WHITE PAPER

An Architect’s View of Hazelcast

By Ben Evans

Java Champion
Tech Fellow at jClarity
www.jclarity.com

WHITE PAPER

An Architect's View of Hazelcast

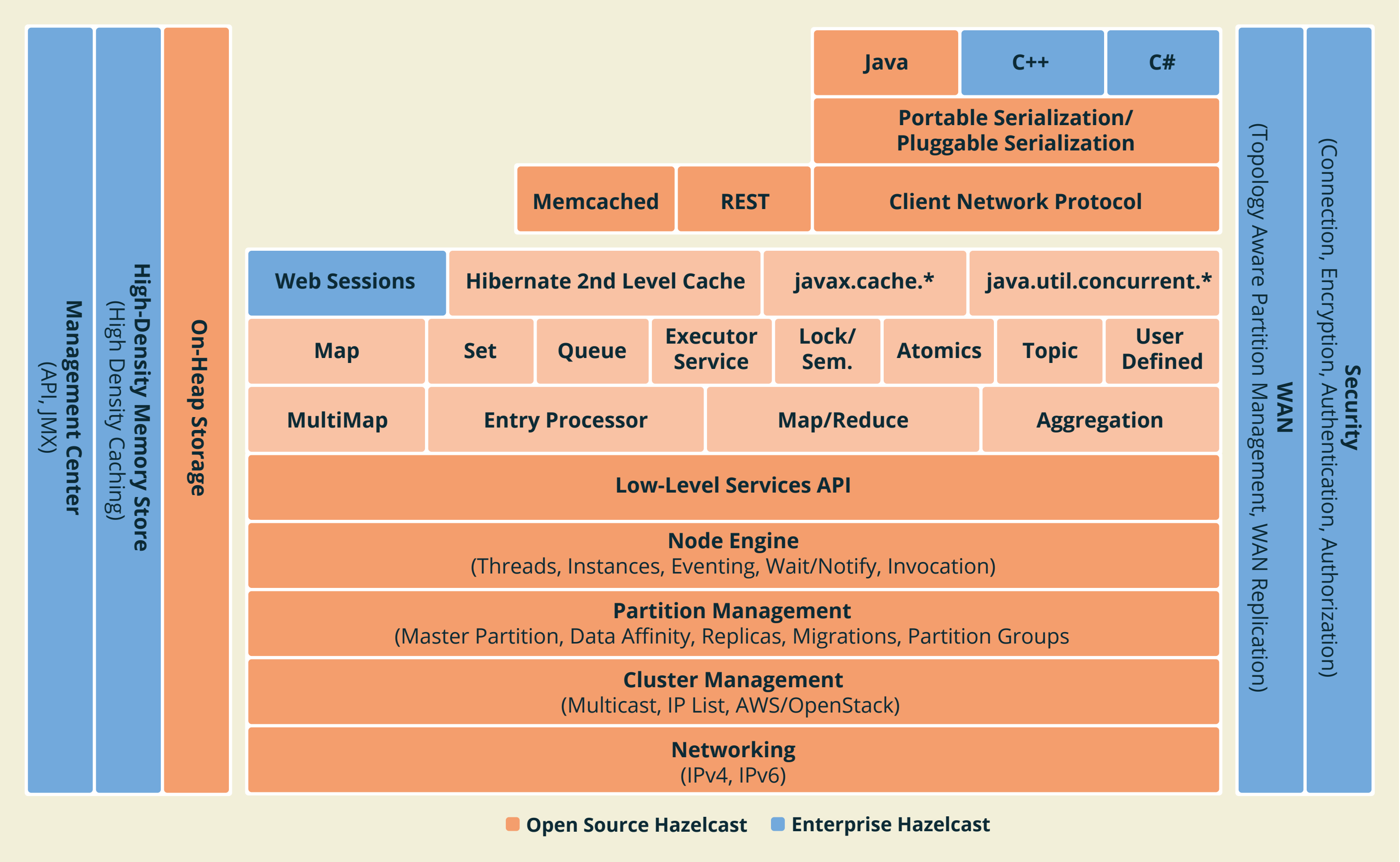
INTRODUCTION TO HAZELCAST ARCHITECTURE

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.

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
- 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.

GETTING STARTED

To include Hazelcast 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>3.4</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 ConcurrentHashMap. 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 ships two implementations for these simple distributed data structures, for reasons that we will see 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, 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 body, 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. Of course, in real production applications, we would use either an XML configuration file, or a technique such as dependency injection to supply the instance of Hazelcast for the application to use.

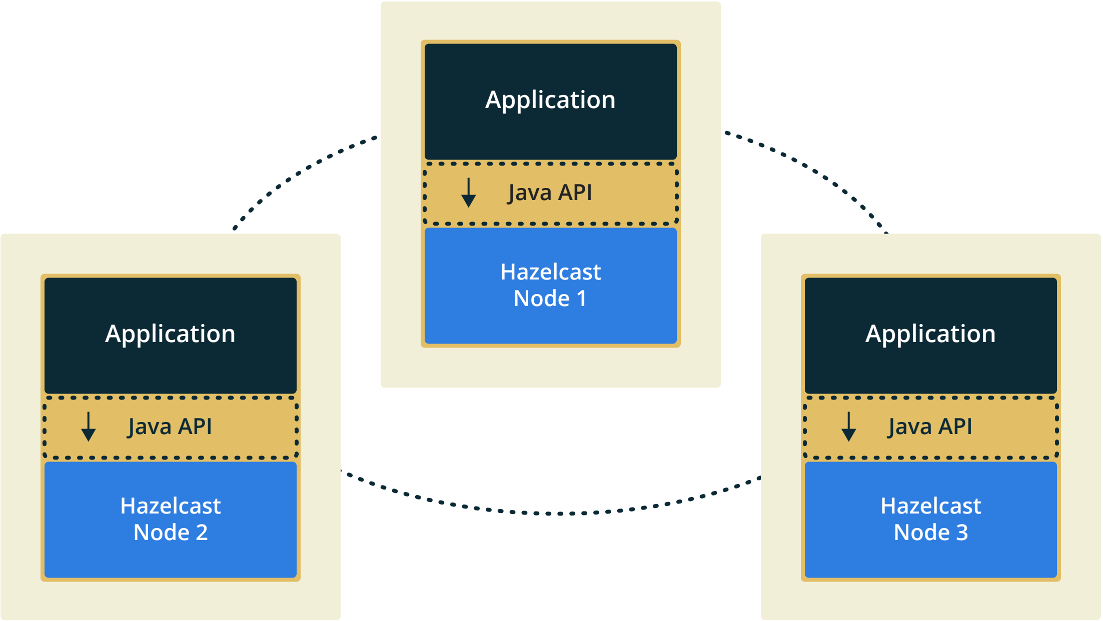
The package com.hazelcast.core contains the core API for Hazelcast. This is mostly expressed as interfaces, but there are some classes in the package as well. One key interface is DistributedObject, which contains basic methods common to all Hazelcast distributed objects. The method getName() returns 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.

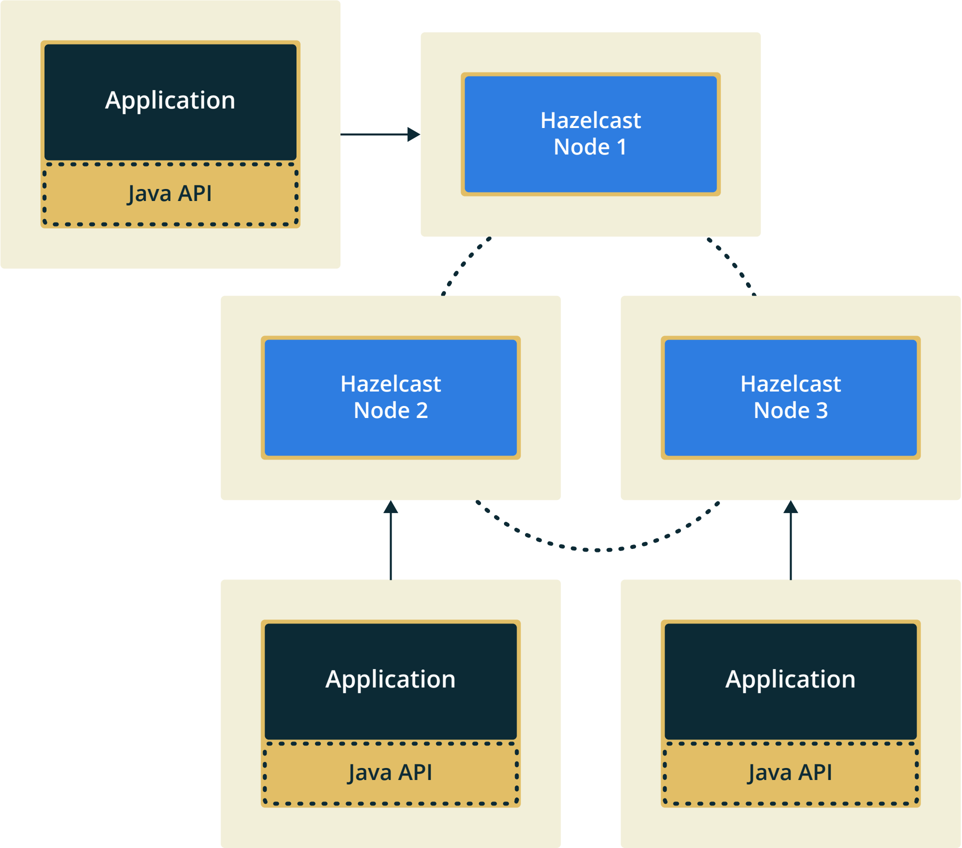
TOPOLOGIES

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:



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 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.

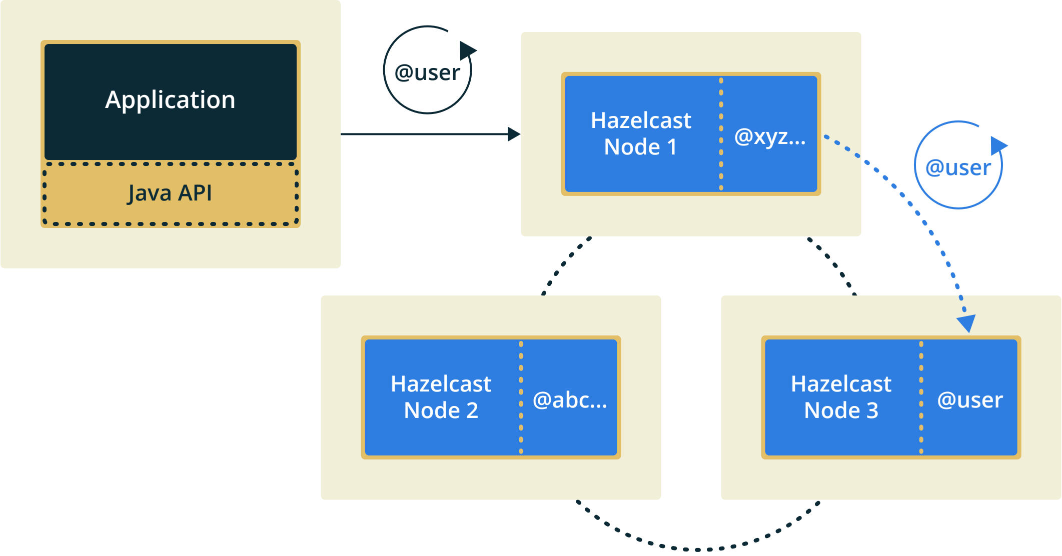
In production applications, the Hazelcast client should be reused between threads and operations. It is designed for multithreaded operation, and creation of a new Hazelcast client is relatively expensive, as it handles cluster events, heartbeating, etc., so as to be transparent to the user.

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 common and 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 the worst case, there is always at least one synchronous backup of every piece of data (unless that was explicitly disabled), so the loss of a single member will never cause data loss.

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 require locking between the two copies to prevent the Lost Update problem. The more backup copies are held, the more costly this locking becomes. For this reason, some data structures, such as Hazelcast's implementation of the atomics, do not allow configuration of the replication behavior. Instead, they provide 1 synchronous and 0 asynchronous backups. Finally, ISet and IList do not allow configuration 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.

IN-MEMORY FORMAT FOR IMAP

Hazelcast's IMap provides the ability to store the data in memory in several different ways. By default, Hazelcast serializes objects to byte arrays as they are stored, and deserializes them as they are read, which simplifies synchronization between Hazelcast nodes. However, this is not as performant as some applications would like because a lot of CPU time can be spent serializing and deserializing.

To compensate for this, Hazelcast provides two other possible storage strategies controlled by the in-memory-format setting. They allow values (but not keys) to be stored in the following formats:

1. BINARY (the default): Values are serialized and deserialized immediately.
2. OBJECT: Values are stored locally as objects, and are only serialized when moving to a different node.
3. CACHED: The values are stored in binary-format, but on first access are deserialized and additionally stored as objects.

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, 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 as 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 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.

Hazelcast’s approach to the CAP theorem notes that network partitions caused by total loss of messages are very rare on modern LANs; instead, Hazelcast applies the CAP theorem over wide-area (and possibly geographically diverse) networks.

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 will give up Consistency (“C”) and remain Available (“A”) whilst Partitioned (“P”). However, unlike some NoSQL implementations, C is not given up unless a network partition occurs.

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.

Fortunately, such partitions are very rare in most networks. Since Hazelcast clients are always made aware of a list of nodes they could connect to, in the much more likely event of loss of a datacenter (or region), clients would simply reconnect to unaffected nodes.

HAZELCAST ENTERPRISE

Hazelcast 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 version is called Hazelcast Enterprise. It adds capabilities targeted for enterprise applications, such as:

- Security – for encrypted and secure usage, even across WAN
- Management center – for operations teams
- C# and C++ clients – for interoperability across more than just JVM applications
- Web sessions – for clustering web applications

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.

THE FUTURE

Hazelcast’s model of commercial open source and support for emerging standards (such as JCache) represents a major shift in the distributed computing market, with several other companies in this space emulating Hazelcast’s approach. Hazelcast is committed to open source and has released a number of tools and auxiliary projects, as well as the core engine.

The success of the in-memory compute model for distributed architecture has been felt across the industry, with analysts (including Gartner and Forrester) now reporting this as a separate, quickly growing category, while at the same time forecasting a much smaller market for NoSQL technologies.

The flexibility of Hazelcast technology means that it can even be used in many of the use cases in which developers have used NoSQL. In future releases, Hazelcast will build upon these successes to move towards High-Density Caching and compute. Hazelcast’s use of familiar data structures and patterns make this a very natural way for developers to think about their distributed architectures, without sacrificing reliability, simplicity or performance.

LINKS TO ADDITIONAL RESOURCES

- http://hazelcast.org/download/ – Hazelcast (open source edition) and code samples.
- http://hazelcast.com/ – Hazelcast Enterprise edition and information related to extended feature set and pricing.
- https://github.com/hazelcast/hazelcast – Hazelcast open source code.
- http://hazelcast.org/documentation/ – Hazelcast Reference Manual in HTML and PDF formats, and Hazelcast Javadoc.
- https://groups.google.com/forum/#!forum/hazelcast – Questions/answers and discussions.
- http://stackoverflow.com – Another resource for questions/answers. Just search for “Hazelcast”.

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