WHITE PAPER

An Architect’s View of Hazelcast IMDG

Updated for Hazelcast IMDG 3.9
August 2017

By Ben Evans

Java Champion
Tech Fellow at jClarity
www.jclarity.com

WHITE PAPER

An Architect’s View of Hazelcast IMDG

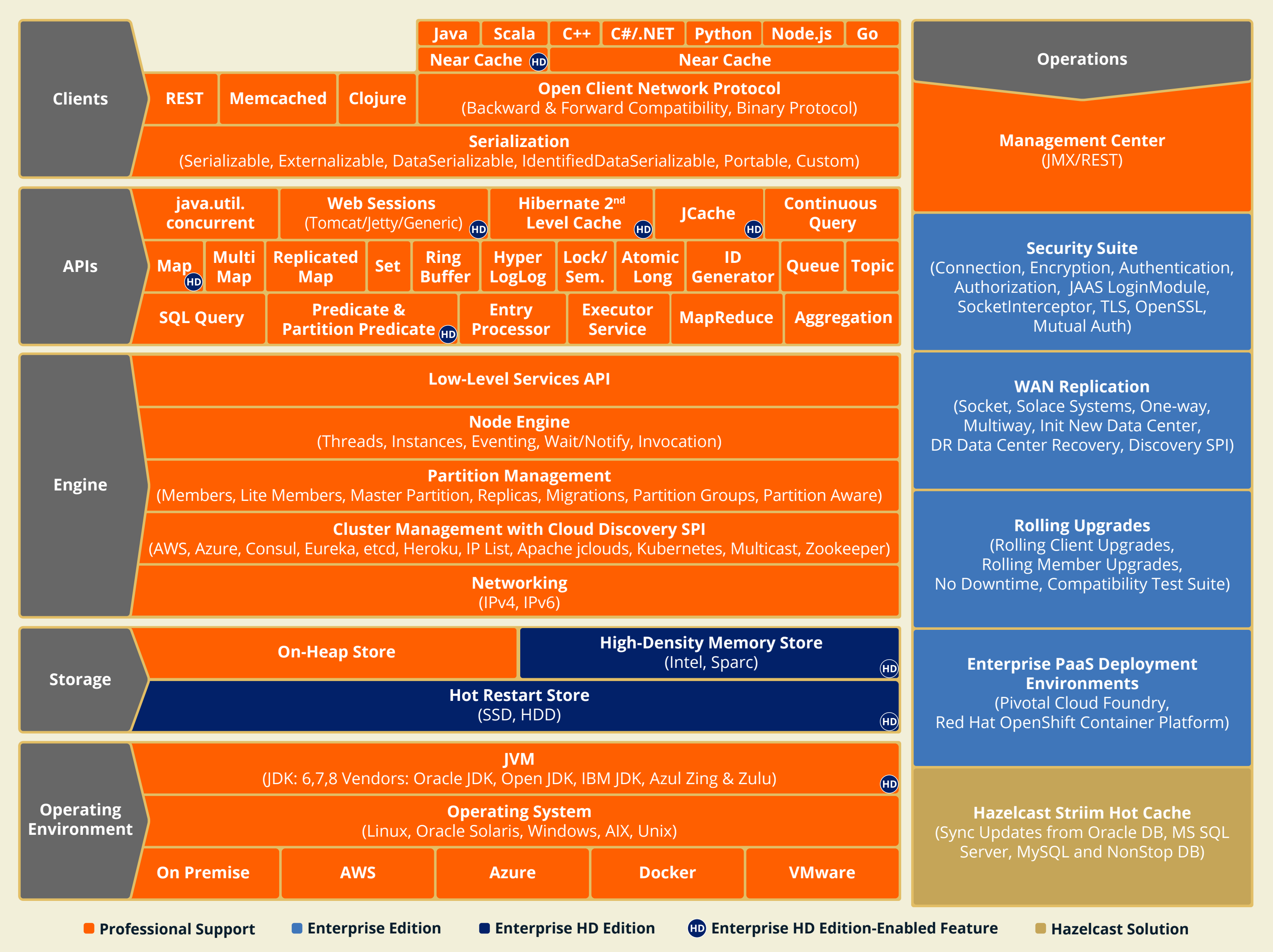
INTRODUCTION TO HAZELCAST IMDG ARCHITECTURE

In this white paper we discuss Hazelcast’s distributed computing technology. This introduction is intended for developers and architects. For documentation more suited to operations engineers or executives (an operational or strategic viewpoint), please see the Links section at the end of this document..

NOTE: This updated document covers some new features from the most recent releases of Hazelcast, including Hot Restart Store, new open-source client APIs for connecting from a wide range of languages environments and enhanced cloud management capabilities covering leading cloud infrastructure technologies.

INTRODUCTION

Hazelcast IMDG 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 *java.util.ConcurrentMap* that can be accessed from multiple application instances of several types such as Java, C++, C#, .NET, Scala, Python, Node.js etc, including Hazelcast instances that are spread out across the network.



Clients

Java | Scala | C++ | C#/.NET | Python | Node.js | Go

Near Cache HD | Near Cache

REST | Memcached | Clojure | Open Client Network Protocol (Backward & Forward Compatibility, Binary Protocol)

Serialization (Serializable, Externalizable, DataSerializable, IdentifiedDataSerializable, Portable, Custom)

APIs

java.util. concurrent | Web Sessions (Tomcat/Jetty/Generic) HD | Hibernate 2nd Level Cache HD | JCache HD | Continuous Query

Map HD | Multi Map | Replicated Map | Set | Ring Buffer | Hyper LogLog | Lock/ Sem. | Atomic Long | ID Generator | Queue | Topic

SQL Query | Predicate & Partition Predicate HD | Entry Processor | Executor Service | MapReduce | Aggregation

Engine

Low-Level Services API

Node Engine (Threads, Instances, Eventing, Wait/Notify, Invocation)

Partition Management (Members, Lite Members, Master Partition, Replicas, Migrations, Partition Groups, Partition Aware)

Cluster Management with Cloud Discovery SPI (AWS, Azure, Consul, Eureka, etcd, Heroku, IP List, Apache jclouds, Kubernetes, Multicast, Zookeeper)

Networking (IPv4, IPv6)

Storage

On-Heap Store | High-Density Memory Store (Intel, Sparc) HD

Hot Restart Store (SSD, HDD) HD

Operating Environment

JVM (JDK: 6,7,8 Vendors: Oracle JDK, Open JDK, IBM JDK, Azul Zing & Zulu) HD

Operating System (Linux, Oracle Solaris, Windows, AIX, Unix)

On Premise | AWS | Azure | Docker | VMware

Operations

Management Center (JMX/REST)

Security Suite (Connection, Encryption, Authentication, Authorization, JAAS LoginModule, SocketInterceptor, TLS, OpenSSL, Mutual Auth)

WAN Replication (Socket, Solace Systems, One-way, Multiway, Init New Data Center, DR Data Center Recovery, Discovery SPI)

Rolling Upgrades (Rolling Client Upgrades, Rolling Member Upgrades, No Downtime, Compatibility Test Suite)

Enterprise PaaS Deployment Environments (Pivotal Cloud Foundry, Red Hat OpenShift Container Platform)

Hazelcast Striim Hot Cache (Sync Updates from Oracle DB, MS SQL Server, MySQL and NonStop DB)

🟧 Professional Support 🟦 Enterprise Edition ■ Enterprise HD Edition HD Enterprise HD Edition-Enabled Feature 🟨 Hazelcast Solution

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.

In order to work effectively with Hazelcast, it is not necessary to deal with the full sophistication present in the architecture. In fact, 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
- 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 on demand, 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 Hazelcast instance 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.

GETTING STARTED – JAVA API EXAMPLE

To include Hazelcast IMDG in a Maven project, just include a dependency stanza like the one below in the dependencies section of your projects POM file and then download the dependency in the usual way:

```
<dependency>
<groupId>com.hazelcast</groupId>
<artifactId>hazelcast</artifactId>
<version>3.6.2</version>
</dependency>
```

The simplest way to get started with Hazelcast is its API, which is largely compatible with *java.util.concurrent* and parts of *java.util*. These classes are familiar to most Java developers and provide a distributed version of collections such as the ConcurrentMap. In addition, the main *com.hazelcast.core* package contains some interfaces used for concurrency primitives, such as IAtomicLong (a distributed sequence number) and ITopic and IQueue (for messaging style applications). Hazelcast allows access to these simple distributed data structures in two different ways - Embedded and Client/Server; we will learn more about this when we discuss cluster topologies.

Code Example

Consider a simple code example – a password updating method. Let us suppose that user passwords are stored as hashes, with a per-user salt to increase security. With Hazelcast IMDG, you can easily make a method that provides the ability to check and update a user password, no matter which machine that a user connected to in a distributed cluster.

```
// Set up Hazelcast once in instance scope, and reuse it on subsequent calls private
final HazelcastInstance hz = Hazelcast.newHazelcastInstance();

// Pass in the username, current password and new password (as plaintext)
public void checkAndUpdatePassword(String username, String passwd, String newPasswd)
{     final Map<String, String> userPasswords = hz.getMap(“user-passwd-hashes”);
   final String hashedPasswd = userPasswords.get(username);
   if (hashPassword(username, passwd).equals(hashedPasswd)) {
      userPasswords.put(username, hashPassword(username, newPasswd));
   }
}
```

First, *newHazelcastInstance* creates the Hazelcast instance as part of this object’s scope. Inside the checkAndUpdatePassword method, getMap obtains a Map instance (actually an instance of an implementation of the Hazelcast IMap interface) from the Hazelcast instance. In most cases, Hazelcast will only create a very lightweight object initially, and will bring real functionality into existence only on first use (this is called lazy initialization).

Next, the real work of the method takes place. The current password, which the user supplied, is hashed by the hashPassword() method, and then it is compared to the value stored in the distributed map. If the values match, then the new password is hashed and updated in the userPasswords map.

In this very simple example, we’re using the factory method Hazelcast::newHazelcastInstance() to get an instance to use. You also have the option of using either an XML configuration file, or a technique such as dependency injection to supply the instance of Hazelcast for the application to use.

The package com.hazelcast.core contains the core API for Hazelcast. This is mostly expressed as interfaces, but there are some classes in the package as well. One key interface is DistributedObject, which contains basic methods common to all Hazelcast distributed objects. The method getName() returns the unique name of the object: for example, allowing it to be retrieved from a HazelcastInstance.

Another important method found on DistributedObject is destroy(), which causes the object to be destroyed across the whole cluster and makes all copies of it available for garbage collection. This is a key pattern for working with distributed objects in Hazelcast – there must be a JVM in the cluster that takes responsibility for cleaning up a distributed object (via calling destroy()). If distributed objects are not correctly cleaned up, then the application will leak memory.

CLIENT APIS

The original API for Hazelcast IMDG is the Java API, but recent versions have added a wide range of other language options. A core set of language APIs are officially supported by Hazelcast, and released as open-source. In addition, recently the Hazelcast user community has added a number of other APIs allowing an even greater number of languages and environments to directly access Hazelcast clusters.

Officially Supported Languages

- Java
- C++/.NET
- Node.js
- Python
- Scala

Community Supported Languages

- Clojure
- Ruby (in progress)

Hazelcast includes an Open Binary Client Protocol. This has support for versioning and backwards compatibility, and allows additional APIs to be written in any client language. The versioning support allows clients and servers to be upgraded independently, and whilst retaining backwards compatibility. The specification is freely available and implementable and specifies a binary, on-the-wire protocol over TCP.

The Hazelcast supported APIs have all been upgraded to use the new open protocol, and this has resulted in significant performance increases. For example low-level benchmarks show Hazelcast IMDG 3.8 as being roughly 30% faster than 3.6, and 3.6 twice as fast as 3.5.

CLOUD PRESENCE

With Hazelcast IMDG 3.8, support for cloud environments and infrastructure technologies has been widened for Hazelcast. This has focused on auto-discovery in popular clouds, using a technology called Cloud Discovery Service Provider Interface (SPI). This feature was added in direct response to a Hazelcast Enhancement Proposal (HEP) – a community request for a new feature.

Supported Cluster Management and Cloud Discovery

- Apache jclouds
- AWS
- Azure
- Consul
- etcd
- IP List
- Eureka
- Kubernetes
- Multicast
- Zookeeper

Supported Containers and Virtual Machines

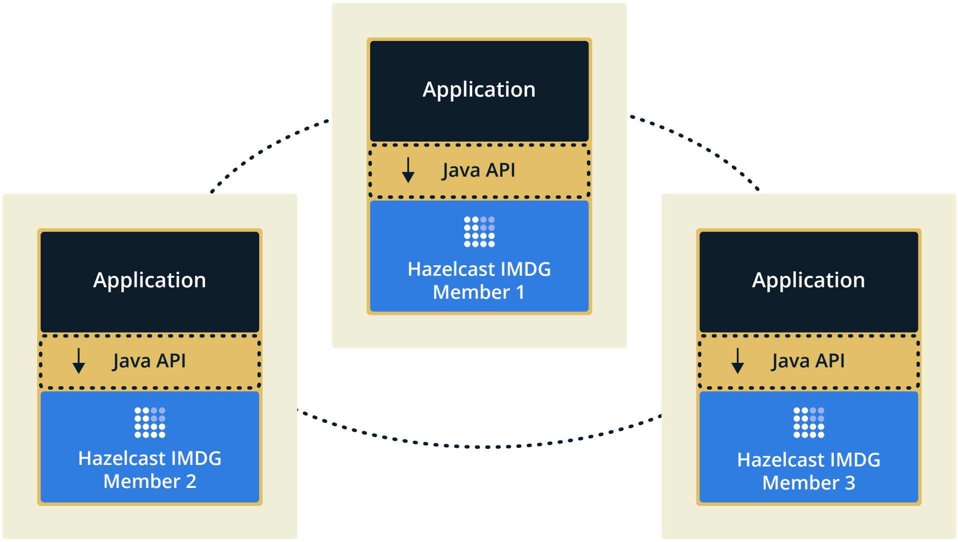
- Azure
- AWS
- Cloud Foundry
- Docker
- OpenShift
- VMware

Cloud technologies are continuing to evolve into a mature marketplace and, in response, Hazelcast is broadening the number and sophistication of the integrations available. For example, the Kubernetes integration provides access to OpenShift and the Apache jclouds library provides access to a large number of cloud providers, including CloudSigma, DigitalOcean, ElasticHosts, Google Compute Engine, Rackspace, ServerLove and SoftLayer.

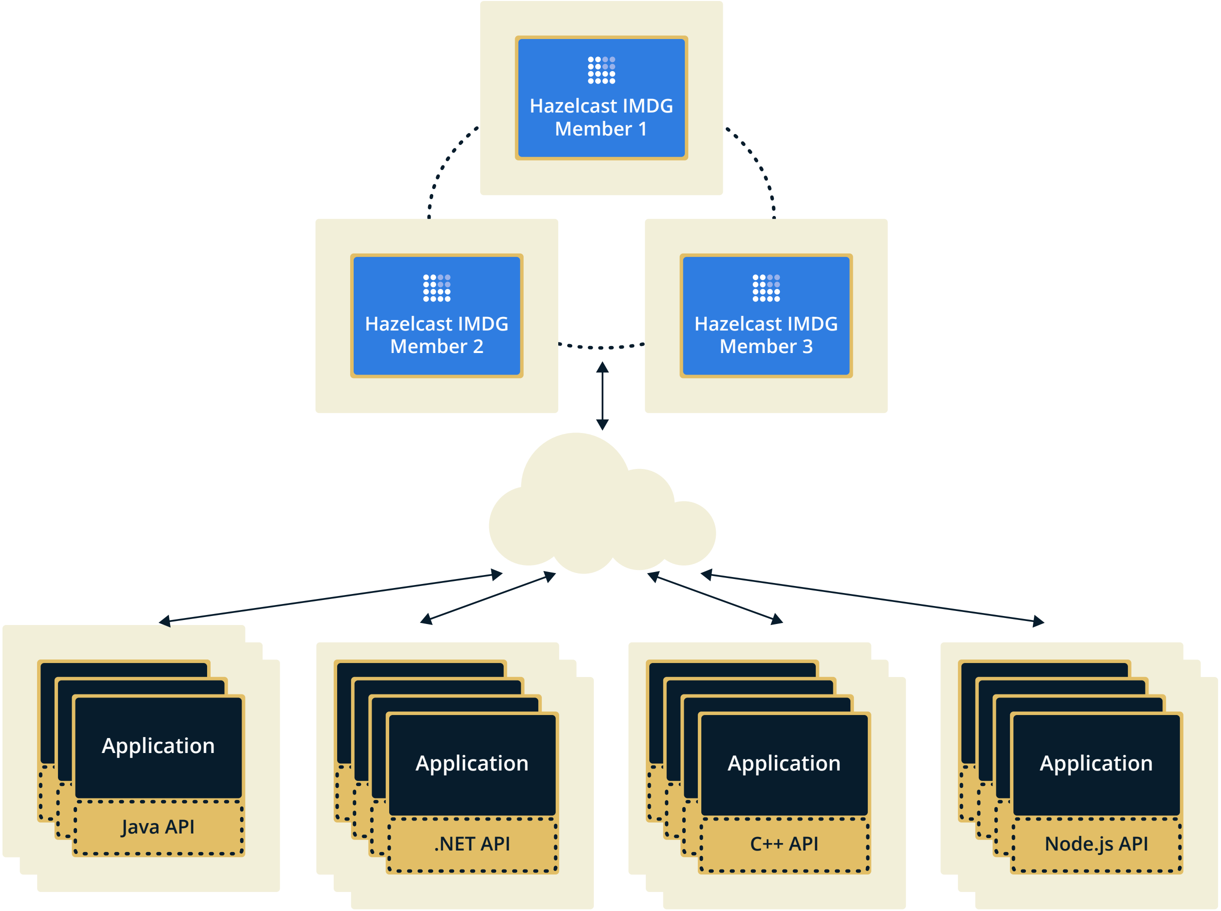
DEPLOYMENT TOPOLOGIES

Hazelcast IMDG supports two modes of operation, either "embedded member", where the JVM containing application code joins the Hazelcast cluster directly, or "client plus member", whereby independent Hazelcast server instances running on same or different host, form the Hazelcast cluster. These two approaches to topology are shown in the following diagrams.

Here is the embedded approach:



Here is the client plus member topology:



Under most circumstances, we recommend client plus member topologies, as it provides greater flexibility in terms of cluster mechanics – member instances 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.

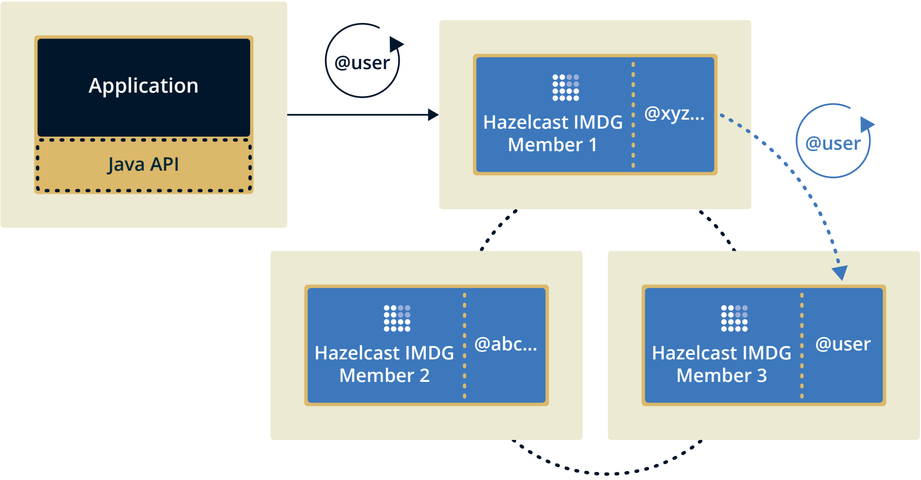
In production applications, the Hazelcast client should be reused between threads and operations. It is designed for multithreaded operation, and creation of a new Hazelcast client is relatively expensive, as it handles cluster events, heartbeating, etc., so as to be transparent to the user.

The Hazelcast API provides full support for both approaches, which is why the core API is expressed as interfaces. For example, Hazelcast’s IMap interface describes the distributed concurrent map that we’ve already met in the first code example. Hazelcast ships two separate implementations of this interface: MapProxyImpl and ClientMapProxy, which correspond to the “embedded member” and “client plus member” approaches to connecting to Hazelcast.

REPLICATION AND SERIALIZATION

Consider the distributed map (IMap). This is one of the most widely-used data structures supported by Hazelcast. To handle data sets too large to fit into a single JVM, Hazelcast partitions the data into local sets and distributes the partitions as evenly as possible, based on the map key. Hazelcast’s elasticity features are enabled by an algorithm that will rebalance the partitions when members join or leave a cluster.

Hazelcast provides a simple naming scheme for controlling which partitions data resides in. This means that the developer can ensure the locality of related data. When Hazelcast’s compute capability is in use, this extends to provide the capability of sending a data processing task (often expressed as a Runnable) to the same partition as the data upon which it will operate.



In the case of a hard failure (e.g. JVM crash or kernel panic), Hazelcast has recovery and failover capabilities to prevent data loss in most of the common failure scenarios.

Hazelcast provides a certain amount of control over how data is replicated. The default is 1 synchronous backup. For data structures such as IMap and IQueue, this can be configured using the config parameters backup-count and async-backup-count. These parameters provide additional backups, but at the cost of higher memory consumption (each backup consumes memory equivalent to the size of the original data structure). The asynchronous option is really designed for low-latency systems where some very small window of data loss is preferable over increased write latency.

It is important to realize that synchronous backups increase the time of the put operations, as the caller waits for backups to be performed. The more backup copies are held, the more time might be needed to execute those backup operations. Some data structures, like AtomicX implementations, ISet or IList have a fixed replication factor i.e. 1 and do not allow configuration to change as of the current version of Hazelcast, but they may provide this in future versions.

The key to how replication takes place is Hazelcast’s use of serialized Java objects. This sophisticated subsystem has evolved considerably as Hazelcast has matured, and it offers several different approaches that provide the developer choices suitable for many different use cases.

For example, one team can choose simplicity using Java’s default and familiar serialization mechanism. However, another team might need to squeeze out every last ounce of performance, and thus may want low-level control to avoid the reflective calls that are inherent to Java’s inbuilt mechanism.

Hazelcast provides a factory-based approach to creating instances as part of the Portable mechanism, which was introduced with Hazelcast 3. This capability also supports multiversioning of classes. It can be very efficient in terms of lookup and queries as it can access and retrieve data at an individual field level. However, to support fast queries, a lot of additional metadata is required. Teams in the extreme high-performance space may seek to avoid this increased data transfer, and fortunately Hazelcast provides a mechanism, called IdentifiedDataSerializable especially for this use case.

	JAVA	PORTABLE	IDENTIFIED DATASERIALIZABLE
Pro	Simple	Full Control	Highest Performance
Con	Uses Reflection	Large Amount of Metadata Transfer	Slower Queries

Hazelcast's replication depends on serialization and can be configured for many use cases. However, there are times when persisting the in-memory data contents to a backing store can be advantageous. This durability comes with a potential performance problem: for full reliability, each change to the in-memory data has to be "written through" to the physical store before the change can be fully confirmed.

There are plenty of use cases where this behavior is overkill, and a short window of potential data loss can be tolerated by the application architecture. In these circumstances, we can configure Hazelcast to "write-behind", and the writes to physical store effectively become asynchronous. Hazelcast provides flexible configuration in the form of the write-delay-seconds, which configures the maximum delay (in seconds) before pending writes are flushed to physical store.

HOT RESTART STORE

In addition to the basic reliability options (write-through and write-behind), Hazelcast IMDG Enterprise HD includes a high-performance option. This feature is known as Hot Restart Store. This persistence feature provides fast cluster restarts by storing the states of the cluster members on disk in a format specially designed for restart performance and to work in concert with SSDs.

Whether the restart is a planned shutdown (including rolling upgrades across a cluster) or a sudden cluster-wide crash (e.g. power outage or total loss of a data center), Hot Restart Store allows full recovery to the previous state of configuration and cluster data.

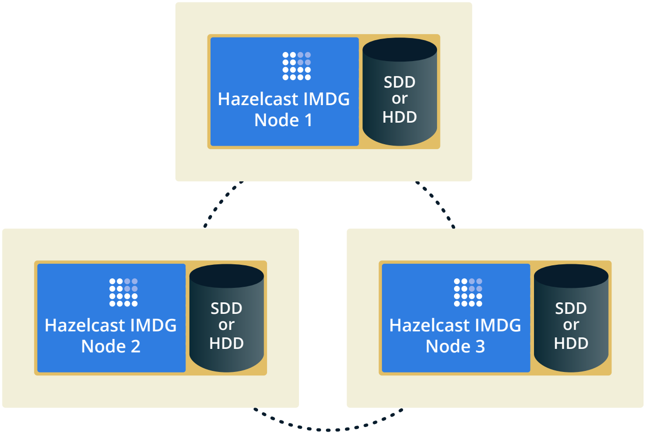
Hot Restart supports the IMap and JCache interfaces, as well as Web Sessions and Hibernate. Further data structures will be delivered in subsequent releases. It is strongly recommended that Hot Restart be used only on SSD drives, as magnetic HDD drives are much slower (typically ~10x). This difference in capability will typically result in restart and recovery times that are not compatible with high-performance use cases.

Hot Restart works by each node controlling its own local snapshot. This provides linear scaling across the cluster, provided one of two best practice approaches is followed:

1. That each node is not configured with more stored data than the size of the node's JVM heap. This is the preferred approach for on-heap storage.
2. In the case of larger stores, HD storage should be used. This use off-heap store, and allows the creation of much larger nodes. This solves some of the limitations of garbage-collected heaps – 100G JVM heaps are not practical to collect.

This has clear best practice implications for the design of high-performance clusters – each node should have a large amount of physical RAM. In the on-heap case, we should use a large JVM heap (not larger than physical RAM) and the Hot Restart size configured to be the same as the heap size. In the off-heap (HD) case, as much memory as we can should be devoted to the HD store.

Hot Restart Start Store – Distributed Persistence



HOT RESTART KEY FEATURES

- Dramatically reduces restart time for huge caches by providing very fast data reload
- Ultra high-performance persistence store, optimized for SSD & mirrored in native memory
- 502,254 writes per second per node and restart times of 39 seconds per 0.80GB per node
- Each node has its own independent store, so enables linear scaling across the cluster
- Restarts in 76 seconds with 100Gb per node (so 76s to reload 1TB on a 10 node cluster)
- Data entirely loaded into RAM on reload – Hazelcast always operates at in-memory speed
- Persistence can be configured per data structure for JCache, Map, Web Sessions and Hibernate

Setup					Reading		Writing	
Storage Type	Data Size	Value Size	Level of par*lism	Partition Threads	Restart Time	Read Throughput	User Threads	Write Throughput
SSD	100 GB	1007 B	4	8	82 sec	1.4 GB/s	16	425,000 ops/sec
SSD	50 GB	495 B	4	40	49 sec	1.2 GB/s	40	400,000 ops/sec
SSD	25 GB	239 B	8	40	39 sec	0.80 GB/s	40	502,254 ops/sec

IN-MEMORY FORMAT FOR IMAP

By default, Hazelcast IMDG serializes objects to byte arrays as they are stored, and deserializes them when they are read. This serialization and deserialization happens only on the client thread. The serialized representation of an object is called the BINARY format. This method shall be used when accessing the data using traditional get(), put() and similar methods.

The binary-format becomes inefficient if the application performs many Queries (Predicate) and Entry Processors operations where serialization/deserialization on the server side is required. To overcome such situations, Hazelcast provides the ability to store the data (values, not keys) in memory in OBJECT format.

The last option is NATIVE format, that is available with Hazelcast Enterprise HD, which allows storing the data in serialized format in Native Memory, which is not subject to garbage collection.

1. BINARY (the default): the value is stored in binary format. Every time the value is needed, it will be deserialized.
2. OBJECT: the value is stored in object (unserialized) format. If a value is needed in an EntryProcessor, this value is used as is and no deserialization occurs. But if a value is needed as result of a *map.get(key)* invocation then an extra step of serialization and deserialization is added. Here is why:
   - *map.put(key, value)*: value gets serialized at the client side and sent to Hazelcast server in a byte array. Server receives the binary array, deserializes it and stores.
   - *map.get(key)*: server serializes the object into binary array, sends over the wire to the client. Client then deserializes the binary array into Java object.
3. NATIVE: equivalent to BINARY but this option enables the map to use HD Memory Store, instead of storing values in JVM heap.

With these three storage schemes, application developers are free to choose whichever is the most efficient for their workload. As always, teams should test their applications to ensure that their settings are optimal, based on the mix of query and other operations they receive.

CACHING API

As an alternative to the distributed concurrent collections view of Hazelcast IMDG, the API also supports a javax.cache implementation. This is the new Java caching standard known as JCache (JSR 107). This provides a standard way of caching objects in Java. JCache has been adopted by all major vendors as an independent standard, and is expected to become a core part of the Java EE 8 specification.

The approach taken is to have a CacheManager that maintains a number of Cache objects. The caches can be thought of being similar to the distributed maps we've already met, but in strict terms they are really instances of Iterable rather than Map.

Code Example

If our application is coding directly against the JCache API, then we don’t necessarily need to make any direct reference to any Hazelcast classes. Instead, a Java Service Provider (SPI) approach is used to obtain an instance of the cache classes, as shown below.

```
public void dictionaryExample() {
   // Obtain a cache manager
   CachingProvider cachingProvider = Caching.getCachingProvider();
   CacheManager cacheManager = cachingProvider.getCacheManager();

   // Configuration for the cache, including type information
   CompleteConfiguration<String, String> config
      = new MutableConfiguration<String, String>()
      .setTypes(String.class, String.class);

   // Create the cache, and get a reference to it cacheManager.
   createCache(“enDeDictionary”, config); Cache<String, String> cache
      = cacheManager.getCache(“enDeDictionary”, String.class, String.class);

   // Populate the cache with a couple of translations
   cache.put(“Monday”, “Montag”);
   cache.put(“Wednesday”, “Mittwoch”);
   System.out.println(cache.get(“Monday”));
}
```

Hazelcast’s implementation of JCache has been designed from the ground up to be compliant with the standard. It is the only compliant implementation that isn’t a wrapper over an existing API, and instead provides first-class support for JCache out of the box.

HAZELCAST AND CAP

The oft-discussed CAP theorem in distributed database theory states that if you have a network that may drop messages, then you cannot have both complete availability and perfect consistency in the event of a partition, instead you must choose one i.e. in the event of a “P” in your cluster, you must choose to be either “C” or “A”

SPLIT BRAIN AND CAP

Hazelcast is an AP product. In the event of a network partition where nodes remain up and connected to different groups of clients (i.e. a split-brain scenario), Hazelcast tends to compromise on Consistency (“C”) to remain Available (“A”) whilst Partitioned (“P”). The effect for the user in the event of a partition would be that clients connected to one partition would see locally consistent results. However, clients connected to different partitions would not necessarily see the same result. For example, an AtomicInteger could now potentially have different values in different partitions.

For transparent data distribution, Hazelcast maps each data entry to a single Hazelcast partition and puts the entry into replicas of that partition. One of the replicas is elected as the primary replica, which is responsible for performing read and write requests on that partition. Backup replicas stay in standby mode until the primary replica fails. By this way, each request hits the most up-to-date version of a particular data entry in a stable cluster. However, since Hazelcast is an AP product due to the CAP theorem, it employs best-effort consistency techniques to keep backup replicas in sync with primaries. Temporary problems in a cluster may cause backup replicas to lose their synchrony with their primary. Hazelcast deals with such problems with an active anti-entropy mechanism.

Hazelcast had always supported handling partitioning for IMap collections, but with release 3.6 Hazelcast’s handling of split-brain has been extended to JCache also. This includes partitioning lost events (which can fire a listener if a partition loss occurs) and support for a minimum size a Hazelcast cluster must have to prevent suspension of the cluster (Cluster Quorum).

CLUSTER QUORUM AND CAP

Cluster Quorum enables you to break the standard implementation of CAP within a Hazelcast cluster by defining minimum number of member instances required in a cluster for the cluster to remain in an operational state. If the number of instances is below the defined minimum at any time, the operations are rejected and the rejected operations return a *QuorumException* to their callers. This allows you to tune Hazelcast cluster towards achieving better Consistency ("C") at the cost of Availability ("A") by rejecting updates that do not pass a minimum threshold. This reduces the chance of concurrent updates to an entry from two partitioned clusters without being a complete CP solution.

RECOVERY FROM "P"

Hazelcast recovers from data partitioning automatically when a network partitioning is resolved. When the network split is resolved and the partitioned clusters begin to see each other, Hazelcast merges the data of partitioned cluster and forms a single cluster with eventually consistent state.

Do note that 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. Hazelcast provides some inbuilt merge policies and also allows you to create your own merge policy by implementing com.hazelcast.map.merge.MapMergePolicy.

HAZELCAST IMDG ENTERPRISE

Hazelcast IMDG is a commercial open source product. The core of the product is free and open source, but many enterprise users require professional support and an extended feature set. The commercial versions are called Hazelcast IMDG Enterprise and Enterprise HD. Hazelcast Enterprise HD guarantees the highest levels of performance and support for the ultimate scale-up and scale-out of your applications. It adds capabilities targeted for enterprise applications, such as:

- **HD Memory** – is a Hazelcast Enterprise HD feature. It allows you to scale in-memory data storage capacity up to Terabytes of main memory in a single JVM. HD Memory eliminates garbage collection issues by minimizing pauses caused by GC, and delivers scaled up and predictable performance.
- **Security** – provides standards-based JAAS and interoperable encryption, authentication, and access control checks to mission-critical applications as well as TLS and OpenSSL support.
- **Management Center** – provides a bird’s-eye view of all cluster activity, along with configurable watermarks for alerts through a web-based user interface and cluster-wide JMX and REST APIs.
- **Hot Restart Store** – distributed persistence with high write performance and low restart times with parallel loading of data across nodes.
- **WAN Replication** – synchronizes multiple Hazelcast clusters in different datacenters for disaster recovery or geographic locality and can be managed centrally through Management Center.
- **Rolling Upgrades** – allows you to upgrade your cluster nodes’ versions without service interruption.
- **Enterprise PaaS Deployment Environments** - enables seamless deployment in Pivotal Cloud Foundry and Red Hat OpenShift PaaS environments

For both the free and enterprise versions, the Hazelcast community is well-known as being friendly, helpful and welcoming to new joiners. There are plenty of great resources available to support teams as they adopt Hazelcast into their architecture.

PARTICIPATE IN THE HAZELCAST COMMUNITY:

www.hazelcast.org

CONTRIBUTE CODE OR REPORT A BUG:

GitHub: https://github.com/hazelcast/hazelcast
Hazelcast Proposals: https://hazelcast.atlassian.net/wiki/display/COM/Hazelcast+Enhancement+Proposals

JOIN THE DISCUSSION:

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

FOLLOW US ONLINE:

Twitter @Hazelcast https://twitter.com/hazelcast
Facebook https://www.facebook.com/hazelcast/
LinkedIn https://www.linkedin.com/company/hazelcast

hazelcast

350 Cambridge Ave, Suite 100, Palo Alto, CA 94306 USA
Email: sales@hazelcast.com **Phone:** +1 (650) 521-5453
Visit us at www.hazelcast.com