sent to the selected queue. Y-axis is the poll count.

Under these charts are Queue Statistics and Queue Operation Statistics tables. The Queue Statistics table provides item and backup item counts in the queue and age statistics of items and backup items at each node, as shown below.

From left to right, this table lists the IP address and port, items and backup items on the queue of each node, and maximum, minimum and average age of items in the queue. You can navigate through the pages using the buttons placed at the bottom right of the table (First, Previous, Next, Last). The order of the listings in each column can be ascended or descended by clicking on column headings.
Queue Operations Statistics table provides information about the operations (offers, polls, events) performed on the queues, as shown below.

From left to right, this table lists the IP address and port of each node, and counts of offers, rejected offers, polls, poll misses and events.

You can select the period in the combo box placed at the top right corner of the window to show the table data. Available values are Since Beginning, Last Minute, Last 10 Minutes and Last 1 Hour.

You can navigate through the pages using the buttons placed at the bottom right of the table (First, Previous, Next, Last). Click on the column headings to ascend or descend the order of the listings.

**18.5.7 Topics**

To monitor your topics’ metrics, click the topic name listed on the left panel under the Topics menu item. A new tab for monitoring that topic instance opens on the right, as shown below.

On top of the page, two charts monitor the Publishes and Receives in real-time. They show the published and received message counts of the cluster, nodes of which are subscribed to the selected topic. The X-axis of both charts show the current system time. To open a chart as a separate dialog, click on the button placed at the top right of each chart.

Under these charts is the Topic Operation Statistics table. From left to right, this table lists the IP addresses and ports of each node, and counts of message published and receives per second in real-time. You can select the period in the combo box placed at top right corner of the table to show the table data. The available values are Since Beginning, Last Minute, Last 10 Minutes and Last 1 Hour.

You can navigate through the pages using the buttons placed at the bottom right of the table (First, Previous, Next, Last). Click on the column heading to ascend or descend the order of the listings.
18.5.8 MultiMaps

MultiMap is a specialized map where you can associate a key with multiple values. This monitoring option is similar to the Maps option: the same monitoring charts and data tables monitor MultiMaps. The differences are that you cannot browse the MultiMaps and re-configure it. Please see the Maps section.

18.5.9 Executors

Executor instances are listed under the Executors menu item on the left. When you click on an executor, a new tab for monitoring that executor instance opens on the right, as shown below.

On top of the page, small charts monitor the pending, started, completed, etc. executors in real-time. The X-axis of all the charts shows the current system time. You can select other small monitoring charts using the button placed at the top right of each chart. When it is clicked, the monitoring options are listed, as shown below.

When you click on a desired monitoring, the chart loads with the selected option. To open a chart as a separate dialog, click on the button placed at top right of each chart. The below monitoring charts are available:

- **Pending**: Monitors the pending executors. Y-axis is the executor count.
18.5. MANAGEMENT CENTER

- **Started**: Monitors the started executors. Y-axis is the executor count.
- **Start Lat. (msec)**: Shows the latency when executors are started. Y-axis is the duration in milliseconds.
- **Completed**: Monitors the completed executors. Y-axis is the executor count.
- **Comp. Time (msec)**: Shows the completion period of executors. Y-axis is the duration in milliseconds.

Under these charts is the **Executor Operation Statistics** table, as shown below.

<table>
<thead>
<tr>
<th>#</th>
<th>Member</th>
<th>Pending</th>
<th>Started/s</th>
<th>Completed/s</th>
<th>Execution Time</th>
<th>Avg Start Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>127.0.0.1:5701</td>
<td>0</td>
<td>2.28</td>
<td>2.28</td>
<td>91.70 ms</td>
<td>0.23 ms</td>
</tr>
<tr>
<td>2</td>
<td>127.0.0.1:5703</td>
<td>0</td>
<td>1.90</td>
<td>1.90</td>
<td>0.02 ms</td>
<td>0.20 ms</td>
</tr>
<tr>
<td>3</td>
<td>127.0.0.1:5702</td>
<td>0</td>
<td>2.28</td>
<td>2.28</td>
<td>0.05 ms</td>
<td>0.23 ms</td>
</tr>
<tr>
<td>4</td>
<td>127.0.0.1:5708</td>
<td>0</td>
<td>1.98</td>
<td>1.98</td>
<td>0.02 ms</td>
<td>0.22 ms</td>
</tr>
<tr>
<td>5</td>
<td>127.0.0.1:5707</td>
<td>0</td>
<td>2.13</td>
<td>2.13</td>
<td>0.03 ms</td>
<td>0.13 ms</td>
</tr>
<tr>
<td>6</td>
<td>127.0.0.1:5706</td>
<td>0</td>
<td>1.90</td>
<td>1.90</td>
<td>0.07 ms</td>
<td>0.18 ms</td>
</tr>
<tr>
<td>7</td>
<td>127.0.0.1:5709</td>
<td>0</td>
<td>2.17</td>
<td>2.17</td>
<td>255.25 ms</td>
<td>0.13 ms</td>
</tr>
<tr>
<td>8</td>
<td>127.0.0.1:5704</td>
<td>0</td>
<td>2.32</td>
<td>2.32</td>
<td>0.10 ms</td>
<td>0.27 ms</td>
</tr>
<tr>
<td>9</td>
<td>127.0.0.1:5713</td>
<td>0</td>
<td>2.28</td>
<td>2.28</td>
<td>0.00 ms</td>
<td>0.22 ms</td>
</tr>
<tr>
<td>10</td>
<td>127.0.0.1:5716</td>
<td>0</td>
<td>1.93</td>
<td>1.93</td>
<td>0.02 ms</td>
<td>0.13 ms</td>
</tr>
</tbody>
</table>

From left to right, this table lists the IP address and port of nodes, the counts of pending, started and completed executors per second, and the execution time and average start latency of executors on each node. You can navigate through the pages using the buttons placed at the bottom right of the table (First, Previous, Next, Last). Click on the column heading to ascend or descend the order of the listings.

18.5.10 Members

Use this menu item to monitor each cluster member (node) and perform operations like running garbage collection (GC) and taking a thread dump. Once you select a member from the menu, a new tab for monitoring that member opens on the right, as shown below.
The CPU Utilization chart shows the percentage of CPU usage on the selected member. The Memory Utilization chart shows the memory usage on the selected member with three different metrics (maximum, used and total memory). You can open both of these charts as separate windows using the button placed at top right of each chart; this gives you a clearer view of the chart.

The window titled Partitions shows which partitions are assigned to the selected member. Runtime is a dynamically updated window tab showing the processor number, the start and up times, and the maximum, total and free memory sizes of the selected member. Next to this, the Properties tab shows the system properties. The Member Configuration window shows the connected Hazelcast cluster’s XML configuration.

The List of Slow Operations gives an overview of detected slow operations which occurred on that member. The data is collected by the SlowOperationDetector.

By clicking on an entry you can open a dialog which shows the stacktrace and detailed information about each slow invocation of this operation.

Besides the aforementioned monitoring charts and windows, you can also perform operations on the selected member through this page. The operation buttons are located at the top right of the page, as explained below:

- **Run GC**: When pressed, garbage collection is executed on the selected member. A notification stating that the GC execution was successful will be shown.
- **Thread Dump**: When pressed, thread dump of the selected member is taken and shown as a separate dialog to the user.
- **Shutdown Node**: It is used to shutdown the selected member.

### 18.5.11 Scripting

You can use the scripting feature of this tool to execute codes on the cluster. To open this feature as a tab, select Scripting located at the toolbar on top. Once selected, the scripting feature opens as shown below.

In this window, the Scripting part is the actual coding editor. You can select the members on which the code will execute from the Members list shown at the right side of the window. Below the members list, a combo box enables you to select a scripting language: currently, JavaScript, Ruby, Groovy and Python languages are supported. After you write your script and press the Execute button, you can see the execution result in the Result part of the window.

> **NOTE:** To use the scripting languages other than JavaScript on a member, the libraries for those languages should be placed in the classpath of that member.

There are Save and Delete buttons on the top right of the scripting editor. To save your scripts, press the Save button after you type a name for your script into the field next to this button. The scripts you saved are listed in the Saved Scripts part of the window, located at the bottom right of the page. Click on a saved script from this list to execute or edit it. If you want to remove a script that you wrote and saved before, select it from this list and press the Delete button.

In the scripting engine you have a HazelcastInstance bonded to a variable named hazelcast. You can invoke any method that HazelcastInstance has via the hazelcast variable. You can see example usage for JavaScript below.
Details of com.hazelcast.map.impl.operation.GetOperation (24 invocations)

Stacktrace
java.lang.Thread.sleep(Native Method)
at java.lang.Thread.sleep(Thread.java:349)
at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
at com.hazelcast.simulator.util.CommonUtil.sleepSeconds(CommonUtil.java:221)
at com.hazelcast.simulator.test.slow.SlowOperationMapTest$SlowMapInterceptor.sleepRecursion(SlowOperationMapTest.java:231)
at com.hazelcast.simulator.test.slow.SlowOperationMapTest$SlowMapInterceptor.sleepRecursion(SlowOperationMapTest.java:234)
at com.hazelcast.simulator.test.slow.SlowOperationMapTest$SlowMapInterceptor.sleepRecursion(SlowOperationMapTest.java:234)
at com.hazelcast.simulator.test.slow.SlowOperationMapTest$SlowMapInterceptor.sleepRecursion(SlowOperationMapTest.java:234)
at com.hazelcast.simulator.test.slow.SlowOperationMapTest$SlowMapInterceptor.sleepRecursion(SlowOperationMapTest.java:234)
at com.hazelcast.simulator.test.slow.SlowOperationMapTest$SlowMapInterceptor.sleepRecursion(SlowOperationMapTest.java:234)
at com.hazelcast.simulator.test.slow.SlowOperationMapTest$SlowMapInterceptor.sleepRecursion(SlowOperationMapTest.java:234)
at com.hazelcast.simulator.test.slow.SlowOperationMapTest$SlowMapInterceptor.sleepRecursion(SlowOperationMapTest.java:234)
at com.hazelcast.map.impl.MapServiceContext$impl interceptionAfterGet(MapServiceContextImpl.java:345)
at com.hazelcast.map.impl.operation.GetOperation.afterRun(GetOperation.java:53)
at com.hazelcast.spi.impl.operationservice.impl.OperationRunnerImpl.afterRun(OperationRunnerImpl.java:209)
at com.hazelcast.spi.impl.operationservice.impl.OperationRunnerImpl.run(OperationRunnerImpl.java:139)
at com.hazelcast.spi.impl.operationexecutor.classic.OperationThread.processOperation(OperationThread.java:154)
at com.hazelcast.spi.impl.operationexecutor.classic.OperationThread.process(OperationThread.java:110)
at com.hazelcast.spi.impl.operationexecutor.classic.OperationThread.doRun(OperationThread.java:101)
at com.hazelcast.spi.impl.operationexecutor.classic.OperationThread.run(OperationThread.java:78)

Operation GetOperation(SlowOperationMapTest)
Start Time Wednesday, May 6th 2015, 3:54:06 pm
Duration 14009 ms

Operation GetOperation(SlowOperationMapTest)
Start Time Wednesday, May 6th 2015, 3:00:21 pm
Duration 14010 ms

Operation GetOperation(SlowOperationMapTest)
Start Time Wednesday, May 6th 2015, 3:54:06 pm
Duration 14006 ms
var name = hazelcast.getName();
var node = hazelcast.getCluster().getLocalMember();
var employees = hazelcast.getMap("employees");
employees.put("1", "John Doe");
employees.get("1"); // will return "John Doe"

### 18.5.12 Console

The Management Center has a console feature that enables you to execute commands on the cluster. For example, you can perform puts and gets on a map, after you set the namespace with the command `ns <name of your map>`. The same is valid for queues, topics, etc. To execute your command, type it into the field below the console and press Enter. Type `help` to see all the commands that you can use.

Open a console window by clicking on the **Console** button located on the toolbar. Below is a sample view with some executed commands.

### 18.5.13 Alerts

You can use the alerts feature of this tool to receive alerts by creating filters. In these filters, you can specify criteria for cluster, nodes or data structures. When the specified criteria are met for a filter, the related alert is shown as a pop-up message on the top right of the page.

Once you click the **Alerts** button located on the toolbar, the page shown below appears.

**Creating Filters for Cluster**

Select the **Cluster Alerts** check box to create a cluster wise filter. Once selected, the next screen asks for the items for which alerts will be created, as shown below.

Select the desired items and click the **Next** button. On the next page (shown below), specify the frequency of checks in **hour** and **min** fields, give a name for the filter, select whether notification e-mails will be sent (to no one, only admin or to all users) and select whether the alert data will be written to the disk (if checked, you can see the alert log at the folder `/users//mancenter`).
18.5. MANAGEMENT CENTER

18.5. MANAGEMENT CENTER

To create an automated alert, choose what you want to check.
- Cluster Alerts -- General alerts on the health of your cluster.
- Member Alerts -- Alerts about memory and thread count of your members.
- Data Type Alerts -- Alerts for data types (map, queue, multimap, executor).

Type your command here, type "help" to see list of commands.
Click on the **Save** button; your filter will be saved and put into the **Filters** part of the page, as shown below.

![Filters](image)

To edit the filter, click on the ![edit](image) icon. To delete the filter, click on the ![delete](image) icon.

**Creating Filters for Cluster Members**

Select **Member Alerts** check box to create filters for some or all members in the cluster. Once selected, the next screen asks for which members the alert will be created. Select the desired members and click on the **Next** button. On the next page (shown below), specify the criteria.

![Create New Filter](image)

Alerts can be created when:

- free memory on the selected nodes is less than the specified number.
- used heap memory is larger than the specified number.
- the number of active threads are less than the specified count.
- the number of daemon threads are larger than the specified count.

When two or more criteria is specified they will be bound with the logical operator **AND**.

On the next page, give a name for the filter, select whether notification e-mails will be sent (to no one, only admin, or to all users) and select whether the alert data will be written to the disk (if checked, you can see the alert log at the folder `/users//mancenter`).

Click on the **Save** button; your filter will be saved and put into the **Filters** part of the page. To edit the filter, click on the ![edit](image) icon. To delete it, click on the ![delete](image) icon.

**Creating Filters for Data Types**
CHAPTER 18. MANAGEMENT

Select the **Data Type Alerts** check box to create filters for data structures. The next screen asks for which data structure (maps, queues, multimaps, executors) the alert will be created. Once a structure is selected, the next screen immediately loads and you then select the data structure instances (i.e. if you selected Maps, it will list all the maps defined in the cluster, you can select one map or more). Select as desired, click on the **Next** button, and select the members on which the selected data structure instances will run.

The next screen, as shown below, is the one where you specify the criteria for the selected data structure.

![Data Type Filter](image)

As the screen shown above shows, you will select an item from the left combo box, select the operator in the middle one, specify a value in the input field, and click on the **Add** button. You can create more than one criteria in this page; those will be bound by the logical operator **AND**.

After you specify the criteria and click the **Next** button, give a name for the filter, select whether notification e-mails will be sent (to no one, only admin or to all users) and select whether the alert data will be written to the disk (if checked, you can see the alert log at the folder `/users//mancenter`).

Click on the **Save** button; your filter will be saved and put into the **Filters** part of the page. To edit the filter, click on the ![Edit](image) icon. To delete it, click on the ![Delete](image) icon.

### 18.5.14 Administration

**NOTE:** This toolbar item is available only to admin users, i.e. the users who initially have **admin** as their both usernames and passwords.*

The **Admin** user can add, edit, and remove users and specify the permissions for the users of Management Center. To perform these operations, click on the **Administration** button located on the toolbar. The page below appears.

To add a user to the system, specify the username, e-mail and password in the **Add/Edit User** part of the page. If the user to be added will have administrator privileges, select **isAdmin** checkbox. **Permissions** checkboxes have two values:

- **Read Only**: If this permission is given to the user, only **Home, Documentation** and **Time Travel** items will be visible at the toolbar at that user’s session. Also, users with this permission cannot update a map configuration, run a garbage collection and take a thread dump on a node, or shutdown a node (please see the **Members** section).
- **Read/Write**: If this permission is given to the user, **Home, Scripting, Console, Documentation and Time Travel** items will be visible. The users with this permission can update a map configuration and perform operations on the nodes.

After you enter/select all fields, click **Save** button to create the user. You will see the newly created user’s username on the left side, in the **Users** part of the page.
To edit or delete a user, select a username listed in the Users. Selected user information appears on the right side of the page. To update the user information, change the fields as desired and click the Save button. To delete the user from the system, click the Delete button.

18.5.15 Time Travel

Time Travel is used to check the status of the cluster at a time in the past. When this item is selected on the toolbar, a small window appears on top of the page, as shown below.

To see the cluster status in a past time, Time Travel should be enabled first. Click on the area where it says OFF (on the right of Time Travel window). It will turn to ON after it asks whether to enable the Time Travel with a dialog: click on Enable in the dialog to enable Time Travel.

Once it is ON, the status of your cluster will be stored on your disk as long as your web server is alive.

You can go back in time using the slider and/or calendar and check your cluster’s situation at the selected time. All data structures and members can be monitored as if you are using the management center normally (charts and data tables for each data structure and members). Using the arrow buttons placed at both sides of the slider, you can go back or further with steps of 5 seconds. It will show status if Time Travel has been ON at the selected time in past; otherwise, all the charts and tables will be shown as empty.

The historical data collected with Time Travel feature are stored in a file database on the disk. These files can be found in the folder <User’s Home Directory>/mancenter<Hazelcast version>, e.g. /home/mancenter3.5. This folder can be changed using the hazelcast.mancenter.home property on the server where Management Center is running.

Time travel data files are created monthly. Their file name format is [group-name]-[year][month].db and [group-name]-[year][month].lg. Time travel data is kept in the *.db files. The files with the extension lg are temporary files created internally and you do not have to worry about them.

Management Center has no automatic way of removing or archiving old time travel data files. They remain in the aforementioned folder until you delete or archive them.
18.5.16 Documentation

To see the documentation, click on the Documentation button located at the toolbar. Management Center manual will appear as a tab.

18.5.17 Suggested Heap Size

For 2 Nodes

<table>
<thead>
<tr>
<th>Mancenter Heap Size</th>
<th># of Maps</th>
<th># of Queues</th>
<th># of Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>256m</td>
<td>3k</td>
<td>1k</td>
<td>1k</td>
</tr>
<tr>
<td>1024m</td>
<td>10k</td>
<td>1k</td>
<td>1k</td>
</tr>
</tbody>
</table>

For 10 Nodes

<table>
<thead>
<tr>
<th>Mancenter Heap Size</th>
<th># of Maps</th>
<th># of Queues</th>
<th># of Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>256m</td>
<td>50</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>1024m</td>
<td>2k</td>
<td>1k</td>
<td>1k</td>
</tr>
</tbody>
</table>

For 20 Nodes

<table>
<thead>
<tr>
<th>Mancenter Heap Size</th>
<th># of Maps</th>
<th># of Queues</th>
<th># of Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>256m*</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>1024m</td>
<td>1k</td>
<td>1k</td>
<td>1k</td>
</tr>
</tbody>
</table>

* With 256m heap, management center is unable to collect statistics.

18.6 Clustered JMX

Hazelcast Enterprise

Clustered JMX via Management Center allows you to monitor clustered statistics of distributed objects from a JMX interface.

18.6.1 Clustered JMX Configuration

In order to configure Clustered JMX, use two command line parameters for your Management Center deployment.

- `-Dhazelcast.mc.jmx.enabled=true` (default is false)
- `-Dhazelcast.mc.jmx.port=9000` (optional, default is 9999)

With embedded Jetty, you do not need to deploy your Management Center application to any container or application server.

You can start Management Center application with Clustered JMX enabled as shown below.

```java
java -Dhazelcast.mc.jmx.enabled=true -Dhazelcast.mc.jmx.port=9999 -jar mancenter-3.3.jar
```

Once Management Center starts, you should see a log similar to below.
18.6. CLUSTERED JMX

INFO: Management Center 3.3
INFO: Starting Management Center JMX Service on port: 9999

You should be able to connect to Clustered JMX interface from the address localhost:9999.

You can use jconsole or any other JMX client to monitor your Hazelcast Cluster. As a sample, below is the jconsole screenshot of the Clustered JMX hierarchy.

18.6.2 API Documentation

The management beans are exposed with the following object name format.

ManagementCenter[cluster name]: type=<object type>, name=<object name>, member="<cluster member IP address>"

Object name starts with ManagementCenter prefix. Then it has the cluster name in brackets followed by a colon. After that, type, name and member attributes follows, each separated with a comma.

- **type** is the type of object. Values are Clients, Executors, Maps, Members, MultiMaps, Queues, Services, and Topics.
- **name** is the name of object.
- **member** is the node address of object (only required if the statistics are local to the node).

A sample bean is shown below.

ManagementCenter[dev]: type=Services, name=OperationService, member="192.168.2.79:5701"

Here is the list of attributes that are exposed from the Clustered JMX interface.

- **ManagementCenter[ClusterName]**
  - Clients
  - Address
  - ClientType
  - Uuid
  - Executors
  - Cluster
    - Name
    - StartedTaskCount
    - CompletedTaskCount
    - CancelledTaskCount
    - PendingTaskCount
  - Maps
    - Cluster
    - Name
    - BackupEntryCount
    - BackupEntryMemoryCost
    - CreationTime
    - DirtyEntryCount
    - Events
    - GetOperationCount
    - HeapCost
    - Hits
18.6. CLUSTERED JMX

- LastAccessTime
- LastUpdateTime
- LockedEntryCount
- MaxGetLatency
- MaxPutLatency
- MaxRemoveLatency
- OtherOperationCount
- OwnedEntryCount
- PutOperationCount
- RemoveOperationCount

• Members
  - ConnectedClientCount
  - HeapFreeMemory
  - HeapMaxMemory
  - HeapTotalMemory
  - HeapUsedMemory
  - IsMaster
  - OwnedPartitionCount

• MultiMaps
  - Cluster
  - Name
  - BackupEntryCount
  - BackupEntryMemoryCost
  - CreationTime
  - DirtyEntryCount
  - Events
  - GetOperationCount
  - HeapCost
  - Hits
  - LastAccessTime
  - LastUpdateTime
  - LockedEntryCount
  - MaxGetLatency
  - MaxPutLatency
  - MaxRemoveLatency
  - OtherOperationCount
  - OwnedEntryCount
  - PutOperationCount
  - RemoveOperationCount

• Queues
  - Cluster
  - Name
  - MinAge
  - MaxAge
  - AvgAge
  - OwnedItemCount
  - BackupItemCount
  - OfferOperationCount
  - OtherOperationsCount
  - PollOperationCount
  - RejectedOfferOperationCount
  - EmptyPollOperationCount
- EventOperationCount
- CreationTime

- **Services**
  - ConnectionManager
    * ActiveConnectionCount
    * ClientConnectionCount
    * ConnectionCount
  - EventService
    * EventQueueCapacity
    * EventQueueSize
    * EventThreadCount
  - OperationService
    * ExecutedOperationCount
    * OperationExecutorQueueSize
    * OperationThreadCount
    * RemoteOperationCount
    * ResponseQueueSize
    * RunningOperationsCount
  - PartitionService
    * ActivePartitionCount
    * PartitionCount
  - ProxyService
    * ProxyCount
  - ManagedExecutor[hz::async]
    * Name
    * CompletedTaskCount
    * MaximumPoolSize
    * PoolSize
    * QueueSize
    * RemainingQueueCapacity
    * Terminated
  - ManagedExecutor[hz::client]
    * Name
    * CompletedTaskCount
    * MaximumPoolSize
    * PoolSize
    * QueueSize
    * RemainingQueueCapacity
    * Terminated
  - ManagedExecutor[hz::global-operation]
    * Name
    * CompletedTaskCount
    * MaximumPoolSize
    * PoolSize
    * QueueSize
    * RemainingQueueCapacity
    * Terminated
  - ManagedExecutor[hz::io]
    * Name
    * CompletedTaskCount
    * MaximumPoolSize
### 18.6.3 New Relic Integration

Use the Clustered JMX interface to integrate Hazelcast Management Center with *New Relic*. To perform this integration, attach New Relic Java agent and provide an extension file that describes which metrics will be sent to New Relic.

Please see [Custom JMX instrumentation by YAML](#) on the New Relic webpage.

Below is an example Map monitoring `.yml` file for New Relic.

```yaml
name: Clustered JMX
version: 1.0
enabled: true

jmx:
  - object_name: ManagementCenter[clusternamex]:type=Maps,name=mapname
    metrics:
      - attributes: PutOperationCount, GetOperationCount, RemoveOperationCount, Hits,
```
CHAPTER 18. MANAGEMENT

BackupEntryCount, OwnedEntryCount, LastAccessTime, LastUpdateTime
  type: simple
  - object_name: ManagementCenter[clustername]:type=Members,name="node address in double quotes"
metrics:
  - attributes: OwnedPartitionCount
    type: simple

Put the .yml file in the extensions folder in your New Relic installation. If an extensions folder does not exist there, create one.

After you set your extension, attach the New Relic Java agent and start Management Center as shown below.

```java
java -javaagent:/path/to/newrelic.jar -Dhazelcast.mc.jmx.enabled=true
    -Dhazelcast.mc.jmx.port=9999 -jar mancenter-3.3.jar
```

If your logging level is set as FINER, you should see the log listing in the file newrelic_agent.log, which is located in the logs folder in your New Relic installation. Below is an example log listing.


Then you can navigate to your New Relic account and create Custom Dashboards. Please see Creating custom dashboards.

While you are creating the dashboard, you should see the metrics that you are sending to New Relic from Management Center in the Metrics section under the JMX folder.

18.6.4 AppDynamics Integration

Use the Clustered JMX interface to integrate Hazelcast Management Center with AppDynamics. To perform this integration, attach AppDynamics Java agent to the Management Center.

For agent installation, refer to Install the App Agent for Java page.
For monitoring on AppDynamics, refer to Using AppDynamics for JMX Monitoring page. After installing AppDynamics agent, you can start Management Center as shown below.

```java
java -javaagent:/path/to/javaagent.jar -Dhazelcast.mc.jmx.enabled=true
-Dhazelcast.mc.jmx.port=9999 -jar mancenter-3.3.jar
```

When Management Center starts, you should see the logs below.

Started AppDynamics Java Agent Successfully.
Hazelcast Management Center starting on port 8080 at path : /mancenter

### 18.7 Clustered REST

**Hazelcast Enterprise**

The Clustered REST API is exposed from Management Center to allow you to monitor clustered statistics of distributed objects.

#### 18.7.1 Enabling Clustered REST

To enable Clustered REST on your Management Center, pass the following system property at startup. This property is disabled by default.

```
-Dhazelcast.mc.rest.enabled=true
```

#### 18.7.2 Clustered REST API Root

The entry point for Clustered REST API is `/rest/`. This resource does not have any attributes.

#### 18.7.3 Clusters Resource

This resource returns a list of clusters that are connected to the Management Center.

##### 18.7.3.0.1 Retrieve Clusters

- **Request Type:** GET
- **URL:** `/rest/clusters`
- **Request:**
  ```
curl http://localhost:8083/mancenter/rest/clusters
  ```
- **Response:** 200 (application/json)
- **Body:**
  ```
  ["dev","qa"]
  ```

#### 18.7.4 Cluster Resource

This resource returns information related to the provided cluster name.
18.7.4.0.2 Retrieve Cluster Information

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}
- **Request:**

  ```
  curl http://localhost:8083/mancenter/rest/clusters/dev/
  ```
- **Response:** 200 (application/json)
- **Body:**

  ```json
  {"masterAddress":"192.168.2.78:5701"}
  ```

18.7.5 Members Resource

This resource returns a list of members belonging to the provided clusters.

18.7.5.0.3 Retrieve Members [GET] [/rest/clusters/{clustername}/members]

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}/members
- **Request:**

  ```
  curl http://localhost:8083/mancenter/rest/clusters/dev/members
  ```
- **Response:** 200 (application/json)
- **Body:**

  ```json
  ["192.168.2.78:5701","192.168.2.78:5702","192.168.2.78:5703","192.168.2.78:5704"]
  ```

18.7.6 Member Resource

This resource returns information related to the provided member.

18.7.6.0.4 Retrieve Member Information

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}/members/{member}
- **Request:**

  ```
  ```
- **Response:** 200 (application/json)
- **Body:**
18.7. CLUSTERED REST

```json
{
  "cluster" : "dev",
  "name" : "192.168.2.78:5701",
  "maxMemory" : 129957888,
  "ownedPartitionCount" : 68,
  "usedMemory" : 60688784,
  "freeMemory" : 24311408,
  "totalMemory" : 85000192,
  "connectedClientCount" : 1,
  "master" : true
}
```

### 18.7.6.0.5 Retrieve Connection Manager Information

- **Request Type:** GET
- **URL:** `/rest/clusters/{clustername}/members/{member}/connectionManager`
- **Request:**

  ```bash
curl http://localhost:8083/mancenter/rest/clusters/dev/members/192.168.2.78:5701/connectionManager
  ```

- **Response:** 200 (application/json)
- **Body:**

  ```json
  {
    "clientConnectionCount" : 2,
    "activeConnectionCount" : 5,
    "connectionCount" : 5
  }
  ```

### 18.7.6.0.6 Retrieve Operation Service Information

- **Request Type:** GET
- **URL:** `/rest/clusters/{clustername}/members/{member}/operationService`
- **Request:**

  ```bash
  ```

- **Response:** 200 (application/json)
- **Body:**

  ```json
  {
    "responseQueueSize" : 0,
    "operationExecutorQueueSize" : 0,
    "runningOperationsCount" : 0,
    "remoteOperationCount" : 1,
    "executedOperationCount" : 461139,
    "operationThreadCount" : 8
  }
  ```
18.7.6.0.7 Retrieve Event Service Information

- Request Type: GET
- URL: /rest/clusters/{clusternam}/members/{member}/eventService
- Request:

  curl http://localhost:8083/mancenter/rest/clusters/dev/members/192.168.2.78:5701/eventService

- Response: 200 (application/json)
- Body:

  
  ```
  {
    "eventThreadCount":5,
    "eventQueueCapacity":1000000,
    "eventQueueSize":0
  }
  ```

18.7.6.0.8 Retrieve Partition Service Information

- Request Type: GET
- URL: /rest/clusters/{clusternam}/members/{member}/partitionService
- Request:


- Response: 200 (application/json)
- Body:

  
  ```
  {
    "partitionCount":271,
    "activePartitionCount":68
  }
  ```

18.7.6.0.9 Retrieve Proxy Service Information

- Request Type: GET
- URL: /rest/clusters/{clusternam}/members/{member}/proxyService
- Request:


- Response: 200 (application/json)
- Body:

  
  ```
  {
    "proxyCount":8
  }
  ```
18.7. CLUSTERED REST

18.7.6.0.10 Retrieve All Managed Executors

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}/members/{member}/managedExecutors
- **Request:**
  
- **Response:** 200 (application/json)
- **Body:**
  
  ```json
  ```

18.7.6.0.11 Retrieve a Managed Executor

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}/members/{member}/managedExecutors/{managedExecutor}
- **Request:**
  
- **Response:** 200 (application/json)
- **Body:**
  
  ```json
  {
    "name":"hz:system",
    "queueSize":0,
    "poolSize":0,
    "remainingQueueCapacity":2147483647,
    "maximumPoolSize":4,
    "completedTaskCount":12,
    "terminated":false
  }
  ```

18.7.7 Clients Resource

This resource returns a list of clients belonging to the provided cluster.

18.7.7.0.12 Retrieve List of Clients

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}/clients
- **Request:**
  
curl http://localhost:8083/mancenter/rest/clusters/dev/clients
- **Response:** 200 (application/json)
- **Body:**
  
  ```json
  ["192.168.2.78:61708"]
  ```
18.7.7.0.13 Retrieve Client Information

- Request Type: GET
- URL: /rest/clusters/{clusternam}/clients/{client}
- Request:
  
  ```
  ```
- Response: 200 (application/json)
- Body:
  
  ```json
  {
    "uuid":"6fae7af6-7a7c-4fa5-b165-cde24cf070f5",
    "address":"192.168.2.78:61708",
    "clientType":"JAVA"
  }
  ```

18.7.8 Maps Resource

This resource returns a list of maps belonging to the provided cluster.

18.7.8.0.14 Retrieve List of Maps

- Request Type: GET
- URL: /rest/clusters/{clustername}/maps
- Request:
  
  ```
curl http://localhost:8083/mancenter/rest/clusters/dev/maps
  ```
- Response: 200 (application/json)
- Body:
  
  ```json
  ["customers","orders"]
  ```

18.7.8.0.15 Retrieve Map Information

- Request Type: GET
- URL: /rest/clusters/{clustername}/maps/{mapName}
- Request:
  
  ```
curl http://localhost:8083/mancenter/rest/clusters/dev/maps/customers
  ```
- Response: 200 (application/json)
- Body:
18.7. CLUSTERED REST

```json
{
  "cluster":"dev",
  "name":"customers",
  "ownedEntryCount":1000,
  "backupEntryCount":1000,
  "ownedEntryMemoryCost":157890,
  "backupEntryMemoryCost":113683,
  "heapCost":297005,
  "lockedEntryCount":0,
  "dirtyEntryCount":0,
  "hits":3001,
  "lastAccessTime":1403608925777,
  "lastUpdateTime":1403608925777,
  "creationTime":1403602693388,
  "putOperationCount":110630,
  "getOperationCount":165945,
  "removeOperationCount":55315,
  "otherOperationCount":0,
  "events":0,
  "maxPutLatency":52,
  "maxGetLatency":30,
  "maxRemoveLatency":21
}
```

18.7.9 MultiMaps Resource

This resource returns a list of multimaps belonging to the provided cluster.

18.7.9.0.16 Retrieve List of MultiMaps

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}/multimaps
- **Request:**
  ```
curl http://localhost:8083/mancenter/rest/clusters/dev/multimaps
  ```
- **Response:** 200 (application/json)
- **Body:**
  ```json
  ["customerAddresses"]
  ```

18.7.9.0.17 Retrieve MultiMap Information

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}/multimaps/{multimapname}
- **Request:**
  ```
curl http://localhost:8083/mancenter/rest/clusters/dev/multimaps/customerAddresses
  ```
- **Response:** 200 (application/json)
- **Body:**
CHAPTER 18. MANAGEMENT

{  
  "cluster":"dev",
  "name":"customerAddresses",
  "ownedEntryCount":996,
  "backupEntryCount":996,
  "ownedEntryMemoryCost":0,
  "backupEntryMemoryCost":0,
  "heapCost":0,
  "lockedEntryCount":0,
  "dirtyEntryCount":0,
  "hits":0,
  "lastAccessTime":1403603095521,
  "lastUpdateTime":1403603095521,
  "creationTime":1403602694158,
  "putOperationCount":166041,
  "getOperationCount":110694,
  "removeOperationCount":55347,
  "otherOperationCount":0,
  "events":0,
  "maxPutLatency":77,
  "maxGetLatency":69,
  "maxRemoveLatency":42  
}

18.7.10 Queues Resource

This resource returns a list of queues belonging to the provided cluster.

18.7.10.0.18 Retrieve List of Queues

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}/queues
- **Request:**
  
curl http://localhost:8083/mancenter/rest/clusters/dev/queues
  
- **Response:** 200 (application/json)
- **Body:**
  
  ["messages"]

18.7.10.0.19 Retrieve Queue Information

- **Request Type:** GET
- **URL:** /rest/clusters/{clustername}/queues/{queueName}
- **Request:**
  
curl http://localhost:8083/mancenter/rest/clusters/dev/queues/messages
  
- **Response:** 200 (application/json)
- **Body:**
18.7.11 Topics Resource

This resource returns a list of topics belonging to the provided cluster.

18.7.11.0.20 Retrieve List of Topics

- **Request Type:** GET
- **URL:** `/rest/clusters/{clustername}/topics`
- **Request:**
  
curl http://localhost:8083/mancenter/rest/clusters/dev/topics

- **Response:** 200 (application/json)
- **Body:**
  
  `["news"]`

18.7.11.0.21 Retrieve Topic Information

- **Request Type:** GET
- **URL:** `/rest/clusters/{clustername}/topics/{topicName}`
- **Request:**
  
curl http://localhost:8083/mancenter/rest/clusters/dev/topics/news

- **Response:** 200 (application/json)
- **Body:**
  
  ```
  {
    "cluster":"dev",
    "name":"news",
    "numberOfPublishes":56370,
    "totalReceivedMessages":56370,
    "creationTime":1403602693411
  }
  ```
18.7.12 Executors Resource

This resource returns a list of executors belonging to the provided cluster.

18.7.12.0.22 Retrieve List of Executors

- Request Type: GET
- URL: /rest/clusters/{clustername}/executors
- Request:
  curl http://localhost:8083/mancenter/rest/clusters/dev/executors
- Response: 200 (application/json)
- Body:
  
  ["order-executor"]

18.7.12.0.23 Retrieve Executor Information [GET] [/rest/clusters/{clustername}/executors/{executorName}]

- Request Type: GET
- URL: /rest/clusters/{clustername}/executors/{executorName}
- Request:
  curl http://localhost:8083/mancenter/rest/clusters/dev/executors/order-executor
- Response: 200 (application/json)
- Body:
  
  {
    "cluster":"dev",
    "name":"order-executor",
    "creationTime":1403602694196,
    "pendingTaskCount":0,
    "startedTaskCount":1241,
    "completedTaskCount":1241,
    "cancelledTaskCount":0
  }
Chapter 19

Security

Hazelcast Enterprise
This chapter provides information on the security features of Hazelcast. These features allow you to perform security activities including intercepting socket connections and remote operations executed by the clients, encrypting the communications between the members at socket level and using SSL socket communication.

19.1 Enabling Security for Hazelcast Enterprise

Hazelcast has an extensible, JAAS based security feature you can use to authenticate both cluster members and clients, and to perform access control checks on client operations. Access control can be done according to endpoint principal and/or endpoint address.

You can enable security declaratively or programmatically, as shown below.

```xml
<hazelcast xsi:schemaLocation="http://www.hazelcast.com/schema/config
 http://www.hazelcast.com/schema/config/hazelcast-config-3.3.xsd"
 xmlns="http://www.hazelcast.com/schema/config"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...

 <security enabled="true">
 ... 
 </security>
 ...
</hazelcast>
```

Config cfg = new Config();
SecurityConfig securityCfg = cfg.getSecurityConfig();
securityCfg.setEnabled( true );

Also, see Setting License Key.

19.2 Socket Interceptor

Hazelcast Enterprise
Hazelcast 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 (like identity checking using Kerberos, etc.). Implement com.hazelcast.nio.MemberSocketInterceptor for members and com.hazelcast.nio.SocketInterceptor for clients.
public class MySocketInterceptor implements MemberSocketInterceptor {
    public void init( SocketInterceptorConfig socketInterceptorConfig ) {
        // initialize interceptor
    }

    void onConnect( Socket connectedSocket ) throws IOException {
        // do something meaningful when connected
    }

    public void onAccept( Socket acceptedSocket ) throws IOException {
        // do something meaningful when accepted a connection
    }
}

<hazelcast>
...

<network>
...

<socket-interceptor enabled="true">
    <class-name>com.hazelcast.examples.MySocketInterceptor</class-name>
    <properties>
        <property name="kerberos-host">kerb-host-name</property>
        <property name="kerberos-config-file">kerb.conf</property>
    </properties>
</socket-interceptor>

</network>
...

</hazelcast>

public class MyClientSocketInterceptor implements SocketInterceptor {
    void onConnect( Socket connectedSocket ) throws IOException {
        // do something meaningful when connected
    }
}

ClientConfig clientConfig = new ClientConfig();
clientConfig.setGroupConfig( new GroupConfig( "dev", "dev-pass" ) )
    .addAddress( "10.10.3.4" );

MyClientSocketInterceptor clientSocketInterceptor = new MyClientSocketInterceptor();
clientConfig.setSocketInterceptor( clientSocketInterceptor );
HazelcastInstance client = HazelcastClient.newHazelcastClient( clientConfig );

19.3 Security Interceptor

Hazelcast Enterprise

Hazelcast allows you to intercept every remote operation executed by the client. This lets you add a very flexible custom security logic. To do this, implement com.hazelcast.security.SecurityInterceptor.

public class MySecurityInterceptor implements SecurityInterceptor {
    public void before( Credentials credentials, String serviceName,
                        String methodName, Parameters parameters )
        throws AccessControlException {
        // credentials: client credentials
public void after( Credentials credentials, String serviceName,
   String methodName, Parameters parameters ) {
    // can be used for logging etc.
}
}

The **before** method will be called before processing the request on the remote server. The **after** method will be called after the processing. Exceptions thrown while executing the **before** method will propagate to the client, but exceptions thrown while executing the **after** method will be suppressed.

## 19.4 Encryption

**Hazelcast Enterprise**

Hazelcast allows you to encrypt the entire socket level communication among all Hazelcast members. Encryption is based on Java Cryptography Architecture. In symmetric encryption, each node uses the same key, so the key is shared. Here is an example configuration for symmetric encryption.

```xml
<hazelcast>
  ...
  <network>
    ...
    <!-- Make sure to set enabled=true
         Make sure this configuration is exactly the same on
         all members
    -->
    <symmetric-encryption enabled="true">
      <!-- encryption algorithm such as
           DES/ECB/PKCS5Padding,
           PBEWithMD5AndDES,
           Blowfish,
           DESede
      -->
      <algorithm>PBEWithMD5AndDES</algorithm>
      <!-- salt value to use when generating the secret key -->
      <salt>thesalt</salt>
      <!-- pass phrase to use when generating the secret key -->
      <password>thepass</password>
      <!-- iteration count to use when generating the secret key -->
      <iteration-count>19</iteration-count>
    </symmetric-encryption>
  </network>
  ...
</hazelcast>
```

### RELATED INFORMATION

Please see the SSL section.
19.5 SSL

**Hazelcast Enterprise**

Hazelcast allows you to use SSL socket communication among all Hazelcast members. To use it, you need to implement `com.hazelcast.nio.ssl.SSLContextFactory` and configure the SSL section in network configuration.

```java
public class MySSLContextFactory implements SSLContextFactory {
    public void init(Properties properties) throws Exception {
    }

    public SSLContext getSSLContext() {
        ...;
        SSLContext sslCtx = SSLContext.getInstance(protocol);
        return sslCtx;
    }
}
```

Hazelcast provides a default SSLContextFactory, `com.hazelcast.nio.ssl.BasicSSLContextFactory`, which uses configured keystore to initialize SSLContext. You define `keyStore` and `keyStorePassword`, and you can set `keyManagerAlgorithm` (default SunX509), `trustManagerAlgorithm` (default SunX509) and `protocol` (default TLS).

```xml
<network>
    ...
    <ssl enabled="true">
        <factory-class-name>
            com.hazelcast.examples.MySSLContextFactory
        </factory-class-name>
        <properties>
            <property name="foo">bar</property>
        </properties>
    </ssl>
    ...
</network>
```

```xml
<network>
    ...
    <ssl enabled="true">
        <factory-class-name>
            com.hazelcast.nio.ssl.BasicSSLContextFactory
        </factory-class-name>
        <properties>
            <property name="keyStore">keyStore</property>
            <property name="keyStorePassword">keyStorePassword</property>
            <property name="keyManagerAlgorithm">SunX509</property>
            <property name="trustManagerAlgorithm">SunX509</property>
            <property name="protocol">TLS</property>
        </properties>
    </ssl>
    ...
</network>
```
19.6. Credentials

Hazelcast client also has SSL support. You can configure Client SSL programmatically as shown below.

```java
Properties props = new Properties();
...
ClientConfig config = new ClientConfig();
config.getSocketOptions().setSocketFactory(new SSLSocketFactory(props));
```

You can also set `keyStore` and `keyStorePassword` with the following system properties.

- `javax.net.ssl.keyStore`
- `javax.net.ssl.keyStorePassword`

**NOTE:** You cannot use SSL when Hazelcast Encryption is enabled.

### 19.6 Credentials

**Hazelcast Enterprise**

One of the key elements in Hazelcast security is the `Credentials` object, which is used to carry all credentials of an endpoint (member or client). Credentials is an interface which extends `Serializable` and has three methods to implement. You can either implement the `Credentials` interface or extend the `AbstractCredentials` class, which is an abstract implementation of `Credentials`.

```java
package com.hazelcast.security;
public interface Credentials extends Serializable {
    String getEndpoint();
    void setEndpoint(String endpoint);
    String getPrincipal();
}
```

Hazelcast calls the `Credentials.setEndpoint()` method when an authentication request arrives at the node before authentication takes place.

```java
package com.hazelcast.security;
...
public abstract class AbstractCredentials implements Credentials, DataSerializable {
    private transient String endpoint;
    private String principal;
    ...
}
```

`UsernamePasswordCredentials`, a custom implementation of `Credentials`, is in the Hazelcast `com.hazelcast.security` package. `UsernamePasswordCredentials` is used for default configuration during the authentication process of both members and clients.

```java
package com.hazelcast.security;
...
public class UsernamePasswordCredentials extends Credentials {
    private byte[] password;
    ...
}
```
19.7 ClusterLoginModule

Hazelcast Enterprise

All security attributes are carried in the `Credentials` object. `Credentials` is used by `LoginModules` during the authentication process. User supplied attributes from `LoginModules` are accessed by `CallbackHandlers`. To access the `Credentials` object, Hazelcast uses its own specialized `CallbackHandler`. During initialization of `LoginModules`, Hazelcast passes this special `CallbackHandler` into the `LoginModule.initialize()` method.

`LoginModule` implementations should create an instance of `com.hazelcast.security.CredentialsCallback` and call the `handle(Callback[] callbacks)` method of `CallbackHandler` during the login process.

`CredentialsCallback.getCredentials()` returns the supplied `Credentials` object.

```java
public class CustomLoginModule implements LoginModule {
    CallbackHandler callbackHandler;
    Subject subject;

    public void initialize( Subject subject, CallbackHandler callbackHandler,
        Map<String, ?> sharedState, Map<String, ?> options ) {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
    }

    public final boolean login() throws LoginException {
        CredentialsCallback callback = new CredentialsCallback();
        try {
            callbackHandler.handle( new Callback[] { callback } );
            credentials = cb.getCredentials();
        } catch ( Exception e ) {
            throw new LoginException( e.getMessage() );
        }
        ...
        ...
    }
}
```

To use the default Hazelcast permission policy, you must create an instance of `com.hazelcast.security.ClusterPrincipal` that holds the `Credentials` object, and you must add it to `Subject.principals` on `LoginModule.commit()` as shown below.

```java
public class MyCustomLoginModule implements LoginModule {
    ...
    public boolean commit() throws LoginException {
        Principal principal = new ClusterPrincipal( credentials );
        subject.getPrincipals().add( principal );

        return true;
    }
    ...
}
```

Hazelcast has an abstract implementation of `LoginModule` that does callback and cleanup operations and holds the resulting `Credentials` instance. `LoginModules` extending `ClusterLoginModule` can access `Credentials, Subject, LoginModule` instances and options, and `sharedState` maps. Extending the `ClusterLoginModule` is recommended instead of implementing all required stuff.
19.8. CLUSTER MEMBER SECURITY

package com.hazelcast.security;
...
public abstract class ClusterLoginModule implements LoginModule {
    protected abstract boolean onLogin() throws LoginException;
    protected abstract boolean onCommit() throws LoginException;
    protected abstract boolean onAbort() throws LoginException;
    protected abstract boolean onLogout() throws LoginException;
}

19.7.1 Enterprise Integration

Using the above API, it should be possible to implement a LoginModule that performs authentication against the Security System of your choice, possibly an LDAP store like Apache Directory or some other corporate standard you have. For example, you may wish to have your clients send an identification token in the Credentials object. This token can then be sent to your back-end security system via the LoginModule that runs on the cluster side.

Additionally, the same system may authenticate the user and also then return the roles that are attributed to the user. These roles can then be used for data structure authorization.

RELATED INFORMATION
Please refer to JAAS Reference Guide for further information.

19.8 Cluster Member Security

Hazelcast Enterprise

Hazelcast supports standard Java Security (JAAS) based authentication between cluster members. To implement it, you configure one or more LoginModules and an instance of com.hazelcast.security.ICredentialsFactory. Although Hazelcast has default implementations using cluster group and group-password and UsernamePasswordCredentials on authentication, it is recommended that you implement the LoginModules and an instance of com.hazelcast.security.ICredentialsFactory according to your specific needs and environment.

<security enabled="true">
  <member-credentials-factory
    class-name="com.hazelcast.examples.MyCredentialsFactory">
    <properties>
      <property name="property1">value1</property>
      <property name="property2">value2</property>
    </properties>
  </member-credentials-factory>
  <member-login-modules>
    <login-module usage="required"
      class-name="com.hazelcast.examples.MyRequiredLoginModule">
      <properties>
        <property name="property3">value3</property>
      </properties>
    </login-module>
    <login-module usage="sufficient"
      class-name="com.hazelcast.examples.MySufficientLoginModule">
      <properties>
        <property name="property4">value4</property>
      </properties>
    </login-module>
    <login-module usage="optional"
      class-name="com.hazelcast.examples.MyOptionalLoginModule">
      <properties>
      </properties>
    </login-module>
  </member-login-modules>
</security>
You can define as many as LoginModules you wanted in configuration. They are executed in the given order. The usage attribute has 4 values: ‘required’, ‘requisite’, ‘sufficient’ and ‘optional’ as defined in javax.security.auth.login.AppConfigurationEntry>LoginModuleControlFlag.

```java
package com.hazelcast.security;
/**
 * ICredentialsFactory is used to create Credentials objects to be used
 * during node authentication before connection accepted by master node.
 */
public interface ICredentialsFactory {
    void configure( GroupConfig groupConfig, Properties properties );
    Credentials newCredentials();
    void destroy();
}
```

Properties defined in configuration are passed to the ICredentialsFactory.configure() method as java.util.Properties and to the LoginModule.initialize() method as java.util.Map.

### 19.9 Native Client Security

#### Hazelcast Enterprise

Hazelcast’s Client security includes both authentication and authorization.

#### 19.9.1 Authentication

The authentication mechanism works the same as cluster member authentication. To implement client authentication, configure a Credential and one or more LoginModules. The client side does not have and does not need a factory object to create Credentials objects like ICredentialsFactory. Credentials must be created at the client side and sent to the connected node during the connection process.

```xml
<security enabled="true">
    <client-login-modules>
        <login-module usage="required"
            class-name="com.hazelcast.examples.MyRequiredClientLoginModule">
            <properties>
                <property name="property3">value3</property>
            </properties>
        </login-module>
        <login-module usage="sufficient"
            class-name="com.hazelcast.examples.MySufficientClientLoginModule">
            <properties>
                <property name="property4">value4</property>
            </properties>
        </login-module>
    </client-login-modules>
</security>
```
19.9. NATIVE CLIENT SECURITY

You can define as many as LoginModules as you want in configuration. Those are executed in the given order. The usage attribute has 4 values: ‘required’, ‘requisite’, ‘sufficient’ and ‘optional’ as defined in javax.security.auth.login.AppConfigurationEntry.LoginModuleControlFlag.

```
<security enabled="true">
  <client-permissions>
    <!-- Principal 'admin' from endpoint '127.0.0.1' has all permissions. -->
    <all-permissions principal="admin">
      <endpoints>
        <endpoint>127.0.0.1</endpoint>
      </endpoints>
    </all-permissions>
    <!-- Principals named 'dev' from all endpoints have 'create', 'destroy', 'put', 'read' permissions for map named 'default'. -->
    <map-permission name="default" principal="dev">
      <actions>
        <action>create</action>
        <action>destroy</action>
        <action>put</action>
        <action>read</action>
      </actions>
    </map-permission>
    <!-- All principals from endpoints '127.0.0.1' or matching to '10.10.*.*' have 'put', 'read', 'remove' permissions for map whose name matches to 'com.foo.entity.*'. -->
    <map-permission name="com.foo.entity.*">
      <endpoints>
        <endpoint>10.10.*.*</endpoint>
        <endpoint>127.0.0.1</endpoint>
      </endpoints>
      <actions>
        <action>put</action>
      </actions>
    </map-permission>
  </client-permissions>
</security>
```

ClientConfig clientConfig = new ClientConfig();
clientConfig.setCredentials( new UsernamePasswordCredentials( "dev", "dev-pass" ) );
HazelcastInstance client = HazelcastClient.newHazelcastClient( clientConfig );

19.9.2 Authorization

Hazelcast client authorization is configured by a client permission policy. Hazelcast has a default permission policy implementation that uses permission configurations defined in the Hazelcast security configuration. Default policy permission checks are done against instance types (map, queue, etc.), instance names (map, queue, name, etc.), instance actions (put, read, remove, add, etc.), client endpoint addresses, and client principal defined by the Credentials object. Instance and principal names and endpoint addresses can be defined as wildcards(*). Please see the Network Configuration section and Using Wildcard section.
<action>read</action>
<action>remove</action>
</actions>
</map-permission>

<!-- Principals named 'dev' from endpoints matching to either
'192.168.1.1-100' or '192.168.2.*'
have 'create', 'add', 'remove' permissions for all queues. -->
<queue-permission name="*" principal="dev">
<endpoints>
  <endpoint>192.168.1.1-100</endpoint>
  <endpoint>192.168.2.*</endpoint>
</endpoints>
<actions>
  <action>create</action>
  <action>add</action>
  <action>remove</action>
</actions>
</queue-permission>

<!-- All principals from all endpoints have transaction permission. -->
<transaction-permission />
</client-permissions>
</security>

Users can also define their own policy by implementing com.hazelcast.security.IPermissionPolicy.

package com.hazelcast.security;

/**
 * IPermissionPolicy is used to determine any Subject’s permissions to perform a security sensitive Hazelcast operation.
 */
public interface IPermissionPolicy {
    void configure( SecurityConfig securityConfig, Properties properties );

    PermissionCollection getPermissions( Subject subject,
        Class<? extends Permission> type );

    void destroy();
}

Permission policy implementations can access client-permissions in configuration by using SecurityConfig.
getClientPermissionConfigs() during configure(SecurityConfig securityConfig, Properties properties) method is called by Hazelcast.

The IPermissionPolicy.getPermissions(Subject subject, Class<? extends Permission> type) method is used to determine a client request that has been granted permission to perform a security-sensitive operation.

Permission policy should return a PermissionCollection containing permissions of the given type for the given Subject. The Hazelcast access controller will call PermissionCollection.implies(Permission) on returning PermissionCollection and will decide if the current Subject has permission to access the requested resources or not.

19.9.3 Permissions

• All Permission
<all-permissions principal="principal">
  <endpoints>
    ...
  </endpoints>
</all-permissions>

- Map Permission

<map-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</map-permission>

Actions: all, create, destroy, put, read, remove, lock, intercept, index, listen

- Queue Permission

<queue-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</queue-permission>

Actions: all, create, destroy, add, remove, read, listen

- Multimap Permission

<multimap-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</multimap-permission>

Actions: all, create, destroy, put, read, remove, listen, lock

- Topic Permission

<topic-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</topic-permission>
Actions: create, destroy, publish, listen

- List Permission

```xml
<list-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</list-permission>
```

Actions: all, create, destroy, add, read, remove, listen

- Set Permission

```xml
<set-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</set-permission>
```

Actions: all, create, destroy, add, read, remove, listen

- Lock Permission

```xml
<lock-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</lock-permission>
```

Actions: all, create, destroy, lock, read

- AtomicLong Permission

```xml
<atomic-long-permission name="name" principal="principal">
  <endpoints>
    ...
  </endpoints>
  <actions>
    ...
  </actions>
</atomic-long-permission>
```

Actions: all, create, destroy, read, modify

- CountDownLatch Permission
19.9. NATIVE CLIENT SECURITY

Actions: all, create, destroy, modify, read

- Semaphore Permission

Actions: all, create, destroy, acquire, release, read

- Executor Service Permission

Actions: all, create, destroy

- Transaction Permission
Chapter 20

Performance

This chapter provides information on the performance features of Hazelcast including slow operations detector, back pressure and data affinity. Moreover, the chapter describes the best performance practices for Hazelcast deployed on Amazon EC2. It also describes the threading models for I/O, events, executors and operations.

20.1 Data Affinity

Data affinity ensures that related entries exist on the same node. If related data is on the same node, operations can be executed without the cost of extra network calls and extra wire data. This feature is provided by using the same partition keys for related data.

Co-location of related data and computation

Hazelcast has a standard way of finding out which member owns/manages each key object. The following operations will be routed to the same member, since all of them are operating based on the same key "key1".

```java
HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
Map mapA = hazelcastInstance.getMap( "mapA" );
Map mapB = hazelcastInstance.getMap( "mapB" );
Map mapC = hazelcastInstance.getMap( "mapC" );

// since map names are different, operation will be manipulating
// different entries, but the operation will take place on the
// same member since the keys ("key1") are the same
mapA.put( "key1", value );
mapB.get( "key1" );
mapC.remove( "key1" );

// lock operation will still execute on the same member
// of the cluster since the key ("key1") is same
hazelcastInstance.getLock( "key1" ).lock();

// distributed execution will execute the ‘Runnable’ on the
// same member since "key1" is passed as the key.
hazelcastInstance.getExecutorService().executeOnKeyOwner( runnable, "key1" );
```

When the keys are the same, entries are stored on the same node. But we sometimes want to have related entries stored on the same node, such as a customer and his/her order entries. We would have a customers map with customerId as the key and an orders map with orderId as the key. Since customerId and orderId are different keys, a customer and his/her orders may fall into different members/nodes in your cluster. So how can we have them stored on the same node? We create an affinity between customer and orders. If we make them part of the same partition then these entries will be co-located. We achieve this by making orderIds PartitionAware.
public class OrderKey implements Serializable, PartitionAware {
    private final long customerId;
    private final long orderId;

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

    public long getCustomerId() {
        return customerId;
    }

    public long getOrderId() {
        return orderId;
    }

    public Object getPartitionKey() {
        return customerId;
    }

    @Override
    public String toString() {
        return "OrderKey{" + "customerId=" + customerId + ", orderId=" + orderId + "}";
    }
}

Notice that OrderKey implements PartitionAware and that getPartitionKey() returns the customerId. This will make sure that the Customer entry and its Orders will be stored on the same node.

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();
Map mapCustomers = hazelcastInstance.getMap( "customers" );
Map mapOrders = hazelcastInstance.getMap( "orders" );

// create the customer entry with customer id = 1
mapCustomers.put( 1, customer );

// now create the orders for this customer
mapOrders.put( new OrderKey( 21, 1 ), order );
mapOrders.put( new OrderKey( 22, 1 ), order );
mapOrders.put( new OrderKey( 23, 1 ), order );

Assume that you have a customers map where customerId is the key and the customer object is the value. You want to remove one of the customer orders and return the number of remaining orders. Here is how you would normally do it.

public static int removeOrder( long customerId, long orderId ) throws Exception {
    IMap<Long, Customer> mapCustomers = instance.getMap( "customers" );
    IMap mapOrders = hazelcastInstance.getMap( "orders" );

    mapCustomers.lock( customerId );
    mapOrders.remove( orderId );
    Set orders = orderMap.keySet(Predicates.equal( "customerId", customerId ));
There are couple of things you should consider.

1. There are four distributed operations there: lock, remove, keySet, unlock. Can you reduce the number of distributed operations?
2. The customer object may not be that big, but can you not have to pass that object through the wire? Think about a scenario where you set order count to the customer object for fast access, so you should do a get and a put, and as a result, the customer object is passed through the wire twice.

Instead, why not move the computation over to the member (JVM) where your customer data resides. Here is how you can do this with distributed executor service.

1. Send a PartitionAware Callable task.
2. Callable does the deletion of the order right there and returns with the remaining order count.
3. Upon completion of the Callable task, return the result (remaining order count). You do not have to wait until the task is completed; since distributed executions are asynchronous, you can do other things in the meantime.

Here is some example code.

```java
HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance();

public int removeOrder( long customerId, long orderId ) throws Exception {
    IExecutorService executorService
        = hazelcastInstance.getExecutorService( "ExecutorService" );

    OrderDeletionTask task = new OrderDeletionTask( customerId, orderId );
    Future<Integer> future = executorService.submit( task );
    int remainingOrders = future.get();

    return remainingOrders;
}

public static class OrderDeletionTask
    implements Callable<Integer>, PartitionAware, Serializable {

private long customerId;
private long orderId;

public OrderDeletionTask() {
}

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

@Override
public Integer call() {
    Map<Long, Customer> customerMap = hazelcastInstance.getMap( "customers" );
    IMap<OrderKey, Order> orderMap = hazelcastInstance.getMap( "orders" );
```
mapCustomers.lock( customerId );
Customer customer = mapCustomers.get( customerId );
Predicate predicate = Predicates.equal( "customerId", customerId );
Set<OrderKey> orderKeys = orderMap.localKeySet( predicate );
int orderCount = orderKeys.size();
for (OrderKey key : orderKeys) {
    if (key.orderId == orderId) {
        orderCount--;
        orderMap.delete( key );
    }
}
mapCustomers.unlock( customerId );

return orderCount;
}

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

The benefits of doing the same operation with distributed ExecutorService based on the key are:

- Only one distributed execution (executorService.submit(task)), instead of four.
- Less data is sent over the wire.
- Since lock/update/unlock cycle is done locally (local to the customer data), lock duration for the Customer entry is much less, thus enabling higher concurrency.

20.2 Back Pressure

Hazelcast uses operations to make remote calls. For example, a map.get is an operation and a map.put is one operation for the primary and one operation for each of the backups, i.e. map.put is executed for the primary and also for each backup. In most cases, there will be a natural balance between the number of threads performing operations and the number of operations being executed. However, there are two situations where this balance and operations can pile up and eventually lead to Out of Memory Exception (OOME):

- Asynchronous calls: With async calls, the system may be flooded with the requests.
- Asynchronous backups: The asynchronous backups may be piling up.

To prevent the system from crashing, Hazelcast provides back pressure. Back pressure works by:

- limiting the number of concurrent operation invocations,
- periodically making an async backup sync.

Back pressure is disabled by default and you can enable it using the following system property:

```
hazelcast.backpressure.enabled
```

To control the number of concurrent invocations, you can configure the number of invocations allowed per partition using the following system property:

```
hazelcast.backpressure.max.concurrent.invocations.per.partition
```
The default value of this system property is 100. Using a default configuration a system is allowed to have \((271 + 1) \times 100 = 27200\) concurrent invocations (271 partitions + 1 for generic operations).

Back pressure is only applied to normal operations. System operations like heartbeats and partition migration operations are not influenced by back pressure. 27200 invocations might seem like a lot, but keep in mind that executing a task on `IExecutor` or acquiring a lock also requires an operation.

If the maximum number of invocations has been reached, Hazelcast will automatically apply an exponential back off policy. This gives the system some time to deal with the load. Using the following system property, you can configure the maximum time to wait before a `HazelcastOverloadException` is thrown:

```
hazelcast.backpressure.backoff.timeout.millis
```

This system property’s default value is 60000 ms.

The Health Monitor keeps an eye on the usage of the invocations. If it sees a member has consumed 70% or more of the invocations, it starts to log health messages.

Apart from controlling the number of invocations, you also need to control the number of pending async backups. This is done by periodically making these backups sync instead of async. This forces all pending backups to get drained. For this, Hazelcast tracks the number of asynchronous backups for each partition. At every \(N\)th call, one synchronization is forced. This \(N\) is controlled through the following property:

```
hazelcast.backpressure.syncwindow
```

This system property’s default value is 100. It means, out of 100 asynchronous backups, Hazelcast makes 1 of them a synchronous one. A randomization is added, so the sync window with default configuration will be between 75 and 125 invocations.

**RELATED INFORMATION**

Please refer to the System Properties section to learn how to configure the system properties.

### 20.3 Threading Model

Your application server has its own threads. Hazelcast does not use these - it manages its own threads.

#### 20.3.1 I/O Threading

Hazelcast uses a pool of threads for I/O. A single thread does not do all the IO: instead, multiple threads do the IO. On each cluster member, the IO-threading is split up in 3 types of IO-threads:

- IO-thread that takes care of accept requests,
- IO-threads that take care of reading data from other members/clients,
- IO-threads that take care of writing data to other members/clients.

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

In case of the read-IO-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, the Packet is handed over to the operation service (please see the Operation Threading section). If the Packet is an event, it is handed over to the event service (please see the Event Threading section).
20.3.2 Event Threading

Hazelcast uses a shared event system to deal with components that rely on events, such as topic, collections, listeners, and Near Cache.

Each cluster member has an array of event threads and each thread has its own work queue. When an event is produced, either locally or remotely, 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.

The following properties can be set to alter the behavior of the system.

- `hazelcast.event.thread.count`: Number of event-threads in this array. Its default value is 5.
- `hazelcast.event.queue.capacity`: Capacity of the work queue. Its default value is 1000000.
- `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 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 lot of 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. Topic A might have a lot of pending messages, and therefore B cannot receive messages because the queue has no capacity and messages for B are dropped.

20.3.3 IExecutor Threading

Executor threading is straight forward. 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. Thus, Executors are fully isolated, but still share the same underlying hardware; most importantly the CPUs.

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

20.3.4 Operation Threading

There are 2 types of operations:

- Operations that are aware of a certain partition, e.g. `IMap.get(key)`.
- Operations that are not partition aware, such as the `IExecutorService.executeOnMember(command, member)` operation.

Each of these operation types has a different threading model, explained below.

20.3.4.1 Partition-aware Operations

To execute partition-aware operations, an array of operation threads is created. The size of this array has a default value of two times the number of cores and a minimum value of 2. This value can be changed using the `hazelcast.operation.thread.count` property.

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

threadIndex = partitionId % partition-thread-count

After the threadIndex is determined, the operation is put in the work queue of that operation-thread. This means that:

- 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.
- each partition belongs to only 1 operation thread. All operations for a partition will always be handled by exactly the same operation-thread.
- 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, 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:

- false sharing of the partition: two completely independent data structures share the same partitions; e.g. if there is a map employees and a map orders, the method employees.get("peter") running on partition 25 may be blocked by a map.get() of orders.get(1234) also running on partition 25. If independent data structure share the same partition, a slow operation on one data structure can slow down the other data structures.
- false sharing of the partition-aware operation-thread: each operation-thread is responsible for executing operations of a number of partitions. For example, thread-1 could be responsible for partitions 0,10,20,… thread-2 for partitions 1,11,21,… etc. If an operation for partition 1 takes 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 starve operations of a thread. As a general rule, the partition thread should be released as soon as possible because operations are not designed to execute long running operations. That is why, for example, it is very dangerous to execute a long running operation using AtomicReference.alter() or an IMap.executeOnKey(), because these operations will block other 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.

20.3.4.2 Non Partition-aware Operations

To execute non partition-aware operations, e.g. 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 has a default value of the number of cores divided by two with a minimum value of 2. It can be changed using the hazelcast.operation.generic.thread.count property. This means that:

- 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.
20.3.4.3 Priority Operations

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

- 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.

- For non partition-aware operations: next to the `genericWorkQueue`, there also is a `genericPriorityWorkQueue`. So 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 be 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 else than trigger the worker to wake up and check the priority queue.

20.3.4.4 Operation-response and Invocation-future

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

```java
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 `genericWorkQueue` where it is executed by a “generic operation 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.

20.3.4.5 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.

20.4 SlowOperationDetector

The `SlowOperationDetector` monitors the operation threads and collects information about all slow operations. An Operation is a task executed by a generic or partition thread (see Operation Threading). An operation is considered as slow when it takes more computation time than the configured threshold.

The `SlowOperationDetector` stores the fully qualified classname of the operation and its stacktrace as well as operation details, start time and duration of each slow invocation. All collected data is available in the Management Center.

The `SlowOperationDetector` is configured via the following system properties.

- `hazelcast.slow.operation.detector.enabled`
20.4.1 Logging of Slow Operations

The detected slow operations are logged as warnings in the Hazelcast log files:

```
WARN 2015-05-07 11:05:30,890 SlowOperationDetector: [127.0.0.1]:5701
  Slow operation detected: com.hazelcast.map.impl.operation.PutOperation
  Hint: You can enable the logging of stacktraces with the following config property: hazelcast.slow.operation.detector.stacktrace.logging.enabled
```

Stacktraces are always reported to the Management Center, but by default they are not printed to keep the log size small. If logging of stacktraces is enabled, the full stacktrace is printed every 100 invocations. All other invocations print a shortened version.

20.4.2 Purging of Slow Operation Logs

Since a Hazelcast cluster can run for a very long time, Hazelcast purges the slow operation logs periodically to prevent an OOME. You can configure the purge interval and the retention time for each invocation.

The purging removes each invocation whose retention time is exceeded. When all invocations are purged from a slow operation log, the log is deleted.

20.5 Hazelcast Performance on AWS

Amazon Web Services (AWS) platform can be an unpredictable environment compared to traditional in-house data centers. This is because the machines, databases or CPUs are shared with other unknown applications in the cloud, causing fluctuations. When you gear up your Hazelcast application from a physical environment to Amazon EC2, you should configure it so that any network outage or fluctuation is minimized and its performance is maximized. This section provides notes on improving the performance of Hazelcast on AWS.

20.5.1 Selecting EC2 Instance Type

Hazelcast is an in-memory data grid that distributes the data and computation to the nodes that are connected with a network, making Hazelcast very sensitive to the network. Not all EC2 Instance types are the same in terms of the network performance. It is recommended that you choose instances that have 10 Gigabit or High network performance for Hazelcast deployments. Please see the below table for the recommended instances.

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>Network Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>m3.2xlarge</td>
<td>High</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>High</td>
</tr>
</tbody>
</table>
### Instance Type Network Performance

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>Network Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>c3.2xlarge</td>
<td>High</td>
</tr>
<tr>
<td>c3.4xlarge</td>
<td>High</td>
</tr>
<tr>
<td>c3.8xlarge</td>
<td>10 Gigabit</td>
</tr>
<tr>
<td>c1.xlarge</td>
<td>High</td>
</tr>
<tr>
<td>cc2.8xlarge</td>
<td>10 Gigabit</td>
</tr>
<tr>
<td>m2.4xlarge</td>
<td>High</td>
</tr>
<tr>
<td>cr1.8xlarge</td>
<td>10 Gigabit</td>
</tr>
</tbody>
</table>

#### 20.5.2 Dealing with Network Latency

Since data is sent and received very frequently in Hazelcast applications, latency in the network becomes a crucial issue. In terms of the latency, AWS cloud performance is not the same for each region. There are vast differences in the speed and optimization from region to region.

When you do not pay attention to AWS regions, Hazelcast applications may run tens or even hundreds of times slower than necessary. The following notes are potential workarounds.

- Create a cluster only within a region. It is not recommended that you deploy a single cluster that spans across multiple regions.
- If a Hazelcast application is hosted on Amazon EC2 instances in multiple EC2 regions, you can reduce the latency by serving the end users' requests from the EC2 region which has the lowest network latency. Changes in network connectivity and routing result in changes in the latency between hosts on the Internet. Amazon has a web service (Route 53) that lets the cloud architects use DNS to route end-user requests to the EC2 region that gives the fastest response. This latency-based routing is based on latency measurements performed over a period of time. Please have a look at Route53.
- Move the deployment to another region. The CloudPing tool gives instant estimates on the latency from your location. By using it frequently, CloudPing can be helpful to determine the regions which have the lowest latency.
- The SpeedTest tool allows you to test the network latency and also the downloading/uploading speeds.

#### 20.5.3 Selecting Virtualization

AWS uses two virtualization types to launch the EC2 instances: Para-Virtualization (PV) and Hardware-assisted Virtual Machine (HVM). According to the tests we performed, HVM provided up to three times higher throughput than PV. Therefore, we recommend you use HVM when you run Hazelcast on EC2.
Chapter 21

Hazelcast Simulator

Hazelcast Simulator is a production simulator used to test Hazelcast and Hazelcast-based applications in clustered environments. It also allows you to create your own tests and perform them on your Hazelcast clusters and applications that are deployed to cloud computing environments. In your tests, you can provide any property that can be specified on these environments (Amazon EC2, Google Compute Engine(GCE), or your own environment): properties such as hardware specifications, operating system, Java version, etc.

Hazelcast Simulator allows you to add potential production problems, such as real-life failures, network problems, overloaded CPU, and failing nodes to your tests. It also provides a benchmarking and performance testing platform by supporting performance tracking and also supporting various out-of-the-box profilers.

Hazelcast Simulator makes use of Apache jclouds®, an open source multi-cloud toolkit that is primarily designed for testing on the clouds like Amazon EC2 and GCE.

You can use Hazelcast Simulator for the following use cases:

- In your pre-production phase to simulate the expected throughput/latency of Hazelcast with your specific requirements.
- To test if Hazelcast behaves as expected when you implement a new functionality in your project.
- As part of your test suite in your deployment process.
- When you upgrade your Hazelcast version.

Hazelcast Simulator is available as a downloadable package on the Hazelcast web site. Please refer to the Installing Simulator section for more information.

21.1 Key Concepts

The following are the key concepts mentioned with Hazelcast Simulator.

- **Test** - A test class for the functionality you want to test, such as a Hazelcast map. This test class may seem like a JUnit test, but it uses custom annotations to define methods for different test phases (e.g. setup, warmup, run, verify).

- **TestSuite** - A property file that contains the name of the test class and the properties you want to set on that test class instance. In most cases, a TestSuite contains a single test class, but you can configure multiple tests within a single TestSuite.

- **Failure** - An indication that something has gone wrong. Failures are picked up by the Agent and sent back to the Coordinator. Please see the descriptions below for the Agent and Coordinator.

- **Worker** - A Java Virtual Machine (JVM) responsible for running a TestSuite. It can be configured to spawn a Hazelcast client or member instance.
• **Agent** - A JVM installed on a piece of hardware. Its main responsibility is spawning, monitoring and terminating **Workers**.

• **Coordinator** - A JVM that can run anywhere, such as on your local machine. **Coordinator** is actually responsible for running the test using the **Agents**. You configure it with a list of **Agent** IP addresses, and you run it by sending a command like “run this testsuite with 10 worker JVMs for 2 hours”.

• **Provisioner** - Spawns and terminates cloud instances, and installs **Agents** on the remote machines. It can be used in combination with EC2 (or any other cloud), but it can also be used in a static setup, such as a local machine or a cluster of machines in your data center.

• **Communicator** - A JVM that enables the communication between the **Agents** and **Workers**.

• **simulator.properties** - The configuration file you use to adapt the Hazelcast Simulator to your business needs (e.g. cloud selection and configuration).

### 21.2 Installing Simulator

Hazelcast Simulator needs a Unix shell to run. Ensure that your local and remote machines are running under Unix, Linux or Mac OS. Hazelcast Simulator may work with Windows using a Unix-like environment such as Cygwin, but that is not officially supported at the moment.

#### 21.2.1 Firewall settings

Please ensure that all remote machines are reachable via TCP ports 22, 9000 and 5701 to 5751 on their external network interface (for example, `eth0`). The first two ports are used by Hazelcast Simulator. The other ports are used by Hazelcast itself. Port 9001 is used on the loopback device on all remote machines for local communication.
21.2.2 Setup of local machine (Coordinator)

Hazelcast Simulator is provided as a separate downloadable package, in zip or tar.gz format. You can download either one here.

After the download is completed, follow the below steps.

- Unpack the tar.gz or zip file to a folder that you prefer to be the home folder for Hazelcast Simulator. The file extracts with the name hazelcast-simulator-<version>. (If you are updating Hazelcast Simulator, perform this same unpacking, but skip the following steps.)

- Add the following lines to the file ~/.bashrc (for Unix/Linux) or to the file ~/.profile (for Mac OS).

  ```bash
  export SIMULATOR_HOME=<extracted folder path>/hazelcast-simulator-<version>
  PATH=$SIMULATOR_HOME/bin:$PATH
  ```

- Create a working folder for your Simulator TestSuite (tests is an example name in the following commands).

  ```bash
  mkdir ~/tests
  ```

- Copy the simulator.properties file to your working folder.

  ```bash
  cp $SIMULATOR_HOME/conf/simulator.properties ~/tests
  ```

21.2.3 Setup of remote machines (Agents, Workers)

After you have installed Hazelcast Simulator as described in the previous section, make sure you create a user on the remote machines upon which you want to run Agents and Workers. The default username used by Hazelcast Simulator is simulator. You can change this in the simulator.properties file in your working folder.

Please ensure that you can connect to the remote machines with the configured username and without password authentication (see the next section). The Provisioner terminates when it needs to access the remote machines and cannot connect automatically.

21.2.4 Setup of public/private key pair

The preferred method for password free authentication is using an RSA (Rivest, Shamir and Adleman cryptosystem) public/private key pair. The RSA key should not require you to enter the pass-phrase manually. A key with a pass-phrase and ssh-agent-forwarding is strongly recommended, but a key without a pass-phrase also works.

21.2.4.1 Local machine (Coordinator)

Make sure you have the files id_rsa.pub and id_rsa in your local ~/.ssh folder.

If you do not have the RSA keys, you can generate a public/private key pair using the following command.

```bash
ssh-keygen -t rsa -C "your_email@example.com"
```

Press [Enter] for all questions. The value for the e-mail address is not relevant in this case. After you execute this command, you should have the files id_rsa.pub and id_rsa in your ~/.ssh folder.

21.2.4.2 Remote machines (Agents, Workers)

Please ensure you have appended the public key (id_rsa.pub) to the ~/.ssh/authorized_keys file on all remote machines (Agents and Workers). You can copy the public key to all your remote machines using the following command.

```bash
ssh-copy-id -i ~/.ssh/id_rsa.pub simulator@remote-ip-address
```
21.2.4.3 SSH connection test

You can check if the connection works as expected using the following command from the Coordinator machine (it will print ok if everything is fine).

```
ssh -o BatchMode=yes simulator@remote-ip-address "echo ok" 2>&1
```

21.3 Setting Up For Amazon EC2

Having installed the Simulator, this section describes how to prepare the Simulator for testing a Hazelcast cluster deployed at Amazon EC2.

To do this, copy the file `SIMULATOR_HOME/conf/simulator.properties` to your working folder and edit this file. You should set the values for the following parameters that are included in this file.

- **CLOUD_PROVIDER**: Maven artifact ID of the cloud provider. In this case it is `aws-ec2` for Amazon EC2. Please refer to the Simulator.Properties File Description section for a full list of cloud providers.
- **CLOUD_IDENTITY**: The path to the file that contains your EC2 access key.
- **CLOUD_CREDENTIAL**: The path to the file that contains your EC2 secret key.
- **MACHINE_SPEC**: The parameter by which you can specify the EC2 instance type, operating system of the instance, EC2 region, etc.

The following is an example of a `simulator.properties` file with the parameters explained above. For this example, you should have created the files `~/ec2.identity` and `~/ec2.credential` that contain your EC2 access key and secret key, respectively.

```
CLOUD_PROVIDER=aws-ec2
CLOUD_IDENTITY=~/ec2.identity
CLOUD_CREDENTIAL=~/ec2.credential
MACHINE_SPEC=hardwareId=c3.xlarge,imageId=us-east-1/ami-1b3b2472
```

**NOTE**: Creating these files in your working folder instead of just setting the access and secret keys in the `simulator.properties` file is for security reasons. It is too easy to share your credentials with the outside world; now you can safely add the `simulator.properties` file in your source repository or share it with other people.

**NOTE**: For the full description of the `simulator.properties` file, please refer to the Simulator.Properties File Description section.

21.4 Setting Up For Google Compute Engine

To prepare the Simulator for testing a Hazelcast cluster deployed at Google Compute Engine (GCE), first you need an e-mail address to be used as a GCE service account. You can obtain this e-mail address in the Admin GUI console of GCE. In this console, select **Credentials** in the menu **API & Auth**. Then, click the **Create New Client ID** button and select **Service Account**. Usually, this e-mail address is in this form: `<your account ID>@developer.gserviceaccount.com`.

Save the `.p12` keystore file that you obtained while creating your Service Account (you will refer to that path). In the `bin` folder of the Hazelcast Simulator package that you downloaded, edit the `setupGce.sh` script to specify the following parameters:

- **GCE_id**: Your developer e-mail address that you obtained in the Admin GUI console of GCE.
- **p12File**: The path to your `.p12` file you saved while you were obtaining your developer e-mail address.

After you run the edited `setupGce.sh` script, the `simulator.properties` file that you need for a proper testing of your instances on GCE is created in the `conf` folder of Hazelcast Simulator.
21.5 Setting Up Machines Manually

You may want to set up Hazelcast Simulator on the environments different than your clusters placed on a cloud: for example, your local machines, a test laboratory, etc. In this case, perform the following steps.

1. Copy the SIMULATOR_HOME/conf/simulator.properties to your working directory.

2. Edit the USER in the simulator.properties file if you want to use a different user name than simulator.

3. Create an RSA key pair or use an existing one. Using the key should not require entering the pass-phrase manually. A key with pass-phrase and ssh-agent-forwarding is strongly recommended, but a key without a pass-phrase will also work.

You can check whether a key pair exists with this command:

```
ls -al ~/.ssh
```

If it does not exist, you can create a key pair on the client machine with this command:

```
ssh-keygen -t rsa
```

You will get a few more questions:

* Enter a file in which to save the key (/home/demo/.ssh/id_rsa):
* Enter a pass-phrase (empty for no pass-phrase): (pass-phrase is optional)

4. Copy the public key into the ~/.ssh/authorized_keys file on the remote machines with this command:

```
ssh-copy-id user@123.45.56.78
```

5. Create the agents.txt file and add the IP addresses of the machines. The content of the agents.txt file with the IP addresses added looks like the following:

```
98.76.65.54 10.28.37.46
```

6. Run the command provisioner --restart to verify.

**NOTE:** For the full description of the simulator.properties file, please refer to the Simulator.Properties File Description section.

21.6 Executing a Simulator Test

After you install and prepare the Hazelcast Simulator for your environment, it is time to perform a test. In the following sections, you are going to verify the setup by running a simple map test with strings as keys and values.

You can start with creating the working folder.

```
mkdir simulator-example
```

A path of working folder needs to be visible in the output of the provisioner/coordinator.
21.6.1 Creating and Editing Properties File

You need to create the file `test.properties` in the working folder. Execute the following command to create and edit this file.

```
cat > test.properties
```

Copy the following lines into the file `test.properties`.

```
class=com.hazelcast.simulator.tests.map.StringStringMapTest
threadCount=10
keyLocality=Random
keyLength=300
valueLength=300
keyCount=100000
putProb=0.2
basename=map
```

The property `class` defines the actual test case and the rest are the properties you want to bind to your test. If a property is not defined in this file, the default value of the property given in your test code is used. Please see the `properties` comment in the `StringStringMapTest`. You will see the following.

```
// properties
public int keyLength = 10;
public int valueLength = 10;
public int keyCount = 10000;
public int valueCount = 10000;
public String basename = "stringStringMap";
public KeyLocality keyLocality = KeyLocality.RANDOM;
public int minNumberOfMembers = 0;
```

After you created the file `test.properties` and set your properties successfully, you need to configure the simulator using the file `simulator.properties`.

Execute the following command to create and edit this file.

```
cat > simulator.properties
```

Copy the following lines into this file and set the properties.

```
CLOUD_PROVIDER=aws-ec2
CLOUD_IDENTITY=/~ec2.identity
CLOUD_CREDENTIAL=/~ec2.credential
MACHINE_SPEC=hardwareId=m3.medium,locationId=us-east-1,imageId=us-east-1/ami-fb8e9292
JDK_FLAVOR=oracle
JDK_VERSION=7
```

Please refer to here for information on `CLOUD_IDENTITY` and `CLOUD_CREDENTIAL`.

**NOTE:** For a full description of the file `simulator.properties`, please see the Simulator.Properties File Description section. You can find the sample simulator properties in the `dist/simulator-tests/simulator.properties`. You can also copy this file to the working folder and then edit according to your needs.
21.6. EXECUTING A SIMULATOR TEST

21.6.2 Running the Test

When in the working folder, execute the following commands step by step to run the test.

1. Starting Instances

First of all, you need agents to run the test on them. Execute the following command to start 4 EC2 instances and install Java and the agents to these instances.

```
provisioner --scale 4
```

The output of the command looks like the following.

```
INFO 09:05:06 Hazelcast Simulator Provisioner
INFO 09:05:06 Version: 0.5, Commit: c6e82c5, Build Time: 18.06.2015 @ 11:58:06 UTC
INFO 09:05:06 SIMULATOR_HOME: /disk1/hazelcast-simulator-0.5
INFO 09:05:07 Loading simulator.properties: /disk1/exampleSandbox/simulator.properties
INFO 09:05:07 =============================================================
INFO 09:05:07 Provisioning 4 aws-ec2 machines
INFO 09:05:07 =============================================================
INFO 09:05:07 Current number of machines: 0
INFO 09:05:07 Desired number of machines: 4
INFO 09:05:07 Using init script:/disk1/hazelcast-simulator-0.5/conf/init.sh
INFO 09:05:07 JDK spec: oracle 7
INFO 09:05:07 Hazelcast version-spec: outofthebox
INFO 09:05:11 Created compute
INFO 09:05:11 Machine spec: hardwareId=m3.medium,locationId=us-east-1,imageId=us-east-1/ami-fb8e9292
INFO 09:05:18 Created template
INFO 09:05:18 Login name to the remote machines: simulator
INFO 09:05:18 Security group: 'simulator' is found in region 'us-east-1'
INFO 09:05:18 Creating machines... (can take a few minutes)
INFO 09:06:18 54.211.146.186 LAUNCHED
INFO 09:06:18 54.166.1.79 LAUNCHED
INFO 09:06:18 54.147.196.63 LAUNCHED
INFO 09:06:18 54.144.235.111 LAUNCHED
INFO 09:06:30 54.211.146.186 JAVA INSTALLED
INFO 09:06:32 54.166.1.79 JAVA INSTALLED
INFO 09:06:32 54.144.235.111 JAVA INSTALLED
INFO 09:06:34 54.147.196.63 JAVA INSTALLED
INFO 09:06:40 54.166.1.79 SIMULATOR AGENT INSTALLED
INFO 09:06:40 Killing Agent on: 54.166.1.79
INFO 09:06:40 Starting Agent on: 54.166.1.79
INFO 09:06:40 54.211.146.186 SIMULATOR AGENT INSTALLED
INFO 09:06:40 Killing Agent on: 54.211.146.186
INFO 09:06:40 Starting Agent on: 54.211.146.186
INFO 09:06:40 54.166.1.79 SIMULATOR AGENT STARTED
INFO 09:06:40 Starting Agent on: 54.211.146.186
INFO 09:06:40 54.211.146.186 SIMULATOR AGENT STARTED
INFO 09:06:42 54.144.235.111 SIMULATOR AGENT INSTALLED
INFO 09:06:42 Killing Agent on: 54.144.235.111
INFO 09:06:42 Starting Agent on: 54.144.235.111
INFO 09:06:43 54.144.235.111 SIMULATOR AGENT STARTED
INFO 09:06:47 54.147.196.63 SIMULATOR AGENT INSTALLED
INFO 09:06:47 Killing Agent on: 54.147.196.63
INFO 09:06:47 Starting Agent on: 54.147.196.63
INFO 09:06:47 54.147.196.63 SIMULATOR AGENT STARTED
INFO 09:06:47 Pausing for machine warmup... (10 sec)
INFO 09:06:57 Duration: 00d 00h 01m 49s
INFO 09:06:57 ==============================================================
INFO 09:06:57 Successfully provisioned 4 aws-ec2 machines
```
INFO 09:06:57 ==============================================================
INFO 09:06:57 Shutting down Provisioner...
INFO 09:06:57 Done!

You can also see the file `agents.txt` that was created automatically by the provisioner in the working folder. The file `agents.txt` includes IP addresses of the started EC2 instances. You can see this file’s content using the following command.

```
less agents.txt
```

First column lists the public IP addresses and the second one lists the private IP addresses. A public IP address is used for the communication between the coordinator and agent. A private IP address is used for the communications between client and member and also between member and member. A private IP address cannot be connected to from the outside of EC2 environment.

### 2. Running the Test Suite

After you created the instances and agents are installed to them, execute the following command to run your test suite.

```
coordinator test.properties
```

Please refer to the Coordinators section for detailed information about the arguments of `coordinator`.

The output looks like the following.

```
INFO 09:57:17 Hazelcast Simulator Coordinator
INFO 09:57:17 Version: 0.5, Commit: c6e82c5, Build Time: 02.07.2015 @ 09:50:21 UTC
INFO 09:57:17 SIMULATOR_HOME: /disk1/hazelcast-simulator-0.5
INFO 09:57:17 Loading simulator.properties: /disk1/exampleSandbox/simulator.properties
INFO 09:57:17 Loading testsuite file: /disk1/exampleSandbox/test.properties
INFO 09:57:17 Loading Hazelcast configuration: /disk1/hazelcast-simulator-0.5/conf/hazelcast.xml
INFO 09:57:17 Loading Hazelcast client configuration: /disk1/hazelcast-simulator-0.5/conf/client-hazelcast.xml
INFO 09:57:17 Loading Log4j configuration for worker: /disk1/hazelcast-simulator-0.5/conf/worker-log4j.properties
INFO 09:57:17 Loading agents file: /disk1/exampleSandbox/agents.txt
INFO 09:57:17 HAZELCAST_VERSION_SPEC: maven=3.5
INFO 09:57:17 --------------------------------------------------------------
INFO 09:57:17 Waiting for agents to start
INFO 09:57:17 --------------------------------------------------------------
INFO 09:57:17 Connect to agent 54.211.146.186 OK
INFO 09:57:17 Connect to agent 54.166.1.79 OK
INFO 09:57:17 Connect to agent 54.147.196.63 OK
INFO 09:57:17 Connect to agent 54.144.235.111 OK
INFO 09:57:17 --------------------------------------------------------------
INFO 09:57:17 All agents are reachable!
INFO 09:57:17 --------------------------------------------------------------
INFO 09:57:21 Performance monitor enabled: false
INFO 09:57:21 Total number of agents: 4
INFO 09:57:21 Total number of Hazelcast member workers: 4
INFO 09:57:21 Total number of Hazelcast client workers: 0
INFO 09:57:21 Agent 54.211.146.186 members: 1 clients: 0 mode: MIXED
INFO 09:57:21 Agent 54.166.1.79 members: 1 clients: 0 mode: MIXED
INFO 09:57:21 Agent 54.147.196.63 members: 1 clients: 0 mode: MIXED
INFO 09:57:21 Agent 54.144.235.111 members: 1 clients: 0 mode: MIXED
INFO 09:57:21 Killing all remaining workers
INFO 09:57:21 Successfully killed all remaining workers
INFO 09:57:21 Starting 4 member workers
INFO 09:57:41 Successfully started member workers
INFO 09:57:41 Skipping client startup, since no clients are configured
INFO 09:57:41 Successfully started a grand total of 4 Workers JVMs after 20120 ms
INFO 09:57:41 Starting testsuite: 2015-07-02__09_57_17
INFO 09:57:41 Tests in testsuite: 1
INFO 09:57:41 Running time per test: 00d 00h 01m 00s
INFO 09:57:41 Expected total testsuite time: 00d 00h 01m 00s
INFO 09:57:41 Running 1 tests sequentially
INFO 09:57:41 --------------------------------------------------------------
Running Test:
TestCase{  
id=  
, class=com.hazelcast.simulator.tests.map.StringStringMapTest  
, keyCount=100000  
, keyLength=300  
, keyLocality=Random  
, putProb=0.2  
, threadCount=10  
, valueLength=300  
}
--------------------------------------------------------------
INFO 09:57:41 Starting Test initialization
INFO 09:57:42 Completed Test initialization
INFO 09:57:42 Starting Test setup
INFO 09:57:44 Completed Test setup
INFO 09:57:44 Starting Test local warmup
INFO 09:57:46 Waiting for localWarmup completion: 00d 00h 00m 00s
INFO 09:57:52 Waiting for localWarmup completion: 00d 00h 00m 06s
INFO 09:57:57 Waiting for localWarmup completion: 00d 00h 00m 12s
INFO 09:58:03 Waiting for localWarmup completion: 00d 00h 00m 18s
INFO 09:58:09 Waiting for localWarmup completion: 00d 00h 00m 24s
INFO 09:58:15 Waiting for localWarmup completion: 00d 00h 00m 30s
INFO 09:58:20 Waiting for localWarmup completion: 00d 00h 00m 35s
INFO 09:58:26 Waiting for localWarmup completion: 00d 00h 00m 41s
INFO 09:58:32 Completed Test local warmup
INFO 09:58:32 Starting Test global warmup
INFO 09:58:33 Completed Test global warmup
INFO 09:58:33 Starting Test start
INFO 09:58:34 Completed Test start
INFO 09:58:34 Test will run for 00d 00h 01m 00s
INFO 09:59:04 Running 00d 00h 00m 30s 50.00% complete
INFO 09:59:34 Running 00d 00h 01m 00s 100.00% complete
INFO 09:59:34 Test finished running
INFO 09:59:34 Starting Test stop
INFO 09:59:36 Completed Test stop
INFO 09:59:37 Starting Test global verify
INFO 09:59:39 Completed Test global verify
INFO 09:59:39 Starting Test local verify
INFO 09:59:41 Completed Test local verify
INFO 09:59:41 Starting Test global tear down
INFO 09:59:43 Finished Test global tear down
INFO 09:59:43 Starting Test local tear down
INFO 09:59:45 Completed Test local tear down
INFO 09:59:45 Terminating workers
INFO 09:59:45 All workers have been terminated
INFO 09:59:45 Starting cool down (10 sec)
INFO 09:59:55 Finished cool down
INFO 09:59:55 Total running time: 133 seconds
INFO 09:59:55 -----------------------------------------------------------------
Now you need the logs and results that the workers generated. You can get these requirements from agents via `provisioner`.

`provisioner --download`

The output looks like the following.

```
INFO 10:05:41 Hazelcast Simulator Provisioner
INFO 10:05:41 Version: 0.5, Commit: c6e82c5, Build Time: 02.07.2015 @ 09:50:21 UTC
INFO 10:05:41 SIMULATOR_HOME: /disk1/hazelcast-simulator-0.5
INFO 10:05:41 Loading simulator.properties: /disk1/exampleSandbox/simulator.properties
INFO 10:05:42 Download artifacts of 4 machines
INFO 10:05:42 Downloading from 54.211.146.186
INFO 10:05:42 Downloading from 54.166.1.79
INFO 10:05:42 Downloading from 54.147.196.63
INFO 10:05:42 Downloading from 54.144.235.111
INFO 10:05:43 Finished Downloading Artifacts of 4 machines
INFO 10:05:43 Shutting down Provisioner...
INFO 10:05:43 Done!
```

The artifacts (log files) are downloaded into the `workers` subfolder of the working folder.

### 4. Terminating the Instances

If want to terminate the instances, execute the following command.

`provisioner --terminate`

If an EC2 machine with an agent running is idle for 2 hours, that machine will automatically terminate itself to prevent running into a big bill.

The output looks like the following.

```
INFO 10:26:46 Hazelcast Simulator Provisioner
INFO 10:26:46 Version: 0.5, Commit: c6e82c5, Build Time: 02.07.2015 @ 09:50:21 UTC
INFO 10:26:46 SIMULATOR_HOME: /disk1/hazelcast-simulator-0.5
INFO 10:26:46 Loading simulator.properties: /disk1/exampleSandbox/simulator.properties
INFO 10:26:46 Terminating 4 aws-ec2 machines (can take some time)
INFO 10:26:46 Terminating 4 of 4, remaining=0
INFO 10:26:46 Shutting down Provisioner...
INFO 10:26:46 Done!
```
21.6.3 Running the Test with a Script

Another option to run the test is using a script. Execute the following command to create a script called, for example, run.sh.

```
cat > run.sh
```

This option is for your convenience. It gathers all the commands used to perform a test into one script. The following is the content of this example run.sh script.

```
#!/bin/bash
set -e
provisioner --scale 4
test -e coordinator test.properties
provisioner --download
```

Note that you should make the script run.sh executable executing the following command.

```
chmod +x run.sh
```

**RELATED INFORMATION**

Please see the Provisioner section and the Coordinator section for more provisioner and coordinator commands.

21.6.4 Using Maven Archetypes

Alternatively, you can execute tests using the Simulator archetype. Please see the following.

```
mvn archetype:generate \
   -DarchetypeGroupId=com.hazelcast.simulator \
   -DarchetypeArtifactId=archetype \
   -DarchetypeVersion=0.5 \
   -DgroupId=yourgroupid \
   -DartifactId=yourproject
```

This creates a fully working Simulator project, including the test having yourgroupid.

1. After this project is generated, go to the created folder and execute the following command.
   
   ```
mvn clean install
   ```

2. Then, go to your working folder.
   
   ```
cd <working folder>
   ```

3. Edit the simulator.properties file as explained in the Editing the Simulator.Properties File section.

4. Run the test from your working folder using the following command.
   
   ```
./run.sh
   ```

The output is the same as shown in the Running the Test section.
21.7 Provisioner

The provisioner is responsible for provisioning (starting/stopping) instances in a cloud. It will start an Operating System instance, install Java, open firewall ports and install Simulator Agents.

You can configure the behavior of the cluster—such as cloud, operating system, hardware, JVM version, Hazelcast version or region—through the file *simulator.properties*. Please see the Simulator.Properties File Description section for more information.

You can use the following arguments with the *provisioner*.

To start a cluster:

```
provisioner --scale 1
```

To scale to a 2 member cluster:

```
provisioner --scale 2
```

To scale back to a 1 member cluster:

```
provisioner --scale 1
```

To terminate all members in the cluster:

```
provisioner --terminate
```

or

```
provisioner --scale 0
```

If you want to restart all agents and also upload the newest JARs to the machines:

```
provisioner --restart
```

To download all the worker home folders (containing logs and whatever has been put inside):

```
provisioner --download
```

This command is also useful if you added a profiling because the profiling information will also be downloaded. The command is also useful when an out of memory exception is thrown because you can download the heap dump.

To remove all the worker home directories:

```
provisioner --clean
```

21.7.1 Accessing the Provisioned Machine

When a machine is provisioned, a user with the name *simulator* is created on the remote machine by default, and that user is added to the sudousers list. Also, the public key of your local user is copied to the remote machine and added to the file `~/.ssh/authorized_keys`. You can login to that machine using the following command.

```
ssh simulator@ip
```

You can change the name of the created user to something else by setting the USER=<somename> property in the file *simulator.properties*. Be careful not to pick a name that is used on the target image: for example, if you use `ec2-user/ubuntu`, and the default user of that image is `ec2-user/ubuntu`, then you can run into authentication problems.
21.8 Coordinator

The Coordinator is responsible for actually running the test using the agents. You can deploy your test on the workers using the following command.

```
coordinator yourtest.properties.
```

This command creates a single worker per agent and runs the test for 60 seconds (the default duration for a Hazelcast Simulator test).

If your test properties file is called `test.properties`, then you can use the following command to have the coordinator pick up your `test.properties` file automatically.

```
coordinator
```

### 21.8.1 Controlling Hazelcast Declarative Configuration

By default, the coordinator uses the files `SIMULATOR_HOME/conf/hazelcast.xml` and `SIMULATOR_HOME/conf/client-hazelcast.xml` to generate the correct Hazelcast configuration. To use your own configuration files instead, use the following arguments:

```
coordinator --clientHzFile=your-client-hazelcast.xml --hzFile your-hazelcast.xml ....
```

### 21.8.2 Controlling Test Duration

You can control the duration of a single test using the `--duration` argument. The default duration is 60 seconds. You can specify your own durations using `m` for minutes, `d` for days or `s` for seconds with this argument.

You can see the usage of the `--duration` argument in the following example commands.

```
coordinator --duration 90s map.properties
coordinator --duration 3m map.properties
coordinator --duration 12h map.properties
coordinator --duration 2d map.properties
```

### 21.8.3 Controlling Client And Workers

By default, the provisioner starts the cluster members. You can also use the `--memberWorkerCount` and `--clientWorkerCount` arguments to control how many members and clients you want to have.

The following command creates a 4 node Hazelcast cluster and 8 clients, and all load will be generated through the clients. It also runs the `map.properties` test for a duration of 12 hours.

```
coordinator --memberWorkerCount 4 --clientWorkerCount 8 --duration 12h map.properties
```

Profiles are usually configured with some clients and some members. If you want to have members and no clients:

```
coordinator --memberWorkerCount 12 --duration 12h map.properties
```

If you want to have a JVM with embedded client plus member and all communication goes through the client:
CHAPTER 21. HAZELCAST SIMULATOR

coordinator --mixedWorkerCount 12 --duration 12h map.properties

If you want to run 2 member JVMs per machine:

cordinator --memberWorkerCount 24 --duration 12h map.properties

As you notice, you can play with the actual deployment.

21.9 Communicator

Communicator enables you to pass messages to Agents, Workers and Tests. You can use messages to simulate various conditions: for example, Hazelcast discomforts like network partitioning and high CPU utilization.

21.9.1 Example

$ communicator --message-address Agent=*,Worker=* spinCore

This will send the message spinCore to all Workers.

Each interaction with Communicator has to specify:

- Message Type
- Message Address

21.9.2 Message Types

- kill - Kills a JVM running a message recipient. In practice, you probably want to send this message to Worker(s) only. The reason for this is you rarely want to kill an Agent and it does not make sense to send this to just a single test; it would kill other tests sharing the same JVM as well.
- blockHzTraffic - Blocks the incoming traffic to TCP port range 5700:5800.
- newMember - Starts a new member. You can send this message to Agents only.
- softKill - Instructs a JVM that is running a message recipient to exit.
- spinCore - Starts a new busy-spinning thread. You can use it to simulate increased CPU consumption.
- unblockTraffic - Open ports blocked by the blockHzTraffic message.
- oom - Forces a message recipient to use all memory and cause an OutOfMemoryError.
- terminateWorker - Terminates a random Worker. This message type can be targeted to an Agent only.

21.9.3 Message Addressing

You can send a message to Agent, Worker or Test. These resources create a naturally hierarchy, making the messaging address hierarchical as well.

Syntax: Agent=<mode>[,Worker=<mode>][,Test=<mode>]

Mode can be either '*' for broadcast or 'R' for a single random destination.

Addressing Example 1:

Agent=*,Worker=R: A message will be routed to all agents, then each agent will pass it to a single random worker, and each worker will pass the message for processing.

Addressing Example 2:

Agent=*,Worker=R,Test=*: A message will be routed to all agents, then each agent will pass the message to a single random worker and workers will pass the message to all tests for processing.
21.9.3.1 Addressing shortcuts

Hierarchical addressing is powerful, but it can be quite verbose. You can use convenient shortcuts, as shown below.

- **--oldest-member**: Sends a message to a worker with the oldest cluster member.
- **--random-agent**: Sends a message to a random agent.
- **--random-worker**: Sends a message to a random worker.

Example: The following command starts a busy-spinning thread in a JVM running a random Worker.

```bash
communicator --random-worker spinCore
```

21.10 Simulator.Properties File Description

The file `simulator.properties` is placed at the `conf` folder of your Hazelcast Simulator. This file is used to prepare the Simulator tests for their proper executions according to your business needs.

**NOTE:** Currently, the main focuses are on the Simulator tests of Hazelcast on Amazon EC2 and Google Compute Engine (GCE). For the preparation of `simulator.properties` for GCE, please refer to the Setting Up For GCE section. The following `simulator.properties` file description is mainly for Amazon EC2.

This file includes the following parameters.

- **CLOUD_PROVIDER**: The Maven artifact ID of your cloud provider. For example, it is `aws-ec2` if you are going to test your Hazelcast on Amazon EC2. For the full list of supported clouds, please refer to [http://jclouds.apache.org/reference/providers/](http://jclouds.apache.org/reference/providers/).
- **CLOUD_IDENTITY**: The full path of the file containing your AWS access key.
- **CLOUD_CREDENTIAL**: The full path of the file containing your AWS secret key.
- **CLOUD_POLL_INITIAL_PERIOD**: The time in milliseconds between the requests (polls) from jclouds® to your cloud. Its default value is 50.
- **CLOUD_POLL_MAX_PERIOD**: The maximum time in milliseconds between the polls to your cloud. Its default value is 1000.
- **CLOUD_BATCH_SIZE**: The number of machines to be started/terminated in one go. For Amazon EC2, its acceptable value is 20.
- **GROUP_NAME**: The prefix for the agent name. You may want to give different names for different test clusters. For GCE, you need to be very careful using multiple group names, since for every port and every group name, a firewall rule is made and you can only have 100 firewall rules. If the name contains `${username}`, this section will be replaced by the actual user that runs the test. This makes it very easy to identify which user owns a certain machine.
- **USER**: The name of the user on your local machine. jclouds® automatically creates a new user on the remote machine with this name as the login name. It also copies the public key of your system to the remote machine and adds it to the file `~/.ssh/authorized_keys`. Therefore, once the instance is created, you can login with the command `ssh <USER>@<IP address>`. Its default value is `simulator`.
- **SSH_OPTIONS**: The options added to SSH. You do not need to change these options.
- **SECURITY_GROUP**: The name of the security group that includes the instances created for the Simulator test. For Amazon EC2, this group will be created automatically if it does not exist. If you do not specify a region for the parameter `MACHINE_SPEC` (using the `locationId` attribute), the region will be `us-east-1`. If a security group already exists, please make sure the ports 22, 9000, 9001 and the ports between 5701 and 5751 are open. For GCE, this parameter is not used.
- **SUBNET_ID**: The VPC Subnet ID for Amazon EC2. If this value is different from `default`, then the instances will be created in EC2 VPC and the parameter `SECURITY_GROUP` will be ignored. For GCE, this parameter is not used.
CHAPTER 21. HAZELCAST SIMULATOR

- **MACHINE_SPEC**: Specifications of the instance to be created. You can specify attributes such as the operating system, Amazon Machine Image (AMI), hardware properties, EC2 instance type and EC2 region. Please see the Setting Up For EC2 section for an example **MACHINE_SPEC** value and please refer to the `TemplateBuilderSpec` class of the `org.jclouds.compute.domain` package at jclouds.org JavaDoc for a full list of machine specifications. Please refer to Amazon EC2 for more information, such as for Amazon EC2 instance types.

- **HAZELCAST_VERSION_SPEC**: The workers can be configured to use a specific version of Hazelcast. By this way, you do not need to depend on the Hazelcast version provided by the simulator. You can configure the Hazelcast version in one of the following ways:
  
  - `outofthebox`: This is the default value provided by the Simulator itself.
  - `maven=<version>`: Used to give a specific version from the maven repository (for examples, `maven=3.2`, `maven=3.3-SNAPSHOT`). Local Hazelcast artifacts will be preferred, so you can check out, for example, an experimental branch and build the artifacts locally. This will all be done on the local machine, not on the agent machine.
  - `bringmyown`: Used to specify your own dependencies. For more information on the values, please see the `--workerClassPath` setting of the Controller.
  - `git=<version>`: If you want the Simulator to use a specific version of Hazelcast from GIT, you can use this parameter (for example, `git=f0288f713` to build a specific revision, or `git=v3.2.3` to build a version from a GIT tag, or `git=<your repository>/<your branch>` to build a version from a branch in a specific repository). Use the parameter `GIT_CUSTOM_REPOSITORIES` to specify custom repositories, explained below. The main Hazelcast repository is always named as `origin`.

- **GIT_BUILD_DIR**: When you set the parameter **HAZELCAST_VERSION_SPEC** to `git=<version>`, the Hazelcast sources will be downloaded to this directory. Its default value is `$HOME/.hazelcast-build/`.

- **GIT_CUSTOM_REPOSITORIES**: Comma separated list of additional GIT repositories to be fetched. Use this parameter when you set the parameter **HAZELCAST_VERSION_SPEC** to `git=<version>` and specify additional repositories. Hazelcast Simulator will always fetch the repository at `https://github.com/hazelcast/hazelcast`. This parameter specifies additional repositories. You can use both remote and local repositories. Remote repositories must be accessible for anonymous and local repositories must be accessible for the current user. Its default value is empty. Only the main Hazelcast repository is used by default.

- **MVN_EXECUTABLE**: This parameter specifies the path to a local Maven installation when you set the parameter **HAZELCAST_VERSION_SPEC** to `git=<version>`. Its default value is `/usr/bin/mvn`.

- **JDK_FLAVOR**: Available flavors are `oracle`, `openjdk`, `ibm` and `outofthebox`. `outofthebox` is the one provided by the image so no software is installed by the Simulator. If you select a flavor different than `outofthebox`, the current behavior is that only 64-bit JVMs are going to be installed. Therefore, make sure that your operating system is 64-bit.

- **JDK_64_BITS**: Specifies whether a 64-bit JVM should be installed or not. For now, only `true` is allowed.

- **JDK_VERSION**: The version of Java to be installed. Oracle and IBM support 6, 7, and 8. OpenJDK supports 6 and 7.

- **PROFILER**: The worker can be configured with a profiler. Available options are `none`, `yourkit`, `hprof`, `perf`, `vtune` and `flightrecorder`. The `yourkit` profiles currently only work on 64-bit Linux (there is no support for Windows or Mac).

- **FLIGHTRECORDER_SETTINGS**: Includes the settings for the `flightrecorder` profiler. For options, please refer to [http://docs.oracle.com/cd/E15289_01/doc.40/e15062/optionxx.htm#BABIECHI](http://docs.oracle.com/cd/E15289_01/doc.40/e15062/optionxx.htm#BABIECHI).

- **YOURKIT_SETTINGS**: Includes the settings for the `yourkit` profiler. When `yourkit` is enabled, a snapshot is created and put in the worker home directory. Therefore, when the artifacts are downloaded, the snapshots are included and can be loaded with your YourKit GUI. Make sure that the path matches the JVM 32/64 bits. The files `11bypagent.so`, which are included in the Simulator, are for YourKit Java Profiler 2013. For more information on the YourKit setting, please see [http://www.yourkit.com/docs/java/help/startup_options.jsp](http://www.yourkit.com/docs/java/help/startup_options.jsp) and [http://www.yourkit.com/docs/java/help/startup_options.jsp](http://www.yourkit.com/docs/java/help/startup_options.jsp).

- **HPROF_SETTINGS**: Includes the settings for the `hprof` profiler, which is a part of the JDK. By default, the file `java.hprof.txt` is created in the worker directory. This file can be downloaded using the command `provisioner --download` after a test has run. For configuration options, please see [http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html](http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html).

- **PERF_SETTINGS**: Includes the settings for the `perf` profiler, available only for Linux. For more information, please see [https://perf.wiki.kernel.org/index.php/Tutorial#Sampling_with_perf_record](https://perf.wiki.kernel.org/index.php/Tutorial#Sampling_with_perf_record).

- **VTUNE_SETTINGS**: Includes the settings for the `vtune` profiler. It requires Intel VTune to be installed on the system. For more information, please see [https://software.intel.com/sites/products/documentation/](https://software.intel.com/sites/products/documentation/)
21.11 Performance and Benchmarking

Hazelcast Simulator can use probes to record throughput and latency while running a test. Hazelcast Simulator can inject a probe into a test, and then it is the responsibility of the test to notify the probe about the start/end of each action.

There are two classes of probes:

- **SimpleProbe**: Counts the number of events. It does not have a notion of start/end.
- **IntervalProbe**: Differentiates between start/end of an action. Used to measure latency.

How to use probes is explained below.

1. Define a probe as a test property. Hazelcast Simulator will inject the appropriate probe implementation.
   ```java
   public class IntIntMapTest { private static final ILogger log = Logger.getLogger(IntIntMapTest.class); private enum Operation { PUT, GET } [...] // Probes will be injected by Hazelcast Simulator public IntervalProbe intervalProbe; public IntervalProbe anotherIntervalProbe; public SimpleProbe simpleProbe;
   ```

2. Use the probe in your test code.
   ```java
   getLatency.started(); map.get(key); getLatency.done();
   ```

3. Configure the probe in your test.properties file.
   ```properties
   probe-intervalProbe=throughput
   probe-simpleProbe=throughput
   ```

The configuration format is `probe-<nameOfField>=<type>`, where `nameOfField` is the name you choose for the probe, and `type` is the type of probe. Please keep in mind that this format is likely to change in future versions of Hazelcast Simulator.

A probe of class **IntervalProbe** can have the following types.

- **latency**: Measures the latency distribution.
- **maxLatency**: Records the highest latency. Unlike the previous probe, it records only the single highest latency measured, not a full distribution.
- **hdr**: Same as latency, but it uses HdrHistogram under the hood. This will replace the latency probe in future versions of Simulator.
- **disabled**: Dummy probe. It does not record anything.

A probe of class **SimpleProbe** can have the following implementations.

- **throughput**: Measures throughput.
- **disabled**: Dummy probe. It does not record anything.

It is important to understand that the class of a probe does not mandate what the probe is actually measuring. Therefore, the tests just know a class of probe, but they do not know if the probe generates, for example, a full latency histogram or just a maximum recorded latency. This detail must be implemented from a point of view of a test.
Chapter 22

WAN

This chapter explains how you can replicate the state of your clusters over Wide Area Network (WAN) environments.

22.1 WAN Replication

There are cases where you need to synchronize multiple clusters to the same state. Synchronization of clusters, also known as WAN Replication, is mainly used for replicating state of different clusters over WAN environments like the Internet.

Imagine you have different data centers in New York, London and Tokyo each running an independent Hazelcast cluster. Every cluster would be operating at native speed in their own LAN (Local Area Network), but you also want some or all recordsets in these clusters to be replicated to each other: updates in the Tokyo cluster should also replicate to London and New York, in the meantime updates in the New York cluster are synchronized to the Tokyo and London clusters.

22.1.1 Configuring WAN Replication

The current WAN Replication implementation supports two different operation modes.

- **Active-Passive**: This mode is mostly used for failover scenarios where you want to replicate an active cluster to one or more passive clusters, for the purpose of maintaining a backup.

- **Active-Active**: Every cluster is equal, each cluster replicate to all other clusters. This is normally used to connect different clients to different clusters for the sake of the shortest path between client and server.

Let’s see how we can configure WAN Replication from the New York cluster to target the London and Tokyo clusters:

```xml
<configuration>
  <wan-replication name="my-wan-cluster">
    <target-cluster group-name="tokyo" group-password="tokyo-pass">
      <replication-impl>com.hazelcast.wan.impl.WanNoDelayReplication</replication-impl>
      <end-points>
        <address>10.2.1.1:5701</address>
        <address>10.2.1.2:5701</address>
      </end-points>
    </target-cluster>
    <target-cluster group-name="london" group-password="london-pass">
      <replication-impl>com.hazelcast.wan.impl.WanNoDelayReplication</replication-impl>
      <end-points>
        <address>10.3.5.1:5701</address>
      </end-points>
    </target-cluster>
  </wan-replication>
</configuration>
```
Using this configuration, the cluster running in New York is replicating to Tokyo and London. The Tokyo and London clusters should have a similar configurations if you want to run in Active-Active mode.

If the New York and London cluster configurations contain the `wan-replication` element and the Tokyo cluster does not, it means New York and London are active endpoints and Tokyo is a passive endpoint.

When using Active-Active Replication, multiple clusters can simultaneously update the same entry in a distributed data structure. You can configure a merge-policy to resolve these potential conflicts.

Hazelcast can configure WAN replication on a per Map basis. Imagine you have different distributed maps, however only one map should be replicated to a target cluster. To achieve this, configure map to be replicated by adding the `wan-replication-ref` element in the map configuration as shown below.

You see that we have `my-shared-map` configured to replicate itself to the cluster targets defined in the earlier `wan-replication` element.

You will also have to define a merge policy for merging replica entries and resolving conflicts during the merge as mentioned before.

### 22.1.2 WAN Replication Additional Information

**RELATED INFORMATION**

You can download the white paper *Hazelcast on AWS: Best Practices for Deployment* from Hazelcast.com.

**RELATED INFORMATION**

Please refer to the *WAN Replication Configuration section* for a full description of Hazelcast WAN Replication configuration.
22.2 Enterprise WAN Replication

22.2.1 Replication implementations

Enterprise WAN replication has two different replication implementations. These are WanNoDelayReplication and WanBatchReplication implementations. You can configure them using the configuration element replication-impl, as shown below.

```
<hazelcast>
    <wan-replication name="my-wan-cluster">
      <target-cluster group-name="tokyo" group-password="tokyo-pass">
        <replication-impl>com.hazelcast.enterprise.wan.replication.WanNoDelayReplication</replication-impl>
      </target-cluster>
    </wan-replication>
</hazelcast>

<hazelcast>
    <wan-replication name="my-wan-cluster">
      <target-cluster group-name="tokyo" group-password="tokyo-pass">
        <replication-impl>com.hazelcast.enterprise.wan.replication.WanBatchReplication</replication-impl>
      </target-cluster>
    </wan-replication>
</hazelcast>
```

WanNoDelayReplication sends replication events to the target cluster as soon as they are generated. WanBatchReplication waits until:

- a pre-defined number of replication events are generated, (please refer to the Wan Replication Batch Size section).
- or a pre-defined amount of time is passed (please refer to the Wan Replication Batch Frequency section).

22.2.2 WAN Replication Batch Size

When WanBatchReplication is preferred as the replication implementation, the maximum size of events that are sent in a single batch can be changed depending on your needs. Default value for batch size is 50.

To change the WanBatchReplication batch size, use the hazelcast.enterprise.wanrep.batch.size property in Hazelcast Enterprise.

You can do this by setting the property on the command line (where xxx is the batch size),

```
-Dhazelcast.enterprise.wanrep.batch.size=xxx
```

or by setting the property inside the hazelcast.xml (where xxx is the requested batch size):

```
<hazelcast>
  <properties>
    <property name="hazelcast.enterprise.wanrep.batch.size">xxx</property>
  </properties>
</hazelcast>
```
22.2.3 WAN Replication Batch Frequency

When using `WanBatchReplication` if the number of WAN replication events generated does not reach `Wan Replication Batch Size`, they are sent to the target cluster after a certain amount of time is passed.

Default value of for this duration is 5 seconds.

To change the `WanBatchReplication` batch sending frequency, set `hazelcast.enterprise.wanrep.batchfrequency.seconds` property.

You can set the property on the command line (where `xxx` is the batch sending frequency in seconds),

```
-Dhazelcast.enterprise.wanrep.batchfrequency.seconds=xxx
```

or by setting the properties inside the `hazelcast.xml` (where `xxx` is the requested batch sending frequency):

```xml
<hazelcast>
  <properties>
    <property name="hazelcast.enterprise.wanrep.batchfrequency.seconds">xxx</property>
  </properties>
</hazelcast>
```

22.2.4 WAN Replication Operation Timeout

After a replication event is sent to the target cluster, the source member waits for an acknowledge that event has reached the target. If confirmation is not received inside a timeout duration window, the event is resent to the target cluster.

Default value of for this duration is 5000 milliseconds.

You can change this duration depending on your network latency. The Hazelcast Enterprise user can set the `hazelcast.enterprise.wanrep.optimeout.millis` property to change the timeout duration.

You can do this by setting the property on the command line (where `xxx` is the timeout duration in milliseconds),

```
-Dhazelcast.enterprise.wanrep.optimeout.millis=xxx
```

or by setting the property inside the `hazelcast.xml` (where `xxx` is the requested timeout duration):

```xml
<hazelcast>
  <properties>
    <property name="hazelcast.enterprise.wanrep.optimeout.millis">xxx</property>
  </properties>
</hazelcast>
```

22.2.5 WAN Replication Queue Capacity

For huge clusters or high data mutation rates, you might need to increase the replication queue size. The default queue size for replication queues is 100000. This means, if you have heavy put/update/remove rates, you might exceed the queue size so that the oldest, not yet replicated, updates might get lost.

To increase the replication queue capacity, the Hazelcast Enterprise user can use the `hazelcast.enterprise.wanrep.queue.capacity` property.

You can do this by setting the property on the command line (where `xxx` is the queue size),

```
-Dhazelcast.enterprise.wanrep.queue.capacity=xxx
```

or by setting the properties inside the `hazelcast.xml` (where `xxx` is the requested queue size):
22.2. ENTERPRISE WAN REPLICATION

22.2.6 Enterprise WAN Replication Additional Information

Each cluster in WAN topology has to have a unique group-name property for a proper handling of forwarded events.

*Please refer to the Enterprise WAN Replication Configuration section for a full description of Hazelcast WAN Replication configuration.*
Chapter 23

Hazelcast Configuration

This chapter covers all the elements and attributes used to configure Hazelcast. It includes the following sections:

- **Configuration Overview**: Provides the options used to configure Hazelcast.
- **Using Wildcard**: Describes the usage of the wildcard character (*) while configuring Hazelcast.
- **Using Variables**: Describes how to use variables in declarative configurations.
- **Composing Declarative Configuration**: Describes how to produce a declarative configuration file out of several configuration files.

The rest of the chapter explains the configuration items listed below.

- Network
- Group
- Map
- MultiMap
- Queue
- Topic
- List
- Set
- Semaphore
- Executor Service
- Serialization
- MapReduce Jobtracker
- Services
- Management Center
- WAN Replication
- Partition Group
- Listeners
- Logging

The chapter ends with the configuration via System Properties.

### 23.1 Configuration Overview

Hazelcast can be configured declaratively (XML) or programmatically (API) or even by a mix of both.

**1- Declarative Configuration**

If you are creating new Hazelcast instance by passing the `null` parameter to `Hazelcast.newHazelcastInstance(null)` or just using an empty factory method (`Hazelcast.newHazelcastInstance()`), Hazelcast will look in two places for the configuration file.
- **System property**: Hazelcast will first check if “hazelcast.config” system property is set to a file path. Example: 
  `-Dhazelcast.config=C:/myhazelcast.xml`.
- **Classpath**: If config file is not set as a system property, Hazelcast will check classpath for `hazelcast.xml` file.

If Hazelcast does not find any configuration file, it will start with the default configuration (`hazelcast-default.xml`) located in `hazelcast.jar`. (Before configuring Hazelcast, please try to work with the default configuration to see if it works for you. Default should be just fine for most users. If not, then consider custom configuration for your environment.)

If you want to specify your own configuration file to create `Config`, Hazelcast supports several ways including filesystem, classpath, InputStream, URL, etc.:

- Config `cfg = new XmlConfigBuilder(xmlFileName).build();`
- Config `cfg = new XmlConfigBuilder(inputStream).build();`
- Config `cfg = new ClasspathXmlConfig(xmlFileName);`
- Config `cfg = new FileSystemXmlConfig(configFilename);`
- Config `cfg = new UrlXmlConfig(url);`
- Config `cfg = new InMemoryXmlConfig(xml);`

2- Programmatic Configuration

To configure Hazelcast programatically, just instantiate a `Config` object and set/change its properties/attributes to your needs. Below is a code sample in which some network, map, map store, and near cache attributes are configured for a Hazelcast instance.

```java
Config config = new Config();
config.getNetworkConfig().setPort(5900);
config.getNetworkConfig().setPortAutoIncrement(false);
NetworkConfig network = config.getNetworkConfig();
JoinConfig join = network.getJoin();
join.getMulticastConfig().setEnabled(false);
join.getTcpIpConfig().addMember("10.45.67.32").addMember("10.45.67.100").setRequiredMember("192.168.10.100").setEnabled(true);
network.getInterfaces().setEnabled(true).addInterface("10.45.67.*");

MapConfig mapConfig = new MapConfig();
mapConfig.setName("testMap");
mapConfig.setBackupCount(2);
mapConfig.getMaxSizeConfig().setSize(10000);
mapConfig.setTimeToLiveSeconds(300);

MapStoreConfig mapStoreConfig = new MapStoreConfig();
mapStoreConfig.setClassName("com.hazelcast.examples.DummyStore").setEnabled(true);
mapConfig.setMapStoreConfig(mapStoreConfig);

NearCacheConfig nearCacheConfig = new NearCacheConfig();
nearCacheConfig.setMaxSize(1000).setMaxIdleSeconds(120).setTimeToLiveSeconds(300);
mapConfig.setNearCacheConfig(nearCacheConfig);
config.addMapConfig(mapConfig);
```

After creating a `Config` object, you can use it to create a new Hazelcast instance.

- `HazelcastInstance hazelcast = Hazelcast.newHazelcastInstance(config);`
23.2. USING WILDCARD

- To create a named HazelcastInstance you should set instanceName of Config object.

```java
Config config = new Config();
config.setInstanceName( "my-instance" );
Hazelcast.newHazelcastInstance( config );
```

- To retrieve an existing HazelcastInstance using its name, use:

```
Hazelcast.getHazelcastInstanceByName( "my-instance" );
```

- To retrieve all existing HazelcastInstances, use:

```
Hazelcast.getAllHazelcastInstances();
```

**NOTE:** Hazelcast performs the schema validation through the hazelcast-config-<version>.xsd file. It throws a meaningful exception if there is an error in the declarative or programmatic configuration.

23.2 Using Wildcard

Hazelcast supports wildcard configuration for all distributed data structures that can be configured using Config (i.e. for all except IAtomicLong, IAtomicReference). Using an asterisk (*) character in the name, different instances of maps, queues, topics, semaphores, etc. can be configured by a single configuration.

A single (only one) asterisk (*) can be placed anywhere inside the configuration name.

For instance, a map named com.hazelcast.test.mymap can be configured using one of the following configurations.

```xml
<map name="com.hazelcast.test.*">
  ...
</map>
```

```xml
<map name="com.hazel*">
  ...
</map>
```

```xml
<map name="*.test.mymap">
  ...
</map>
```

```xml
<map name="com.*test.mymap">
  ...
</map>
```

Or a queue 'com.hazelcast.test.myqueue':

```xml
<queue name="*hazelcast.test.myqueue">
  ...
</queue>
```

```xml
<queue name="com.hazelcast.*.myqueue">
  ...
</queue>
```
23.3 Using Variables

In your Hazelcast and/or Hazelcast Client declarative configuration, you can use variables to set the values of the elements. This is valid when you set a system property programmatically or you use the command line interface. You can use a variable in the declarative configuration to access the values of the system properties you set.

For example, see the following command that sets two system properties.

```bash
-Dgroup.name=dev -Dgroup.password=somepassword
```

Let’s get the values of these system properties in the declarative configuration of Hazelcast, as shown below.

```xml
<hazelcast>
  <group>
    <name>${group.name}</name>
    <password>${group.password}</password>
  </group>
</hazelcast>
```

This also applies to the declarative configuration of Hazelcast Client, as shown below.

```xml
<hazelcast-client>
  <group>
    <name>${group.name}</name>
    <password>${group.password}</password>
  </group>
</hazelcast-client>
```

If you do not want to rely on the system properties, you can use the `XmlConfigBuilder` and explicitly set a `Properties` instance, as shown below.

```java
Properties properties = new Properties();
// fill the properties, e.g. from database/LDAP, etc.
XmlConfigBuilder builder = new XmlConfigBuilder();
builder.setProperties(properties);
Config config = builder.build();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(config);
```

23.4 Composing Declarative Configuration

You can compose the declarative configuration of your Hazelcast or Hazelcast Client from multiple declarative configuration snippets. In order to compose a declarative configuration, you can use the `<import/>` element to load different declarative configuration files. Please see the following examples.

Let’s say you want to compose the declarative configuration for Hazelcast out of two configurations: `development-group-config.xml` and `development-network-config.xml`. These two configurations are shown below.

`development-group-config.xml`:

```xml
<hazelcast>
  <group>
    <name>dev</name>
    <password>dev-pass</password>
  </group>
</hazelcast>
```
23.4. **COMPOSING DECLARATIVE CONFIGURATION**

**development-network-config.xml:**

```xml
<hazelcast>
    <network>
        <port auto-increment="true" port-count="100">5701</port>
        <join>
            <multicast enabled="true">
                <multicast-group>224.2.2.3</multicast-group>
                <multicast-port>54327</multicast-port>
            </multicast>
        </join>
    </network>
</hazelcast>
```

To get your example Hazelcast declarative configuration out of the above two, use the `<import/>` element as shown below.

```xml
<hazelcast>
    <import resource="development-group-config.xml"/>
    <import resource="development-network-config.xml"/>
</hazelcast>
```

This feature also applies to the declarative configuration of Hazelcast Client. Please see the following examples.

**client-group-config.xml:**

```xml
<hazelcast-client>
    <group>
        <name>dev</name>
        <password>dev-pass</password>
    </group>
</hazelcast-client>
```

**client-network-config.xml:**

```xml
<hazelcast-client>
    <network>
        <cluster-members>
            <address>127.0.0.1:7000</address>
        </cluster-members>
    </network>
</hazelcast-client>
```

To get a Hazelcast Client declarative configuration from the above two examples, use the `<import/>` element as shown below.

```xml
<hazelcast-client>
    <import resource="client-group-config.xml"/>
    <import resource="client-network-config.xml"/>
</hazelcast-client>
```

**NOTE:** You can only use `<import/>` element on top level of the XML hierarchy.

- XML resources can be loaded from classpath and filesystem. Please see the following example.
<hazelcast>
  <import resource="file:///etc/hazelcast/development-group-config.xml"/>
  <!-- loaded from filesystem -->
  <import resource="classpath:development-network-config.xml"/>
  <!-- loaded from classpath -->
</hazelcast>

- You can use property placeholders in the <import/> elements. Please see the following example.

<hazelcast>
  <import resource="${environment}-group-config.xml"/>
  <import resource="${environment}-network-config.xml"/>
</hazelcast>

### 23.5 Network Configuration

**NOTE:** When explaining configuration elements and attributes, most of the sections below use the tags used in the declarative configurations. We assume that the reader is familiar with these tags’ programmatic equivalents, since declarative and programmatic approaches have the similar tag/method names. For example, the `port-count` tag in declarative configuration is equivalent to `setPortCount` in programmatic configuration.

All network related configuration is performed via the `network` element in the Hazelcast XML configuration file or the class `NetworkConfig` when using programmatic configuration. Let’s first give the examples for these two approaches. Then we will look at its sub-elements and attributes.

**Declarative:**

```
<network>
  <public-address></public-address>
  <port auto-increment="true" port-count="100">5701</port>
  <outbound-ports>
    <ports>0</ports>
  </outbound-ports>
  <reuse-address>false</reuse-address>
  <join>
    <multicast enabled="true">murphy</multicast>
    <multicast-group>224.2.2.3</multicast-group>
    <multicast-port>54327</multicast-port>
  </join>
  <interfaces enabled="false">
    <interface>127.0.0.1</interface>
  </interfaces>
  <ssl enabled="false" />
  <socket-interceptor enabled="false" />
  <symmetric-encryption enabled="false" />
```

```
23.5. NETWORK CONFIGURATION

<algorithm>PBEWithMD5AndDES</algorithm>
<salt>thesalt</salt>
<password>thepass</password>
<iteration-count>19</iteration-count>
</symmetric-encryption>

Programmatic:

Network netConfig = new NetworkConfig();
netConfig.setReuseAddress( false );

AwsConfig awsConfig = new AwsConfig();
awsConfig.setTagKey( "5551234" ).setTagValue( "Node1234" );

It has the following sub-elements which are described in the following sections.

- Public Address
- Port
- Outbound Ports
- Reuse Address
- Join
- Interfaces
- SSL
- Socket Interceptor
- Symmetric Encryption

23.5.1 Public Address

The public-address overrides the public address of a node. By default, a node selects its socket address as its public address. But behind a network address translation (NAT), two endpoints (nodes) may not be able to see/access each other. If both nodes set their public addresses to their defined addresses on NAT, then that way they can communicate with each other. In this case, their public addresses are not an address of a local network interface but a virtual address defined by NAT. It is optional to set and useful when you have a private cloud.

23.5.2 Port

You can specify the ports that Hazelcast will use to communicate between cluster members. Its default value is 5701. The following are example configurations.

Declarative:

<network>
  <port port-count="20" auto-increment="false">5701</port>
</network>

Programmatic:

Config config = new Config();
config.getNetworkConfig().setPort( "5701" );
   .setPortCount( "20" ).setPortAutoIncrement( false );

Port has the following attributes.
• **port-count**: By default, Hazelcast will try 100 ports to bind. Meaning that, if you set the value of port as 5701, as members are joining to the cluster, Hazelcast tries to find ports between 5701 and 5801. You can choose to change the port count in the cases like having large instances on a single machine or willing to have only a few ports to be assigned. The parameter `port-count` is used for this purpose, whose default value is 100.

• **auto-increment**: According to the above example, Hazelcast will try to find free ports between 5701 and 5801. Normally, you will not need to change this value, but it will come very handy when needed. You may also want to choose to use only one port. In that case, you can disable the auto-increment feature of `port` by setting `auto-increment` to `false`.

The parameter `port-count` is ignored when the above configuration is made.

### 23.5.3 Outbound Ports

By default, Hazelcast lets the system pick up an ephemeral port during socket bind operation. But security policies/firewalls may require you to restrict outbound ports to be used by Hazelcast-enabled applications. To fulfill this requirement, you can configure Hazelcast to use only defined outbound ports. The following are example configurations.

**Declarative:**

```xml
<network>
  <outbound-ports>
    <!-- ports between 33000 and 35000 -->
    <ports>33000-35000</ports>
    <!-- comma separated ports -->
    <ports>37000,37001,37002,37003</ports>
    <ports>38000,38500-38600</ports>
  </outbound-ports>
</network>
```

**Programmatic:**

```java
NetworkConfig networkConfig = config.getNetworkConfig();
// ports between 35000 and 35100
networkConfig.addOutboundPortDefinition("35000-35100");
// comma separated ports
networkConfig.addOutboundPortDefinition("36001, 36002, 36003");
networkConfig.addOutboundPort(37000);
networkConfig.addOutboundPort(37001);
...
```

**Note:** You can use port ranges and/or comma separated ports.

As shown in the programmatic configuration, you use the method `addOutboundPort` to add only one port. If you need to add a group of ports, then use the method `addOutboundPortDefinition`.

In the declarative configuration, the element `ports` can be used for both single and multiple port definitions.

### 23.5.4 Reuse Address

When you shutdown a cluster member, the server socket port will be in the `TIME_WAIT` state for the next couple of minutes. If you start the member right after shutting it down, you may not be able to bind it to the same port because it is in the `TIME_WAIT` state. If you set the `reuse-address` element to `true`, the `TIME_WAIT` state is ignored and you can bind the member to the same port again.
The following are example configurations.

Declarative:

```xml
<network>
  <reuse-address>true</reuse-address>
</network>
```

Programmatic:

```java
...
NetworkConfig networkConfig = config.getNetworkConfig();
networkConfig.setReuseAddress( true );
...```

23.5.5 Join

The join configuration element is used to enable the Hazelcast instances to form a cluster, i.e. to join the members. Three ways can be used to join the members: discovery by TCP/IP, by multicast, and by discovery on AWS (EC2 auto-discovery). The following are example configurations.

Declarative:

```xml
<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>
      <trusted-interfaces>
        <interface>192.168.1.102</interface>
      </trusted-interfaces>
    </multicast>
    <tcp-ip enabled="false">
      <required-member>192.168.1.104</required-member>
      <member>192.168.1.104</member>
      <members>192.168.1.105,192.168.1.106</members>
    </tcp-ip>
    <aws enabled="false">
      <access-key>my-access-key</access-key>
      <secret-key>my-secret-key</secret-key>
      <region>us-west-1</region>
      <host-header>ec2.amazonaws.com</host-header>
      <security-group-name>hazelcast-sg</security-group-name>
      <tag-key>type</tag-key>
      <tag-value>hz-nodes</tag-value>
    </aws>
  </join>
</network>
```

Programmatic:

```java
Config config = new Config();
NetworkConfig network = config.getNetworkConfig();
JoinConfig join = network.getJoin();
```
join.getMulticastConfig().setEnabled( false )
   .addTrustedInterface( "192.168.1.102" );
join.getTcpIpConfig().addMember( "10.45.67.32" ).addMember( "10.45.67.100" )
   .setRequiredMember( "192.168.10.100" ).setEnabled( true );

The join element has the following sub-elements and attributes.

### 23.5.5.1 multicast element

The multicast element includes parameters to fine tune the multicast join mechanism.

- **enabled**: Specifies whether the multicast discovery is enabled or not, true or false.
- **multicast-group**: The multicast group IP address. Specify it when you want to create clusters within the same network. Values can be between 224.0.0.0 and 239.255.255.255. Default value is 224.2.2.3.
- **multicast-port**: The multicast socket port that the Hazelcast member listens to and sends discovery messages through. Default value is 54327.
- **multicast-time-to-live**: Time-to-live value for multicast packets sent out to control the scope of multicasts. See more information here.
- **multicast-timeout-seconds**: Only when the nodes are starting up, this timeout (in seconds) specifies the period during which a node waits for a multicast response from another node. For example, if you set it as 60 seconds, each node will wait for 60 seconds until a leader node is selected. Its default value is 2 seconds.
- **trusted-interfaces**: Includes IP addresses of trusted members. When a node wants to join to the cluster, its join request will be rejected if it is not a trusted member. You can give an IP addresses range using the wildcard (*) on the last digit of IP address (e.g. 192.168.1.* or 192.168.1.100-110).

### 23.5.5.2 tcp-ip element

The tcp-ip element includes parameters to fine tune the TCP/IP join mechanism.

- **enabled**: Specifies whether the TCP/IP discovery is enabled or not. Values can be true or false.
- **required-member**: IP address of the required member. Cluster will only formed if the member with this IP address is found.
- **member**: IP address(es) of one or more well known members. Once members are connected to these well known ones, all member addresses will be communicated with each other. You can also give comma separated IP addresses using the members element.

**NOTE:** tcp-ip element also accepts the interface parameter. Please refer to the Interfaces element description.

- **connection-timeout-seconds**: Defines the connection timeout. This is the maximum amount of time Hazelcast is going to try to connect to a well known member before giving up. Setting it to a too low value could mean that a member is not able to connect to a cluster. Setting it to a too high value means that member startup could slow down because of longer timeouts (e.g. when a well known member is not up). Increasing this value is recommended if you have many IPs listed and the members cannot properly build up the cluster. Its default value is 5.

### 23.5.5.3 aws element

The aws element includes parameters to allow the nodes to form a cluster on the Amazon EC2 environment.

- **enabled**: Specifies whether the EC2 discovery is enabled or not, true or false.
- **access-key, secret-key**: Access and secret keys of your account on EC2.
23.5. NETWORK CONFIGURATION

- **region**: The region where your nodes are running. Default value is `us-east-1`. You need to specify this if the region is other than the default one.
- **host-header**: The URL that is the entry point for a web service. It is optional.
- **security-group-name**: Name of the security group you specified at the EC2 management console. It is used to narrow the Hazelcast nodes to be within this group. It is optional.
- **tag-key, tag-value**: To narrow the members in the cloud down to only Hazelcast nodes, you can set these parameters as the ones you specified in the EC2 console. They are optional.
- **connection-timeout-seconds**: The maximum amount of time Hazelcast will try to connect to a well known member before giving up. Setting this value too low could mean that a member is not able to connect to a cluster. Setting the value too high means that member startup could slow down because of longer timeouts (for example, when a well known member is not up). Increasing this value is recommended if you have many IPs listed and the members cannot properly build up the cluster. Its default value is 5.

**NOTE:** If you are using a cloud provider other than AWS, you can use the programmatic configuration to specify a TCP/IP cluster. The members will need to be retrieved from that provider (e.g. JClouds).

### 23.5.5.3.1 AWSClient Configuration

To make sure EC2 instances are found correctly, you can use the `AWSClient` class. It determines the private IP addresses of EC2 instances to be connected. Give the `AWSClient` class the values for the parameters that you specified in the `aws` element, as shown below. You will see whether your EC2 instances are found.

```java
public static void main( String[] args )
throws Exception{
    AwsConfig config = new AwsConfig();
    config.setSecretKey( ... );
    config.setSecretKey( ... );
    config.setRegion( ... );
    config.setSecurityGroupName( ... );
    config.setTagKey( ... );
    config.setTagValue( ... );
    config.setEnabled( true );
    AWSClient client = new AWSClient( config );
    List<String> ipAddresses = client.getPrivateIpAddresses();
    System.out.println( "addresses found:" + ipAddresses );
    for ( String ip: ipAddresses ) {
        System.out.println( ip );
    }
}
```

### 23.5.6 Interfaces

You can specify which network interfaces that Hazelcast should use. Servers mostly have more than one network interface, so you may want to list the valid IPs. Range characters ("*" and "]") can be used for simplicity. 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 will not start on that node.

The following are example configurations.

**Declarative:**

```
<hazelcast>
...
<network>
...```
Programmatic:

```
Config config = new Config();
NetworkConfig network = config.getNetworkConfig();
InterfacesConfig interface = network.getInterfaces();
    interface.setEnabled(true)
    .addInterface("192.168.1.3");
```

### 23.5.7 SSL

This is a Hazelcast Enterprise feature, please see the Security chapter.

### 23.5.8 Socket Interceptor

This is a Hazelcast Enterprise feature, please see the Security chapter.

### 23.5.9 Symmetric Encryption

This is a Hazelcast Enterprise feature, please see the Security chapter.

### 23.5.10 IPv6 Support

Hazelcast supports IPv6 addresses seamlessly (This support is switched off by default, please see the note at the end of this section).

All you need is to define IPv6 addresses or interfaces in network configuration. The only current limitation is that you cannot define wildcard IPv6 addresses in the TCP/IP join configuration (`tcp-ip` element). Interfaces configuration does not have this limitation, you can configure wildcard IPv6 interfaces in the same way as IPv4 interfaces.

```
<port auto-increment="true">5701</port>
<join>
    <multicast enabled="false">
        <multicast-group>FF02:0:0:0:0:0:0:1</multicast-group>
        <multicast-port>54327</multicast-port>
    </multicast>
    <tcp-ip enabled="true">
        <member>[fe80::223:6cff:fe93:7c7e]:5701</member>
        <interface>192.168.1.0-7</interface>
        <interface>192.168.1.*</interface>
        <interface>fe80:0:0:0:45c5:47ee:fe15:493a</interface>
    </tcp-ip>
```
23.6  GROUP CONFIGURATION

JVM has two system properties for setting the preferred protocol stack (IPv4 or IPv6) as well as the preferred address family types (inet4 or inet6). On a dual stack machine, IPv6 stack is preferred by default, you can change this through the `java.net.preferIPv4Stack=<true|false>` system property. When querying name services, JVM prefers IPv4 addresses over IPv6 addresses and will return an IPv4 address if possible. You can change this through `java.net.preferIPv6Addresses=<true|false>` system property.

Also see additional details on IPv6 support in Java.

NOTE: IPv6 support has been switched off by default, since some platforms have issues using the IPv6 stack. Some other platforms such as Amazon AWS have no support at all. To enable IPv6 support, just set configuration property `hazelcast.prefer.ipv4.stack` to false. Please refer to the System Properties section for details.

23.6  Group Configuration

This configuration is used to create multiple Hazelcast clusters. The cluster members (nodes) and clients having the same group configuration (i.e., the same group name and password) forms a private cluster.

Each cluster will have its own group and it will not interfere with other clusters. The name of the element to configure cluster groups is `group`.

The following are example configurations.

Declarative:

```
<group>
  <name>testCluster</name>
  <password>1q2w3e</password>
</group>
```

Programmatic:

```
GroupConfig groupConfig = new GroupConfig();
groupConfig.setName( "testCluster" ).setPassword( "1q2w3e" );
```

It has the following elements.

- **name**: Name of the group to be created.
- **password**: Password of the group to be created.

23.7  Map Configuration

The following are example map configurations.

Declarative:
CHAPTER 23. HAZELCAST CONFIGURATION

```xml
<hazelcast>
  <map name="default">
    <in-memory-format>BINARY</in-memory-format>
    <backup-count>0</backup-count>
    <async-backup-count>1</async-backup-count>
    <read-backup-data>true</read-backup-data>
    <time-to-live-seconds>0</time-to-live-seconds>
    <max-idle-seconds>0</max-idle-seconds>
    <eviction-policy>LRU</eviction-policy>
    <max-size policy="PER_NODE">5000</max-size>
    <eviction-percentage>25</eviction-percentage>
    <near-cache>
      <invalidate-on-change>true</invalidate-on-change>
      <cache-local-entries>false</cache-local-entries>
    </near-cache>
    <map-store enabled="true">
      <write-delay-seconds>60</write-delay-seconds>
      <write-batch-size>1000</write-batch-size>
      <write-coalescing>true</write-coalescing>
    </map-store>
    <indexes>
      <index ordered="true">id</index>
      <index ordered="false">name</index>
    </indexes>
    <entry-listeners>
      <entry-listener include-value="true" local="false">
        com.hazelcast.examples.EntryListener
      </entry-listener>
    </entry-listeners>
  </map>
</hazelcast>

Programmatic:

```java
MapConfig mapConfig = new MapConfig();
mapConfig.setName("default").setInMemoryFormat("BINARY");

mapConfig.setBackupCount("0").setAsyncBackupCount("1").
  setReadBackupData("true");

MapStoreConfig mapStoreConfig = mapConfig.getMapStoreConfig();
mapStoreConfig.setWriteDelaySeconds("60").
  setWriteBatchSize("1000");

MapIndexConfig mapIndexConfig = mapConfig.getMapIndexConfig();
mapIndexConfig.setAttribute("id").setOrdered("true");
mapIndexConfig.setAttribute("name").setOrdered("false");
```

Map configuration has the following elements.

- **in-memory-format**: Determines how the data will be stored in memory. It has two values: BINARY and OBJECT. BINARY is the default option, storing the data in serialized binary format. If set to OBJECT, data will be stored in deserialized form.
- **backup-count**: Defines the count of synchronous backups. If it is set as 1, for example, backup of a partition will be placed on 1 other node. If it is 2, it will be placed on 2 other nodes.
- **async-backup-count**: The number of synchronous backups. Count behavior is the same as that of backup-count element.
• read-backup-data: When set to true, enables reading of local backup entries.
• time-to-live-seconds: Maximum time in seconds for each entry to stay in the map.
• max-idle-seconds: Maximum time in seconds for each entry to stay idle in the map.
• eviction-policy: Policy for evicting entries. It has three values: NONE, LRU (Least Recently Used) and LFU (Least Frequently Used). If set to NONE, no items will be evicted.
• max-size: Maximum size of the map (i.e. maximum entry count of the map). When the maximum size is reached, the map is evicted based on the eviction policy defined. It has four attributes: PER_NODE (Maximum number of map entries in each JVM), PER_PARTITION (Maximum number of map entries within each partition), USED_HEAP_SIZE (Maximum used heap size in megabytes for each JVM) and USED_HEAP_PERCENTAGE (Maximum used heap size percentage for each JVM).
• eviction-percentage: When the map reaches max-size, this specified percentage of the map will be evicted.
• merge-policy: Policy for merging maps after a split-brain syndrome was detected and the different network partitions need to be merged. The available merge policy classes are explained below.
  – HigherHitsMapMergePolicy causes the merging entry to be merged from the source map to the destination map if the source entry has more hits than the destination one.
  – LatestUpdateMapMergePolicy causes the merging entry to be merged from the source map to the destination map if the source entry has updated more recently than the destination entry. This policy can only be used if the machine clocks are in sync.
  – PassThroughMergePolicy causes the merging entry to be merged from source to destination map unless merging entry is null.
  – PutIfAbsentMapMergePolicy causes the merging entry to be merged from source to destination map if it does not exist in the destination map.
• statistics-enabled: You can retrieve statistics information like owned entry count, backup entry count, last update time, and locked entry count by setting statistics-enabled to true. The method for retrieving the statistics is getLocalMapStats().
• wan-replication-ref: Hazelcast can replicate some or all of the cluster data. For example, suppose you can have 5 different maps but you want only one of these maps to replicate across clusters. To achieve this, you mark the maps to be replicated by adding this element in the map configuration.
• optimize-queries: This element is used to increase the speed of query processes in the map. It only works when in-memory-format is set as BINARY and performs a pre-caching on the entries queried.

23.7.1 Map Store

• class-name: Name of the class implementing MapLoader and/or MapStore.
• write-delay-seconds: Number of seconds to delay to call the MapStore.store(key, value). If the value is zero then it is write-through so MapStore.store(key, value) will be called as soon as the entry is updated. Otherwise it is write-behind so updates will be stored after write-delay-seconds value by calling Hazelcast.storeAll(map). Default value is 0.
• write-batch-size: Used to create batch chunks when writing map store. In default mode, all map entries will be tried to be written in one go. To create batch chunks, the minimum meaningful value for write-batch-size is 2. For values smaller than 2, it works as in default mode.
• write-coalescing: In write-behind mode, by default Hazelcast coalesces updates on a specific key, i.e. applies only the last update on it. You can set this element to false to store all updates performed on a key to the data store.

23.7.2 Near Cache

Most of the map near cache properties have the same names and tasks explained in the map properties above. Below are the ones specific to near cache.

• invalidate-on-change: Determines whether the cached entries get evicted if the entries are updated or removed.
• cache-local-entries: If you want the local entries to be cached, set this element’s value to true.
23.7.3 Indexes

This configuration lets you index the attributes of an object in the map and also order these attributes. See the above declarative and programmatic configuration examples.

23.7.4 Entry Listeners

This configuration lets you add listeners (listener classes) for the map entries. You can also set the attribute `include-value` to `true` if you want the entry event to contain the entry values, and you can set `local` to `true` if you want to listen to the entries on the local node.

23.8 MultiMap Configuration

The following are the example MultiMap configurations.

Declarative:

```xml
<hazelcast>
<multimap name="default">
  <backup-count>0</backup-count>
  <async-backup-count>1</async-backup-count>
  <value-collection-type>SET</value-collection-type>
  <entry-listeners>
    <entry-listener include-value="false" local="false">
      com.hazelcast.examples.EntryListener
    </entry-listener>
  </entry-listeners>
</map>
</hazelcast>
```

Programmatic:

```java
MultiMapConfig mmConfig = new MultiMapConfig();
mmConfig.setName( "default" );

mmConfig.setBackupCount( "0" ).setAsyncBackupCount( "1" );

mmConfig.setValueCollectionType( "SET" );
```

Most of the MultiMap configuration elements and attributes have the same names and functionalities explained in the Map Configuration section. Below are the ones specific to MultiMap.

- **statistics-enabled**: You can retrieve some statistics like owned entry count, backup entry count, last update time, locked entry count by setting this parameter’s value as “true”. The method for retrieving the statistics is `getLocalMultiMapStats()`.
- **value-collection-type**: Type of the value collection. It can be `Set` or `List`.

23.9 Queue Configuration

The following are example queue configurations.

Declarative:
23.10 Topic Configuration

The following are example topic configurations.

Declarative Configuration:

```xml
<hazelcast>
...
</hazelcast>
```

Programmatic:

```java
Config config = new Config();
QueueConfig queueConfig = config.getQueueConfig();
queueConfig.setName("MyQueue").setBackupCount("1").setMaxSize("0").setStatisticsEnabled("true");
queueConfig.getQueueStoreConfig().setEnabled("true").setClassName("com.hazelcast.QueueStoreImpl").setProperty("binary", "false");
```

Queue configuration has the following elements.

- **max-size**: Maximum number of items in the Queue.
- **backup-count**: Number of synchronous backups. Queue is a non-partitioned data structure, so all entries of a Set resides in one partition. When this parameter is ‘1’, it means there will be 1 backup of that Set in another node in the cluster. When it is ‘2’, 2 nodes will have the backup.
- **async-backup-count**: Number of asynchronous backups.
- **empty-queue-ttl**: Used to purge unused or empty queues. If you define a value (time in seconds) for this element, then your queue will be destroyed if it stays empty or unused for that time.
- **item-listeners**: Lets you add listeners (listener classes) for the queue items. You can also set the attribute `include-value` to `true` if you want the item event to contain the item values.
- **queue-store**: Includes the queue store factory class name and the properties `binary`, `memory limit` and `bulk load`. Please refer to Queue Persistence.
- **statistics-enabled**: If set to `true`, you can retrieve statistics for this Queue using the method `getLocalQueueStats()`.

---

23.10 TOPIC CONFIGURATION

<queue name="default">
  <max-size>0</max-size>
  <backup-count>1</backup-count>
  <async-backup-count>0</async-backup-count>
  <empty-queue-ttl>-1</empty-queue-ttl>
  <item-listeners>
    <item-listener>
      com.hazelcast.examples.ItemListener
    </item-listener>
  </item-listeners>
</queue>

<queue-store>
  <class-name>com.hazelcast.QueueStoreImpl</class-name>
  <properties>
    <property name="binary">false</property>
    <property name="memory-limit">10000</property>
    <property name="bulk-load">500</property>
  </properties>
</queue-store>

Programmatic:

```java
Config config = new Config();
QueueConfig queueConfig = config.getQueueConfig();
queueConfig.setName( "MyQueue" ).setBackupCount( "1" )
  .setMaxSize( "0" ).setStatisticsEnabled( "true" );
queueConfig.getQueueStoreConfig()
  .setEnabled ( "true" )
  .setClassName( "com.hazelcast.QueueStoreImpl" )
  .setProperty( "binary", "false" );
```
Programmatic Configuration:

```
TopicConfig topicConfig = new TopicConfig();
topicConfig.setGlobalOrderingEnabled( true );
topicConfig.setStatisticsEnabled( true );
topicConfig.setName( "yourTopicName" );
MessageListener<String> implementation = new MessageListener<String>() {
    @Override
    public void onMessage( Message<String> message ) {
      // process the message
    }
};
topicConfig.addMessageListenerConfig( new ListenerConfig( implementation ) );
HazelcastInstance instance = Hazelcast.newHazelcastInstance()
```

Topic configuration has the following elements.

- **statistics-enabled**: Default is `true`, meaning statistics are calculated.
- **global-ordering-enabled**: Default is `false`, meaning there is no global order guarantee.
- **message-listeners**: Lets you add listeners (listener classes) for the topic messages.

Topic related but not topic specific configuration parameters

- `'hazelcast.event.queue.capacity'`: default value is 1,000,000.
- `'hazelcast.event.queue.timeout.millis'`: default value is 250.
- `'hazelcast.event.thread.count'`: default value is 5.

**RELATED INFORMATION**

For description of these parameters, please see *Global Event Configuration*.  

### 23.11 Reliable Topic Configuration

The following are example Reliable Topic configurations.

**Declarative:**

```
<reliable-topic name="default">
  <statistics-enabled>true</statistics-enabled>
  <message-listeners>
    <message-listener>
      ...
    </message-listener>
  </message-listeners>
  <read-batch-size>10</read-batch-size>
  <topic-overload-policy>BLOCK</topic-overload-policy>
</reliable-topic>
```
23.12. List Configuration

Programmatic:

```java
Config config = new Config();
ReliableTopicConfig rtConfig = config.getReliableTopicConfig();
rtConfig.setTopicOverloadPolicy( "BLOCK" )
    .setReadBatchSize( 10 )
    .setStatisticsEnabled( "true" );
```

Reliable Topic configuration has the following elements.

- **statistics-enabled**: Enables or disables the statistics collection for the Reliable Topic. The default value is `true`.
- **message-listener**: Message listener class that listens to the messages when they are added or removed.
- **read-batch-size**: Minimum number of messages that Reliable Topic will try to read in batches. The default value is 10.
- **topic-overload-policy**: Policy to handle an overloaded topic. Available values are `DISCARD_OLDEST`, `DISCARD_NEWEST`, `BLOCK` and `ERROR`. The default value is `BLOCK`.

23.12 List Configuration

The following are example list configurations.

Declarative:

```xml
<list name="default">
  <backup-count>1</backup-count>
  <async-backup-count>0</async-backup-count>
  <max-size>10</max-size>
  <item-listeners>
    <item-listener>
      com.hazelcast.examples.ItemListener
    </item-listener>
  </item-listeners>
</list>
```

Programmatic:

```java
Config config = new Config();
CollectionConfig collectionList = config.getCollectionConfig();
collectionList.setName( "MyList" ).setBackupCount( "1" )
    .setMaxSize( "10" );
```

List configuration has the following elements.

- **backup-count**: Number of synchronous backups. List is a non-partitioned data structure, so all entries of a List reside in one partition. When this parameter is ‘1’, there will be 1 backup of that List in another node in the cluster. When it is ‘2’, 2 nodes will have the backup.
- **async-backup-count**: Number of asynchronous backups.
- **max-size**: The maximum number of entries for this List.
- **item-listeners**: Lets you add listeners (listener classes) for the list items. You can also set the attribute `include-value` to `true` if you want the item event to contain the item values, and you can set the attribute `local` to `true` if you want to listen the items on the local node.
23.13 Set Configuration

The following are the example set configurations.

Declarative:

```xml
<set name="default">
  <backup-count>1</backup-count>
  <async-backup-count>0</async-backup-count>
  <max-size>10</max-size>
  <item-listeners>
    <item-listener>
      com.hazelcast.examples.ItemListener
    </item-listener>
  </item-listeners>
</set>
```

Programmatic:

```java
Config config = new Config();
CollectionConfig collectionSet = config.getCollectionConfig();
collectionSet.setName( "MySet" ).setBackupCount( "1" )
    .setMaxSize( "10" );
```

Set configuration has the following elements.

- **backup-count**: Count of synchronous backups. Set is a non-partitioned data structure, so all entries of a Set reside in one partition. When this parameter is ‘1’, it means there will be 1 backup of that Set in another node in the cluster. When it is ‘2’, 2 nodes will have the backup.
- **async-backup-count**: Count of asynchronous backups.
- **max-size**: The maximum number of entries for this Set.
- **item-listeners**: Lets you add listeners (listener classes) for the list items. You can also set the attributes include-value to true if you want the item event to contain the item values, and you can set local to true if you want to listen to the items on the local node.

23.14 Ringbuffer Configuration

The following are example Ringbuffer configurations.

Declarative:

```xml
<ringbuffer name="default">
  <capacity>1000</capacity>
  <time-to-live-seconds>0</time-to-live-seconds>
  <backup-count>1</backup-count>
  <async-backup-count>0</async-backup-count>
  <in-memory-format>BINARY</in-memory-format>
</ringbuffer>
```

Programmatic:

```java
Config config = new Config();
RingbufferConfig rbConfig = config.getRingbufferConfig();
rbConfig.setCapacity( 1000 )
    .setTimeToLiveSeconds( 0 )
    .setBackupCount( 1 )
    .setAsyncBackupCount( 0 )
    .setInMemoryFormat( "BINARY" );
```
23.15 Semaphore Configuration

Ringbuffer configuration has the following elements.

- **capacity**: Total number of items in the Ringbuffer. The default value is 10000.
- **time-to-live-seconds**: Duration that the Ringbuffer retains the items before deleting them. When it is set to 0, it will be disabled. The default value is 0.
- **backup-count**: Number of synchronous backups. The default value is 1.
- **async-backup-count**: Number of asynchronous backups. The default value is 0.
- **in-memory-format**: In-memory format of an item when stored in the Ringbuffer. Available values are OBJECT and BINARY. The default value is BINARY.

The following are example semaphore configurations.

**Declarative:**

```xml
<semaphore name="semaphore">
  <backup-count>1</backup-count>
  <async-backup-count>0</async-backup-count>
  <initial-permits>3</initial-permits>
</semaphore>
```

**Programmatic:**

```java
Config config = new Config();
SemaphoreConfig semaphoreConfig = config.getSemaphoreConfig();
semaphoreConfig.setName( "semaphore" ).setBackupCount( "1" ).setInitialPermits( "3" );
```

It has below elements.

- **initial-permits**: the thread count to which the concurrent access is limited. For example, if you set it to “3”, concurrent access to the object is limited to 3 threads.
- **backup-count**: Number of synchronous backups.
- **async-backup-count**: Number of asynchronous backups.

23.16 Executor Service Configuration

Following are the example configurations for executor service.

**Declarative:**

```xml
<executor-service name="exec">
  <pool-size>1</pool-size>
  <queue-capacity>10</queue-capacity>
  <statistics-enabled>true</statistics-enabled>
</executor-service>
```

**Programmatic:**

```java
Config config = new Config();
ExecutorConfig executorConfig = config.getExecutorConfig();
executorConfig.setPoolSize( "1" ).setQueueCapacity( "10" ).setStatisticsEnabled( true );
```
CHAPTER 23. HAZELCAST CONFIGURATION

Executor service configuration has the following elements.

- **pool-size**: The number of executor threads per Member for the Executor.
- **queue-capacity**: Executor’s task queue capacity.
- **statistics-enabled**: Some statistics like pending operations count, started operations count, completed operations count, cancelled operations count can be retrieved by setting this parameter’s value as **true**. The method for retrieving the statistics is `getLocalExecutorStats()`.

### 23.17 Cache Configuration

Following are examples of cache configurations.

**Declarative:**

```xml
<cache name="default">
  <key-type class-name="exampleClass"></key-type>
  <value-type class-name="exampleClass"></value-type>
  <statistics-enabled>true</statistics-enabled>
  <management-enabled>true</management-enabled>
  <read-through>false</read-through>
  <write-through>true</write-through>
  <cache-loader-factory class-name="exampleClass"></cache-loader-factory>
  <cache-writer-factory class-name="exampleClass"></cache-writer-factory>
  <expiry-policy-factory class-name="exampleClass"></expiry-policy-factory>
  <cache-entry-listeners>
    <cache-entry-listener old-value-required="true">
      <cache-entry-listener-factory class-name="exampleClass"></cache-entry-listener-factory>
      <cache-entry-event-filter-factory class-name="exampleClass"></cache-entry-event-filter-factory>
    </cache-entry-listener>
  </cache-entry-listeners>
  <in-memory-format>BINARY</in-memory-format>
  <backup-count>1</backup-count>
  <async-backup-count>0</async-backup-count>
  <eviction size="10000" max-size-policy="ENTRY_COUNT"></eviction>
</cache>
```

**Programmatic:**

```java
ICache<Object, Object> cache = cacheManager.getCache().unwrap(ICache.class);
CacheConfig cacheConfig = cache.getConfiguration(CacheConfig.class);

cacheConfig.setStatisticsEnabled( true ).setManagementEnabled( true );
cacheConfig.setReadThrough( true ).setWriteThrough( true );
  .setBackupCount( "1" )
  .setAsyncBackupCount( "0" );

CacheEvictionConfig cacheEvictionConfig = cacheConfig.getEvictionConfig();
cacheEvictionConfig.setSize( "10000" ).setMaxSizePolicy( CacheMaxSizePolicy.ENTRY_COUNT );
```

Cache configuration has the following elements.

- **name**: Name of the cache.
- **key-type**: Type of the keys provided as a full class name.
- **value-type**: Type of the values provided as a full class name.
- **statistics-enabled**: If set as **true**, you can retrieve statistics for this cache using the `getLocalCacheStatistics()` method.
• **management-enabled**: Defines whether the management is enabled on this cache. If you set it to `true`, you can monitor this cache on the Hazelcast Management Center.
• **read-through**: If you want to use the read-through caching mode, set this value to `true`.
• **write-through**: If you want to use the write-through caching mode, set this value to `true`.
• **cache-loader-factory**: Defines the cache loader factory class name.
• **cache-writer-factory**: Defines the cache writer factory class name.
• **expiry-policy-factory**: Defines the expiry policy factory class name.
• **cache-entry-listeners**: Includes the list of cache entry listeners.
• **in-memory-format**: Data type that will be used for storing the records. Possible values are `BINARY` (default), `OBJECT` and `NATIVE`. If you set it to `BINARY`, the keys and values will be stored as binary data. If you set it to `OBJECT`, the values will be stored in their object forms. If it is `NATIVE`, the keys and values will be stored in the native memory.
• **backup-count**: Count of synchronous backups.
• **async-backup-count**: Count of asynchronous backups.
• **eviction**: When the maximum size is reached, the cache is evicted based on the eviction policy. It has the following attributes.
  - **size**: The cache size can be any integer between 0 and `Integer.MAX_VALUE`. Default value is 0.
  - **max-size-policy**: Available policies are listed below.
    * `ENTRY_COUNT`: Maximum number of cache entries in the cache. This is the default value.
    * `USED_NATIVE_MEMORY_SIZE`: Maximum used native memory size in megabytes for each JVM.
    * `USED_NATIVE_MEMORY_PERCENTAGE`: Maximum used native memory size percentage for each JVM.
    * `FREE_NATIVE_MEMORY_SIZE`: Maximum free native memory size in megabytes for each JVM.
    * `FREE_NATIVE_MEMORY_PERCENTAGE`: Maximum free native memory size percentage for each JVM.
  - **eviction-policy**: Available policies are LRU (Least Recently Used) and LFU (Least Frequently Used). Default value is LRU.

### 23.18 Serialization Configuration

The following are example serialization configurations.

**Declarative:**

```xml
<serialization>
  <portable-version>2</portable-version>
  <use-native-byte-order>true</use-native-byte-order>
  <byte-order>BIG_ENDIAN</byte-order>
  <enable-compression>true</enable-compression>
  <enable-shared-object>false</enable-shared-object>
  <allow-unsafe>true</allow-unsafe>
  <data-serializable-factories>
    <data-serializable-factory factory-id="1001">abc.xyz.Class</data-serializable-factory>
  </data-serializable-factories>
  <portable-factories>
    <portable-factory factory-id="9001">xyz.abc.Class</portable-factory>
  </portable-factories>
  <serializers>
    <global-serializer>abc.Class</global-serializer>
    <serializer type-class="Employee" class-name="com.EmployeeSerializer">
```
CHAPTER 23. HAZELCAST CONFIGURATION

Programmatic:

```java
Config config = new Config();
SerializationConfig szrConfig = config.getSerializationConfig();
szrConfig.setPortableVersion("2").setUseNativeByteOrder(true);
szrConfig.setAllowUnsafe(true).setEnableCompression(true);
szrConfig.setCheckClassDefErrors(true);

GlobalSerializerConfig globSrzConfig = szrConfig.getGlobalSerializerConfig();
globSrzConfig.setClassName("abc.Class");

SerializerConfig serializerConfig = szrConfig.getSerializerConfig();
serializerConfig.setTypeClass("Employee")
    .setClassName("com.EmployeeSerializer");
```

Serialization configuration has the following elements.

- **portable-version**: Defines versioning of the portable serialization. Portable version differentiates two of the same classes that have changes, such as adding/removing field or changing a type of a field.
- **use-native-byte-order**: Set to `true` to use native byte order for the underlying platform.
- **byte-order**: Defines the byte order that the serialization will use: `BIG_ENDIAN` or `LITTLE_ENDIAN`. The default value is `BIG_ENDIAN`.
- **enable-compression**: Enables compression if default Java serialization is used.
- **enable-shared-object**: Enables shared object if default Java serialization is used.
- **allow-unsafe**: Set to `true` to allow `unsafe` to be used.
- **data-serializable-factory**: The DataSerializableFactory class to be registered.
- **portable-factory**: The PortableFactory class to be registered.
- **global-serializer**: The global serializer class to be registered if no other serializer is applicable.
- **serializer**: The class name of the serializer implementation.
- **check-class-def-errors**: When set to `true`, the serialization system will check for class definitions error at start and will throw a Serialization Exception with an error definition.

### 23.19 MapReduce Jobtracker Configuration

The MapReduce JobTracker configuration sets the behavior of the Hazelcast MapReduce framework. Every JobTracker is capable of running multiple MapReduce jobs at the same time. Therefore, one configuration is a shared resource for all jobs created by the same JobTracker. The configuration gives full control over the expected load behavior and thread counts to be used.

Declarative:

```xml
<job-tracker name="default">
    <max-thread-size>0</max-thread-size>
    <queue-size>0</queue-size>
    <retry-count>0</retry-count>
    <chunk-size>1000</chunk-size>
    <communicate-stats>true</communicate-stats>
    <topology-changed-strategy>CANCEL_RUNNING_OPERATION</topology-changed-strategy>
</job-tracker>
```
Programmatic:

```
Config config = new Config();
JobTrackerConfig JTcfg = config.getJobTrackerConfig()
JTcfg.setName( "default" ).setQueueSize( "0" )
   .setChunkSize( "1000" );
```

MapReduce JobTracker configuration has the following elements.

- **max-thread-size**: The maximum thread pool size of the JobTracker.
- **queue-size**: The maximum number of tasks that can wait to be processed. A value of 0 means an unbounded queue. Very low numbers can prevent successful execution since the job might not be correctly scheduled or intermediate chunks are lost.
- **retry-count**: Currently not used but reserved for later use where the framework will automatically try to restart / retry operations from an available save point.
- **chunk-size**: Defines the number of emitted values before a chunk is sent to the reducers. If your emitted values are big or you want to better balance your work, you might want to change this to a lower or higher value. A value of 0 means immediate transmission, but remember that low values mean higher traffic costs. A very high value might cause an OutOfMemoryError if the emitted values do not fit into heap memory before being sent to reducers. To prevent this, you might want to use a combiner to pre-reduce values on mapping nodes.
- **communicate-stats**: Defines if statistics (for example, about processed entries) are transmitted to the job emitter. This might be used to show some kind of progress to a user inside of an UI system, but it produces additional traffic. If not needed, you might want to deactivate this.
- **topology-changed-strategy**: Defines how the MapReduce framework will react on topology changes while executing a job. Currently, only CANCEL_RUNNING_OPERATION is fully supported, which throws an exception to the job emitter (will throw a `com.hazelcast.mapreduce.TopologyChangedException`).

### 23.20 Services Configuration

This configuration is used for Hazelcast Service Provider Interface (SPI). The following are example configurations.

**Declarative:**

```
<services enable-defaults="true">
   <service enabled="true">
      <name>MyService</name>
      <class-name>MyServiceClass</class-name>
      <properties>
         <property>
            <property name="com.property.foo">value</property>
         </property>
      </properties>
      <configuration>
         abcConfig
      </configuration>
   </service>
</services>
```

**Programmatic:**

```
Config config = new Config();
ServicesConfig servicesConfig = config.getServicesConfig();

servicesConfig.setEnableDefaults( true );
```
ServiceConfig svcConfig = servicesConfig.getServiceConfig();
svcConfig.setEnabled( true ).setName( "MyService" )
    .setClassName( "MyServiceClass" );
 svcConfig.setProperty( "com.property.foo", "value" );

SPI configuration has the following elements.

- name: Name of the service to be registered.
- className: Name of the class that you develop for your service.
- properties: The custom properties that you can add to your service. You enable/disable these properties and set their values using this element.
- configuration: You can include configuration items which you develop using the Config object in your code.

### 23.21 Management Center Configuration

Management Center configuration is used to enable/disable Hazelcast Management Center and specify a time frequency for which the tool is updated with the cluster information.

The example configurations are shown below.

**Declarative:**

```xml
<management-center enabled="true" update-interval="3">http://localhost:8080/mancenter</management-center>
```

**Programmatic:**

```java
Config config = new Config();
config.getManagementCenterConfig().setEnabled( true )
    .setUrl( "http://localhost:8080/mancenter" )
    .setUpdateInterval( 3 );
```

Management Center configuration has the following attributes.

- enabled: Set to true to enable Management Center.
- url: The URL where Management Center will work.
- updateInterval: The time frequency (in seconds) for which Management Center will take information from Hazelcast cluster.

### 23.22 WAN Replication Configuration

The following are example WAN replication configurations.

**Declarative Configuration:**

```xml
<wan-replication name="my-wan-cluster">
    <target-cluster group-name="tokyo" group-password="tokyo-pass">
        <replication-impl>com.hazelcast.wan.impl.WanNoDelayReplication</replication-impl>
        <end-points>
            <address>10.2.1.1:5701</address>
            <address>10.2.1.2:5701</address>
        </end-points>
    </target-cluster>
</wan-replication>
```
23.23. ENTERPRISE WAN REPLICATION CONFIGURATION

Programmatic Configuration:

```java
Config config = new Config();
WanReplicationConfig wrConfig = config.getWanReplicationConfig();
WanTargetClusterConfig wtcConfig = wrConfig.getWanTargetClusterConfig();
wrConfig.setName("my-wan-cluster");
wtcConfig.setGroupName("tokyo").setGroupPassword("tokyo-pass");
wtcConfig.setReplicationImplObject("com.hazelcast.wan.impl.WanNoDelayReplication");
```

WAN replication configuration has the following elements.

- **name**: Name for your WAN replication configuration.
- **target-cluster**: Creates a group and its password.
- **replication-impl**: Name of the class implementation for the WAN replication.
- **end-points**: IP addresses of the cluster members for which the WAN replication is implemented.

23.23. Enterprise WAN Replication Configuration

The following are example Enterprise WAN replication configurations.

Declarative Configuration:

```xml
<wan-replication name="my-wan-cluster">
    <target-cluster group-name="tokyo" group-password="tokyo-pass">
        <replication-impl>com.hazelcast.enterprise.wan.replication.WanNoDelayReplication</replication-impl>
        <end-points>
            <address>10.2.1.1:5701</address>
            <address>10.2.1.2:5701</address>
        </end-points>
    </target-cluster>
</wan-replication>

<wan-replication name="my-wan-cluster-batch" snapshot-enabled="false">
    <target-cluster group-name="london" group-password="london-pass">
        <replication-impl>com.hazelcast.enterprise.wan.replication.WanBatchReplication</replication-impl>
        <end-points>
            <address>10.3.5.1:5701</address>
            <address>10.3.5.2:5701</address>
        </end-points>
    </target-cluster>
</wan-replication>
```
Programmatic Configuration:

```java
Config config = new Config();

// No delay replication config
WanReplicationConfig wrConfig = new WanReplicationConfig();
WanTargetClusterConfig wtcConfig = wrConfig.getWanTargetClusterConfig();
wrConfig.setName("my-wan-cluster");
wtcConfig.setGroupName("tokyo").setGroupPassword("tokyo-pass");
wtcConfig.setReplicationImpl("com.hazelcast.enterprise.wan.replication.WanNoDelayReplication");
List<String> endpoints = new ArrayList<String>();
endpoints.add("10.2.1.1:5701");
endpoints.add("10.2.1.1:5701");
wtcConfig.setEndpoints(endpoints);
config.addWanReplicationConfig(wrConfig);

// Batch replication config
WanReplicationConfig wrConfig = new WanReplicationConfig();
WanTargetClusterConfig wtcConfig = wrConfig.getWanTargetClusterConfig();
wrConfig.setName("my-wan-cluster-batch");
wrConfig.setSnapshotEnabled(false);
wtcConfig.setGroupName("london").setGroupPassword("london");
wtcConfig.setReplicationImpl("com.hazelcast.enterprise.wan.replication.WanBatchReplication");
List<String> batchEndpoints = new ArrayList<String>();
batchEndpoints.add("10.3.5.1:5701");
batchEndpoints.add("10.3.5.2:5701");
wtcConfig.setEndpoints(batchEndpoints);
config.addWanReplicationConfig(wrConfig);
```

Enterprise WAN replication configuration has the following elements.

- **name**: Name for your WAN replication configuration.
- **snapshot-enabled**: Only valid when used with `WanBatchReplication`. When set to `true`, only the latest events (based on key) are selected and sent in a batch.
- **target-cluster**: Configures target cluster’s group name and password.
- **replication-impl**: Name of the class implementation for the Enterprise WAN replication.
- **end-points**: IP addresses of the cluster members for which the Enterprise WAN replication is implemented.

### 23.23.1 IMap and ICache WAN Configuration

To enable WAN replication for an IMap or ICache instance, you can use `wan-replication-ref` element. Each IMap and ICache instance can have different WAN replication configurations.

Declarative Configuration:

```xml
<wan-replication name="my-wan-cluster">
...
</wan-replication>
<map name="testMap">
  <wan-replication-ref name="testWanRef">
    <merge-policy>TestMergePolicy</merge-policy>
    <republishing-enabled>false</republishing-enabled>
  </wan-replication-ref>
</map>
```
Programmatic Configuration:

```java
Config config = new Config();
WanReplicationConfig wrConfig = new WanReplicationConfig();
WanTargetClusterConfig wtcConfig = wrConfig.getWanTargetClusterConfig();
wrConfig.setName("my-wan-cluster");
...
config.addWanReplicationConfig(wrConfig);

WanReplicationRef wanRef = new WanReplicationRef();
wanRef.setName("my-wan-cluster");
wанRef.setMergePolicy(PassThroughMergePolicy.class.getName());
wанRef.setRepublishingEnabled(true);
config.getMapConfig("testMap").setWanReplicationRef(wanRef);

WanReplicationRef cacheWanRef = new WanReplicationRef();
cacheWanRef.setName("my-wan-cluster");
cacheWanRef.setRepublishingEnabled(true);
config.getCacheConfig("testCache").setWanReplicationRef(cacheWanRef);
```

**NOTE:** Caches that are created dynamically do not support WAN replication functionality. Cache configurations should be defined either declaratively (by XML) or programmatically on both source and target clusters.

**wan-replication-ref** has the following elements:

- **name:** Name of **wan-replication** configuration. IMap or ICache instance uses this **wan-replication** config. Please refer to the Enterprise WAN Replication Configuration section for details about **wan-replication** configuration.
- **merge-policy:** Resolve conflicts that are occurred when target cluster already has the replicated entry key.
- **republishing-enabled:** When enabled, an incoming event to a member is forwarded to target cluster of that member.

**Merge policies:**

4 merge policy implementations for IMap and 2 merge policy implementations for ICache are provided out of the box.

**IMap** has the following merge policies:

- **com.hazelcast.map.merge.PutIfAbsentMapMergePolicy:** Incoming entry merges from the source map to the target map if it does not exist in the target map.
- **com.hazelcast.map.merge.HigherHitsMapMergePolicy:** Incoming entry merges from the source map to the target map if the source entry has more hits than the target one.
- **com.hazelcast.map.merge.PassThroughMergePolicy:** Incoming entry merges from the source map to the target map unless the incoming entry is not null.
• `com.hazelcast.map.merge.LatestUpdateMapMergePolicy`: Incoming entry merges from the source map to the target map if the source entry has been updated more recently than the target entry. Please note that this merge policy can only be used when the clusters' clocks are in sync.

ICache has the following merge policies:

• `com.hazelcast.cache.merge.HigherHitsCacheMergePolicy`: Incoming entry merges from the source cache to the target cache if the source entry has more hits than the target one.
• `com.hazelcast.cache.merge.PassThroughCacheMergePolicy`: Incoming entry merges from the source cache to the target cache unless the incoming entry is not null.

### 23.24 Partition Group Configuration

Hazelcast distributes key objects into partitions using a consistent hashing algorithm. Those partitions are assigned to nodes. An entry is stored in the node that owns the partition to which the entry’s key is assigned. The total partition count is 271 by default; you can change it with the configuration property `hazelcast.map.partition.count`. Please see the System Properties section.

Along with those partitions, there are also copies of the partitions as backups. Backup partitions can have multiple copies due to the backup count defined in configuration, such as first backup partition, second backup partition, etc. A node cannot hold more than one copy of a partition (ownership or backup). By default, Hazelcast distributes partitions and their backup copies randomly and equally among cluster nodes, assuming all nodes in the cluster are identical.

But what if some nodes share the same JVM or physical machine or chassis and you want backups of these nodes to be assigned to nodes in another machine or chassis? What if processing or memory capacities of some nodes are different and you do not want an equal number of partitions to be assigned to all nodes?

You can group nodes in the same JVM (or physical machine) or nodes located in the same chassis. Or you can group nodes to create identical capacity. We call these groups partition groups. Partitions are assigned to those partition groups instead of to single nodes. Backups of these partitions are located in another partition group.

When you enable partition grouping, Hazelcast presents three choices for you to configure partition groups.

• You can group nodes automatically using the IP addresses of nodes, so nodes sharing the same network interface will be grouped together. All members on the same host (IP address or domain name) will be a single partition group. This helps to avoid data loss when a physical server crashes, because multiple replicas of the same partition are not stored on the same host. But if there are multiple network interfaces or domain names per physical machine, that will make this assumption invalid.

```xml
<partition-group enabled="true" group-type="HOST_AWARE"/>
```

Config config = ...;
PartitionGroupConfig partitionGroupConfig = config.getPartitionGroupConfig();
partitionGroupConfig.setEnabled(true)
    .setGroupType(MemberGroupType.HOST_AWARE);

• You can do custom grouping using Hazelcast’s interface matching configuration. This way, you can add different and multiple interfaces to a group. You can also use wildcards in the interface addresses. For example, the users can create rack aware or data warehouse partition groups using custom partition grouping.

```xml
<partition-group enabled="true" group-type="CUSTOM">
    <member-group>
        <interface>10.10.0.0</interface>
        <interface>10.10.3.0</interface>
        <interface>10.10.5.0</interface>
    </member-group>
</partition-group>
```
<member-group>
    <interface>10.10.0.*</interface>
    <interface>10.10.1.*</interface>
    <interface>10.10.2.*</interface>
</member-group>

Config config = ...;
PartitionGroupConfig partitionGroupConfig = config.getPartitionGroupConfig();
partitionGroupConfig.setEnabled(true)
    .setGroupType(MemberGroupType.CUSTOM);

MemberGroupConfig memberGroupConfig = new MemberGroupConfig();
memberGroupConfig.addInterface("10.10.0.*")
    .addInterface("10.10.3.*")
    .addInterface("10.10.5.*");

MemberGroupConfig memberGroupConfig2 = new MemberGroupConfig();
memberGroupConfig2.addInterface("10.10.10-100")
    .addInterface("10.10.1.")
    .addInterface("10.10.2.");

partitionGroupConfig.addMemberGroupConfig(memberGroupConfig);
partitionGroupConfig.addMemberGroupConfig(memberGroupConfig2);

- You can give every member its own group. Each member is a group of its own and primary and backup
  partitions are distributed randomly (not on the same physical member). This gives the least amount of
  protection and is the default configuration for a Hazelcast cluster.

<partition-group enabled="true" group-type="PER_MEMBER" />

Config config = ...;
PartitionGroupConfig partitionGroupConfig = config.getPartitionGroupConfig();
partitionGroupConfig.setEnabled(true)
    .setGroupType(MemberGroupType.PER_MEMBER);

23.25 Listener Configurations

You can add or remove event listeners to/from the related object using the Hazelcast API.

The downside of attaching listeners using this API is the possibility of missing events between the creation of an
object and registering the listener. To overcome this race condition, Hazelcast allows you to register listeners in
configuration. You can register listeners using declarative, programmatic, or Spring configuration.

23.25.0.1 MembershipListener

- Declarative Configuration

<listeners>
    <listener>com.hazelcast.examples.MembershipListener</listener>
</listeners>

- Programmatic Configuration

    config.addListenerConfig(
        new ListenerConfig("com.hazelcast.examples.MembershipListener")
    );
23.25.0.2 DistributedObjectListener

- Declarative Configuration

```xml
<listeners>
  <listener>com.hazelcast.examples.DistributedObjectListener</listener>
</listeners>
```

- Programmatic Configuration

```java
config.addListenerConfig(
  new ListenerConfig( "com.hazelcast.examples.DistributedObjectListener" ) );
```

- Spring XML configuration

```xml
<hz:listeners>
  <hz:listener class-name="com.hazelcast.spring.DummyDistributedObjectListener"/>
  <hz:listener implementation="dummyDistributedObjectListener"/>
</hz:listeners>
```

23.25.0.3 MigrationListener

- Declarative Configuration

```xml
<listeners>
  <listener>com.hazelcast.examples.MigrationListener</listener>
</listeners>
```

- Programmatic Configuration

```java
config.addListenerConfig(
  new ListenerConfig( "com.hazelcast.examples.MigrationListener" ) );
```

- Spring XML configuration

```xml
<hz:listeners>
  <hz:listener class-name="com.hazelcast.spring.DummyMigrationListener"/>
  <hz:listener implementation="dummyMigrationListener"/>
</hz:listeners>
```
23.25. LISTENER CONFIGURATIONS

23.25.0.4 LifecycleListener

- Declarative Configuration

```xml
<listeners>
  <listener>com.hazelcast.examples.LifecycleListener</listener>
</listeners>
```

- Programmatic Configuration

```java
config.addListenerConfig(
    new ListenerConfig( "com.hazelcast.examples.LifecycleListener" ) );
```

- Spring XML configuration

```xml
<hz:listeners>
  <hz:listener class-name="com.hazelcast.spring.DummyLifecycleListener"/>
  <hz:listener implementation="dummyLifecycleListener"/>
</hz:listeners>
```

23.25.0.5 EntryListener for IMap

- Declarative Configuration

```xml
<map name="default">
  ...
  <entry-listeners>
    <entry-listener include-value="true" local="false">
      com.hazelcast.examples.EntryListener
    </entry-listener>
  </entry-listeners>
</map>
```

- Programmatic Configuration

```java
mapConfig.addEntryListenerConfig(
    new EntryListenerConfig( "com.hazelcast.examples.EntryListener", false, false ) );
```

- Spring XML configuration

```xml
<hz:map name="default">
  <hz:entry-listeners>
    <hz:entry-listener include-value="true" class-name="com.hazelcast.spring.DummyEntryListener"/>
    <hz:entry-listener implementation="dummyEntryListener" local="true"/>
  </hz:entry-listeners>
</hz:map>
```

23.25.0.6 EntryListener for MultiMap

- Declarative Configuration
CHAPTER 23. HAZELCAST CONFIGURATION

<multimap name="default">
   <value-collection-type>SET</value-collection-type>
   <entry-listeners>
      <entry-listener include-value="true" local="false">
         com.hazelcast.examples.EntryListener
      </entry-listener>
   </entry-listeners>
</multimap>

- Programmatic Configuration

multiMapConfig.addEntryListenerConfig(
   new EntryListenerConfig( "com.hazelcast.examples.EntryListener", false, false )
);

- Spring XML configuration

<hz:multimap name="default" value-collection-type="LIST">
   <hz:entry-listeners>
      <hz:entry-listener include-value="true"
         class-name="com.hazelcast.spring.DummyEntryListener"/>
      <hz:entry-listener implementation="dummyEntryListener" local="true"/>
   </hz:entry-listeners>
</hz:multimap>

23.25.0.7 ItemListener for IQueue

- Declarative Configuration

<queue name="default">
   ...
   <item-listeners>
      <item-listener include-value="true">
         com.hazelcast.examples.ItemListener
      </item-listener>
   </item-listeners>
</queue>

- Programmatic Configuration

queueConfig addItemListenerConfig(
   new ItemListenerConfig( "com.hazelcast.examples.ItemListener", true )
);

- Spring XML configuration

<hz:queue name="default">
   ...
   <hz:item-listeners>
      <hz:item-listener include-value="true"
         class-name="com.hazelcast.spring.DummyItemListener"/>
   </hz:item-listeners>
</hz:queue>
23.26. LOGGING CONFIGURATION

23.25.0.8 MessageListener for ITopic

- Declarative Configuration

```xml
<topic name="default">
  <message-listeners>
    <message-listener>
      com.hazelcast.examples.MessageListener
    </message-listener>
  </message-listeners>
</topic>
```

- Programmatic Configuration

```java
topicConfig.addMessageListenerConfig(new ListenerConfig( "com.hazelcast.examples.MessageListener" ));
```

- Spring XML configuration

```xml
<hz:topic name="default">
  <hz:message-listeners>
    <hz:message-listener
      class-name="com.hazelcast.spring.DummyMessageListener"/>
  </hz:message-listeners>
</hz:topic>
```

23.25.0.9 ClientListener

- Declarative Configuration

```xml
<listeners>
  <listener>com.hazelcast.examples.ClientListener</listener>
</listeners>
```

- Programmatic Configuration

```java
topicConfig.addMessageListenerConfig(new ListenerConfig( "com.hazelcast.examples.ClientListener" ));
```

- Spring XML configuration

```xml
<hz:listeners>
  <hz:listener
    class-name="com.hazelcast.spring.DummyClientListener"/>
  <hz:listener
    implementation="dummyClientListener"/>
</hz:listeners>
```

23.26 Logging Configuration

Hazelcast has a flexible logging configuration and does not depend on any logging framework except JDK logging. It has built-in adaptors for a number of logging frameworks and it also supports custom loggers by providing logging interfaces.

To use built-in adaptors, set the `hazelcast.logging.type` property to one of the predefined types below.
• jdk: JDK logging (default)
• log4j: Log4j
• slf4j: Slf4j
• none: disable logging

You can set `hazelcast.logging.type` through declarative configuration, programmatic configuration, or JVM system property.

**NOTE:** If you choose to use log4j or slf4j, you should include the proper dependencies in the classpath.

**Declarative Configuration**

```xml
<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">

    ....

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

**Programmatic Configuration**

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

**System Property**

- Using JVM parameter: `java -Dhazelcast.logging.type=slf4j`
- Using System class: `System.setProperty( "hazelcast.logging.type", "none" );`;

If the provided logging mechanisms are not satisfactory, you can implement your own using the custom logging feature. To use it, implement the `com.hazelcast.logging.LoggerFactory` and `com.hazelcast.logging.ILogger` interfaces and set the system property `hazelcast.logging.class` as your custom LoggerFactory class name.

`-Dhazelcast.logging.class=foo.bar.MyLoggingFactory`

You can also listen to logging events generated by Hazelcast runtime by registering LogListeners to LoggingService.

```java
LogListener listener = new LogListener() {
    public void log( LogEvent logEvent ) {
        // do something
    }
};
HazelcastInstance instance = Hazelcast.newHazelcastInstance();
LoggingService loggingService = instance.getLoggingService();
loggingService.addLogListener( Level.INFO, listener );
```
23.27 System Properties

Hazelcast has system properties to tune some aspects of Hazelcast. You can set them as property name and value pairs through declarative configuration, programmatic configuration or JVM system property.

Declarative Configuration

```xml
<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">

....

<properties>
  <property name="hazelcast.property.foo">value</property>

....

</properties>
</hazelcast>
```

Programmatic Configuration

```
Config config = new Config();
config.setProperty( "hazelcast.property.foo", "value" );
```

System Property

1. Using JVM parameter: `java -Dhazelcast.property.foo=value`
2. Using System class: `System.setProperty( "hazelcast.property.foo", "value" );`

The table below lists the system properties with their descriptions in alphabetical order.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hazelcast.application.validation.token</td>
<td></td>
<td>string</td>
<td>This property can be used to verify that Hazelcast nodes only join when their application level configuration is the same.</td>
</tr>
<tr>
<td>hazelcast.backpressure.backoff.timeout.millis</td>
<td>60000</td>
<td>int</td>
<td>Controls the maximum timeout in milliseconds to wait for an invocation space to be available. The value needs to be equal to or larger than 0.</td>
</tr>
<tr>
<td>hazelcast.backpressure.enabled</td>
<td>false</td>
<td>bool</td>
<td>Enable back pressure.</td>
</tr>
<tr>
<td>hazelcast.backpressure.max.concurrent.invocations.per.partition</td>
<td>100</td>
<td>int</td>
<td>The maximum number of concurrent invocations per partition.</td>
</tr>
<tr>
<td>hazelcast.backpressure.syncwindow</td>
<td>1000</td>
<td>string</td>
<td>Used when back pressure is enabled.</td>
</tr>
<tr>
<td>hazelcast.cache.invalidation.batch.enabled</td>
<td>true</td>
<td>bool</td>
<td>Specifies whether the cache invalidation event batch sending is enabled or not.</td>
</tr>
<tr>
<td>hazelcast.cache.invalidation.batch.size</td>
<td>100</td>
<td>int</td>
<td>Defines the maximum number of cache invalidation events to be drained and sent to the event listeners in a batch.</td>
</tr>
<tr>
<td>hazelcast.cache.invalidation.batchfrequency.seconds</td>
<td>5</td>
<td>int</td>
<td>Specifies the frequency in seconds of cache invalidation event batch sending.</td>
</tr>
<tr>
<td>hazelcast.clientengine.thread.count</td>
<td>1000000</td>
<td>string</td>
<td>Defines the maximum number of threads to process non-partition-aware client requests, like <code>map.size()</code>, query, executor tasks, etc. Default count is 20 times number of cores.</td>
</tr>
<tr>
<td>hazelcast.client.event.queue.capacity</td>
<td>1000000</td>
<td>string</td>
<td>Default value of the capacity of the executor that handles incoming event packets.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Default Value</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>---------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>hazelcast.client.event.thread.count</td>
<td>5</td>
<td>string</td>
<td>Thread count for handling incoming event packets.</td>
</tr>
<tr>
<td>hazelcast.client.heartbeat.interval</td>
<td>10000</td>
<td>string</td>
<td>The frequency of heartbeat messages sent by the clients to members.</td>
</tr>
<tr>
<td>hazelcast.client.heartbeat.timeout</td>
<td>300000</td>
<td>string</td>
<td>Timeout for the heartbeat messages sent by the client to members.</td>
</tr>
<tr>
<td>hazelcast.client.invocation.timeout.seconds</td>
<td>120</td>
<td>int</td>
<td>Time to give up the invocation when a member in the member list is not reachable.</td>
</tr>
<tr>
<td>hazelcast.client.max.no.heartbeat.seconds</td>
<td>300</td>
<td>int</td>
<td>Time after which the node assumes the client is dead and closes its connections to the client.</td>
</tr>
<tr>
<td>hazelcast.client.shuffle.member.list</td>
<td>true</td>
<td>string</td>
<td>The client shuffles the given member list to prevent all clients to connect to the same node when this property is true. When it is set to false, the client tries to connect to the nodes in the given order.</td>
</tr>
<tr>
<td>hazelcast.connect.all.wait.seconds</td>
<td>120</td>
<td>int</td>
<td>Minimum interval in milliseconds to consider a connection error as critical.</td>
</tr>
<tr>
<td>hazelcast.connection.monitor.interval</td>
<td>3</td>
<td>int</td>
<td>Maximum IO error count before disconnecting from a node.</td>
</tr>
<tr>
<td>hazelcast.connection.monitor.max.faults</td>
<td>100000</td>
<td>int</td>
<td>Event queue capacity capacity of internal event queue.</td>
</tr>
<tr>
<td>hazelcast.event.queue.capacity</td>
<td>10000000</td>
<td>int</td>
<td>Event queue capacity.</td>
</tr>
<tr>
<td>hazelcast.event.queue.timeout.millis</td>
<td>250</td>
<td>int</td>
<td>Timeout to enqueue events to event queue.</td>
</tr>
<tr>
<td>hazelcast.event.thread.count</td>
<td>5</td>
<td>int</td>
<td>Number of event handler threads.</td>
</tr>
<tr>
<td>hazelcast.graceful.shutdown.max.wait</td>
<td>600</td>
<td>int</td>
<td>Maximum wait in seconds during graceful shutdown.</td>
</tr>
<tr>
<td>hazelcast.health.monitoring.delay.seconds</td>
<td>30000</td>
<td>int</td>
<td>Health monitoring logging interval in milliseconds.</td>
</tr>
<tr>
<td>hazelcast.health.monitoring.level</td>
<td>SILENT</td>
<td>string</td>
<td>Health monitoring logging level.</td>
</tr>
<tr>
<td>hazelcast.heartbeat.interval.seconds</td>
<td>1</td>
<td>int</td>
<td>Heartbeat send interval in seconds.</td>
</tr>
<tr>
<td>hazelcast.icmp.enabled</td>
<td>false</td>
<td>bool</td>
<td>Enable ICMP ping.</td>
</tr>
<tr>
<td>hazelcast.icmp.timeout</td>
<td>1000</td>
<td>int</td>
<td>ICMP timeout in milliseconds.</td>
</tr>
<tr>
<td>hazelcast.icmp.ttl</td>
<td>0</td>
<td>int</td>
<td>ICMP TTL (maximum)</td>
</tr>
<tr>
<td>hazelcast.initial.min.cluster.size</td>
<td>0</td>
<td>int</td>
<td>Initial expected cluster size.</td>
</tr>
<tr>
<td>hazelcast.initial.wait.seconds</td>
<td>0</td>
<td>int</td>
<td>Initial time in seconds for waiting to start a node.</td>
</tr>
<tr>
<td>hazelcast.io.balancer.interval.seconds</td>
<td>20</td>
<td>int</td>
<td>Number of socket input threads that will be executed on backup members.</td>
</tr>
<tr>
<td>hazelcast.io.input.thread.count</td>
<td>3</td>
<td>int</td>
<td>Number of threads performing socket input.</td>
</tr>
<tr>
<td>hazelcast.io.output.thread.count</td>
<td>3</td>
<td>int</td>
<td>Number of threads performing socket output.</td>
</tr>
<tr>
<td>hazelcast.io.thread.count</td>
<td>3</td>
<td>int</td>
<td>Number of threads performing socket output.</td>
</tr>
<tr>
<td>hazelcast.jcache.provider.type</td>
<td>string</td>
<td></td>
<td>Type of the JCache provider.</td>
</tr>
<tr>
<td>hazelcast.jmx</td>
<td>false</td>
<td>bool</td>
<td>Enable JMX agent.</td>
</tr>
<tr>
<td>hazelcast.jmx.detailed</td>
<td>false</td>
<td>bool</td>
<td>Enable cluster-wide stats for List, Map, ReplicatedMap, Queue, Set and Topic in JMX agent.</td>
</tr>
<tr>
<td>hazelcast.logging.type</td>
<td>jdk</td>
<td>enum</td>
<td>Name of logging framework type.</td>
</tr>
<tr>
<td>hazelcast.map.expiry.delay.seconds</td>
<td>10</td>
<td>int</td>
<td>Useful to deal with some possible edge cases.</td>
</tr>
<tr>
<td>hazelcast.map.load.chunk.size</td>
<td>1000</td>
<td>int</td>
<td>Chunk size for MapLoader.loadAllKeys().</td>
</tr>
<tr>
<td>hazelcast.map.replica.wait.seconds.for.scheduled.tasks</td>
<td>10</td>
<td>int</td>
<td>Scheduler delay for map scheduled tasks.</td>
</tr>
<tr>
<td>hazelcast.map.write.behind.queue.capacity</td>
<td>50000</td>
<td>int</td>
<td>Maximum write-behind queue capacity.</td>
</tr>
<tr>
<td>hazelcast.master.confirmation.interval.seconds</td>
<td>30</td>
<td>int</td>
<td>Interval at which node sends master confirmation messages.</td>
</tr>
</tbody>
</table>

...
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hazelcast.max.join.merge.target.seconds</td>
<td>20</td>
<td>int</td>
<td>Split-brain merge timeout</td>
</tr>
<tr>
<td>hazelcast.max.join.seconds</td>
<td>300</td>
<td>int</td>
<td>Join timeout, maximum time to try to join before giving</td>
</tr>
<tr>
<td>hazelcast.max.no.heartbeat.seconds</td>
<td>300</td>
<td>int</td>
<td>Maximum timeout of heartbeat</td>
</tr>
<tr>
<td>hazelcast.max.no.master.confirmation.seconds</td>
<td>450</td>
<td>int</td>
<td>Max timeout of master confirmation</td>
</tr>
<tr>
<td>hazelcast.max.wait.seconds.before.join</td>
<td>20</td>
<td>int</td>
<td>Maximum wait time before joining</td>
</tr>
<tr>
<td>hazelcast.mc.max.visible.instance.count</td>
<td>100</td>
<td>int</td>
<td>Management Center maximum visible instance count</td>
</tr>
<tr>
<td>hazelcast.mc.max.visible.slow.operations.count</td>
<td>10</td>
<td>int</td>
<td>Management Center maximum visible slow operations count</td>
</tr>
<tr>
<td>hazelcast.mc.url.change.enabled</td>
<td>true</td>
<td>bool</td>
<td>Enable Memcache client request listener service</td>
</tr>
<tr>
<td>hazelcast.member.list.publish.interval.seconds</td>
<td>600</td>
<td>int</td>
<td>Interval at which master node publishes a member list</td>
</tr>
<tr>
<td>hazelcast.member.list.publish.interval.seconds</td>
<td>true</td>
<td>bool</td>
<td>Enable Hazelcast shutdownhook thread</td>
</tr>
<tr>
<td>hazelcast.merge.first.run.delay.seconds</td>
<td>300</td>
<td>int</td>
<td>Initial run delay of split-brain merge</td>
</tr>
<tr>
<td>hazelcast.merge.next.run.delay.seconds</td>
<td>120</td>
<td>int</td>
<td>Run interval of split brain merge</td>
</tr>
<tr>
<td>hazelcast.migration.min.delay.on.member.removed.seconds</td>
<td>5</td>
<td>int</td>
<td>Minimum delay (in seconds) for member removal</td>
</tr>
<tr>
<td>hazelcast.operation.backup.timeout.millis</td>
<td>5</td>
<td>int</td>
<td>Maximum time a caller waits for backup response</td>
</tr>
<tr>
<td>hazelcast.operation.call.timeout.millis</td>
<td>600000</td>
<td>int</td>
<td>Timeout to wait for a response</td>
</tr>
<tr>
<td>hazelcast.operation.generic.thread.count</td>
<td>-1</td>
<td>int</td>
<td>Number of generic operation handler threads</td>
</tr>
<tr>
<td>hazelcast.operation.thread.count</td>
<td>-1</td>
<td>int</td>
<td>Number of operation threads</td>
</tr>
<tr>
<td>hazelcast.partition.backup.sync.interval</td>
<td>30</td>
<td>int</td>
<td>Interval for syncing backup replicas</td>
</tr>
<tr>
<td>hazelcast.partition.count</td>
<td>271</td>
<td>int</td>
<td>Total partition count</td>
</tr>
<tr>
<td>hazelcast.partition.max.parallel.replications</td>
<td>5</td>
<td>int</td>
<td>Maximum number of parallel replications</td>
</tr>
<tr>
<td>hazelcast.partition.migration.interval</td>
<td>300</td>
<td>int</td>
<td>Timeout for partition migration</td>
</tr>
<tr>
<td>hazelcast.partition.table.send.interval</td>
<td>15</td>
<td>int</td>
<td>Interval for publishing partition table to all members</td>
</tr>
<tr>
<td>hazelcast.partitioning.strategy.class</td>
<td>null</td>
<td>string</td>
<td>Class name implementing PartitioningStrategy</td>
</tr>
<tr>
<td>hazelcast.performance.monitor.max.rolled.file.count</td>
<td>10</td>
<td>int</td>
<td>The PerformanceMonitor maximum rolled file count</td>
</tr>
<tr>
<td>hazelcast.performance.monitor.max.rolled.file.size.mb</td>
<td>10</td>
<td>int</td>
<td>The performance monitor maximum rolled file size</td>
</tr>
<tr>
<td>hazelcast.performance.monitoring.enabled</td>
<td>true</td>
<td>bool</td>
<td>Enable the performance monitor</td>
</tr>
<tr>
<td>hazelcast.performance.monitoring.delay.seconds</td>
<td>int</td>
<td>int</td>
<td>The delay in seconds for the performance monitor</td>
</tr>
<tr>
<td>hazelcast.prefer.ipv4.stack</td>
<td>true</td>
<td>bool</td>
<td>Prefer IPv4 network in partitioning</td>
</tr>
<tr>
<td>hazelcast.query.max.local.partition.limit.for.precheck</td>
<td>3</td>
<td>int</td>
<td>Maximum value of local partition limit for precheck</td>
</tr>
<tr>
<td>hazelcast.query.result.size.limit</td>
<td>-1</td>
<td>int</td>
<td>Result size limit for query operations</td>
</tr>
<tr>
<td>hazelcast.rest.enabled</td>
<td>true</td>
<td>bool</td>
<td>Enable REST client request listener</td>
</tr>
<tr>
<td>hazelcast.shutdownhook.enabled</td>
<td>true</td>
<td>bool</td>
<td>Enable Hazelcast shutdownhook</td>
</tr>
<tr>
<td>hazelcast.slow.operation.detector.enabled</td>
<td>true</td>
<td>bool</td>
<td>Enables/disables the SlowOperationDetector</td>
</tr>
<tr>
<td>hazelcast.slow.operation.detector.log.purge.interval.seconds</td>
<td>300</td>
<td>int</td>
<td>Purge interval for slow operation log purge</td>
</tr>
<tr>
<td>hazelcast.slow.operation.detector.log.retention.seconds</td>
<td>36000</td>
<td>int</td>
<td>Defines the retention time for slow operation log retention</td>
</tr>
<tr>
<td>hazelcast.slow.operation.detector.stacktrace.logging.enabled</td>
<td>false</td>
<td>bool</td>
<td>Enables if the stacktrace logging is enabled</td>
</tr>
<tr>
<td>hazelcast.slow.operation.detector.threshold.millis</td>
<td>10000</td>
<td>int</td>
<td>Defines a threshold above which an operation is considered slow</td>
</tr>
<tr>
<td>hazelcast.socket.bind.any</td>
<td>true</td>
<td>bool</td>
<td>Bind both server-socket and client-socket</td>
</tr>
</tbody>
</table>

Note: The properties are listed in alphabetical order.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Default Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hazelcast.socket.client.bind</td>
<td>true</td>
<td>bool</td>
<td>Bind client socket to a local interface when connecting to a remote server socket. When set to false, client socket is not bound to any interface.</td>
</tr>
<tr>
<td>hazelcast.socket.client.bind.any</td>
<td>true</td>
<td>bool</td>
<td>Bind client-sockets to any local interface. If not set, hazelcast.socket.bind.any will be used as default.</td>
</tr>
<tr>
<td>hazelcast.socket.client.receive.buffer.size</td>
<td>-1</td>
<td>int</td>
<td>Hazelcast creates all connections with receive buffer size set according to the hazelcast.socket.receive.buffer.size. When it detects a connection opened by a client, then it adjusts the receive buffer size according to this property. It is in kilobytes and the default value is -1.</td>
</tr>
<tr>
<td>hazelcast.socket.client.send.buffer.size</td>
<td>-1</td>
<td>int</td>
<td>Hazelcast creates all connections with send buffer size set according to the hazelcast.socket.send.buffer.size. When it detects a connection opened by a client, then it adjusts the send buffer size according to this property. It is in kilobytes and the default value is -1.</td>
</tr>
<tr>
<td>hazelcast.socket.connect.timeout.seconds</td>
<td>0</td>
<td>int</td>
<td>Socket connection timeout in seconds. Socket.connect() will be blocked until either connection is established or connection is refused or this timeout passes. Default is 0, means infinite.</td>
</tr>
<tr>
<td>hazelcast.socket.keep.alive</td>
<td>true</td>
<td>bool</td>
<td>Socket set keep alive (SO_KEEPALIVE).</td>
</tr>
<tr>
<td>hazelcast.socket.linger.seconds</td>
<td>0</td>
<td>int</td>
<td>Set socket SO_LINGER option.</td>
</tr>
<tr>
<td>hazelcast.socket.no.delay</td>
<td>true</td>
<td>bool</td>
<td>Socket set TCP no delay.</td>
</tr>
<tr>
<td>hazelcast.socket.receive.buffer.size</td>
<td>32</td>
<td>int</td>
<td>Socket receive buffer (SO_RCVBUF) size in KB. If you have a very fast network (e.g. 10gbit) and/or you have large entries, then you may benefit from below tune the size. For example, a send/receive buffer size of 1024kB is a safe starting point for a 10gbit network.</td>
</tr>
<tr>
<td>hazelcast.socket.send.buffer.size</td>
<td>32</td>
<td>int</td>
<td>Socket send buffer (SO_SNDBUF) size in KB.</td>
</tr>
<tr>
<td>hazelcast.socket.server.bind.any</td>
<td>true</td>
<td>bool</td>
<td>Bind server-socket to any local interface. If not set, hazelcast.socket.bind.any will be used as default.</td>
</tr>
<tr>
<td>hazelcast.tcp.join.port.try.count</td>
<td>3</td>
<td>int</td>
<td>The number of incremental ports, starting with the port number defined in the network configuration, that will be used to connect to a host (which is defined without a port in TCP/IP member list while a node is searching for a cluster).</td>
</tr>
<tr>
<td>hazelcast.unsafe.mode</td>
<td>auto</td>
<td>string</td>
<td>Other than the default &quot;auto&quot;, it has two other values. One of them is &quot;disabled&quot; to explicitly disable the Unsafe usage in your platform. The other one is &quot;enforced&quot; to enforce the usage of Unsafe even if your platform does not support it. The default value &quot;auto&quot; automatically detects whether the usage of Unsafe is suitable for a given platform. This property can only be set by passing a JVM-wide system property.</td>
</tr>
<tr>
<td>hazelcast.version.check.enabled</td>
<td>true</td>
<td>bool</td>
<td>Enable Hazelcast new version check on startup.</td>
</tr>
<tr>
<td>hazelcast.wait.seconds.before.join</td>
<td>5</td>
<td>int</td>
<td>Wait time before join operation.</td>
</tr>
</tbody>
</table>
Chapter 24

Network Partitioning - Split Brain Syndrome

Imagine that you have a 10-node cluster and that the network is divided into two in a way that 4 servers cannot see the other 6. As a result, you end up having two separate clusters: 4-node cluster and 6-node cluster. Members in each sub-cluster think that the other nodes are dead even though they are not. This situation is called Network Partitioning (a.k.a. Split-Brain Syndrome).

However, these two clusters have a combination of the 271 (using default) primary and backup partitions. It’s very likely that not all of the 271 partitions, including both primaries and backups, exist in both mini-clusters. Therefore, from each mini-cluster’s perspective, data has been lost as some partitions no longer exist (they exist on the other segment).

24.1 Understanding Partition Recreation

If a MapStore was in use, those lost partitions would be reloaded from some database, making each mini-cluster complete. Each mini-cluster will then recreate the missing primary partitions and continue to store data in them, including backups on the other nodes.

24.2 Understanding Backup Partition Creation

When primary partitions exist without a backup, a backup version problem will be detected and a backup partition will be created. When backups exist without a primary, the backups will be promoted to primary partitions and new backups will be created with proper versioning. At this time, both mini-clusters have repaired themselves with all 271 partitions with backups, and continue to handle traffic without any knowledge of each other. Given that they have enough remaining memory (assumption), they are just smaller and can handle less throughput.

24.3 Understanding The Update Overwrite Scenario

If a MapStore is in use and the network to the database is available, one or both of the mini-clusters will write updates to the same database. There is a potential for the mini-clusters to overwrite the same cache entry records if modified in both mini-clusters. This overwrite scenario represents a potential data loss, and thus the database design should consider an insert and aggregate on read or version strategy rather than update records in place.

If the network to the database is not available, then based on the configured or coded consistency level or transaction, entry updates are held in cache or updates are rejected (fully synchronous and consistent). When held in cache, the updates will be considered dirty and will be written to the database when it becomes available. You can view the dirty entry counts per cluster member in the Management Center web console (please see the Map Monitoring section).
24.4 What Happens When The Network Failure Is Fixed

Since it is a network failure, there is no way to programmatically avoid your application running as two separate independent clusters. But what will happen after the network failure is fixed and connectivity is restored between these two clusters? Will these two clusters merge into one again? If they do, how are the data conflicts resolved, because you might end up having two different values for the same key in the same map?

When the network is restored, all 271 partitions should exist in both mini-clusters and they should all undergo the merge. Once all primaries are merged, all backups are rewritten so their versions are correct. You may want to write a merge policy using the `MapMergePolicy` interface that rebuilds the entry from the database rather than from memory.

The only metadata available for merge decisions are from the `EntryView` interface that includes object size (cost), hits count, last updated/stored dates, and a version number that starts at zero and is incremented for each entry update. You could also create your own versioning scheme or capture a time series of deltas to reconstruct an entry.

24.5 How Hazelcast Split Brain Merge Happens

Here is, step by step, how Hazelcast split brain merge happens:

1. The oldest member of the cluster checks if there is another cluster with the same `group-name` and `group-password` in the network.
2. If the oldest member finds such a cluster, then it figures out which cluster should merge to the other.
3. Each member of the merging cluster will do the following.
   - Pause.
   - Take locally owned map entries.
   - Close all of its network connections (detach from its cluster).
   - Join to the new cluster.
   - Send merge request for each of its locally owned map entry.
   - Resume.

Each member of the merging cluster rejoins the new cluster and sends a merge request for each of its locally owned map entries. Two important points:

- The smaller cluster will merge into the bigger one. If they have equal number of members then a hashing algorithm determines the merging cluster.
- Each cluster may have different versions of the same key in the same map. The destination cluster will decide how to handle merging entry based on the `MergePolicy` set for that map. There are built-in merge policies such as `PassThroughMergePolicy`, `PutIfAbsentMapMergePolicy`, `HigherHitsMapMergePolicy` and `LatestUpdateMapMergePolicy`. You can develop your own merge policy by implementing `com.hazelcast.map.merge.MapMergePolicy`. You should set the full class name of your implementation to the `merge-policy` configuration.

```java
public interface MergePolicy {
    /**
     * Returns the value of the entry after the merge
     * of entries with the same key. Returning value can be
     * You should consider the case where existingEntry is null.
     * @param mapName name of the map
     * @param mergingEntry entry merging into the destination cluster
     * @param existingEntry existing entry in the destination cluster
     * @return final value of the entry. If returns null then entry will be removed.
     */
    Object merge( String mapName, EntryView mergingEntry, EntryView existingEntry );
}
```
24.6 Specifying Merge Policies

Here is how merge policies are specified per map:

```xml
<hazelcast>
  ...
  <map name="default">
    <backup-count>1</backup-count>
    <eviction-policy>NONE</eviction-policy>
    <max-size>0</max-size>
    <eviction-percentage>25</eviction-percentage>
    
    <!-- While recovering from split-brain (network partitioning), map entries in the small cluster will merge into the bigger cluster based on the policy set here. When an entry merge into the cluster, there might an existing entry with the same key already. Values of these entries might be different for that same key. Which value should be set for the key? Conflict is resolved by the policy set here. Default policy is hz.ADD_NEW_ENTRY

    There are built-in merge policies such as
    com.hazelcast.map.merge.PassThroughMergePolicy; entry will be added if there is no existing entry for the key.
    com.hazelcast.map.merge.PutIfAbsentMapMergePolicy; entry will be added if the merging entry doesn't exist in the cluster.
    com.hazelcast.map.merge.HigherHitsMapMergePolicy; entry with the higher hits wins.
    com.hazelcast.map.merge.LatestUpdateMapMergePolicy; entry with the latest update wins.
    -->

    <merge-policy>MY_MERGE_POLICY_CLASS</merge-policy>
  </map>
  ...
</hazelcast>
```

**NOTE:** Map is the only Hazelcast distributed data structure that merges after a split brain syndrome. For the other data structures (e.g. Queue, Topic, IdGenerator, etc.), one instance of that data structure is chosen after split brain syndrome.
Chapter 25

License Questions

Hazelcast is distributed using the Apache License 2, therefore permissions are granted to use, reproduce and distribute it along with any kind of open source and closed source applications.

Hazelcast Enterprise is a commercial product of Hazelcast, Inc. and is distributed under a commercial license that must be acquired before using it in any type of released software. Feel free to contact Hazelcast sales department for more information on commercial offers.

Depending on the used feature-set, Hazelcast has certain runtime dependencies which might have different licenses. Following are dependencies and their respective licenses.

25.1 Embedded Dependencies

Embedded dependencies are merged (shaded) with the Hazelcast codebase at compile-time. These dependencies become an integral part of the Hazelcast distribution.

For license files of embedded dependencies, please see the license directory of the Hazelcast distribution, available at our download page.

minimal-json:

minimal-json is a JSON parsing and generation library which is a part of the Hazelcast distribution. It is used for communication between the Hazelcast cluster and the Management Center.

minimal-json is distributed under the MIT license and offers the same rights to add, use, modify, and distribute the source code as the Apache License 2.0 that Hazelcast uses. However, some other restrictions might apply.

25.2 Runtime Dependencies

Depending on the used features, additional dependencies might be added to the dependency set. Those runtime dependencies might have other licenses. See the following list of additional runtime dependencies.

Spring Framework:

Hazelcast offers a tight integration into the Spring Framework. Hazelcast can be configured and controlled using Spring.

The Spring Framework is distributed under the terms of the Apache License 2 and therefore it is fully compatible with Hazelcast.

Hibernate:

Hazelcast integrates itself into Hibernate as a second-level cache provider.

Hibernate is distributed under the terms of the Lesser General Public License 2.1, also known as LGPL. Please read carefully the terms of the LGPL since restrictions might apply.
Apache Tomcat:
Hazelcast Enterprise offers native integration into Apache Tomcat for web session clustering.
Apache Tomcat is distributed under the terms of the Apache License 2 and therefore fully compatible with Hazelcast.

Eclipse Jetty:
Hazelcast Enterprise offers native integration into Jetty for web session clustering.
Jetty is distributed with a dual licensing strategy. It is licensed under the terms of the Apache License 2 and under the Eclipse Public License v1.0, also known as EPL. Due to the Apache License, it is fully compatible with Hazelcast.

JCache API (JSR 107):
Hazelcast offers a native implementation for JCache (JSR 107), which has a runtime dependency to the JCache API.
The JCache API is distributed under the terms of the so called Specification License. Please read carefully the terms of this license since restrictions might apply.

Boost C++ Libraries:
Hazelcast Enterprise offers a native C++ client, which has a link-time dependency to the Boost C++ Libraries.
The Boost Libraries are distributed under the terms of the Boost Software License, which is very similar to the MIT or BSD license. Please read carefully the terms of this license since restrictions might apply.
Chapter 26

Common Exception Types

You may see the following exceptions in any Hazelcast operation when the following situations occur:

- **HazelcastInstanceNotActiveException**: Thrown when `HazelcastInstance` is not active (already shutdown or being shutdown) during an invocation.

- **HazelcastOverloadException**: Thrown when the system will not handle more load due to an overload. This exception is thrown when back pressure is enabled.

- **DistributedObjectDestroyedException**: Thrown when an already destroyed `DistributedObject` (IMap, IQueue, etc.) is accessed and when a method call is done over a destroyed object.

- **MemberLeftException**: Thrown when a member leaves during an invocation or execution.

Hazelcast also throws the following exceptions in the cases of overall system problems such as networking issues and long pauses:

- **PartitionMigratingException**: Thrown when an operation is executed on a partition, but that partition is currently being moved around.

- **TargetNotMemberException**: Thrown when an operation is sent to a machine that is not a member of the cluster.

- **CallerNotMemberException**: Thrown when an operation was sent by a machine which is not a member in the cluster when the operation is executed.

- **WrongTargetException**: Thrown when an operation is executed on the wrong machine, mostly because the partition that operation belongs to is already moved to some other member.
Chapter 27

Frequently Asked Questions

27.1 Why 271 as the default partition count?

The partition count of 271, being a prime number, is a good choice because it will be distributed to the nodes almost evenly. For a small to medium sized cluster, the count of 271 gives an almost even partition distribution and optimal-sized partitions. As your cluster becomes bigger, you should make this count bigger to have evenly distributed partitions.

27.2 Is Hazelcast thread safe?

Yes. All Hazelcast data structures are thread safe.

How do nodes discover each other?

When a node is started in a cluster, it will dynamically and automatically be discovered. There are three types of discovery.

- Multicast discovery: nodes in a cluster discover each other by multicast, by default.
- Discovery by TCP/IP: the first node created in the cluster (leader) will form a list of IP addresses of other joining nodes and will send this list to these nodes so the nodes will know each other.
- If your application is placed on Amazon EC2, Hazelcast has an automatic discovery mechanism. You will give your Amazon credentials and the joining node will be discovered automatically.

Once nodes are discovered, all the communication between them will be via TCP/IP. RELATED INFORMATION

Please refer to the Hazelcast Cluster Discovery section for detailed information.

27.3 What happens when a node goes down?

Once a node is gone (crashes), the following happens since data in each node has a backup in other nodes.

- First, the backups in other nodes are restored.
- Then, data from these restored backups are recovered.
- And finally, backups for these recovered data are formed.

So eventually, no data is lost.
27.4 How do I test the connectivity?

If you notice that there is a problem with a node joining a cluster, you may want to perform a connectivity test between the node to be joined and a node from the cluster. You can use the `iperf` tool for this purpose. For example, you can execute the below command on one node (i.e. listening on port 5701).

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

And you can execute the below command on the other node.

```
iperf -c <IP address> -d -p 5701
```

The output should include connection information, such as the IP addresses, transfer speed, and bandwidth. Otherwise, if the output says `No route to host`, it means a network connection problem exists.

27.5 How do I choose keys properly?

When you store a key and value in a distributed Map, Hazelcast serializes the key and value, and stores the byte array version of them in local ConcurrentHashMaps. These ConcurrentHashMaps use `equals` and `hashCode` methods of byte array version of your key. It does not take into account the actual `equals` and `hashCode` implementations of your objects. So it is important that you choose your keys in a proper way.

Implementing `equals` and `hashCode` is not enough, it is also important that the object is always serialized into the same byte array. All primitive types like String, Long, Integer, etc. are good candidates for keys to be used in Hazelcast. An unsorted Set is an example of a very bad candidate because Java Serialization may serialize the same unsorted set in two different byte arrays.

27.6 How do I reflect value modifications?

Hazelcast always return a clone copy of a value. Modifying the returned value does not change the actual value in the map (or multimap, list, set). You should put the modified value back to make changes visible to all nodes.

```java
V value = map.get( key );
value.updateSomeProperty();
map.put( key, value );
```

Collections which return values of methods (such as `IMap.keySet`, `IMap.values`, `IMap.entrySet`, `MultiMap.get`, `MultiMap.remove`, `IMap.keySet`, `IMap.values`) contain cloned values. These collections are NOT backed up by related Hazelcast objects. Therefore, changes to them are NOT reflected in the originals, and vice-versa.

27.7 How do I test my Hazelcast cluster?

Hazelcast allows you to create more than one instance on the same JVM. Each member is called `HazelcastInstance` and each will have its own configuration, socket and threads, so you can treat them as totally separate instances.

This enables you to write and to run cluster unit tests on a single JVM. Because you can use this feature for creating separate members different applications running on the same JVM (imagine running multiple web applications on the same JVM), you can also use this feature for testing your Hazelcast cluster.

Let’s say you want to test if two members have the same size of a map.

```java
@Test
public void testTwoMemberMapSizes() {
    // start the first member
    HazelcastInstance h1 = Hazelcast.newHazelcastInstance();
    // get the map and put 1000 entries
```
Map map1 = h1.getMap( "testmap" );
for ( int i = 0; i < 1000; i++ ) {
    map1.put( i, "value" + i );
}
// check the map size
assertEquals( 1000, map1.size() );
// start the second member
HazelcastInstance h2 = Hazelcast.newHazelcastInstance();
// get the same map from the second member
Map map2 = h2.getMap( "testmap" );
// check the size of map2
assertEquals( 1000, map2.size() );
// check the size of map1 again
assertEquals( 1000, map1.size() );
}

In the test above, everything happens in the same thread. When developing a multi-threaded test, you need to carefully handle coordination of the thread executions. It is highly recommended that you use CountDownLatch for thread coordination (you can certainly use other ways). Here is an example where we need to listen for messages and make sure that we got these messages.

@Test
public void testTopic() {
    // start two member cluster
    HazelcastInstance h1 = Hazelcast.newHazelcastInstance();
    HazelcastInstance h2 = Hazelcast.newHazelcastInstance();
    String topicName = "TestMessages";
    // get a topic from the first member and add a messageListener
    ITopic<String> topic1 = h1.getTopic( topicName );
    final CountDownLatch latch1 = new CountDownLatch( 1 );
    topic1.addMessageListener( new MessageListener() {
        public void onMessage( Object msg ) {
            assertEquals( "Test1", msg );
            latch1.countDown();
        }
    });
    // get a topic from the second member and add a messageListener
    ITopic<String> topic2 = h2.getTopic(topicName);
    final CountDownLatch latch2 = new CountDownLatch( 2 );
    topic2.addMessageListener( new MessageListener() {
        public void onMessage( Object msg ) {
            assertEquals( "Test1", msg );
            latch2.countDown();
        }
    });
    // publish the first message, both should receive this
    topic1.publish( "Test1" );
    // shutdown the first member
    h1.shutdown();
    // publish the second message, second member's topic should receive this
    topic2.publish( "Test1" );
    try {
        // assert that the first member's topic got the message
        assertTrue( latch1.await( 5, TimeUnit.SECONDS ) );
        // assert that the second members' topic got two messages
        assertTrue( latch2.await( 5, TimeUnit.SECONDS ) );
    } catch ( InterruptedException ignored ) {
}
You can start Hazelcast members with different configurations. Remember to call `Hazelcast.shutdownAll()` after each test case to make sure that there is no other running member left from the previous tests.

```java
@After
public void cleanup() throws Exception {
    Hazelcast.shutdownAll();
}
```

For more information please check our existing tests.

### 27.8 Does Hazelcast support hundreds of nodes?

Yes. Hazelcast performed a successful test on Amazon EC2 with 200 nodes.

### 27.9 Does Hazelcast support thousands of clients?

Yes. However, there are some points you should consider. The environment should be LAN with a high stability and the network speed should be 10 Gbps or higher. If the number of nodes is high, the client type should be selected as Dummy, not Smart Client. In the case of Smart Clients, since each client will open a connection to the nodes, these nodes should be powerful enough (for example, more cores) to handle hundreds or thousands of connections and client requests. Also, you should consider using near caches in clients to lower the network traffic. And you should use the Hazelcast releases with the NIO implementation (which starts with Hazelcast 3.2). Also, you should configure the clients attentively. Please refer to the Java Client section section for configuration notes.

### 27.10 What is the difference between old LiteMember and new Smart Client?

LiteMember supports task execution (distributed executor service), smart client does not. Also, LiteMember is highly coupled with cluster, smart client is not.

### 27.11 How do you give support?

We have two support services: community and commercial support. Community support is provided through our Mail Group and StackOverflow web site. For information on support subscriptions, please see Hazelcast.com.

### 27.12 Does Hazelcast persist?

No. However, Hazelcast provides `MapStore` and `MapLoader` interfaces. For example, when you implement the `MapStore` interface, Hazelcast calls your store and load methods whenever needed.

### 27.13 Can I use Hazelcast in a single server?

Yes. But please note that Hazelcast’s main design focus is multi-node clusters to be used as a distribution platform.
27.14 How can I monitor Hazelcast?

Hazelcast Management Center is what you use to monitor and manage the nodes running Hazelcast. In addition to monitoring the overall state of a cluster, you can analyze and browse data structures in detail, you can update map configurations, and you can take thread dumps from nodes.

Moreover, JMX monitoring is also provided. Please see the Monitoring with JMX section for details.

27.15 How can I see debug level logs?

By changing the log level to “Debug”. Below are sample lines for log4j logging framework. Please see the Logging Configuration section to learn how to set logging types.

First, set the logging type as follows.

```java
String location = "log4j.configuration";
String logging = "hazelcast.logging.type";
System.setProperty( logging, "log4j" );
/**if you want to give a new location. **/
System.setProperty( location, "file:/path/mylog4j.properties" );
```

Then set the log level to “Debug” in the properties file. Below is example content.

```
# direct log messages to stdout #
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p [%c{1}] - %m%n
log4j.logger.com.hazelcast=debug
#log4j.logger.com.hazelcast.cluster=debug
#log4j.logger.com.hazelcast.partition=debug
#log4j.logger.com.hazelcast.partition.InternalPartitionService=debug
#log4j.logger.com.hazelcast.nio=debug
#log4j.logger.com.hazelcast.hibernate=debug
```

The line `log4j.logger.com.hazelcast=debug` is used to see debug logs for all Hazelcast operations. Below this line, you can select to see specific logs (cluster, partition, hibernate, etc.).

27.16 What is the difference between client-server and embedded topologies?

In the embedded topology, nodes include both the data and application. This type of topology is the most useful if your application focuses on high performance computing and many task executions. Since application is close to data, this topology supports data locality.

In the client-server topology, you create a cluster of nodes and scale the cluster independently. Your applications are hosted on the clients, and the clients communicate with the nodes in the cluster to reach data.

Client-server topology fits better if there are multiple applications sharing the same data or if application deployment is significantly greater than the cluster size (for example, 500 application servers vs. 10 node cluster).
27.17 How do I know it is safe to kill the second node?

Programmatically:

```java
PartitionService partitionService = hazelcastInstance.getPartitionService().isClusterSafe()
if (partitionService().isClusterSafe()) {
    hazelcastInstance.shutdown(); // or terminate
}
```

Or declaratively:

```java
PartitionService partitionService = hazelcastInstance.getPartitionService().isClusterSafe()
if (partitionService().isLocalMemberSafe()) {
    hazelcastInstance.shutdown(); // or terminate
}
```

**RELATED INFORMATION**

Please refer to the Cluster-Member Safety Check section for more information.

27.18 When do I need Native Memory solutions?

Native Memory solutions can be preferred:

- when the amount of data per node is large enough to create significant garbage collection pauses.
- when your application requires predictable latency.

27.19 Is there any disadvantage of using near-cache?

The only disadvantage when using Near Cache is that it may cause stale reads.

27.20 Is Hazelcast secure?

Hazelcast supports symmetric encryption, secure sockets layer (SSL), and Java Authentication and Authorization Service (JAAS). Please see the Security chapter for more information.

27.21 How can I set socket options?

Hazelcast allows you to set some socket options such as `SO_KEEPALIVE`, `SO_SNDBUF`, and `SO_RCVBUF` using Hazelcast configuration properties. Please see `hazelcast.socket.*` properties explained in the System Properties section.

27.22 I periodically see client disconnections during idle time?

In Hazelcast, socket connections are created with the `SO_KEEPALIVE` option enabled by default. In most operating systems, default keep-alive time is 2 hours. If you have a firewall between clients and servers which is configured to reset idle connections/sessions, make sure that the firewall’s idle timeout is greater than the TCP keep-alive defined in the OS.

For additional information please see:

- Using TCP keepalive under Linux
- Microsoft TechNet
**27.23 How to get rid of “java.lang.OutOfMemoryError: unable to create new native thread”?**

If you encounter an error of `java.lang.OutOfMemoryError: unable to create new native thread`, it may be caused by exceeding the available file descriptors on your operating system, especially if it is Linux. This exception is usually thrown on a running node, after a period of time when the thread count exhausts the file descriptor availability.

The JVM on Linux consumes a file descriptor for each thread created. The default number of file descriptors available in Linux is usually 1024. If you have many JVMs running on a single machine, it is possible to exceed this default number.

You can view the limit using the following command.

```
# ulimit -a
```

At the operating system level, Linux users can control the amount of resources (and in particular, file descriptors) used via one of the following options.

1. Editing the `limits.conf` file:

```
# vi /etc/security/limits.conf
```

```
testuser soft nofile 4096

testuser hard nofile 10240
```

2. Or using the `ulimit` command:

```
# ulimit -Hn
```

```
10240
```

The default number of process per users is 1024. Adding the following to your `$HOME/.profile` could solve the issue:

```
# ulimit -u 4096
```

**27.24 Does repartitioning wait for Entry Processor?**

Repartitioning is the process of redistributing the partition ownerships. Hazelcast performs the repartitioning in the cases where a node leaves the cluster or joins the cluster. If a repartitioning will happen while an entry processor is active in a node processing on an entry object, the repartitioning waits for the entry processor to complete its job.

**27.25 Why do Hazelcast instances on different machines not see each other?**

Assume you have two instances on two different machines and you develop a configuration as shown below.

```java
Config config = new Config();
NetworkConfig network = config.getNetworkConfig();

JoinConfig join = network.getJoin();
join.getMulticastConfig().setEnabled(false);
join.getTcpIpConfig().addMember("IP1")
    .addMember("IP2") .setEnabled(true);
network.getInterfaces() .setEnabled(true)
    .addInterface("IP1") .addInterface("IP2");
```
When you create the Hazelcast instance, you have to pass the configuration to the instance. If you create the instances without passing the configuration, each instance starts but cannot see each other. Therefore, a correct way to create the instance is the following:

```java
HazelcastInstance instance = Hazelcast.newHazelcastInstance(config);
```

The following is an incorrect way:

```java
HazelcastInstance instance = Hazelcast.newHazelcastInstance();
```

### 27.26 What Does “Replica: 1 has no owner” Mean?

When you start more nodes after the first one is started, you will see `replica: 1 has no owner` entry in the newly started node’s log. There is no need to worry about it since it refers to a transitory state. It only means the replica partition is not ready/assigned yet and eventually it will be.
## Chapter 28

## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-phase Commit</td>
<td>2-phase commit protocol is an atomic commitment protocol for distributed systems. It consists of two phases: preparation and commit. It is used to coordinate the activities of two or more transactions.</td>
</tr>
<tr>
<td>ACID</td>
<td>A set of properties (Atomicity, Consistency, Isolation, Durability) guaranteeing that transactions are processed correctly. ACID is also known as ACID properties.</td>
</tr>
<tr>
<td>Cache</td>
<td>A high-speed access area that can be either a reserved section of main memory or storage device.</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>Garbage collection is the recovery of storage that is being used by an application when that application is no longer using the storage.</td>
</tr>
<tr>
<td>Hazelcast Cluster</td>
<td>It is a virtual environment formed by Hazelcast nodes communicating with each other.</td>
</tr>
<tr>
<td>Hazelcast Partitions</td>
<td>Memory segments containing the data. Hazelcast is built-on the partition concept, it uses partitions to store and process data.</td>
</tr>
<tr>
<td>IMDG</td>
<td>An in-memory data grid (IMDG) is a data structure that resides entirely in memory, and is distributed. IMDG is used for high performance and low latency.</td>
</tr>
<tr>
<td>Invalidation</td>
<td>The process of marking an object as being invalid across the distributed cache.</td>
</tr>
<tr>
<td>Java heap</td>
<td>Java heap is the space that Java can reserve and use in memory for dynamic memory allocation.</td>
</tr>
<tr>
<td>LRU, LFU</td>
<td>LRU and LFU are two of eviction algorithms. LRU is the abbreviation for Least Recently Used. It is used to determine which entry to remove from the cache.</td>
</tr>
<tr>
<td>+Member</td>
<td>A Hazelcast instance. Depending on your Hazelcast usage, it can refer to a server or a Java virtual machine.</td>
</tr>
<tr>
<td>Multicast</td>
<td>It is a type of communication where data is addressed to a group of destination nodes simultaneously.</td>
</tr>
<tr>
<td>Near Cache</td>
<td>It is a caching model. When it is enabled, an object retrieved from a remote node is put into the local cache.</td>
</tr>
<tr>
<td>NoSQL</td>
<td>“Not Only SQL”. It is a database model that provides a mechanism for storage and retrieval of data. It is also known as the NoSQL database model.</td>
</tr>
<tr>
<td>Race Condition</td>
<td>This condition occurs when two or more threads can access shared data and they try to change it at the same time.</td>
</tr>
<tr>
<td>RSA</td>
<td>An algorithm developed by Rivest, Shamir and Adleman to generate, encrypt and decrypt keys for secure data transmissions.</td>
</tr>
<tr>
<td>Serialization</td>
<td>Process of converting an object into a stream of bytes in order to store the object or transmit it to another system.</td>
</tr>
<tr>
<td>Split Brain</td>
<td>Split brain syndrome, in a clustering context, is a state in which a cluster of nodes gets divided (or partitioned)</td>
</tr>
<tr>
<td>Transaction</td>
<td>Means a sequence of information exchange and related work (such as data store updating) that is transactional. It ensures data integrity.</td>
</tr>
</tbody>
</table>