HAZELCAST

An Architect's View of the Hazelcast Platform

Introduction to the Hazelcast Platform Architecture

White Paper

White Paper

An Architect's View of the Hazelcast Platform

In this white paper, we discuss the distributed computing and storage technologies that make up the Hazelcast Platform. 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 document supersedes the original "An Architects View of Hazelcast IMDG."

Table of Contents

Introduction

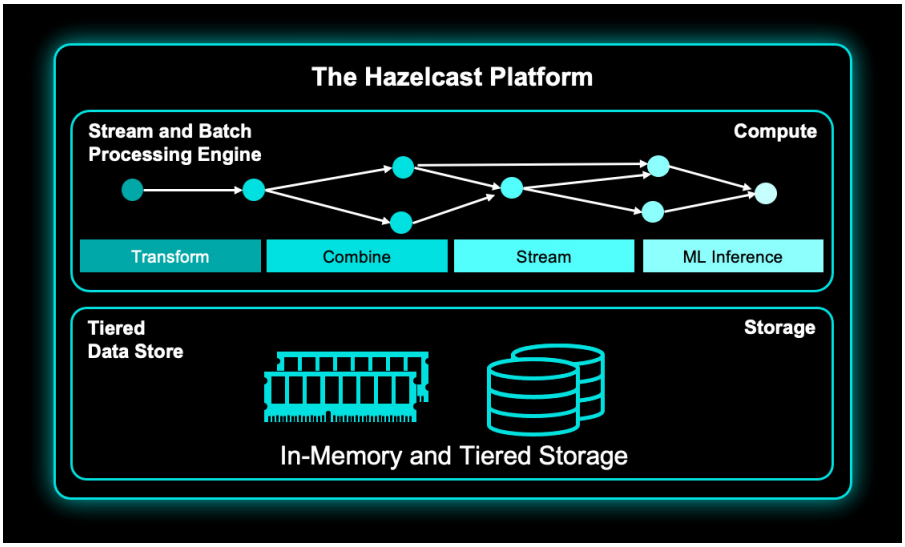
The Hazelcast Platform is the evolution of a distributed computing and storage initiative that has been running since 2008 as an open source project. If you have heard of Hazelcast, it's likely you'll recall it as being an in-memory store and this was one of its early use cases. Its popularity rose from it being a very easy-to-implement solution to the problem of slow databases.

It is a well-regarded piece of software with nearly five thousand stars on GitHub and around 100 million server starts per month at the time of writing. It is used in some of the most business-critical systems in the world, from credit card transaction processing to train monitoring and scheduling. There's a pretty good chance you're interacting with a system built using Hazelcast almost every day.

Since those early days, Hazelcast has become a true distributed compute and storage platform that offers capabilities that would usually require software from multiple projects and vendors to achieve. Today, with the 5.x line of releases, Hazelcast offers technical features such as:

- Distributed stream processing
- SQL for both data-in-motion (aka "streaming data") and data-at-rest
- Tiered storage between a super-fast memory layer and various disk layers
- First-class support for JSON
- A wide variety of data structures such as maps, queues, topics, lists, and sets
- Out-of-the-box integration connectors to Apache Kafka, relational and NoSQL databases, HDFS, and other data platforms
- Support for a wide range of client APIs such as Java, .NET, Python, C++, Go, and Node.js
- Integration with Kubernetes, Docker, and all the major cloud providers
- CP-based subsystem built on RAFT that provides distributed concurrency primitives
- Minimal third-party dependencies make it ideal for embedded and edge computing solutions

Conceptually, Hazelcast combines a batch and stream processing engine with an ultra-low-latency tiered storage layer. This acts as a data platform for compute and data serving to real-time transactional applications. You are free to choose both capabilities or just one of them. For example, you can use Hazelcast purely as a storage layer integrating many outside data sources. Or you could use it as a stream processing engine to provide advanced ETL, moving data from one place to another.



What Makes Hazelcast Special?

The key capability that is quite unique and sets it apart from most data solutions today is Hazelcast's ability to combine compute and data storage in the same cluster processes.

Today, you'll most probably have system architectures that move data from storage to compute and back again, particularly for batch processing systems. But we've seen this pattern for even newer systems where data moves from NoSQL databases to compute processes and then back. Even what is thought of as the cutting edge of event-driven systems still moves data back and forth.

The ability to reduce these "data hops" drives many benefits. Simpler architectures mean lower vendor and hardware costs, less complexity and lower administrative overhead, and much lower latency.

Operational Simplicity

The primary runtime capabilities that the Hazelcast Platform provides include:

- Elasticity
- Redundancy
- High performance

Elasticity means that Hazelcast clusters can grow or reduce capacity on demand, simply by adding or removing nodes. Elasticity has been built into Hazelcast from day one, and it will become obvious to users that scaling a Hazelcast cluster is a far simpler proposition than other data platforms such as Redis or MongoDB.

Redundancy means that data is replicated across the cluster to enable high availability so that any node failure will not result in data loss. You can configure the Hazelcast data replication policy, which defaults to one synchronous backup copy, to adjust for the level of resilience you require.

High performance is achieved through the in-memory architecture, a distributed computing framework, and other optimizations such as green threads which reduce the overhead of operating-system-level context switching. Benchmark results over the past several years have demonstrated superior Hazelcast performance compared to other high-speed data platforms.

To support these capabilities, Hazelcast has a concept of members, which are Hazelcast instances (running in Java virtual machines) that make up a Hazelcast cluster. Members are typically deployed on a one-per-hardware-server basis. A cluster provides a single extended environment where data can be synchronized between (and processed by) members of the cluster.

Containers and the Cloud

Finally, a word about containerization and the cloud. Hazelcast provides first-class support for running clusters in Kubernetes using Helm Charts or our custom operators. Hazelcast also provides official Docker images for the Hazelcast Platform.
https://docs.hazelcast.com/operator/latest/

For the cloud, you can choose to run your own Hazelcast clusters and we provide discovery mechanisms that automatically detect the environment and allow clusters to form in AWS, Azure, and GCP. Or you can make use of Hazelcast Cloud which is a fully managed service run by Hazelcast experts. They take care of the reliability and running of the cluster, freeing you up to develop your applications.

You can find out more at https://cloud.hazelcast.com where you can create a free AWS-based Hazelcast cluster.

Hazelcast clusters can be connected in hybrid or multi-cloud topologies using our Enterprise WAN Replication feature. This is also supported in the Hazelcast Cloud Managed Service.

Technical Use Cases

Hazelcast excels at a number of technical use cases.

- Stream processing
- Low-latency JSON and object data store for applications
- Extract-transform-load (ETL)
- Digital integration hub / materialized views
- Database acceleration
- Batch processing
- Mainframe integration

Getting Started – Simple SQL

A quick way to get started with Hazelcast is to use Docker and the SQL interface to the cluster. So, assuming you have Docker already installed, proceed with the following steps:

1. **Create a new docker network.**

```
docker network create hazelcast-network
```

2. **Start a Hazelcast member.**
   You can start more to form a cluster, but one will do, for now. Remember to update the port numbers if you do.

```
docker run \
    -it \
    --network hazelcast-network \
    --rm \
    -e HZ_CLUSTERNAME=hello-world \
    -p 5701:5701 hazelcast/hazelcast:5.1
```

3. **Start the SQL shell.**
   Replace the `$DOCKER_IP` placeholder with the IP address of your member's Docker container you just started, by checking the console output from step 2.

```
docker run --network hazelcast-network -it --rm hazelcast/hazelcast:5.1 hz-cli
--targets hello-world@$DOCKER_IP sql
```

4. **Create a map and add some data to the map.**

```
CREATE MAPPING my_distributed_map TYPE IMap OPTIONS
('keyFormat'='varchar','valueFormat'='varchar');
SINK INTO my_distributed_map VALUES
('1', 'John'),
('2', 'Mary'),
('3', 'Jane');
```

5. **Now we can select the data from the map.**

```
SELECT * FROM my_distributed_map;
```

This is obviously a very simple example. You can find out more about the SQL capabilities in our documentation.
https://docs.hazelcast.com/hazelcast/latest/sql/sql-overview

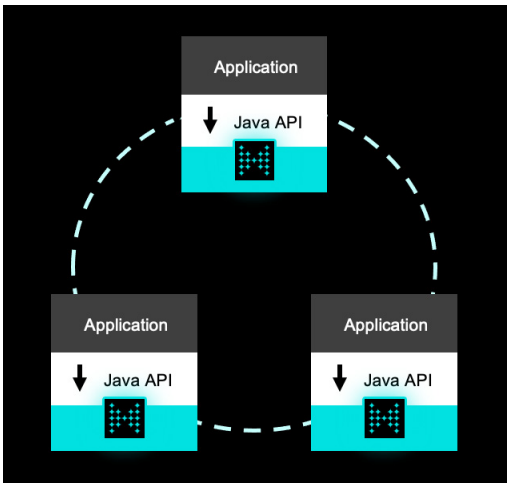
If you don't wish to use the SQL interface, there are always the programmatic APIs to work with the numerous data structures and distribute compute / streaming functionality.

The following documentation shows examples of how to use these APIs directly within applications to access data.
https://docs.hazelcast.com/hazelcast/latest/getting-started/get-started-docker

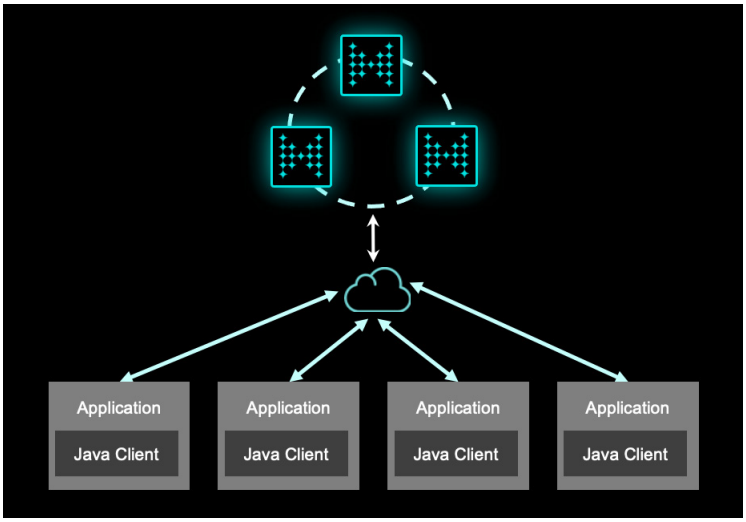
Deployment Topologies

Hazelcast supports two modes of operation: either “embedded” where the applications built on the Hazelcast library connect with each other to form a cluster, or “client-server” in which separate Hazelcast server instances connect to form a cluster. These two approaches to topology are shown in the following diagrams.

Here is the embedded approach:



Here is the client-server topology:



Under most circumstances, we recommend client-server topologies since it provides greater flexibility in terms of cluster mechanics. Member instances can be taken down and restarted without any negative impact on the overall system since the Hazelcast client simply reconnects to another member of the cluster. In other words, client-server topologies isolate application code from purely cluster-level events.

Data Replication within a Cluster

Hazelcast partitions data into local sets and distributes these partitions as evenly as possible across the cluster members, based on a key hash partitioning scheme. Hazelcast clients also contain this partitioning table and so they can send any operation directly to the cluster member that contains the requisite data for the API call, for example, `map.get(key)`.

Hazelcast provides a simple scheme for controlling which partitions data resides in. With this, a developer can choose to ensure the locality of related data. When Hazelcast's compute capability is in use, this extends to provide the capability of sending data processing tasks such as a stream or batch processing job, or a Java **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 and system downtime. This resilience capability is based on intracluster replication that creates replicas known as "backups" in Hazelcast terminology. These backups are copies of data distributed on other Hazelcast members that can be used if the primary data copy is inaccessible due to a failure on the respective member server.

Elasticity features are enabled by an algorithm that will rebalance the partitions when members join or leave a cluster. These operations are performed automatically and there is no need for any manual partitioning intervention.

Hazelcast provides a certain amount of control over how data is replicated. The default is to create one synchronous backup. For data structures such as **Map** and **Queue**, 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) and higher write latency. The asynchronous option is 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 write operations, as the caller waits for backups to be performed. The more backup copies that are held, the more time might be needed to execute those backup operations.

Polyglot Data Support

Hazelcast is an object store and when data is moving from clients to cluster members or between cluster members, it must be serialized before being placed on the network. To provide polyglot data transfer, that is to enable data to be read between different languages, there are two options.

JSON Support

The first and most common option is to represent complex data structures as JSON. The Hazelcast cluster and all client languages recognize JSON data. JSON can be stored as a value in a map using a new `HazelcastJsonValue` class. Hazelcast will recognize this type and enable queries over the properties of the JSON data. Initial benchmarks show a 4x increase in throughput over traditional document stores. The JSON ability is available from all Hazelcast clients including Java, C++, .Net, Python, Node.js, and Go.

Along with JSON support in the API, you can also work with JSON and query it hierarchically using the SQL Interface and JsonPath syntax. For example, given a JSON set of country data, the following query would return a projection of cities per country.

```
SELECT JSON_QUERY(this, '$.countries..city' WITH WRAPPER) FROM countries;
```

For more information see the documentation on working with JSON in SQL.
https://docs.hazelcast.com/hazelcast/latest/sql/working-with-json

Compact Serialization

A more performant and memory-efficient way of storing data structures in Hazelcast is to use the compact serialization features introduced since Hazelcast Platform 5.0. Its main benefits are listed as:

- Separates the schema from the data and stores it per type, not per object which results in less memory and bandwidth usage compared to other formats
- Does not require a class to implement an interface or change the source code of the class in any way
- Supports schema evolution which permits adding or removing fields, or changing the types of fields
- Can work with no configuration or any kind of factory/serializer registration for Java classes and Java records
- Platform and language independent
- Supports partial deserialization of fields, without deserializing the whole objects during queries or indexing

For more information on compact serialization please read our documentation
https://docs.hazelcast.com/hazelcast/latest/serialization/compact-serialization

Note: While there are older serialization options available for Hazelcast, we recommend choosing one of the two described above for future use.

JavaScript Object Support

The Node.js client enables the ability to save JavaScript objects directly to the Hazelcast cluster, and in the same manner as JSON, you will be able to query these on any property of the object using the Hazelcast Predicate API.

SQL and Streaming SQL

The Hazelcast Platform provides an advanced SQL capability. Standard SQL DML is provided that allows not only joining data across different maps in a Hazelcast cluster but also the ability to join these maps to "outside" data sources such as an Apache Kafka topic or data in a file system. We call this our federated query engine.

Hazelcast goes beyond standard SQL that deals with data-at-rest and combines it with the capability to execute SQL commands against data in motion, for example, an event stream from a Kafka topic or an event stream of updates from a relational database.

Where some data integration solutions require all sources to feed data through a Kafka transaction log, Hazelcast allows the potential to leave data in situ and draw down on it when required.

Hazelcast is connected to outside data sources by using the SQL `CREATE MAPPING` command. The following example shows how a Kafka topic can be mapped.

```
CREATE MAPPING my_topic
TYPE Kafka
OPTIONS (
    'keyFormat'='int',
    'valueFormat'='varchar',
    'bootstrap.servers' = '127.0.0.1:9092'
);
```

Once mapped, standard SQL select statements can be written against the mapping `my_topic`, for example, a stream of clicks from a Kafka topic could be joined and enriched with reference data already stored in a Hazelcast map.

Using streaming SQL, it is possible to write jobs that continually update Hazelcast maps with data from a Kafka topic. We can go further and build materialized views using SQL that joins streaming data-in-motion to data-at-rest. This example shows just that:

```
CREATE JOB ingest_trades AS
SINK INTO trade_map
SELECT trades.id, trades.ticker, companies.company, trades.amount
FROM trades
JOIN companies
ON companies.ticker = trades.ticker;
```

In the above example, we are creating a job, this is a piece of streaming SQL that will run continuously in the Hazelcast cluster. It is taking a Kafka stream of trades and writing into a Hazelcast map called `trade_map`. As the stream is running it also enriches it by joining a `companies` Hazelcast map that provides the company name.

To find out more about Hazelcast SQL and streaming SQL read our documentation at:
https://docs.hazelcast.com/hazelcast/latest/sql/sql-overview

Windowing

In stream processing, time-based views of data are very important. Hazelcast provides functionality to add time-based windows with aggregations over these unbounded streams of data.

Concepts such as tumbling, sliding, and session windows are supported, and within these timed windows rolling state can be aggregated. For example, you can the sales of a particular item within a 24-hour period.

To find out more about windowing and event time please read our documentation at:
https://docs.hazelcast.com/hazelcast/latest/pipelines/event-time#time-windowing

Batch and Stream Processing

While the streaming SQL interface is an excellent choice for most ETL and materialized view use cases, for finer-grained control Hazelcast also provides a lower-level Pipeline API with which to build not only stream processing but also batch processing jobs.

This processing engine has proven to provide extremely low latency processing capabilities that far exceed the abilities of similar streaming technologies such as Apache Flink or Apache Spark Streaming. An independent research paper found that Hazelcast outperformed these technologies by a factor of 5000x.
https://dl.acm.org/doi/pdf/10.1145/3427921.3450242

Additionally, a recent Forrester Wave for Streaming Analytics found Hazelcast to have a stronger “current offering” score than the commercial companies offering support for Kafka Streams and Apache Flink amongst others.
https://hazelcast.com/lp/forrester-streaming-analytics-wave-2021/

As mentioned, the Pipeline API allows more custom control over how Hazelcast deals with its input data sources (sources) and then how it passes this data on (sinks). Hazelcast provides an extensive range of connectors that allow integration with the most popular databases, file systems, and streaming sources.
https://hazelcast.com/hub/#type=connector

The following is an example of the Java Pipeline API in action:

```
    Pipeline pipeline = Pipeline.create();
    pipeline.readFrom(TestSources.itemStream(10))
      .withoutTimestamps()
      .filter(event -> event.sequence() % 2 == 0)
      .setName("filter out odd numbers")
      .writeTo(Sinks.logger());

    HazelcastInstance hz = Hazelcast.bootstrappedInstance();

    hz.getJet().newJob(pipeline);
```

In the above example we create a pipeline from a source called `TestSources` that emits its data every ten seconds, filters out the odd numbers, and then writes the output to a sink which in this case is a console logger.

While the Pipeline API is a purely Java-based API, Hazelcast does also allow for different stages of a pipeline to call out to external services using the `mapUsingPython` operator to call out to a Python function found on the attached file system or by using `mapUsingServiceAsync` which provides the ability to call out to a service on the network, for example, `grpc`.

The same capabilities can be applied to finite streams of data, commonly known as batch jobs by using the `BatchStage` functionality.

The entire capabilities of the Hazelcast processing engine are far too wide to cover in this short guide, so please be sure to read more in our documentation here:
https://docs.hazelcast.com/hazelcast/latest/pipelines/overview

Tiered Storage

Hazelcast originally started its life as an in-memory data grid for the acceleration of slow databases. Since those early days, Hazelcast has since added the ability to persist data, most recently with its Hot Restart and Persistence features. These persistence features allowed for an exact mirror of the in-memory store to disk. So, if you had a cluster of a total size of 500GB, that is exactly what would be persisted to disk.

Tiered Storage was introduced in the Hazelcast Platform 5.1 to replace the above-mentioned persistence capabilities. Tiered Storage now offers the ability to grow and persist data without the need for any other external databases. But importantly it also enables the different tiers to differ in size and now it is possible to have durable persistence storage that far exceeds the capacity of the in-memory layer.

What's more, Hazelcast automatically takes care of moving data between the different tiers of storage. For example, if some data is stored in a slow disk-based tier, Hazelcast will retrieve this and place it in the memory tier and onto the application requesting it.

HD Memory

By default, Hazelcast serializes objects to byte arrays as they are stored and deserializes them when they are read. This serialization and deserialization happen 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 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, which is available with the Hazelcast Platform - Enterprise Edition. This allows storing the data in serialized format in native memory, outside of the cluster member JVM, and therefore is not subject to garbage collection.

The net benefit of using **NATIVE** (or HD Memory Store) is that it allows a much bigger in-memory storage per cluster member. As a guide, using HD Memory a cluster member can store somewhere in the region of 200GB, while standard JVM-based GC usually performs in the 10-20GB range. In this way, the number of Hazelcast cluster members can be reduced and potentially with it, hardware instance numbers, maintenance complexity, and costs.

1. **BINARY** (the default): The value is stored in binary format. Every time the value is needed, it will be deserialized unless you are using compact serialization which we described in the prior section.
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 the result of a `map.get(key)` invocation, an extra step of serialization and deserialization is added. Here is why:
    a. `map.put(key, value):` Value gets serialized at the client-side and sent to the Hazelcast server in a byte array. The server receives the binary array, deserializes it, and stores it.
    b. `map.get(key):` The server serializes the object into a binary array and sends it over the wire to the client. The client then deserializes the binary array into a Java object.
3. **NATIVE:** Equivalent to **BINARY**, but this option enables the map to use HD Memory Store, instead of storing values in the 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 queries and other operations they receive.

HD Memory is available as a feature in the Hazelcast Platform Enterprise edition.

Hazelcast and CAP

The often-discussed CAP theorem in distributed database theory states that if you have a network that may drop messages, you cannot have both perfect availability and perfect consistency in the event of a partition. Instead, you must choose one. That is, in the event of a "P" (network partition) in your cluster, you must choose to be either "C" (consistent) or "A" (available).

Split-Brain and CAP

Hazelcast is primarily 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") while 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 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 using this process, 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 synchrony with their primary. Hazelcast deals with such problems with an active anti-entropy mechanism.

Cluster Split-Brain Protection and CAP

Cluster Split-Brain Protection enables you to break the standard implementation of CAP within a Hazelcast cluster by defining a minimum number of member instances required 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 the Hazelcast cluster towards achieving better consistency at the cost of availability 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. The reason is node failures are detected eventually by an internal failure detector implementation. Relatedly, operations are also rejected eventually, just after the failed nodes are detected. For this reason, Hazelcasts Cluster Split-Brain Protection solution offers only a best-effort solution to minimize data loss on network partitioning failures.

Recovery from "P"

Hazelcast recovers from the split-brain issue automatically when the network problem is resolved. When the network recovers and the partitioned clusters begin to see each other, Hazelcast merges the data of the partitioned cluster and forms a single cluster. The merge process runs in the background and the cluster eventually merges all the data of the split clusters.

Note that each cluster may have different versions of the same key in the same map. The destination cluster will decide how to handle the merging entry based on the MergePolicy set for that map. Hazelcast provides some inbuilt merge policies and allows you to create your own merge policy by implementing `com.hazelcast.map. merge.MapMergePolicy`.

CP Subsystem

Hazelcast also provides a CP Subsystem to sit alongside the existing AP structures. CP Subsystem contains new implementations of Hazelcast's concurrency APIs on top of the Raft consensus algorithm. As the name of the module implies, these implementations are CP with respect to the CAP Theorem and they live alongside AP data structures in the same Hazelcast cluster. They maintain linearizability in all cases, including client and server failures, network partitions, and prevent split-brain situations. There has also been a reiteration and clarification of the distributed execution and failure semantics of the APIs, as well as a lot of general improvements to these APIs.

Lastly, a brand-new API has been introduced, `FencedLock`, that covers various failure models that can be faced in distributed environments. Verifying the new CP Subsystem takes place via an extensive Jepsen test suite. Jepsen tools have been used to discover many subtle bugs as development took place, and to verify proper behavior after fixing the bugs. There are now Jepsen tests for `IAtomicLong`, `IAtomicReference`, `ISemaphore`, and `FencedLock`.

Split Brain Resistant Data Structures

Hazelcast also provides other data structures that can be used where split brain is a concern. Conflict-Free Replicated Data Types (CRDTs) are a data structure that can be replicated across multiple members in the Hazelcast cluster, where the replicas can be updated independently and concurrently without coordination between the replicas, and where it is always mathematically possible to resolve inconsistencies which might result after a split-brain event. Hazelcast provides a PN Counter CRDT type that can be incremented and decremented.

Obtaining a unique ID within a distributed system is also problematic and prone to providing duplicates during network partitions, to solve this Hazelcast also provides a `FlakeIdGenerator` which is a coordination-free structure that provides a unique ID even during a network split in the cluster.

For more information on the CP Subsystem please read our documentation.
https://docs.hazelcast.com/hazelcast/latest/cp-subsystem/cp-subsystem

Hazelcast Platform – Enterprise Edition

The Hazelcast Platform is an open source product with additional commercial-only capabilities. The core functionality is free and open source, but many enterprise users require professional support and an extended feature set. The commercial version is called Hazelcast Platform - Enterprise Edition. This offering guarantees the highest levels of performance and support for the ultimate scale-up and scale-out of your applications. It adds capabilities not found in the open source edition which are targeted for enterprise applications, such as the following:

- **HD Memory** – This allows you to scale in-memory data storage capacity up to hundreds of GB of main memory in a single cluster member. HD Memory eliminates garbage collection issues by minimizing pauses caused by GC and delivers scaled-up and predictable performance.
- **Security Suite** – Provides role-based access control (RBAC) to all data structures with connectivity to LDAP and Active Directory. Interoperable encryption, authentication, and access control checks to mission-critical applications as well as TLS and OpenSSL support, with multiple certification zones (member to member, cluster to cluster, client to cluster).
- **Management Center** – Provides a bird's-eye view of all cluster activity through a web-based user interface and cluster-wide JMX and REST APIs. It lets you monitor data structures, stream, and batch processing jobs.
- **Tiered Storage** – Allows clusters to durably store data beyond the limits of memory. Hazelcast Tiered Storage automatically moves data between the different tiers as and when it is needed by your applications. Choose different storage tiers based on speed and cost requirements, such as memory, SSD, or blob/object stores such as Amazon S3.
- **WAN Replication** – Synchronizes multiple Hazelcast clusters in different datacenters for disaster recovery or geographic locality and can be managed centrally through the Management Center.
- **Rolling Upgrades** – This allows you to upgrade your cluster nodes' versions without service interruption.
- **Blue/Green Deployments** – Reduce downtime and risk by running two identical Hazelcast Enterprise clusters called blue and green. One of the clusters provides production services to clients while the other cluster can be upgraded with new application code. Hazelcast Blue/Green Deployment functionality allows clients to be migrated from one cluster to another without client interaction or cluster downtime. All clients of a cluster may be migrated, or groups of clients can be moved with the use of label filtering and block/allow lists.
- **Automatic Disaster Recovery Fail-Over** – Provides clients connected to Hazelcast Platform Enterprise clusters with the ability to automatically fail-over to a disaster recovery cluster should there be an unplanned outage of the primary production Hazelcast cluster.
- **Dynamic Configuration Persistence** – Allows the persistence of any configuration changes made to a cluster while it is operational. While the open source edition does allow configuration changes in a running cluster it does not provide persistence of these. DCP also provides REST API endpoints that allow dynamic reloading of changed configuration from an XML or YAML file.

For both the open source and enterprise versions, the Hazelcast community is well-known for 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.

Links

Participate in the Hazelcast community:
https://hazelcast.com/dev/

Contribute code or report a bug:
GitHub: https://github.com/hazelcast/hazelcast

Download software or sign up for the cloud:
https://hazelcast.com/get-started/

Read our documentation:
https://docs.hazelcast.com/

Join the discussion:
Slack https://slack.hazelcast.com/
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