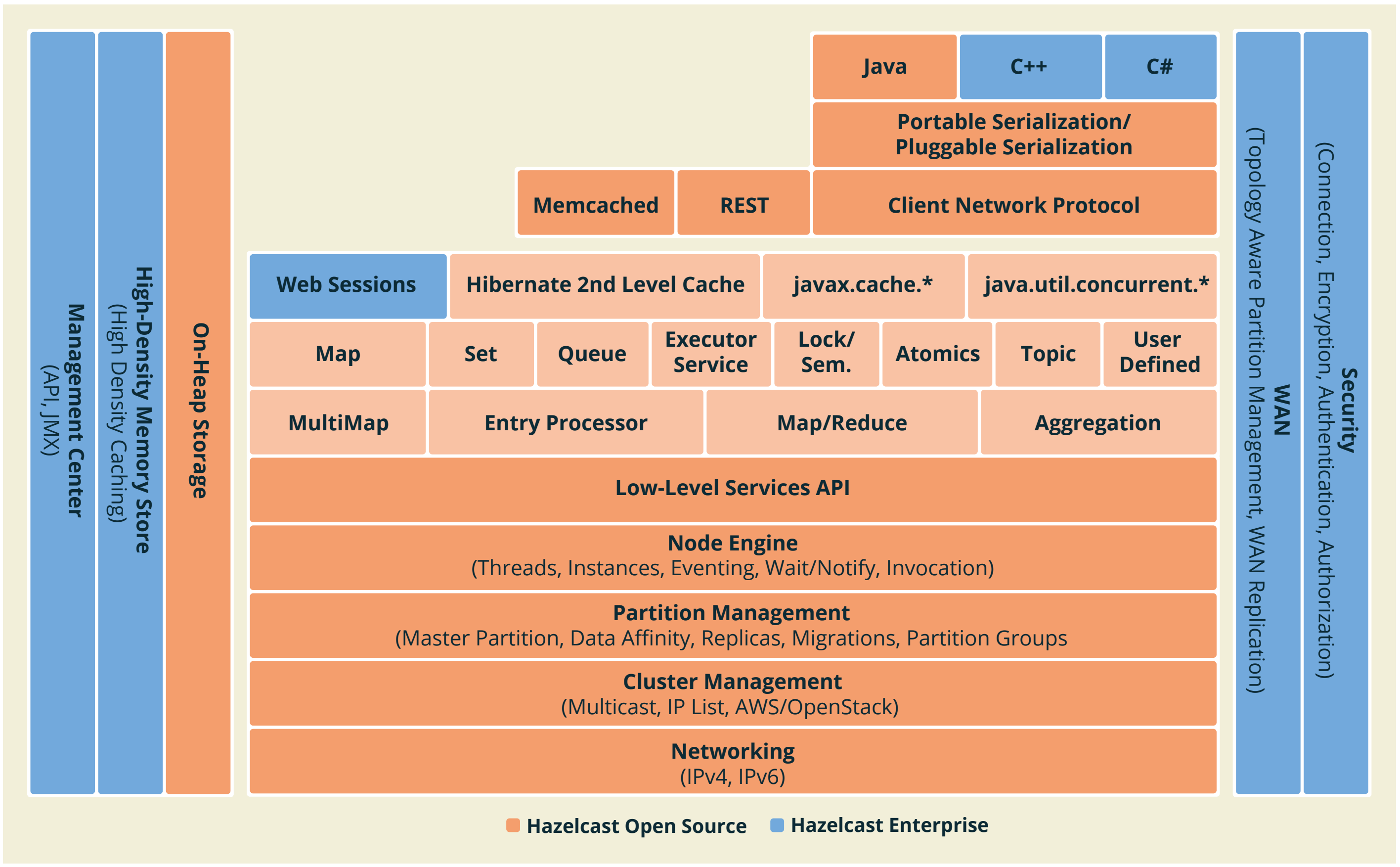
Hazelcast Deployment and Operations Guide

For Hazelcast version 3.4

TABLE OF CONTENTS

Introduction

Hazelcast provides a convenient and familiar interface for developers to work with distributed data structures and other aspects of in-memory computing. For example, in its simplest configuration, Hazelcast can be treated as an implementation of the familiar ConcurrentHashMap that can be accessed from multiple JVMs (Java Virtual Machine), including JVMs that are spread out across the network. However, the Hazelcast architecture has sufficient flexibility and advanced features that it can be used in a large number of different architectural patterns and styles. The following schematic represents the basic architecture of Hazelcast.



However, it is not necessary to deal with the overall sophistication present in the architecture in order to work effectively with Hazelcast, and many users are happy integrating purely at the level of the `java.util.concurrent` or `javax.cache` APIs.

The core Hazelcast technology:

- Is open source
- Is written in Java
- Supports Java 6, 7 and 8 SE
- Uses minimal dependencies
- Has simplicity as a key concept

The primary capabilities that Hazelcast provides include:

- Elasticity
- Redundancy
- High performance

Elasticity means that Hazelcast clusters can grow capacity simply by adding new nodes. Redundancy means that you have great flexibility when you configure Hazelcast clusters for data replication policy (which defaults to one synchronous backup copy). To support these capabilities, Hazelcast has a concept of members - members are JVMs that join a Hazelcast cluster. A cluster provides a single extended environment where data can be synchronized between (and processed by) members of the cluster.

PURPOSE OF THIS DOCUMENT

If you are a Hazelcast user planning to go into production with a Hazelcast-backed application or you are curious about the practical aspects of deploying and running such an application, this guide will provide an introduction to the most important aspects of deploying and operating a successful Hazelcast installation.

In addition to this guide, there is a host of useful resources available online including the product documentation, Hazelcast forums, books, webinars and blog posts. Where applicable, each section of this document provides links to further reading if you would like to delve more deeply into a particular topic.

Hazelcast also offers support, training and consulting to help you get the most out of the product and to ensure successful deployment and operation. Visit hazelcast.com/services for more information.

HAZELCAST VERSIONS

This document is current to Hazelcast version 3.4 with some coverage of upcoming features in version 3.5. It is not explicitly backward-compatible to earlier versions, but may still substantially apply.

Network Architecture and Configuration

TOPOLOGIES

Hazelcast supports two modes of operation, either “embedded member”, where the JVM containing application code joins the Hazelcast cluster directly, or “client plus member”, whereby a secondary JVM (which may be on the same host, or a different one) is responsible for joining the Hazelcast cluster. These two approaches to topology are shown in the following diagrams.

Here is the embedded approach:

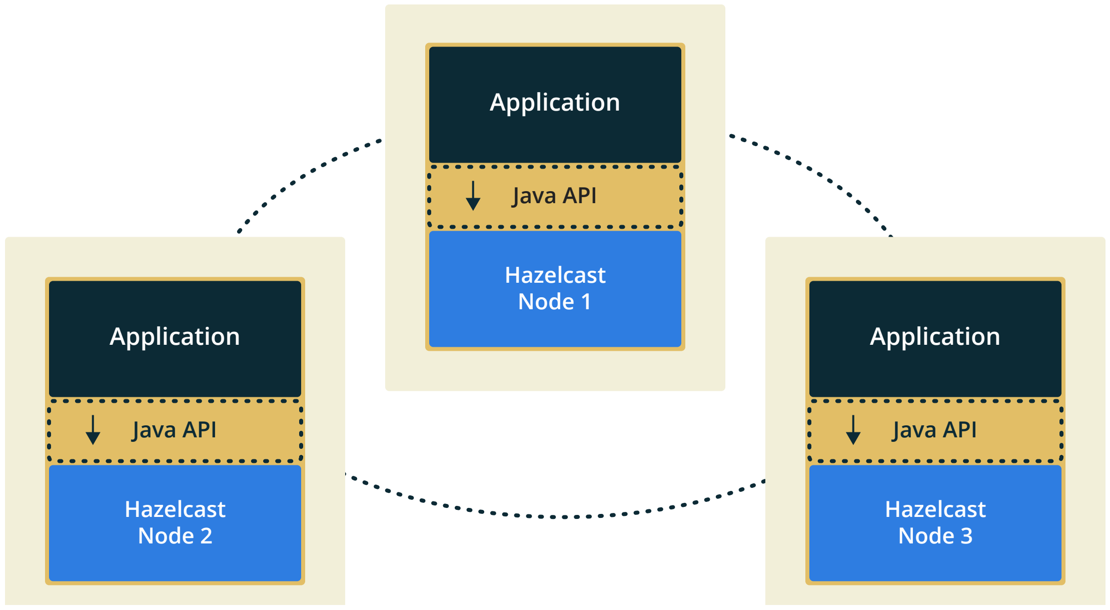


Figure 1: Hazelcast embedded topology

Here is the client plus member topology:

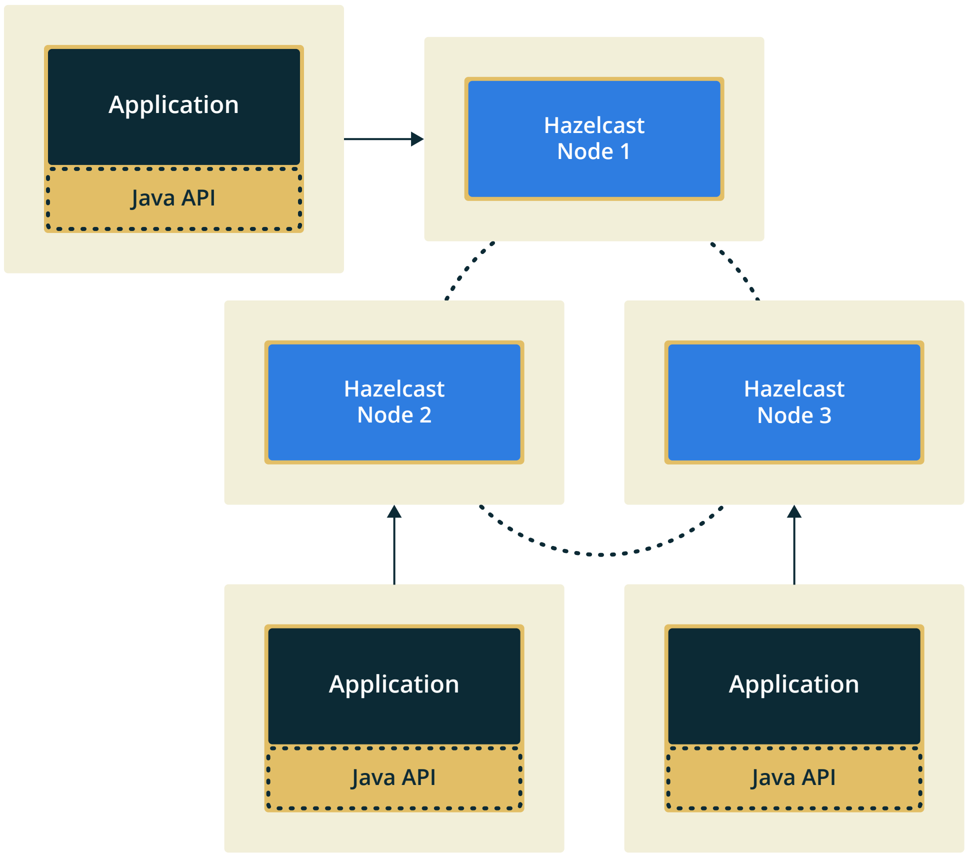


Figure 2: Hazelcast client-server topology

Under most circumstances, we recommend client plus member topologies, as it provides greater flexibility in terms of cluster mechanics - member JVMs can be taken down and restarted without any impact to the overall application, as the Hazelcast client will simply reconnect to another member of the cluster. Another way of saying this is that client plus member topologies isolate application code from purely cluster-level events.

Hazelcast allows clients to be configured within the client code (programmatically), or by XML, or by properties files. Configuration uses properties files (handled by the class `com.hazelcast.client.config.ClientConfigBuilder`) and XML (via `com.hazelcast.client.config.XmlClientConfigBuilder`). Clients have quite a few configurable parameters, including known members of the cluster. Hazelcast will discover the other members as soon as they are online, but they need to connect first. In turn, this requires the user to configure enough addresses to ensure that the client can connect into the cluster somewhere.

In production applications, the Hazelcast client should be reused between threads and operations. It is designed for multithreaded operation, and creation of a new Hazelcast client is relatively expensive, as it handles cluster events, heartbeating, etc., so as to be transparent to the user.

ADVANTAGES OF EMBEDDED ARCHITECTURE

The main advantage of using the embedded architecture is its simplicity. Because the Hazelcast services run in the same JVMs as the application, there are no extra servers to deploy, manage or maintain. This simplicity applies especially when the Hazelcast cluster is tied directly to the embedded application.

ADVANTAGES OF CLIENT-SERVER ARCHITECTURE

For most use cases, however, there are significant advantages to using the client-server architecture. Broadly, they are as follows:

1. Cluster member lifecycle is independent of application lifecycle
2. Resource isolation
3. Problem isolation
4. Shared Infrastructure
5. Better scalability

Cluster Member Node Lifecycle Independent of Application Lifecycle

The practical lifecycle of Hazelcast member nodes is usually different from any particular application instance. When Hazelcast is embedded in an application instance, the embedded Hazelcast node must necessarily be started and shut down in concert with its co-resident application instance and vice versa. This is often not ideal and may lead to increased operational complexity. When Hazelcast nodes are deployed as separate server instances, they and their client application instances may be started and shut down independently of one other.

Resource Isolation

When Hazelcast is deployed as a member on its own dedicated host, it does not compete with the application for CPU, memory and I/O resources. This makes Hazelcast performance more predictable and reliable.

Easier Problem Isolation

Because Hazelcast member activity is isolated on its own server, it's easier to identify the cause of any pathological behavior. For example, if there is a memory leak in the application causing heap usage to grow without bounds, the memory activity of the application is not obscured by the co-resident memory activity of Hazelcast services. The same holds true for CPU and I/O issues. When application activity is isolated from Hazelcast services, symptoms are automatically isolated and easier to recognize.

Shared Infrastructure

The client server architecture is appropriate when using Hazelcast as a shared infrastructure such that multiple applications, especially those under the control of different work groups, use the same cluster or clusters.

Better Scalability

The client-server architecture has a better, more flexible scalability profile. When you need to scale, simply add more Hazelcast servers. With the client-server deployment model, the separate client and server scalability concerns may be addressed separately.

Achieve Very Low Latency with Client-Server

If you need very low latency data access, but you also want the scalability advantages of the client-server deployment model, consider configuring the clients to use Near Cache. This will ensure that frequently used data is kept in local memory on the application JVM.

Further Reading:

- Online Documentation, Near Cache:
  http://docs.hazelcast.org/docs/latest/manual/html/map-nearcache.html

CLUSTER DISCOVERY PROTOCOLS

Hazelcast supports three options for cluster creation and discovery when nodes start up:

- Multicast
- TCP
- Amazon EC2 auto-discovery, when running on Amazon Web Services

Once a node has joined a cluster, all further network communication is performed via TCP.

Multicast

The advantage of multicast discovery is its simplicity and flexibility. As long as Hazelcast's local network supports multicast, the cluster members do not need to know each other's specific IP addresses when they start up; they just multicast to all other members on the subnet. This is especially useful during development and testing. In production environments, if you want to avoid accidentally joining the wrong cluster, then use Group Configuration.

Further Reading:

- Online Documentation, Group Configuration: http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#group-configuration

TCP

When using TCP for cluster discovery, the specific IP address of at least one other cluster member must be specified in configuration. Once a new node discovers another cluster member, the cluster will inform the new node of the full cluster topology, so the complete set of cluster members need not be specified in the configuration. However, we recommend that you specify the addresses of at least two other members in case one of those members is not available at startup time.

Amazon EC2 Auto Discovery

Hazelcast on Amazon EC2 supports TCP and EC2 auto-discovery, which is similar to multicast. It is useful when you do not want to or cannot provide the list of possible IP addresses. To configure your cluster to use EC2 Auto Discovery, disable cluster joining over multicast and TCP/IP, enable AWS, and provide your credentials (access and secret keys).

Further Reading:

For detailed information on cluster discovery and network configuration for Multicast, TCP and EC2, see the following documentation:

- Mastering Hazelcast, Network Configuration: http://hazelcast.org/mastering-hazelcast/chapter-11/
- Online Documentation, Hazelcast Cluster Discovery: http://docs.hazelcast.org/docs/3.4/manual/html-single/hazelcast-documentation.html#hazelcast-cluster-discovery

FIREWALLS, NAT, NETWORK INTERFACES AND PORTS

Hazelcast's default network configuration is designed to make cluster startup and discovery simple and flexible out of the box. It's also possible to tailor the network configuration to fit the specific requirements of your production network environment.

If your server hosts have multiple network interfaces, you may customize the specific network interfaces Hazelcast should use. You may also restrict which hosts are allowed to join a Hazelcast cluster by specifying a set of trusted IP addresses or ranges. If your firewall restricts outbound ports, you may configure Hazelcast to use specific outbound ports allowed by the firewall. Nodes behind network address translation (NAT) in, for example, a private cloud may be configured to use a public address.

Further Reading:

- Mastering Hazelcast, Network Configuration: http://hazelcast.org/mastering-hazelcast/chapter-11/
- Online Documentation, Network Configuration: http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#network-configuration
- Online Documentation, Network Interfaces: http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#interfaces
- Online Documentation, Outbound Ports: http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#outbound-ports

WAN REPLICATION (ENTERPRISE FEATURE)

If, for example, you have multiple data centers for geographic locality or disaster recovery and you need to synchronize data across the clusters, Hazelcast Enterprise supports wide-area network (WAN) replication. WAN replication operates in either active-passive mode where an active cluster backs up to a passive cluster, or active-active mode where each participating cluster replicates to all others.

You may configure Hazelcast to replicate all data or restrict replication to specific shared data structures. In certain cases, you may need to adjust the replication queue size. The default replication queue size is 100,000, but in high volume cases, a larger queue size may be required to accommodate all of the replication messages.

Further Reading:

- Online Documentation, WAN Replication: http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#wan

Lifecycle, Maintenance and Updates

When operating a Hazelcast installation over time, planning for certain lifecycle events will ensure high uptime and smooth operation. Before moving your Hazelcast application into production, you will want to have policies in place for handling various aspects of your installation such as:

- Changes in cluster and network configuration
- Startup and shutdown procedures
- Application, software and hardware updates

CONFIGURATION MANAGEMENT

Some Map configuration options may be updated after a cluster has started—for example, TTL and TTI via file or programmatic configuration or via the Management Center. Other configuration options can't be changed on a running cluster. Hazelcast will not accept nor communicate configuration of joining nodes that differs from the cluster configuration. The following are configurations that are to remain the same at all nodes in a cluster and may not be changed after cluster startup:

- Group name and password
- Application validation token
- Partition count
- Partition group
- Joiner

The use of a file change monitoring tool is recommended to ensure proper configuration across the cluster.

Further Reading:

- Online Documentation: http://docs.hazelcast.org/docs/latest/manual/html/configurationoverview.html
- Mastering Hazelcast eBook: http://hazelcast.org/mastering-hazelcast/chapter-2/#configuring-hazelcast

CLUSTER STARTUP

Hazelcast cluster startup is typically as simple as starting all of the nodes. Cluster formation and operation will happen automatically. However, in certain use cases, you may need to coordinate the startup of the cluster in a particular way. In a cache use case, for example, where shared data is loaded from an external source such as a database or web service, you may want to ensure the data is substantially loaded into the Hazelcast cluster before initiating normal operation of your application.

Data and Cache Warming

Data from an external source via a custom `MapLoader` implementation may be loaded either lazily or eagerly, depending on configuration. The Hazelcast instance will immediately return lazy-loaded maps from calls to `getMap()`. Alternately, the Hazelcast instance will block calls to `getMap()` until all of the data is loaded from the `MapLoader`.

Further Reading:

- Online Documentation: http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#mapstorefactory-and-maploaderlifecyclesupport-interfaces

TOPOLOGY CHANGES: NODES JOINING AND LEAVING

The oldest node in the cluster is responsible for managing a partition table that maps the ownership of Hazelcast's data partitions to the nodes in the cluster. When the topology of the cluster changes—either a node joins or leaves the cluster—the oldest node rebalances the partitions across the extant nodes to ensure equitable distribution of data, then initiates the process of moving partitions according to the new partition table. While a partition is in transit to its new node, only requests for data in that partition will block. When a node leaves the cluster, the nodes that hold the backups of the partitions held by the exiting node promote those backup partitions to be primary partitions and are immediately available for access. To avoid data loss, it is important to ensure that all the data in the cluster has been backed up again before taking down other nodes. To shut down a node gracefully, call the `HazelcastInstance.shutdown()` method which will block until there is no active data migration and at least one backup of that node's partitions are synced with the new primary ones. To ensure that the entire cluster is in a "safe" state (rather than just a single node), you may call `PartitionService.isClusterSafe()`. If `PartitionService.isClusterSafe()` returns true, it is safe to take down another node. You may also use the Management Center to determine if the cluster or a given node is in a safe state. See the Management Center section below.

Non-map data structures, e.g. `Lists`, `Sets`, `Queues`, etc., are backed up according to their backup count configuration, but their data is not distributed across multiple nodes. If a node with a non-map data structure leaves the cluster, its backup node will become the primary for that data structure and it will be backed up to another node. Because the partition map changes when nodes join and leave the cluster, be sure not to store object data to a local filesystem if you persist objects via `MapStore` and `MapLoader` interfaces. The partitions that a particular node is responsible for will almost certainly change over time, rendering locally persisted data inaccessible when the partition table changes.

Further Reading:

- Online Documentation, Data Partitioning: http://docs.hazelcast.org/docs/latest/manual/html/datapartitioning.html#data-partitioning
- Online Documentation, Partition Service: http://docs.hazelcast.org/docs/latest/manual/html/javaclient.html#partition-service
- Online Documentation, Safe Shutdown: http://docs.hazelcast.org/docs/latest/manual/html/faq.html#how-do-i-know-it-is-safe-to-kill-the-second-node

MAINTENANCE AND SOFTWARE UPDATES

When you need to perform software updates and hardware maintenance, you will most likely be able to do so without incurring downtime. Make sure that you have enough memory and CPU headroom in your cluster to allow for smooth operation when a cluster member node that you are updating is removed from service.

There are four flavors of update that each require different considerations vis-a-vis the state of a running cluster and its data:

1. **Hardware, operating system or JVM updates.** All of these may be updated live on a running cluster without scheduling a maintenance window. **Note:** Hazelcast supports Java versions 6-8. While not a best practice, JVMs of any supported Java version may be freely mixed and matched between the cluster and its clients and between individual members of a cluster.
2. **Live updates to user application code that executes only on the client side.** These updates may be performed against a live cluster with no downtime. Even if the new client-side user code defines new Hazelcast data structures, these are automatically created in the cluster. As other clients are upgraded they will be able to use these new structures. Changes to classes that define existing objects stored in Hazelcast are subject to some restrictions. Adding new fields to classes of existing objects is always allowed. However, removing fields or changing the type of a field will require special consideration. See the section on object schema changes below.

3. **Live updates to user application code that executes on cluster members and on cluster clients.** Clients may be updated and restarted without any interruption to cluster operation.
4. **Updates to Hazelcast libraries.** Prior to Hazelcast 3.5, all members and clients of a running cluster must run the same major and minor version of Hazelcast. Patch-level upgrades are guaranteed to work with each other.

Live Updates to Cluster Member Nodes

In most cases, maintenance and updates may be performed on a live, running cluster without incurring downtime. However, when performing a live update, you must take certain precautions to ensure the continuous availability of the cluster and the safety of its data. When you remove a node from service, its data backups on other nodes becomes active and the cluster automatically creates new backups and rebalances data across the new cluster topology. Before stopping another member node, you must ensure that the cluster has been fully backed up and is once again in a safe, high-availability state.

The following steps will ensure cluster data safety and high availability when performing maintenance or software updates:

1. Remove one member node from service. You may either kill the JVM process, call `HazelcastInstance.shutdown()`, or use the Management Center. **Note:** when you stop a member, all locks and semaphore permits held by that member will be released.
2. Perform the required maintenance or updates on that node's host.
3. Restart the node. The cluster will once again automatically rebalance its data based on the new cluster topology.
4. Wait until the cluster is in a safe state before removing any more nodes from service. The cluster is in a safe state when all of its members are in a safe state. A member is in a safe state when all of its data has been backed up to other nodes according to the backup count. You may call `HazelcastInstance.getPartitionService().isClusterSafe()` to determine whether the entire cluster is in a safe state You may also call `HazelcastInstance.getPartitionService().isMemberSafe(Member member)` to determine whether a particular node is in a safe state. Likewise, the Management Center displays the current safety of the cluster on its dashboard.
5. Continue this process for all remaining member nodes.

Live Updates to Clients

A client is a process that is connected to a Hazelcast cluster with either Hazelcast's client library (Java, C++, C#, .Net), REST or Memcached interfaces. Restarting clients has no effect on the state of the cluster or its members, so they may be taken out of service for maintenance or updates at any time and in any order. However, any locks or semaphore permits acquired by a client instance will be automatically released. In order to stop a client JVM, you may kill the JVM process or call `HazelcastClient.shutdown()`.

Live Updates to User Application Code that Executes on Both Clients and Cluster Members

Live updates to user application code on cluster members nodes is supported where:

- Existing class definitions do not change (i.e., you are only adding new classes definitions, not changing existing ones)
- The same Hazelcast version is used on all members and clients

Examples of what is allowed are new `EntryProcessors`, `ExecutorService`, `Runnable`, `Callables`, Map/Reduce and `Predicates`. Because the same code must be present on both clients and members, you should ensure the code is installed on all of the cluster members before invoking that code from a client. As a result, all cluster members must be updated before any client is.

Procedure:

1. Remove one member node from service
2. Update the user libraries on the member node
3. Restart the member node
4. Wait until the cluster is in a safe state before removing an more nodes from service
5. Continue this process for all remaining member nodes
6. Update clients in any order

Object Schema Changes

When you release new versions of user code that uses Hazelcast data, take care to ensure that the object schema for that data in the new application code is compatible with the existing object data in Hazelcast—or, implement custom deserialization code to convert the old schema into the new schema. Hazelcast supports a number of different serialization methods, one of which—the Portable interface—directly supports the use of multiple versions of the same class in different class loaders. See below for more information on different serialization options.

If you are using object persistence via `MapStore` and `MapLoader` implementations, be sure to handle object schema changes there as well. Depending on the scope of object schema changes in user code updates, it may be advisable to schedule a maintenance window to perform those updates. This will avoid unexpected problems with deserialization errors associated with updating against a live cluster.

HAZELCAST SOFTWARE UPDATES

Prior to Hazelcast version 3.5, all members and clients must run the same major and minor version of Hazelcast. Different patch-level updates are guaranteed to work with each other. For example, Hazelcast version 3.4.0 will work with 3.4.1 and 3.4.2, allowing for live updates of those versions against a running cluster.

Starting with version 3.5, Hazelcast will support updating clients with different minor versions. For example, Hazelcast 3.5.x clients will work with Hazelcast version 3.6.x. However, major and minor version updates cluster members must be performed concurrently which will require scheduled a maintenance window to bring the cluster down. Only patch-level updates are supported on members of a running cluster.

Live Updates of Hazelcast Libraries on Clients

Where compatibility is guaranteed, the procedure for updating Hazelcast libraries on clients is as follows:

1. Take any number of clients out of service
2. Update the Hazelcast libraries on each client
3. Restart each client
4. Continue this process until all clients are updated

Updates to Hazelcast Libraries on Cluster Members

For patch-level Hazelcast updates, use the procedure for live updates on member nodes described above.

For major and minor-level Hazelcast version updates, use the following procedure:

1. Schedule a window for cluster maintenance
2. Start the maintenance window
3. Stop all cluster members
4. Update Hazelcast libraries on all cluster member hosts
5. Restart all cluster members
6. Return the cluster to service

Optimization

Aside from standard code optimization in your application, there are a few Hazelcast-specific optimization considerations to keep in mind when preparing for a new Hazelcast deployment.

DEDICATED, HOMOGENEOUS HARDWARE RESOURCES

The first, easiest and most effective optimization strategy for Hazelcast is to ensure that Hazelcast services are allocated their own dedicated machine resources. Using dedicated, properly sized hardware (or virtual hardware) ensures that Hazelcast nodes have ample CPU, memory and network resources without competing with other processes or services.

Hazelcast distributes load evenly across all of its nodes and assumes that the resources available to each of its nodes is homogeneous. In a cluster with a mix of more powerful and less powerful machines, the weaker nodes will cause bottlenecks, leaving the stronger nodes under-utilized. For predictable performance, it's best to use equivalent hardware for all Hazelcast nodes.

PARTITION COUNT

Hazelcast's default partition count is 271, which is a good choice for clusters of up to 50 nodes and ~25 – 30 GB of data. Up to this threshold, partitions are small enough that any rebalancing of the partition map when nodes join or leave the cluster doesn't disturb smooth operation of the cluster. However, with larger clusters and bigger data sets, a larger partition count helps to rebalance data more efficiently across nodes.

An optimum partition size is anything between 50MB – 100MB. Therefore, while designing the cluster, size the data that will be distributed across all nodes and determine a number of partitions such that that no partition size exceeds 100MB. If the default count of 271 results in heavily-loaded partitions, increase the partition count to the point where current data load plus headroom for a future increase in data load keeps per-partition size under 100MB.

Important: If you change the partition count from the default, be sure to use a prime number of partitions. This will help minimize collision of keys across partitions, ensuring more consistent lookup times. For further reading on the advantages of using a prime number of partitions, see http://www.quora.com/Does-making-array-size-a-prime-number-help-in-hash-table-implementation-Why.

Important: If you are an Enterprise customer using the High-Density Data Store with large data sizes, we recommend a large increase in partition count, starting with 5009 or higher.

The partition count cannot be changed after a cluster is created, so if you have a larger cluster, be sure to test for and set an optimum partition count prior to deployment. If you need to change the partition count after a cluster is running, you will need to schedule a maintenance window to update the partition count and restart the cluster.

DEDICATED NETWORK INTERFACE CONTROLLER FOR HAZELCAST MEMBERS

Provisioning a dedicated physical network interface controller (NIC) for Hazelcast member nodes ensures smooth flow of data, including business data and cluster health checks, across servers. Sharing network interfaces between a Hazelcast instance and another application could result in choking the port, thus causing unpredictable cluster behavior.

NETWORK SETTINGS

Adjust TCP buffer size

TCP uses a congestion window to determine how many packets it can send at one time; the larger the congestion window, the higher the throughput. The maximum congestion window is related to the amount of buffer space that the kernel allocates for each socket. For each socket, there is a default value for the buffer size, which may be changed by using a system library call just before opening the socket. The buffer size for both the receiving and sending sides of the socket may be adjusted.

To achieve maximum throughput, it is critical to use optimal TCP socket buffer sizes for the links you are using to transmit data. If the buffers are too small, the TCP congestion window will never open up fully, therefore throttling the sender. If the buffers are too large, the sender can overrun the receiver such that the sending host is faster than the receiving host, which will cause the receiver to drop packets and the TCP congestion window to shut down.

Hazelcast, by default, configures I/O buffers to 32KB, but these are configurable properties and may be changed in Hazelcast configuration with the following configuration paramaters:

- `hazelcast.socket.receive.buffer.size`
- `hazelcast.socket.send.buffer.size`

Typically, throughput may be determined by the following formulae:

```
TPS = Buffer Size / Latency
```

and

```
Buffer Size = RTT (Round Trip Time) * Network Bandwidth
```

To increase TCP Max Buffer Size in Linux, see the following settings:

- `net.core.rmem.max`
- `net.core.wmem.max`

To increase TCP auto-tuning by Linux, see the following settings:

- `net.ipv4.tcp.rmem`
- `net.ipv4.tcp.wmem`

Further Reading:

- http://www.onlamp.com/pub/a/onlamp/2005/11/17/tcp_tuning.html

GARBAGE COLLECTION

Keeping track of garbage collection statistics is vital to optimum Java performance, especially if you run the JVM with large heap sizes. Tuning the garbage collector for your use case is often a critical performance practice prior to deployment. Likewise, knowing what baseline garbage collection behavior looks like and monitoring for behavior outside of normal tolerances will keep you apprised of potential memory leaks and other pathological memory usage.

Minimize Heap Usage

The best way to minimize the performance impact of garbage collection is to keep heap usage small. Maintaining a small heap can save countless hours of garbage collection tuning and will provide much higher stability and predictability across your entire application. Even if your application uses very large amounts of data, you can still keep your heap small by using Hazelcast High-Density Memory Store.

Some common off-the-shelf GC tuning parameters for Hotspot and OpenJDK:

```
-XX:+UseParallelOldGC
-XX:+UseParallelGC
-XX:+UseCompressedOops
```

To enable GC logging, use the following JVM arguments:

```
-XX:+PrintGCDetails
-verbose:gc
-XX:+PrintGCTimeStamp
```

HIGH-DENSITY MEMORY STORE (ENTERPRISE FEATURE)

Hazelcast High-Density Memory Store is an in-memory storage option that uses native, off-heap memory to store object data instead of the JVM heap, allowing you to keep terabytes of data in memory without incurring the overhead of garbage collection.

Available to Hazelcast Enterprise customers, the High-Density Memory Store is a perfect solution for those who want the performance gains of large amounts of in-memory data, need the predictability of well-behaved Java memory management and don't want to waste time and effort on meticulous and fragile garbage collection tuning.

Important: If you are an Enterprise customer using the High-Density Data Store with large data sizes, we recommend a large increase in partition count, starting with 5009 or higher. See the Partition Count section above for more information. Also, if you intend to pre-load very large amounts of data into memory (tens, hundreds, or thousands of gigabytes), be sure to profile the data load time and to take that startup time into account prior to deployment.

Further Reading:

- Online Documentation: http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#high-density-memory-store
- Hazelcast Resources: http://hazelcast.com/resources/enterprise-lightning-talk-high-density-memory-store/

OPTIMIZING QUERIES

Add Indexes for Queried Fields

For queries on fields with ranges, you can use an ordered index.

Set optimizeQuery to "true" in Map Configuration

This will cause Hazelcast to cache a deserialized form of the object under query in memory. This removes the overhead of object deserialization per query, but will increase heap usage.

Object "in-memory-format"

An alternative to setting optimize query to "true" is to set the queried object's in-memory format to "object." This will force that object to be always kept in object format, resulting in faster access for queries, but also higher heap usage. It will also incur an object serialization step on every remote "get" operation.

Further Reading:

- Hazelcast Blog: http://blog.hazelcast.com/2013/09/21/in-memory-format/
- Online Documentation: http://docs.hazelcast.org/docs/latest/manual/html/map-inmemoryformat.html

Implement the "Portable" Interface on Queried objects

The Portable interface allows for individual fields to be accessed without the overhead of deserialization or reflection and supports query and indexing support without full-object deserialization.

Further Reading:

- Hazelcast Blog: http://blog.hazelcast.com/2013/12/27/for-faster-hazelcast-queries/
- Online Documentation: http://docs.hazelcast.org/docs/latest/manual/html/portableserialization.html

OPTIMIZING SERIALIZATION

Hazelcast supports a range of object serialization mechanisms, each with their own costs and benefits. Choosing the best serialization scheme for your data and access patterns can greatly increase the performance of your cluster. An in-depth discussion of the various serialization methods is referenced below, but here is an at-a-glance summary:

java.io.Serializable

Benefits:

- Standard Java
- Does not require custom serialization implementation

Costs:

- Not as memory- or CPU-efficient as other options

java.io.Externalizable

Benefits over standard Java serialization:

- Allows client-provided implementation

Benefits:

- Standard Java
- More memory- and CPU-efficient than built-in Java serialization

Costs:

- Requires a custom serialization implementation

com.hazelcast.nio.serialization.DataSerializable

Optimization over standard Java Serialization:

- Doesn't store class metadata

Benefits:

- More memory- and CPU-efficient than built-in Java serialization

Costs:

- Not standard Java
- Requires a custom serialization implementation
- Uses reflection

com.hazelcast.nio.serialization.IdentifiedDataSerializable

Optimization over standard Java Serialization

- Doesn't use reflection

Benefits:

- Can help manage object schema changes by making object instantiation into the new schema from older version instance explicit
- More memory-efficient than built-in Java serialization, more CPU-efficient than DataSerializable

Costs:

- Not standard Java

- Requires a custom serialization implementation
- Requires configuration and implementation of a factory method

com.hazelcast.nio.serialization.Portable

Optimization over other serialization schemes:

- Supports partial deserialization during queries

Benefits:

- More CPU-efficient than other serialization schemes in cases where you don't need access to the entire object
- Doesn't use reflection
- Supports versioning

Costs:

- Not standard Java
- Requires a custom serialization implementation
- Requires implementation of factory and class definition
- Class definition (metadata) is sent with object data—but only once per class

Pluggable serialization libraries, e.g. Kryo

Benefits:

- Convenient and flexible
- Can be stream or byte-array based

Costs:

- Often requires serialization implementation
- Requires plugin configuration. Sometimes requires class annotations

SERIALIZATION OPTIMIZATION RECOMMENDATIONS

- Use `IMap.set()` on maps instead of `IMap.put()` if you don't need the old value. This eliminates unnecessary deserialization of the old value.
- Set "native byte order" and "allow unsafe" to "true" in Hazelcast configuration. Setting the native byte array and unsafe options to true enables fast copy of primitive arrays like `byte[]`, `long[]`, etc. in your object.
- Compression—Compression is supported only by `Serializable` and `Externalizable`. It has not been applied to other serializable methods because it is much slower (around three orders of magnitude slower than not using compression) and consumes a lot of CPU. However, it can reduce binary object size by an order of magnitude.
- SharedObject—If set "true", the Java serializer will back-reference an object pointing to a previously serialized instance. If set "false", every instance is considered unique and copied separately even if they point to the same instance. The default configuration is false.

Further Reading:

- Tutorial: http://hazelcast.com/resources/maximizing-hazelcast-performance-with-serialization/
- Webinar: http://hazelcast.com/resources/maximizing-hazelcast-performance-serialization/
- Kryo Serializer: http://blog.hazelcast.com/2013/10/16/kryo-serializer/
- Performance Top Five: http://blog.hazelcast.com/2014/04/08/performance-top-5-1-map-put-vs-map-set/

EXECUTOR SERVICE OPTIMIZATIONS

Hazelcast's `IExecutorService` is an extension of Java's built-in `ExecutorService` that allows for distributed execution and control of tasks. There are a number of options to Hazelcast's executor service that will have an impact on performance.

Number of Threads

An executor queue may be configured to have a specific number of threads dedicated to executing enqueued tasks. Set the number of threads appropriate to the number of cores available for execution. Too few threads will reduce parallelism, leaving cores idle while too many threads will cause context switching overhead.

Bounded Execution Queue

An executor queue may be configured to have a maximum number of entries. Setting a bound on the number of enqueued tasks will put explicit back-pressure on enqueuing clients by throwing an exception when the queue is full. This will avoid the overhead of enqueuing a task only to be cancelled because its execution takes too long. It will also allow enqueuing clients to take corrective action rather than blindly filling up work queues with tasks faster than they can be executed.

Avoid Blocking Operations in Tasks

Any time spent blocking or waiting in a running task is thread execution time wasted while other tasks wait in the queue. Tasks should be written such that they perform no potentially blocking operations (e.g., network or disk I/O) in their `run()` or `call()` methods.

Locality of Reference

By default, tasks may be executed on any member node. Ideally, however, tasks should be executed on the same machine that contains the data the task requires to avoid the overhead of moving remote data to the local execution context. Hazelcast's executor service provides a number of mechanisms for optimizing locality of reference.

- Send tasks to a specific member—using `ExecutorService.executeOnMember()`, you may direct execution of a task to a particular node
- Send tasks to a key owner—if you know a task needs to operate on a particular map key, you may direct execution of that task to the node that owns that key
- Send tasks to all or a subset of members—if, for example, you need to operate on all of the keys in a map, you may send tasks to all members such that each task operates on the local subset of keys, then return the local result for further processing in a Map/Reduce-style algorithm

Scaling Executor Services

If you find that your work queues consistently reach their maximum and you have already optimized the number of threads and locality of reference and removed any unnecessary blocking operations in your tasks, you may first try to scale up the hardware of the overburdened members by adding cores and, if necessary, more memory.

When you have reached diminishing returns on scaling up (such that the cost of upgrading a machine outweighs the benefits of the upgrade), you can scale out by adding more nodes to your cluster. The distributed nature of Hazelcast is perfectly suited to scaling out and you may find in many cases that it is as easy as just configuring and deploying addition virtual or physical hardware.

Executor Service Tips and Best Practices

Work Queue Is Not Partitioned

Each member-specific executor will have its own private work-queue. Once a job is placed on that queue, it will not be taken by another member. This may lead to a condition such that one member has a lot of unprocessed work while another is idle. This can either be the result of an application call such as the following:

```
for(;;){
     iexecutorservice.submitToMember(mytask, member)
}
```

This could also be the result of an imbalance caused by the application, such as in the following scenario: all products by a particular manufacturer are kept in one partition. When a new, very popular product gets released by that manufacturer, the resulting load puts a huge pressure on that single partition while others remain idle.

Work Queue Has Unbounded Capacity by Default

This can lead to `OutOfMemoryError` because the number of queued tasks can grow without bounds. This can be solved by setting the `<queue-capacity>` property on the executor service. If a new task is submitted while the queue is full, the call will not block, but immediately throw a `RejectedExecutionException` that the application must handle.

No Load Balancing

There is currently no load balancing available for tasks that can run on any member. If load balancing is needed, it may be done by creating an `IExecutorService` proxy that wraps the one returned by Hazelcast. Using the members from the `ClusterService` or member information from `SPI:MembershipAwareService`, it could route "free" tasks to a specific member based on load.

Destroying Executors

An `IExecutorService` must be shut down with care because it will shut down all corresponding executors in every member and subsequent calls to proxy will result in a `RejectedExecutionException`. When the executor is destroyed and later a `HazelcastInstance.getExecutorService` is done with the id of the destroyed executor, a new executor will be created as if the old one never existed.

Exceptions in Executors

When a task fails with an exception (or an error), this exception will not be logged by Hazelcast by default. This comports with the behavior of Java's `ThreadPoolExecutorService`, but it can make debugging difficult. There are, however, some easy remedies: either add a try/catch in your runnable and log the exception. Or wrap the runnable/callable in a proxy that does the logging; the last option will keep your code a bit cleaner.

Further Reading:

- Mastering Hazelcast—Distributed Executor Service: http://hazelcast.org/mastering-hazelcast/chapter-6/
- Hazelcast Documentation: http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#executor-service

NEAR CACHE

Access to small-to-medium, read-mostly data sets may be sped up by creating a Near Cache. This cache maintains copies of distributed data in local memory for very fast access.

Benefits:

- Avoids the network and deserialization costs of retrieving frequently-used data remotely

Costs:

- Increased memory consumption in the local JVM
- High invalidation rates may outweigh the benefits of locality of reference
- Strong consistency is not maintained; you may read stale data

Further Reading:

- http://blog.hazelcast.com/2012/09/06/hazelcast-tip-when-to-use-near-cache/
- http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#near-cache

CLIENT EXECUTOR POOL SIZE

The Hazelcast client uses an internal executor service (different from the distributed `IExecutorService`) to perform some of its internal operations. By default, the thread pool for that executor service is configured to be the number of cores on the client machine times five—e.g., on a 4-core client machine, the internal executor service will have 20 threads. In some cases, increasing that thread pool size may increase performance.

Further Reading:

- http://docs.hazelcast.org/docs/latest/manual/html/javaclient.html#executorpoolsize

CLUSTERS WITH MANY (HUNDREDS) OF NODES OR CLIENTS

Very large clusters of hundreds of nodes are possible with Hazelcast, but stability will depend heavily on your network infrastructure and ability to monitor and manage that many servers. Distributed executions in such an environment will be more sensitive to your application's handling of execution errors, timeouts and the optimization of task code.

In general, you may get better results with smaller clusters of Hazelcast members running on more powerful hardware and a higher number of Hazelcast clients. When running large numbers of clients, network stability will still be a significant factor in overall stability. If you are running in Amazon's EC2, hosting clients and servers in the same zone is beneficial. Using Near Cache on read-mostly data sets will reduce server load and network overhead. You may also try increasing the number of threads in the client executor pool (see above).

Further Reading:

- Hazelcast Blog: http://blog.hazelcast.com/2014/02/13/hazelcast-with-100-nodes/
- Hazelcast Blog: http://blog.hazelcast.com/2014/02/28/hazelcast-with-hundreds-of-clients/
- Online Documentation: http://docs.hazelcast.org/docs/latest/manual/html/javaclient.html#executorpoolsize

BASIC OPTIMIZATION RECOMMENDATIONS

- 8 cores per Hazelcast server instance
- Minimum of 8 GB RAM per Hazelcast member (if not using the High-Density Memory Store)
- Dedicated NIC per Hazelcast member
- Linux—any distribution
- All member nodes should run within same subnet
- All member nodes should be attached to the same network switch

Cluster Sizing

To determine the size of the cluster you will need for your use case, you must first be able to answer the following questions:

- What is your expected data size?
- What are you data access patterns?
- What is your read/write percentage?
- Are you doing more key-based lookups or predicates?
- What are your throughput requirements?
- What are your latency requirements?
- What is your fault tolerance and how many backups do you require?

SIZING CONSIDERATIONS

Once you know the size, access patterns, throughput, latency and fault tolerance requirements of your application, you can use the following rules of thumb to help you determine the size of your cluster.

Memory Headroom

Once you know the size of your working set of data, you can start sizing your memory requirements. Data in Hazelcast is both active data and backup data for high availability, so the total memory footprint will be the size of your active data plus the size of your backup data. If your fault tolerance allows for just a single backup, then each member of the Hazelcast cluster will contain a 1:1 ratio of active data to backup data for a total memory footprint of two times the active data. If your fault tolerance requires two backups, then that ratio climbs to 1:2 active to backup data for a total memory footprint of three times your active data set. If you use only heap memory, each Hazelcast node with a 4GB heap should accommodate a maximum of 3.5 GB of total data (active and backup). If you use the High-Density Data 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 redistributed across the remaining members. For this reason, we recommend that you plan to use only 60% of available memory, with 40% headroom to handle node failure or shutdown.

Very Low-Latency Requirements

If your application requires very low latency, consider using an embedded deployment. This configuration will deliver the best latency characteristics. Another solution for ultra-low-latency infrastructure could be to use `ReplicatedMap`. `ReplicatedMap` is a distributed data structure that stores an exact replica of data on each node. This way, all of the data is always present on every node in the cluster, thus preventing a network hop across to other nodes in the case of a `map.get()` request. Otherwise, the isolation and scalability gains of using a client-server deployment are preferable.

CPU Sizing

As a rule of thumb, we recommend a minimum of 8 cores per Hazelcast server instance. You may need more cores if your application is CPU-heavy in, for example, a high throughput distributed executor service deployment.

EXAMPLE: SIZING A CACHE USE CASE

Consider an application that uses Hazelcast as a data cache. The active memory footprint will be the total number of objects in the cache times the average object size. The backup memory footprint will be the active memory footprint times the backup count. The total memory footprint is the active memory footprint plus the backup memory footprint:

> Total memory footprint = (total objects * average object size) + (total objects * average object size * backup count)

For this example, let's stipulate the following requirements:

- 50 GB of active data
- 40,000 transactions per second
- 70:30 ratio of reads to writes via map lookups
- Less than 500ms latency per transaction
- A backup count of 2

Cluster Size Using the High-Density Memory Store

Since we have 50 GB of active data, our total memory footprint will be:

> 50 GB + 50 GB * 2 (backup count) = 150 GB.

Add 40% memory headroom and you will need a total of 250 GB of RAM for data.

To satisfy this use case, you will need 3 Hazelcast nodes, each running a 4 GB heap with ~84 GB of data off-heap in the High-Density Data Store.

Note: You cannot have a backup count greater than or equal to the number of nodes available in the cluster—Hazelcast will ignore higher backup counts and will create the maximum number of backup copies possible. For example, Hazelcast will only create two backup copies in a cluster of three nodes, even if the backup count is set equal to or higher than three.

Note: No node in a Hazelcast cluster will store primary as well as its own backup.

Cluster Size Using Only Heap Memory

Since it's not practical to run JVMs with greater than a four GB heap, you will need a minimum of 42 JVMs, each with a four GB heap to store 150 GB of active and backup data as a four GB JVM would give approximately 3.5 GB of storage space. Add the 40% headroom discussed earlier, for a total of 250 GB of usable heap, then you will need ~72 JVMs, each running with four GB heap for active and backup data. Considering that each JVM has some memory overhead and Hazelcast's rule of thumb for CPU sizing is eight cores per Hazelcast server instance, you will need at least 576 cores and upwards of 300 GB of memory.

Summary

150 GB of data, including backups.

High-Density Memory Store:

- 3 Hazelcast nodes
- 24 cores
- 256 GB RAM

Heap-only:

- 72 Hazelcast nodes
- 576 cores
- 300 GB RAM

Security (Enterprise Features)

Hazelcast Enterprise offers a rich set of JAAS-based security features you can use to authenticate cluster members and client, and to perform access control checks on client operations.

Socket Interceptor

The socket interceptor allows you to intercept socket connections before a node joins a cluster or a client connects to a node. This provides the ability to add custom hooks to the cluster join operation and perform connection procedures (like identity checking using Kerberos, etc.).

Security Interceptor

The security interceptor allows you to intercept every remote operation executed by the client. This lets you add very flexible custom security logic.

Encryption

All socket-level communication among all Hazelcast members may be encrypted. Encryption is based on the Java Cryptography Architecture.

SSL

All Hazelcast members can use SSL socket communication between them.

Credentials and ClusterLoginModule

The Credentials interface and `ClusterLoginModule` allow you to implement custom credentials checking. The default implementation that comes with Hazelcast uses a username/password scheme.

Cluster Member Security

Hazelcast Enterprise supports standard Java Security (JAAS) based authentication between cluster members.

Native Client Security

Hazelcast's client security includes both authentication and authorization via configurable permissions policies.

Further Reading:

- http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#security

Deployment and Scaling Runbook

The following is a sample set of procedures for deploying and scaling a Hazelcast cluster:

1. Ensure you have the appropriate Hazelcast jars (`hazelcast-ee` for Enterprise) installed. Normally `hazelcast-all-<version>.jar` is sufficient for all operations, but you may also install the smaller hazelcast-<version>.jar on member nodes and `hazelcast-client-<version>.jar` for clients.
2. If not configured programmatically, Hazelcast looks for a hazelcast.xml configuration file for server operations and `hazelcast-client.xml` configuration file for client operations. Place all the configurations at their respective places so that they can be picked by their respective applications (Hazelcast server or an application client).
3. Make sure that you have provided the IP addresses of a minimum of two Hazelcast server nodes and the IP address of the joining node itself, if there are more than two nodes in the cluster, in both the configurations. This is required to avoid new nodes failing to join the cluster if the IP address that was configured does not have any server instance running on it.
   Note: A Hazelcast member looks for a running cluster at the IP addresses provided in its configuration. For the upcoming member to join a cluster, it should be able to detect the running cluster on any of the IP addresses provided. The same applies to clients as well.
4. Enable "smart" routing on clients. This is done to avoid a client sending all of its requests to the cluster routed through a Hazelcast member, hence bottlenecking that member. A smart client connects with all Hazelcast server instances and sends all of its requests directly to the respective member node. This also ensures better latency and throughput in accessing data stored in Hazelcast servers.

   Further Reading:

   - Hazelcast Blog: http://blog.hazelcast.com/2013/06/03/whats-new-in-hazelcast-3/
   - Online Documentation: http://docs.hazelcast.org/docs/latest/manual/html/javaclient.html

5. Make sure that all nodes are reachable by every other node in the cluster and are also accessible by clients (ports, network, etc).
6. Start Hazelcast server instances first. This is not mandatory but a recommendation to avoid clients timing out or complaining that no Hazelcast server is found if clients are started before server.
7. Enable/start a network log collecting utility. `nmon` is perhaps the most commonly used tool and is very easy to deploy.
8. To add more server nodes to an already running cluster, just start a server instance with similar configuration to other nodes with a possible addition of the IP address of the new node. You do not require a maintenance window to add more nodes to an already running Hazelcast cluster.
   Note: When a node is added or removed in a Hazelcast cluster, clients may see a little pause time, but this is normal. This is essentially the time required by Hazelcast servers to rebalance the data on the arrival or departure of a member node.
   Note: There is no need to change anything on the clients when adding more server nodes to the running cluster. Clients will update themselves automatically to connect to new nodes once the new node has successfully joined the cluster.
   Note: Rebalancing of data (primary plus backup) on arrival or departure (forced or unforced) of a node is an automated process and no manual intervention is required.
9. Ensure you have configured an adequate backup count based on your SLAs.
10. When using distributed computing features such as `IExecutorService`, `EntryProcessors`, Map/Reduce or `Aggregators`, any change in application logic or in the implementation of above features must also be installed on member nodes. All the member nodes must be restarted after new code is deployed using the typical cluster re-deployment process:
    a. Shutdown servers
    b. Deploy the new application jar on servers' classpath
    c. Start servers

Failure Detection And Recovery

While smooth and predictable operation is the norm, occasional failure of hardware and software is inevitable. But with the right detection, alerts and recovery processes in place, your cluster will tolerate failure without incurring unscheduled downtime.

COMMON CAUSES OF NODE FAILURE

The most common causes of node failure are garbage collection pauses and network connectivity issues, both of which can cause a node to fail to respond to health checks and thus be removed from the cluster.

HEALTH MONITORING & ALERTS

Hazelcast provides multi-level tolerance configurations in a cluster:

1. Garbage collection tolerance—when a node fails to respond to health check probes on the existing socket connection but is actually responding to health probes sent on a new socket, it can be presumed to be stuck either in a long GC or another long running task. Adequate tolerance levels configured here may allow the node to come back from its stuck state within permissible SLAs.
2. Network tolerance—when a node is temporarily unreachable by any means. Temporary network communication errors may cause nodes to become unresponsive. In such a scenario, adequate tolerance levels configured here will allow the node to return to healthy operation within permissible SLAs.

See below for more details:
http://docs.hazelcast.org/docs/3.4/manual/html-single/hazelcast-documentation.html#advanced-configuration-properties

You should establish tolerance levels for garbage collection and network connectivity and then set monitors to raise alerts when those tolerance thresholds are crossed. Customers with a Hazelcast subscription can use the extensive monitoring capabilities of the Management Center to set monitors and alerts.

In addition to the Management Center, we recommend that you use `jstat` and keep verbose GC logging turned on and use a log scraping tool like Splunk or similar to monitor GC behavior. Back-to-back full GCs and anything above 90% heap occupancy after a full GC should be cause for alarm.

Hazelcast dumps a set of information to the console of each instance that may further be used for to create alerts. The following is a detail of those properties:

- `processors`—number of available processors in machine
- `physical.memory.total`—total memory
- `physical.memory.free`—free memory
- `swap.space.total`—total swap space
- `swap.space.free`—available swap space
- `heap.memory.used`—used heap space
- `heap.memory.free`—available heap space
- `heap.memory.total`—total heap memory
- `heap.memory.max`—max heap memory
- `heap.memory.used/total`—ratio of used heap to total heap
- `heap.memory.used/max`—ratio of used heap to max heap
- `minor.gc.count`—number of minor GCs that have occurred in JVM
- `minor.gc.time`—duration of minor GC cycles

- `major.gc.count`–number of major GCs that have occurred in JVM
- `major.gc.time`–duration of all major GC cycles
- `load.process`–the recent CPU usage for the particular JVM process; negative value if not available
- `load.system`–the recent CPU usage for the whole system; negative value if not available
- `load.systemAverage`–system load average for the last minute. The system load average is the sum of the number of runnable entities queued to the available processors and the number of entities running on available processors averaged over a period of time
- `thread.count`–number of threads currently allocated in the JVM
- `thread.peakCount`–peak number of threads allocated in the JVM
- `event.q.size`–size of the event queue

  Note: Hazelcast uses internal executors to perform various operations that read tasks from a dedicated queue. Some of the properties below belong to such executors:

- `executor.q.async.size`–Async Executor Queue size. Async Executor is used for async APIs to run user callbacks and is also used for some Map/Reduce operations.
- `executor.q.client.size`– Size of Client Executor: Queue that feeds to the executor that perform client operations
- `executor.q.query.size`–Query Executor Queue size: Queue that feeds to the executor that execute queries
- `executor.q.scheduled.size`–Scheduled Executor Queue size: Queue that feeds to the executor that performs scheduled tasks
- `executor.q.io.size`–IO Executor Queue size: Queue that feeds to the executor that performs I/O tasks
- `executor.q.system.size`–System Executor Queue size: Executor that processes system-like tasks for cluster/partition
- `executor.q.operation.size`–Number of pending operations. When an operation is invoked, the invocation is sent to the correct machine and put in a queue to be processed. This number represents the number of operations in that queue
- `executor.q.priorityOperation.size`–Same as `executor.q.operation.size`. Only there are two types of operations - normal and priority. Priority operations end up in a separate queue
- `executor.q.response.size`–number of pending responses in the response queue. Responses from remote executions are added to the response queue to be sent back to the node invoking the operation (e.g. the node sending a map.put for a key it does not own)
- `operations.remote.size`–number of invocations that need a response from a remote Hazelcast server instance
- `operations.running.size`–number of operations currently running on this node
- `proxy.count`–number of proxies
- `clientEndpoint.count`–number of client endpoints
- `connection.active.count`–number of currently active connections
- `client.connection.count`–number of current client connections

MANAGEMENT CENTER (SUBSCRIPTION AND ENTERPRISE FEATURE)

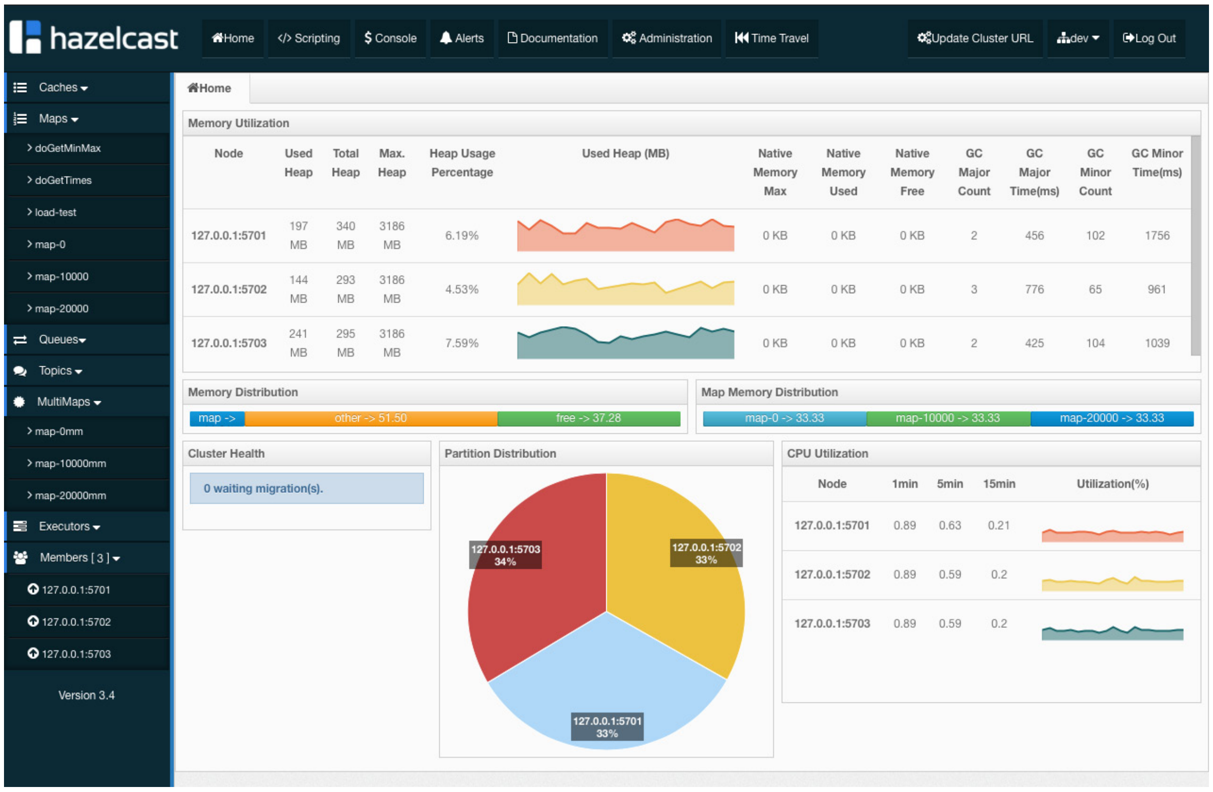
The Hazelcast Management Center is a product available to Hazelcast Enterprise and subscription customers that facilitates monitoring and management of Hazelcast clusters. In addition to monitoring overall cluster state, you can also analyze and browse your data structures in detail, update map configurations and take thread dump from nodes. With its scripting and console module, you can run scripts (JavaScript, Ruby, Groovy, and Python) and commands on your nodes.

Cluster-Wide Statistics and Monitoring

While each member node has a JMX management interface that expose per-node monitoring capabilities, the Management Center collects the all of the individual member node statistics to provide cluster-wide JMX and REST management APIs, making it a central hub for all of your cluster's management data. In a production environment, the Management Center is the best way to monitor the behavior of the entire cluster, both through its web-based user interface and through its cluster-wide JMX and REST APIs.

Web Interface Home Page

The home page of the Management Center provides a dashboard-style overview. For each node, it displays at-a-glance statistics that may be used to quickly gauge the status and health of each member and the cluster as a whole.



Memory Utilization

Node	Used Heap	Total Heap	Max. Heap	Heap Usage Percentage	Used Heap (MB)	Native Memory Max	Native Memory Used	Native Memory Free	GC Major Count	GC Major Time(ms)	GC Minor Count	GC Minor Time(ms)
127.0.0.1:5701	197 MB	340 MB	3186 MB	6.19%		0 KB	0 KB	0 KB	2	456	102	1756
127.0.0.1:5702	144 MB	293 MB	3186 MB	4.53%		0 KB	0 KB	0 KB	3	776	65	961
127.0.0.1:5703	241 MB	295 MB	3186 MB	7.59%		0 KB	0 KB	0 KB	2	425	104	1039

Memory Distribution

map -> | other -> 51.50 | free -> 37.28

Map Memory Distribution

map-0 -> 33.33 | map-10000 -> 33.33 | map-20000 -> 33.33

Cluster Health

0 waiting migration(s).

CPU Utilization

Node	1min	5min	15min	Utilization(%)
127.0.0.1:5701	0.89	0.63	0.21	
127.0.0.1:5702	0.89	0.59	0.2	
127.0.0.1:5703	0.89	0.59	0.2	

Figure 3: Management Center Home

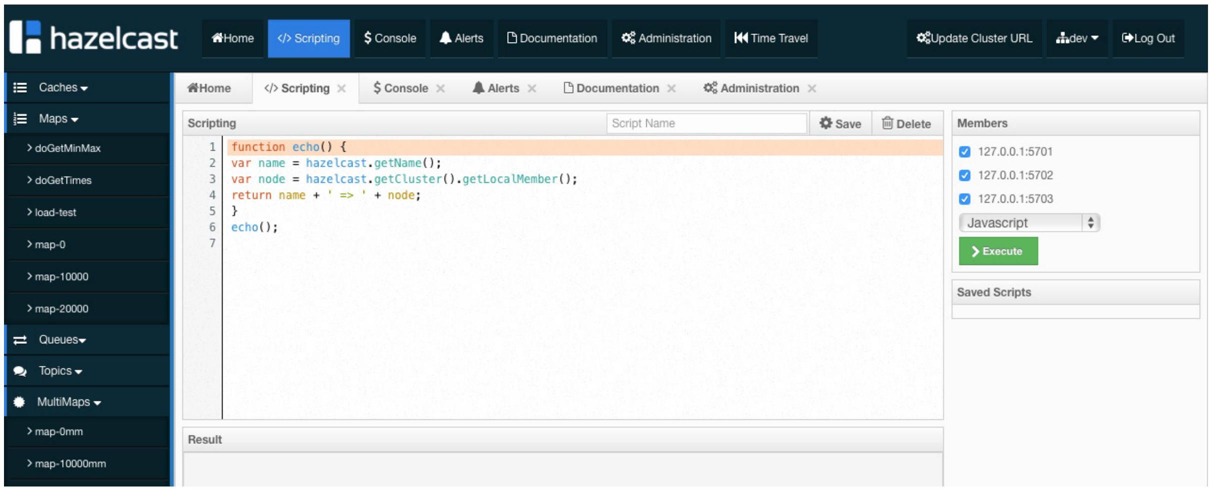
Home page statistics per node:

- Used heap
- Total heap
- Max heap
- Heap usage percentage
- A graph of used heap over time
- Max native memory
- Used native memory

- Major GC count
- Major GC time
- Minor GC count
- Minor GC time
- CPU utilization of each node over time

Home page cluster-wide statistics:

- Total memory distribution by percentage across map data, other data, and free memory
- Map memory distribution by percentage across all the map instances
- Distribution of partitions across members



```
function echo() {
var name = hazelcast.getName();
var node = hazelcast.getCluster().getLocalMember();
return name + ' => ' + node;
}
echo();
```

✓ 127.0.0.1:5701
✓ 127.0.0.1:5702
✓ 127.0.0.1:5703
Javascript

Figure 4: Management Center Tools

Management Center Tools

The toolbar menu provides access to various resources and functions available in the Management Center. These include:

- Home—loads the Management Center home page
- Scripting—allows ad-hoc Javascript, Ruby, Groovy, or Python scripts to be executed against the cluster
- Console—provides a terminal-style command interface to view information about and to manipulate cluster members and data structures
- Alerts—allows custom alerts to be set and managed (see Monitoring Cluster Health below)
- Documentation—loads the Management Center documentation
- Administration—provides user access management (available to admin users only)
- Time Travel—provides a view onto cluster statistics in the past

Data Structure and Member Management

The Caches, Maps, Queues, Topics, MultiMaps, and Executors pages each provide a drill-down view onto the operational statistics of individual data structures. The Members page provides a drill-down view onto the operational statistics of individual cluster members, including CPU and memory utilization, JVM Runtime statistics and properties, and member configuration. It also provides facilities to run GC, take thread dumps, and shut down each member node.

Monitoring Cluster Health

The "Cluster Health" section on the Management Center home page describes current backup and partition migration activity. While a member's data is being backed up, the Management Center will show an alert indicating that the the cluster is vulnerable to data loss if that node is removed from service before the backup is complete.

When a member node is removed from service, the cluster health section will show an alert while the data is re-partitioned across the cluster indicating that the cluster is vulnerable to data loss if any further nodes are removed from service before the repartitioning is complete.

You may also set alerts to fire under specific conditions. In the "Alerts" tab, you can set alerts based on the state of cluster members as well as alerts based on the status of particular data types. For one or more members and for one or more data structures of a given type on one or more members, you can set alerts to fire when certain watermarks are crossed.

When an alert fires, it will show up as an orange warning pane overlaid on the Management Center web interface.

Available member alert watermarks:

- Free memory has dipped below a given threshold
- Used heap memory has grown beyond a given threshold
- Number of active threads has dipped below a given threshold
- Number of daemon threads has grown above a given threshold

Available Map and MultiMap alert watermarks (greater than, less than, or equal to a given threshold):

- Entry count
- Entry memory size
- Backup entry count
- Backup entry memory size
- Dirty entry count
- Lock count
- Gets per second
- Average get latency
- Puts per second
- Average put latency
- Removes per second
- Average remove latency
- Events per second

Available Queue alert watermarks (greater than, less than, or equal to a given threshold):

- Item count
- Backup item count
- Maximum age
- Minimum age

- Average age
- Offers per second
- Polls per second

Available executor alert watermarks (greater than, less than, or equal to a given threshold):

- Pending task count
- Started task count
- Completed task count
- Average remove latency
- Average execution latency

Further Reading:

- Management Center Product Information: http://hazelcast.com/products/management-center/
- Online Documentation, Management Center: http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#management-center
- Online Documentation, Clustered JMX Interface: http://docs.hazelcast.org/docs/latest/manual/html/clusteredjmx.html
- Online Documentation, Clustered REST Interface: http://docs.hazelcast.org/docs/latest/manual/html/clusteredrest.html#clustered-rest

USING JMX OR REST FOR MONITORING

Each Hazelcast node exposes a JMX management interface that include statistics about distributed data structures and the state of that node's internals. The Management Center described above provides a centralized JMX and REST management API that collects all of the operational statistics for the entire cluster.

As an example of what you can achieve with JMX beans for an `IMap`, you may want to raise alerts when the latency of accessing the map increases beyond an expected watermark that you established in your load-testing efforts. This could also be the result of high load, long GC or other potential problems that you might have already created alerts for, so consider the output the following bean properties:

```
localTotalPutLatency
localTotalGetLatency
localTotalRemoveLatency
localMaxPutLatency
localMaxGetLatency
localMaxRemoveLatency
```

Similarly, you may also make use the `HazelcastInstance` bean that exposes information about the current node and all other cluster members.

For example, you may use the following properties to raise appropriate alerts or for general monitoring:

- `memberCount`—if this is lower than the count of expected members in the cluster, raise an alert
- `members`—returns a list of all members connected in the cluster
- `shutdown`—shutdown hook for that node
- `clientConnectionCount`—returns the number of client connections. Raise an alert if lower than expected number of clients.
- `activeConnectionCount`—total active connections

Further Reading:

- Online Documentation, Monitoring With JMX: http://docs.hazelcast.org/docs/3.4/manual/html-single/hazelcast-documentation.html#monitoring-with-jmx
- Online Documentation, Clustered JMX: http://docs.hazelcast.org/docs/latest/manual/html/clusteredjmx.html
- Online Documentation, Clustered REST: http://docs.hazelcast.org/docs/latest/manual/html/clusteredrest.html

RECOMMENDATIONS FOR SETTING ALERTS

We recommend setting alerts for at least the following incidents:

- CPU usage consistently over 90% for a specific time period
- Heap usage alerts:
  - Increasing old gen after every full GC while heap occupancy is below 80% should be treated as a moderate alert
  - Over 80% heap occupancy after a full GC should be treated as a red alert.
  - Too-frequent full GCs
- Node left event
- Node join event
- SEVERE or ERROR in Hazelcast logs

ACTIONS AND REMEDIES FOR ALERTS

When an alert fires on a node, it's important to gather as much data about the ailing JVM as possible before shutting it down.

Logs. Collect Hazelcast server logs from all server instances. If running in a client-server topology, also collect client application logs before a restart.

Thread dumps. Make sure you take thread dumps of the ailing JVM using either the Management Center or `jstack`. Take multiple snapshots of thread dumps at 3 – 4 second intervals.

Heap dumps. Make sure you take heap dumps and histograms of the ailing JVM using jmap

Further Reading:

- What to do in case of an OOME: http://blog.hazelcast.com/2013/12/30/out-of-memory-and-hazelcast/
- What to do when one or more partitions become unbalanced (e.g., a partition becomes so large, it can't fit in memory): http://blog.hazelcast.com/2013/08/25/controlled-partitioning/
- What to do when a queue store has reached its memory limit: http://blog.hazelcast.com/2013/12/26/overflow-in-hazelcast-queue-store/
- http://blog.scoutapp.com/articles/2009/07/31/understanding-load-averages
- http://docs.oracle.com/javase/7/docs/jre/api/management/extension/com/sun/management/OperatingSystemMXBean.html

SPLIT-BRAIN

Under certain cases of network failure, some cluster members may become unreachable to each other, yet still be fully operational and may even be able to see some, but not all, of the extant cluster members. From the perspective of each node, the unreachable members will appear to have gone offline. Under these circumstances, what was once a single cluster will have been divided into two or more clusters. This is known as network partitioning, or the "Split-Brain Syndrome."

Consider a five-node cluster as depicted in Figure 5:

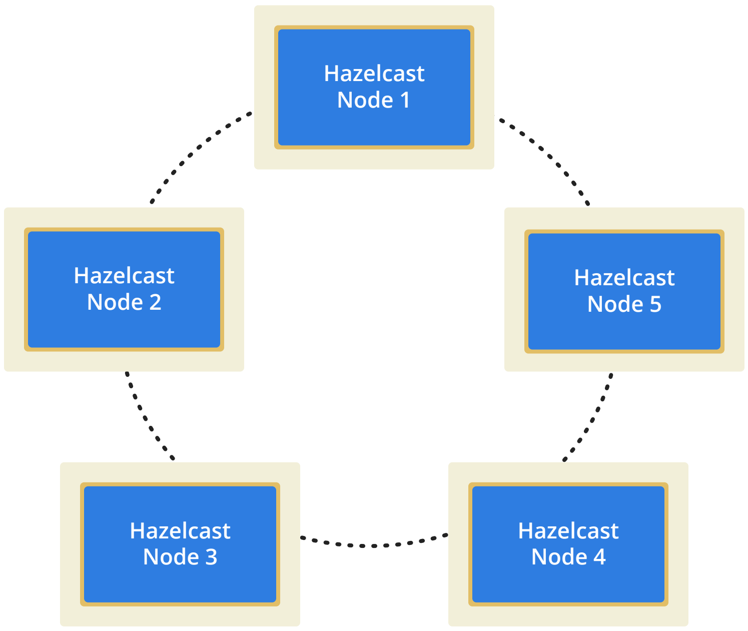


Figure 5

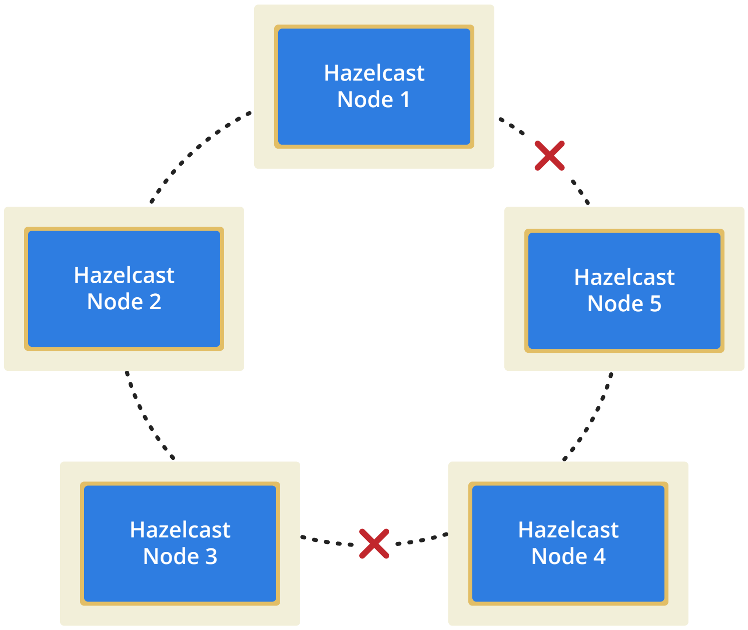


Figure 6: Network failure isolates nodes one and two from node three

All five nodes have working network connections to each other and respond to health check heartbeat pings. If a network failure causes communication to fail between nodes four and five and the rest of the cluster

(Figure 6), from the perspective of nodes one, two, and three, nodes four and five will appear to have gone offline. However, from the perspective of nodes four and five, the opposite is true: nodes one through three appear to have gone offline (Figure 7).

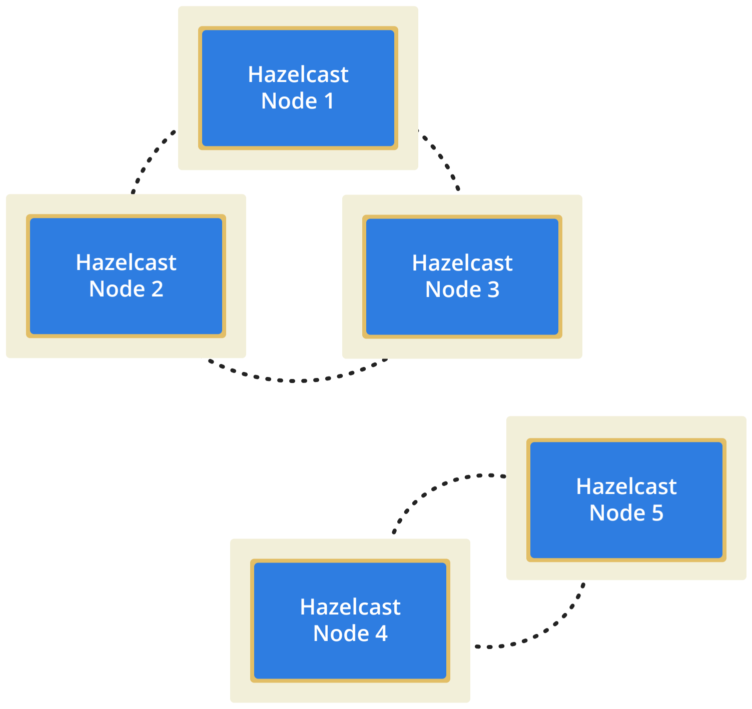


Figure 7: Split-Brain

How to respond to a split-brain scenario depends on whether consistency of data or availability of your application is of primary concern. In either case, because a Split-Brain scenario is caused by a network failure, you must initiate an effort to identify and correct the network failure. Your cluster cannot be brought back to steady state operation until the underlying network failure is fixed.

If availability is of primary concern, especially if there is little danger of data becoming inconsistent across clusters (e.g., you have a primarily read-only caching use case) then you may keep both clusters running until the network failure has been fixed. Alternately, if data consistency is of primary concern, it may make sense to remove the clusters from service until the split-brain is repaired.

Once the network is repaired, the multiple clusters must be merged back together into a single cluster. This normally happens by default and the multiple sub-clusters created by the split-brain merge again to re-form the original cluster. This is how Hazelcast resolves the split-brain condition:

1. Perform internal election to determine the original cluster. The election is performed based on algorithms internal to Hazelcast, but it mainly relies on the number of connected nodes, amount of data, most recent transactions, etc.
2. After the election, the losing cluster is merged into the winning cluster.
3. How the merge is carried out is based on merge policies. The default merge policy forces the merging cluster to join with a clean, initialized state—blank memory. Hazelcast also allows users to write custom merge policies to decide what the merging cluster should do before it can re-join the cluster.

Hazelcast 3.5 will include a cluster quorum configuration that allows you to specify the minimum number of member nodes required for the cluster to remain in an operational state. If the number of machines is below the defined minimum at any time, operations are rejected and the rejected operations return a `QuorumException` to their callers.

Further Reading:

- Online documentation on split-brain: http://docs.hazelcast.org/docs/3.4/manual/html-single/hazelcast-documentation.html#network-partitioning-split-brain-syndrome
- Online Documentation, Network Partitioning: http://docs.hazelcast.org/docs/latest/manual/html/networkpartitioning.html
- Hazelcast 3.5 Snapshot Documentation, Cluster Quorum: http://docs.hazelcast.org/docs/latest-dev/manual/html/clusterquorum.html

RECOVERY FROM A PARTIAL OR TOTAL FAILURE

Under normal circumstances, Hazelcast members are self-recoverable as in the following scenarios:

- Automatic split-brain resolution
- Hazelcast allowing stuck/unreachable nodes to come back within configured tolerance levels (see above in the document for more details)

However, in the rare case when a node is declared unreachable by Hazelcast because it fails to respond, but the rest of the cluster is still running, use the following procedure to for recovery:

1. Collect Hazelcast server logs from all server nodes, active and unresponsive
2. Collect Hazelcast client logs or application logs from all clients
3. If the cluster is running and one or more member nodes was ejected from the cluster because it was stuck, take heap dump of any stuck member nodes
4. If the cluster is running and one or more member nodes is ejected from the cluster because it was stuck, take thread dumps of server nodes including any stuck member nodes. For taking thread dumps, you may use the Java utilities `jstack`, `jconsole` or any other JMX client
5. If the cluster is running and one or more member nodes are ejected from the cluster because it was stuck, collect `nmon` logs from all nodes in the cluster
6. After collecting all of the necessary artifacts, shut down the rogue node(s) by calling shutdown hooks (see next section, Cluster Member Shutdown, for more details) or through JMX beans if using a JMX client
7. After shutdown, start the server node(s) and wait for them to join the cluster. After successful joining, Hazelcast will rebalance the data across new nodes

Important: Hazelcast allows persistence based on Hazelcast callback APIs, which allow you to store cached data in an underlying data store in a write-through or write-behind pattern and reload into cache for cache warm-up or disaster recovery. See link for more details:

http://docs.hazelcast.org/docs/latest/manual/html-single/hazelcast-documentation.html#map-store

Cluster Member Shutdown

- `HazelcastInstance.shutdown()` is graceful so it waits all backups to be completed. You may also use the web-based user interface in the Management Center to shut down a particular cluster member. See the Management Center section above for details
- Make sure to shut down Hazelcast instance on shutdown; in a web application, do it in context destroy event - http://blog.hazelcast.com/2012/09/18/hazelcast-tip-do-not-forget-to-shutdown-hazelcast-on-context-destroy/
- To perform graceful shutdown in a web container, see http://stackoverflow.com/questions/18701821/hazelcast-prevents-the-jvm-from-terminating--Tomcat hooks; Tomcat-independent way to detect JVM shutdown and safely call `Hazelcast.shutdownAll()`
- If an instance crashes or you forced it to shutdown ungracefully, any data that is unwritten to cache, any enqueued write-behind data, and any data that has not yet been backed up will be lost

License Management

If you have a license for Hazelcast Enterprise, you will receive a unique license key from Hazelcast Support that will enable the Hazelcast enterprise capabilities. Ensure the license key file is available on the filesystem of each member and configure the path to it using either declarative, programmatic, or Spring configuration or set the following system property:

```
-Dhazelcast.enterprise.license.key=/path/to/license/key
```

How to Upgrade or Renew Your License

If you wish to upgrade your license or renew your existing license before it expires, contact Hazelcast support to receive a new license. To install the new license, replace the license key on each member host and restart each node, one node at a time, similar to the process described in the "Live Updates to Cluster Member Nodes" section above.

Important: If your license expires in a running cluster or Management Center, do not restart any of the cluster members or the Management Center JVM. Hazelcast will not start with an expired or invalid license. Reach out to Hazelcast support to resolve any issues with an expired license.

Further Reading:

- Online Documentation, Installing Hazelcast Enterprise: http://docs.hazelcast.org/docs/latest/manual/html/installinghzenterprise.html

How to Report Issues to Hazelcast

HAZELCAST SUPPORT SUBSCRIBERS

Offered by the creators of Hazelcast, support subscriptions are the best ways to leverage the power of Hazelcast. With timely responses, critical patches and technical support, subscription customers will get the help they need to achieve a higher level of productivity and quality.

Learn more about Hazelcast support subscriptions:
http://hazelcast.com/services/support/

If you are a current Hazelcast support subscriber with an issue, you can contact Hazelcast either through a ticketing system Zendesk:
https://hazelcast.zendesk.com/

or direct email to:
support@hazelcast.com

Based on your contract, you will have users registered with Hazelcast Zendesk and you should use those credentials to log into Zendesk and submit a ticket.

When submitting a ticket with Hazelcast, provide as much information and data as possible:

1. Detailed description of incident – what happened and when.
2. Details of use case
3. Hazelcast logs
4. Thread dumps from all server nodes
5. Heap dumps
6. Networking logs
7. Time of incident
8. Reproducible test case (optional: Hazelcast engineering may ask for it if required)

Support SLA

Based on your support contract with Hazelcast, you are entitled to pre-defined SLAs for your support issues. However, each issue has a level that defines the severity of the issue in order to garner proper attention. Therefore, setting an appropriate severity level is very important. You should have received details on support process and SLAs in a "Welcome to Hazelcast Support" email.

When you register an issue with Hazelcast support, it is also important to provide detailed information as requested by Hazelcast support engineers and to be prompt in your communication with Hazelcast support. This ensures timely resolution of issues.

HAZELCAST OPEN SOURCE USERS

Hazelcast has an active open source community of developers and users. If you are a Hazelcast open source user, you will find a wealth of information and a forum for discussing issues with Hazelcast developers and other users at the Hazelcast Google Group and on Stack Exchange:

https://groups.google.com/forum/#!forum/hazelcast
http://stackoverflow.com/questions/tagged/hazelcast

You may also file and review issues on the Hazelcast issue tracker on GitHub:
https://github.com/hazelcast/hazelcast/issues

To see all of the resources available to the Hazelcast community, please visit the community page on Hazelcast.org:
http://hazelcast.org/get-involved/

hazelcast
350 Cambridge Ave, Suite 50, Palo Alto, CA 94306 USA
Email: sales@hazelcast.com **Phone:** +1 (650) 521-5453
Visit us at www.hazelcast.com