- Home
- Product
- Services
- Downloads
- Documentation
- Company
- Testimonials

Hazelcast is **free** because it is **priceless** and Hazelcast is **open source** because it is **confident**!

Search

Hazelcast Documentation

Table of Contents

1. Introduction

Hazelcast is a clustering and highly scalable data distribution platform for Java. Hazelcast helps architects and developers to easily design and develop faster, highly scalable and reliable applications for their businesses.

- Distributed implementations of `java.util.{Queue, Set, List, Map}`

- Distributed implementation of `java.util.concurrent.ExecutorService`
- Distributed implementation of `java.util.concurrency.locks.Lock`
- Distributed `Topic` for publish/subscribe messaging
- Transaction support and J2EE container integration via JCA
- Distributed listeners and events
- Support for cluster info and membership events
- Dynamic HTTP session clustering
- Dynamic clustering
- Dynamic scaling to hundreds of servers
- Dynamic partitioning with backups
- Dynamic fail-over
- Super simple to use; include a single jar
- Super fast; thousands of operations per sec.
- Super small; less than a MB
- Super efficient; very nice to CPU and RAM

To install Hazelcast:

- Download hazelcast-_version_.zip from www.hazelcast.com
- Unzip hazelcast-_version_.zip file
- Add hazelcast.jar file into your classpath

Hazelcast is pure Java. JVMs that are running Hazelcast will dynamically cluster. Although by default Hazelcast will use multicast for discovery, it can also be configured to only use TCP/IP for environments where multicast is not available or preferred (Click here for more info). Communication among cluster members is always TCP/IP with Java NIO beauty. Default configuration comes with 1 backup so if one node fails, no data will be lost. It is as simple as using `java.util.{Queue, Set, List, Map}`. Just add the hazelcast.jar into your classpath and start coding.

2. Distributed Data Structures

Common Features of all Hazelcast Data Structures:

Data in the cluster is almost evenly distributed (partitioned) across all nodes. So each node carries ~ (1/n total-data) + backups , n being the number of nodes in the cluster.

- If a member goes down, its backup replica that also holds the same data, will dynamically redistribute the data including the ownership and locks on them to remaining live nodes. As a result, no data will get lost. **When a new node joins the cluster, new node takes ownership(responsibility) and load of -some- of the entire data in the cluster. Eventually the new node will carry almost (1/n** total-data) + backups and becomes the new partition reducing the load on others.

- There is no single cluster master or something that can cause single point of failure. Every node in the cluster has equal rights and responsibilities. No-one is superior. And no dependency on external 'server' or 'master' kind of concept.

Here is how you can retrieve existing data structure instances (map, queue, set, lock, topic, etc.) and how you can listen for instance events to get notified when an instance is created or destroyed.

```
import java.util.Collection;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.Instance;
import com.hazelcast.core.InstanceEvent;
import com.hazelcast.core.InstanceListener;
public class Sample implements InstanceListener {
        public static void main(String[] args) {
                Sample sample = new Sample();
                Hazelcast.addInstanceListener(sample);
                Collection<Instance> instances = Hazelcast.getInstances();
                for (Instance instance : instances) {
                     System.out.println(instance.getInstanceType() + "," + instance.getId());
                }
        }
        public void instanceCreated(InstanceEvent event) {
               Instance instance = event.getInstance();
               System.out.println("Created " + instance.getInstanceType() + "," + instance.getId());
        }
        public void instanceDestroyed(InstanceEvent event) {
              Instance instance = event.getInstance();
              System.out.println("Destroyed " + instance.getInstanceType() + "," + instance.getId());
       }
}
```

2.1. Distributed Queue

Hazelcast distributed queue is an implementation of `java.util.concurrent.BlockingQueue`.

```
import com.hazelcast.core.Hazelcast;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;

BlockingQueue<MyTask> q = Hazelcast.getQueue("tasks");
q.put(new MyTask());
MyTask task = q.take();

boolean offered = q.offer(new MyTask(), 10, TimeUnit.SECONDS);
```

```
task = q.poll (5, TimeUnit.SECONDS);
if (task != null) {
        //process task
}
```

If you have 10 million tasks in your "tasks" queue and you are running that code over 10 JVMs (or servers), then each server carries 1 million task objects (plus backups). FIFO ordering will apply to all queue operations cluster-wide. User objects (such as `MyTask` in the example above), that are (en/de)queued have to be `Serializable`.

Maximum capacity per JVM and the TTL (Time to Live) for a queue can be configured as shown in the example below.

```
<hazelcast>
        ...
        <queue name="tasks">
        <!--
            Maximum size of the queue. When a JVM's local queue size reaches the maximum,
            all put/offer operations will get blocked until the queue size
            of the JVM goes down below the maximum.
            Any integer between 0 and Integer.MAX_VALUE. 0 means Integer.MAX_VALUE. Default is 0.
        -->
        <max-size-per-jvm>10000</max-size-per-jvm>

        <!--
            Maximum number of seconds for each item to stay in the queue. Items that are
            not consumed in <time-to-live-seconds> will get automatically evicted from the queue.
            Any integer between 0 and Integer.MAX_VALUE. 0 means infinite. Default is 0.
        -->
        <time-to-live-seconds>0</time-to-live-seconds>
    </queue>
</hazelcast>
```

As of version 1.9.3, distributed queues are backed by distributed maps. Thus, queues can have custom backup counts and persistent storage. Hazelcast will generate cluster-wide unique id for each element in the queue.

Sample configuration:

```
<hazelcast>
        ...
        <queue name="tasks">
        <!--
            Maximum size of the queue. When a JVM's local queue size reaches the maximum,
            all put/offer operations will get blocked until the queue size
            of the JVM goes down below the maximum.
            Any integer between 0 and Integer.MAX_VALUE. 0 means Integer.MAX_VALUE. Default is 0.
        -->
        <max-size-per-jvm>10000</max-size-per-jvm>

        <!--
            Name of the map configuration that will be used for the backing distributed
            map for this queue.
        -->
        <backing-map-ref>queue-map</backing-map-ref>
    </queue>
```

```
    <map name="queue-map">

        <backup-count>1</backup-count>

        <map-store enabled="true">

            <class-name>com.your,company.storage.DBMapStore</class-name>

            <write-delay-seconds>0</write-delay-seconds>

        </map-store>

        ...

    </map>
</hazelcast>
```

If the backing map has no `map-store` defined then your distributed queue will be in-memory only.

If the backing map has a `map-store` defined then Hazelcast will call your `MapStore` implementation to persist queue elements. Even if you reboot your cluster Hazelcast will rebuild the queue with its content. When implementing a `MapStore` for the backing map, note that type of the`key` is always `Long` and values are the elements you place into the queue. So make sure `MapStore.loadAllKeys` returns `Set<Long>` for instance.

2.2. Distributed Topic

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

Messages are ordered, meaning, listeners(subscribers) will process the messages in the order they are actually published. If cluster member M publishes messages m1, m2, m3...mn to a topic T, then Hazelcast makes sure that all of the subscribers of topic T will receive and process m1, m2, m3...mn in order.

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

public class Sample implements MessageListener {

        public static void main(String[] args) {
                Sample sample = new Sample();
                Topic topic = Hazelcast.getTopic ("default");
                topic.addMessageListener(sample);
                topic.publish ("my-message-object");
        }
```

```
        public void onMessage(Object msg) {
                System.out.println("Message received = " + msg);
        }
}
```

2.3. Distributed Map

Just like queue and set, Hazelcast will partition your map entries; and almost evenly distribute onto all Hazelcast members. Distributed maps have 1 backup (replica-count) by default so that if a member goes down, we don't lose data. Backup operations are synchronous so when a `map.put(key, value)` returns, it is guaranteed that the entry is replicated to one other node. For the reads, it is also guaranteed that `map.get(key)` returns the latest value of the entry. Consistency is strictly enforced.

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

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

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

`Hazelcast.getMap()` actually returns `com.hazelcast.core.IMap` which extends `java.util.concurrent.ConcurrentMap` interface. So methods like `ConcurrentMap.putIfAbsent(key,value)` and `ConcurrentMap.replace(key,value)` can be used on distributed map as shown in the example below.

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

Customer getCustomer (String id) {
        ConcurrentMap<String, Customer> map = Hazelcast.getMap("customers");
        Customer customer = map.get(id);
        if (customer == null) {
                customer = new Customer (id);
                customer = map.putIfAbsent(id, customer);
        }
        return customer;
}

public boolean updateCustomer (Customer customer) {
        ConcurrentMap<String, Customer> map = Hazelcast.getMap("customers");
        return (map.replace(customer.getId(), customer) != null);
}

public boolean removeCustomer (Customer customer) {
    ConcurrentMap<String, Customer> map = Hazelcast.getMap("customers");
```

```
    return map.remove(customer.getId(), customer) );
}
```

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

Also see distribute map internals.

2.3.1. Backups

Hazelcast will distribute map entries onto multiple JVMs (cluster members). Each JVM holds some portion of the data but we don't want to lose data when a member JVM crashes. To provide data-safety, Hazelcast allows you to specify the number of backup copies you want to have. That way data on a JVM will be synchronously copied onto other JVM(s). By default, Hazelcast will have one backup copy.

Backup operations are *synchronous*. When a `map.put(key,value)` call returns, it means entry is updated on the both owner and backup JVMs.

If backup count >= 1, then each member will carry both owned entries and backup copies of other member(s). So for the `map.get(key)` call, it is possible that calling member has backup copy of that key but by default, `map.get(key)` will always read the value from the actual owner of the key for consistency. It is possible to enable backup reads by changing the configuration. Enabling backup reads will give you greater performance.

```
<hazelcast>
    ...
    <map name="default">
        <!--
            Number of backups. If 1 is set as the backup-count for example,
            then all entries of the map will be copied to another JVM for
            fail-safety. Valid numbers are 0 (no backup), 1, 2, 3.
        -->
        <backup-count>1</backup-count>

        <!--
            Can we read the local backup entries? Default value is false for
            strong consistency. Being able to read backup data will give you
            greater performance.
        -->
        <read-backup-data>false</read-backup-data>

        ...

    </map>
</hazelcast>
```

Q. If I have only one backup-copy then, will I always lose data if two JVMs crash at the same time?

Not always. Cluster member list is the same on each member. Hazelcast will backup each member's data onto next members in the member list. Let say you have a cluster with members A, B, C, D, E, F, G and the backup-count is 1, then Hazelcast will copy A's data onto B, B's data onto C... and G's data onto A. If A and B crashes at the same time then you will lose data because B was the backup of A. But A and C crashes at the same time, you won't lose any data because B was the backup of A and D was the backup of C. So you will only lose that if `sequential-JVM-crash-count > backup-count`.

2.3.2. Eviction

Hazelcast also supports policy based eviction for distributed map. Currently supported eviction policies are LRU (Least Recently Used) and LFU (Least Frequently Used). This feature enables Hazelcast to be used as a distributed cache.

If `time-to-live-seconds` is not 0 then entries older than `time-to-live-seconds` value will get evicted, regardless of the eviction policy set.

Here is a sample configuration for eviction:

```
<hazelcast>
    ...
    <map name="default">
        <!--
            Number of backups. If 1 is set as the backup-count for example,
            then all entries of the map will be copied to another JVM for
            fail-safety. Valid numbers are 0 (no backup), 1, 2, 3.
        -->
        <backup-count>1</backup-count>

        <!--
            Maximum number of seconds for each entry to stay in the map. Entries that are
            older than <time-to-live-seconds> and not updated for <time-to-live-seconds>
            will get automatically evicted from the map.
            Any integer between 0 and Integer.MAX_VALUE. 0 means infinite. Default is 0.
        -->
        <time-to-live-seconds>0</time-to-live-seconds>

        <!--
            Maximum number of seconds for each entry to stay idle in the map. Entries that are
            idle(not touched) for more than <max-idle-seconds> will get
            automatically evicted from the map.
            Entry is touched if get, put or containsKey is called.
            Any integer between 0 and Integer.MAX_VALUE.
            0 means infinite. Default is 0.
        -->
        <max-idle-seconds>0</max-idle-seconds>

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

```
        -->
        <eviction-policy>LRU</eviction-policy>

        <!--
            Maximum size of the map. When max size is reached,
            map is evicted based on the policy defined.
            Any integer between 0 and Integer.MAX_VALUE. 0 means
            Integer.MAX_VALUE. Default is 0.
        -->
        <max-size>5000</max-size>

        <!--
            When max. size is reached, specified percentage of
            the map will be evicted. Any integer between 0 and 100.
            If 25 is set for example, 25% of the entries will
            get evicted.
        -->
        <eviction-percentage>25</eviction-percentage>
       <!--
            Specifies when eviction will be started. Default value is 3.
           So every 3 (+up to 5 for performance reasons) seconds
           eviction will be kicked of. Eviction is costly operation, setting
           this number too low, can decrease the performance.
       -->
      <eviction-delay-seconds>3</eviction-delay-seconds>
    </map>
</hazelcast>
```

2.3.3. Persistence

Hazelcast allows you to load and store the distributed map entries from/to a persistent datastore such as relational database.

If a loader implementation is provided, when `get(key)` is called, if the map entry doesn't exist in-memory then Hazelcast will call your loader implementation to load the entry from a datastore.

If a store implementation is provided, when `put(key,value)` is called, Hazelcast will call your store implementation to store the entry into a datastore. Hazelcast can call your implementation to store the entries synchronously (write-through) with no-delay or asynchronously (write-behind) with delay and it is defined by the `write-delay-seconds` value in the configuration.

If it is write-through, when the `map.put(key,value)` call returns, you can be sure that

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

If it is write-behind, when the `map.put(key,value)` call returns, you can be sure that

- In-Memory entry is updated

- In-Memory backup copies are successfully created on other JVMs (if backup-count is greater than 0)
- The entry is marked as `dirty` so that after `write-delay-seconds`, it can be persisted.

Same behavior goes for the `remove(key` and `MapStore.delete(key)`.

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

When write-through is used, Hazelcast will call `MapStore.store(key,value)` and `MapStore.delete(key)` for each entry update. When write-behind is used, Hazelcast will call `MapStore.store(map)`, and `MapStore.delete(collection)` to do all writes in a single call.

Here is a sample configuration:

```
<hazelcast>
    ...
    <map name="default">
        ...
        <map-store enabled="true">
            <!--
               Name of the class implementing MapLoader and/or MapStore.
               The class should implement at least of these interfaces and
               contain no-argument constructor. Note that the inner classes are not supported.
            -->
            <class-name>com.hazelcast.examples.DummyStore</class-name>
            <!--
               Number of seconds to delay to call the MapStore.store(key, value).
               If the value is zero then it is write-through so MapStore.store(key, value)
               will be called as soon as the entry is updated.
               Otherwise it is write-behind so updates will be stored after write-delay-seconds
               value by calling Hazelcast.storeAll(map). Default value is 0.
            -->
            <write-delay-seconds>0</write-delay-seconds>
        </map-store>
    </map>
</hazelcast>
```

As of 1.9.3 `MapLoader` has the new `MapLoader.loadAllKeys` API. It is used for pre-populating the in-memory map when the map is first touched/used. If `MapLoader.loadAllKeys` returns NULL then nothing will be loaded. Your `MapLoader.loadAllKeys` implementation can return all or some of the keys. You may select and return only the `hot` keys, for instance. Also note that this is the fastest way of pre-populating the map as Hazelcast will optimize the loading process by having each node loading owned portion of the entries.

2.3.4. Query

Hazelcast partitions your data and spreads across cluster of servers. You can surely iterate over the map entries and look for certain entries you are interested in but this is not very efficient as you will have to bring entire entry set and iterate locally. Instead, Hazelcast allows you to run distributed queries on your distributed map.

Let's say you have a "employee" map containing values of `Employee` objects:

```
import java.io.Serializable;
public class Employee implements Serializable {
        private String name;
        private int age;
        private boolean active;
        private double salary;
        public Employee(String name, int age, boolean live, double price) {
            this.name = name;
            this.age = age;
            this.active = live;
            this.salary = price;
        }
        public Employee() {
        }
        public String getName() {
            return name;
        }
        public int getAge() {
            return age;
        }
        public double getSalary() {
            return salary;
        }
        public boolean isActive() {
            return active;
        }
}
```

Now you are looking for the employees who are active and with age less than 30. Hazelcast allows you to find these entries in two different ways:

`SqlPredicate` takes regular SQL where clause. Here is an example:

```
import com.hazelcast.core.IMap;
import com.hazelcast.query.SqlPredicate;
IMap map = Hazelcast.getMap("employee");
Set<Employee> employees = (Set<Employee>) map.values(new SqlPredicate("active AND age < 30"));
```

Supported SQL syntax:

- AND/OR
    - `<expression> AND <expression> AND <expression>...`
        - `active AND age>30`
        - `active=false OR age = 45 OR name = 'Joe'`
        - `active AND (age >20 OR salary < 60000)`
- `=, !=, <, <=, >, >=`
    - `<expression> = value`
        - `age <= 30`
        - `name ="Joe"`

        - `salary != 50000`

- BETWEEN
    - `<attribute> [NOT] BETWEEN <value1> AND <value2>`
        - `age BETWEEN 20 AND 33 (same as age >=20 AND age<=33)`
        - `age NOT BETWEEN 30 AND 40 (same as age <30 OR age>40)`
- LIKE
    - `<attribute> [NOT] LIKE 'expression'` `%` (percentage sign) is placeholder for many characters, _ (underscore) is placeholder for only one character.
        - `name LIKE 'Jo%'` (true for 'Joe', 'Josh', 'Joseph' etc.)
        - `name LIKE 'Jo_'` (true for 'Joe'; false for 'Josh')
        - `name NOT LIKE 'Jo_'` (true for 'Josh'; false for 'Joe')
        - `name LIKE 'J_s%'` (true for 'Josh', 'Joseph'; false 'John', 'Joe')
- IN
    - `<attribute> [NOT] IN (val1, val2, ...)`
        - `age IN (20, 30, 40)`
        - `age NOT IN (60, 70)`

Examples:

- `active AND (salary >= 50000 OR (age NOT BETWEEN 20 AND 30))`
- `age IN (20, 30, 40) AND salary BETWEEN (50000, 80000)`

If SQL is not enough or programmable queries are preferred then JPA criteria like API can be used. Here is an example:

```
import com.hazelcast.core.IMap;
import com.hazelcast.query.Predicate;
import com.hazelcast.query.PredicateBuilder;
import com.hazelcast.query.EntryObject;
IMap map = Hazelcast.getMap("employee");
EntryObject e = new PredicateBuilder().getEntryObject();
Predicate predicate = e.is("active").and(e.get("age").lessThan(30));
Set<Employee> employees = (Set<Employee>) map.values(predicate);
```

Hazelcast distributed queries will run on each member in parallel and only results will return the caller. When a query runs on a member, Hazelcast will iterate through the entire owned entries and find the matching ones. Can we make this even faster? Yes by indexing the mostly queried fields. Just like you would do for your database. Of course, indexing will add overhead for each `write` operation but queries will be a lot faster. If you are querying your map a lot then make sure to add indexes for most frequently queried fields. So if your `active and age < 30` query, for example, is used a lot then make sure you add index for `active` and `age` fields. Here is how:

```
IMap imap = Hazelcast.getMap("employees");
imap.addIndex("age", true);        // ordered, since we have ranged queries for this field
imap.addIndex("active", false);    // not ordered, because boolean field cannot have range
```

API `IMap.addIndex(fieldName, ordered)` is used for adding index. For a each indexed field, if you have -ranged- queries such as `age>30`, `age BETWEEN 40 AND 60` then `ordered` parameter should be `true`, otherwise set it to `false`.

2.3.5. Near Cache

Map entries in Hazelcast are partitioned across the cluster. Imagine that you are reading key `k` so many times and `k` is owned by another member in your cluster. Each `map.get(k)` will be a remote operation; lots of network trips.

If you have a map that is read-mostly then you should consider creating a `Near Cache` for the map so that reads can be much faster and consume less network traffic.

All these benefits don't come free. When using near cache, you should consider the following issues:

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

Near cache is highly recommended for the maps that are read-mostly.

Here is a near-cache configuration for a map :

```
<hazelcast>
    ...
    <map name="my-read-mostly-map">
        ...
        <near-cache>
            <!--
                Maximum number of seconds for each entry to stay in the near cache. Entries that are
                older than <time-to-live-seconds> will get automatically evicted from the near cache.
                Any integer between 0 and Integer.MAX_VALUE. 0 means infinite. Default is 0.
            -->
            <time-to-live-seconds>0</time-to-live-seconds>
            <!--
                Maximum number of seconds each entry can stay in the near cache as untouched (not-read).
                Entries that are not read (touched) more than <max-idle-seconds> value will get removed
                from the near cache.
                Any integer between 0 and Integer.MAX_VALUE. 0 means
                Integer.MAX_VALUE. Default is 0.
            -->
            <max-idle-seconds>60</max-idle-seconds>
```

```
            <!--
                Valid values are:
                NONE (no extra eviction, <time-to-live-seconds> may still apply),
                LRU  (Least Recently Used),
                LFU  (Least Frequently Used).
                NONE is the default.
                Regardless of the eviction policy used, <time-to-live-seconds> will still apply.
            -->
            <eviction-policy>LRU</eviction-policy>
            <!--
                Maximum size of the near cache. When max size is reached,
                cache is evicted based on the policy defined.
                Any integer between 0 and Integer.MAX_VALUE. 0 means
                Integer.MAX_VALUE. Default is 0.
            -->
            <max-size>5000</max-size>
            <!--
                Should the cached entries get evicted if the entries are changed (updated or removed).
                true of false. Default is true.
            -->
            <invalidate-on-change>true</invalidate-on-change>
        </near-cache>
    </map>
</hazelcast>
```

2.3.6. Entry Statistics

Hazelcast keeps extra information about each map entry such as creationTime, lastUpdateTime, lastAccessTime, number of hits, version, and this information is exposed to the developer via `IMap.getMapEntry(key)` call. Here is an example:

```
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.MapEntry;

MapEntry entry = Hazelcast.getMap("quotes").getMapEntry("1");
System.out.println ("size in memory  : " + entry.getCost();
System.out.println ("creationTime    : " + entry.getCreationTime();
System.out.println ("expirationTime  : " + entry.getExpirationTime();
System.out.println ("number of hits  : " + entry.getHits();
System.out.println ("lastAccessedTime: " + entry.getLastAccessTime();
System.out.println ("lastUpdateTime  : " + entry.getLastUpdateTime();
System.out.println ("version         : " + entry.getVersion();
System.out.println ("isValid         : " + entry.isValid();
System.out.println ("key             : " + entry.getKey();
System.out.println ("value           : " + entry.getValue();
System.out.println ("oldValue        : " + entry.setValue(newValue);
```

2.4. Distributed MultiMap

`MultiMap` is a specialized map where you can associate a key with multiple values. Just like any other distributed data structure implementation in Hazelcast, `MultiMap` is distributed/partitioned and thread-safe.

```
import com.hazelcast.core.MultiMap;
```

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

// a multimap to hold <customerId, Order> pairs
MultiMap<String, Order> mmCustomerOrders = Hazelcast.getMultiMap("customerOrders");
mmCustomerOrders.put("1", new Order ("iPhone", 340));
mmCustomerOrders.put("1", new Order ("MacBook", 1200));
mmCustomerOrders.put("1", new Order ("iPod", 79));

// get orders of the customer with customerId 1.
Collection<Order> colOrders = mmCustomerOrders.get ("1");
for (Order order : colOrders) {
        // process order
}

// remove specific key/value pair
boolean removed = mmCustomerOrders.remove("1", new Order ("iPhone", 340));
```

2.5. Distributed Set

Distributed Set is distributed and concurrent implementation of `java.util.Set`. Set doesn't allow duplicate elements, so elements in the set should have proper hashCode and equals methods.

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

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

2.6. Distributed List

Distributed List is very similar to distributed set, but it allows duplicate elements.

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

java.util.List list = Hazelcast.getList("IBM-Quote-Frequency");
list.add(new Price(10));
list.add(new Price(11));
list.add(new Price(12));
```

```
list.add(new Price(11));
list.add(new Price(12));

//....
Iterator it = list.iterator();
while (it.hasNext()) {
        Price price = (Price) it.next();
        //analyze
}
```

2.7. Distributed Lock

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

Lock lock = Hazelcast.getLock(myLockedObject);
lock.lock();
try {
    // do something here
} finally {
    lock.unlock();
}
```

`java.util.concurrent.locks.Lock.tryLock()` with timeout is also supported. All operations on the Lock that `Hazelcast.getLock(Object obj)` returns are cluster-wide and Lock behaves just like `java.util.concurrent.lock.ReentrantLock`.

```
if (lock.tryLock (5000, TimeUnit.MILLISECONDS)) {
    try {
       // do some stuff here..
    }
    finally {
      lock.unlock();
    }
}
```

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

2.8. Distributed Events

Hazelcast allows you to register for entry events to get notified when entries added, updated or removed. Listeners are cluster-wide. When a member adds a listener, it is actually registering for events originated in any member in the cluster. When a new member joins, events originated at the new member will also be delivered.

All events are ordered, meaning, listeners will receive and process the events in the order they are actually occurred.

```
import java.util.Queue;
import java.util.Map;
import java.util.Set;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.ItemListener;
import com.hazelcast.core.EntryListener;
import com.hazelcast.core.EntryEvent;

public class Sample implements ItemListener, EntryListener {

	public static void main(String[] args) {
		Sample sample = new Sample();
		Queue queue = Hazelcast.getQueue ("default");
		Map   map   = Hazelcast.getMap   ("default");
		Set   set   = Hazelcast.getSet   ("default");
		//listen for all added/updated/removed entries
		queue.addItemListener(sample, true);
		set.addItemListener  (sample, true);
		map.addEntryListener (sample, true);
		//listen for an entry with specific key
		map.addEntryListener (sample, "keyobj");
	}

	public void entryAdded(EntryEvent event) {
		System.out.println("Entry added key=" + event.getKey() + ", value=" + event.getValue());
	}

	public void entryRemoved(EntryEvent event) {
		System.out.println("Entry removed key=" + event.getKey() + ", value=" + event.getValue()
	}

	public void entryUpdated(EntryEvent event) {
		System.out.println("Entry update key=" + event.getKey() + ", value=" + event.getValue())
	}

	public void entryEvicted(EntryEvent event) {
		System.out.println("Entry evicted key=" + event.getKey() + ", value=" + event.getValue()
	}

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

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

3. Cluster Monitoring Tool

Cluster Monitoring Tool is a simple web application (war file) that enables you to monitor your running Hazelcast cluster. It is distributed with Hazelcast as of 1.8.1 version.

It is not using JMX, so the effect to your running cluster is negligible. You can only monitor cluster that runs the same version of Hazelcast. In order to use it, you should first upgrade to 1.8.1 and then use the tool.

To use it

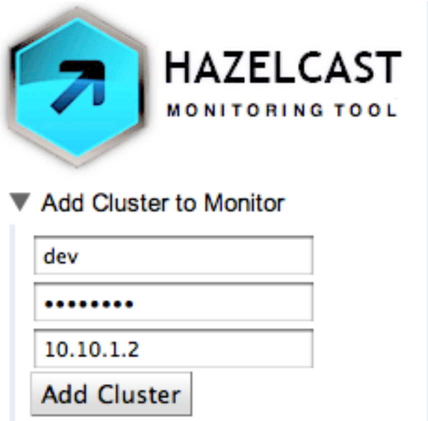
1. Download hazelcast-`<`version`>`.zip.

2. Locate hazelcast-monitor-`<`version`>`.war under main directory of unzipped directory.

3. Deploy it to your web container. For Tomcat, simply copy the war file to webapps directory.

4. Go to http://localhost:8080/hazelcast-monitor-1.8.1/

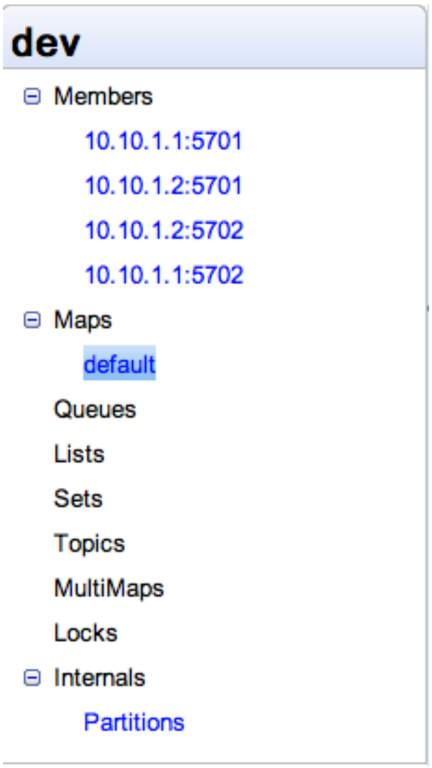
5. You need to add a cluster to monitor.



To add a cluster, type

- cluster-group-name (default is dev),
- cluster-group-password (default is dev-pass)
- ip:port of the one of the members.(ex 10.10.1.1, 10.10.1.2:5702). Default port is 5701.
- and press Add Cluster button.

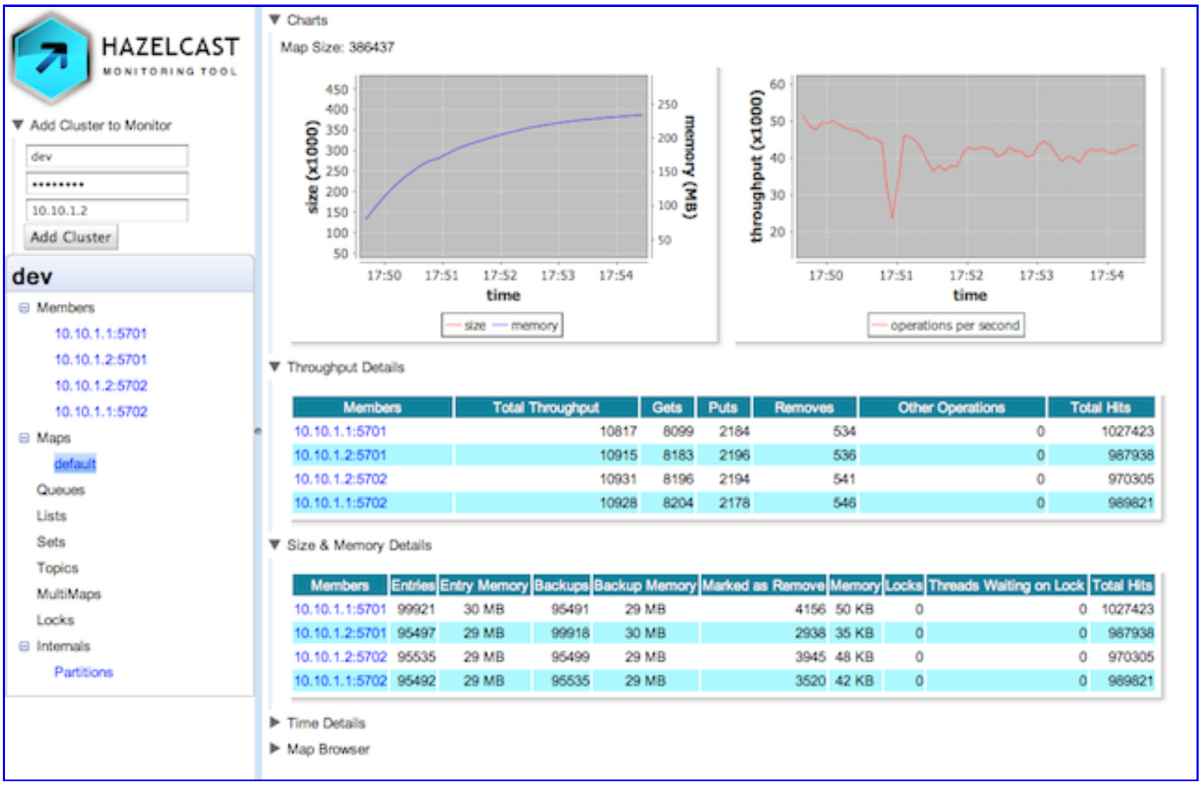
Note that you can add more than one cluster to monitor.



You can see

- Cluster members
- Map, Queue, ... instances. On 1.8.1 only Map instances can be monitored.
- Internals

6. Expand Maps tree and click on any map instance. If you can't see any instance then your cluster does not use any map.



On right side following panels will be displayed

Charts*: When you click on a map instance you will see two charts displaying:

1. Size&Memory
2. Throughput chart

Size& Memory

It shows the number of entries in your map and the memory that those entries occupy. The values are aggregation of all members. To see the details, like how the entries are partitioned among members, see Size&Memory Details panel. _Throughput_

This chart shows how many map operations per second your cluster is doing in total. Again it is aggregation of all members. To see how each member is doing see Throughput Details panel.

Table Panels*: Under charts there is three table panels.

1. Throughput Details
2. Size&Memory Details
3. Times Details

These tables show you the details per member.

Throughput Details

Shows the number of map operations;gets, puts, removes and others per second for each member.

Size&Memory Details

Displays how your entries are partitioned among members. How many entries and backups each member owns. In Hazelcast the preceding member is the backup of the previous. So you can watch how your entries are backed up and what happens when a member leaves the cluster. If your cluster is doing lots of puts and removes, you will notice that the owned and backup numbers will not exactly match. This doesn't mean that Hazelcast does not backup properly. The mismatch happens because tool makes a call to each node, asking their internal details. It is a distributed task and not all members reply at the same time. That's why the numbers may not be exactly same. But they should be very very close.

Also notice the marked as remove part. When you remove an entry, hazelcast will mark it as removed but will delay the actual removal process. Those columns show number of entries waiting for removal and their memory size.

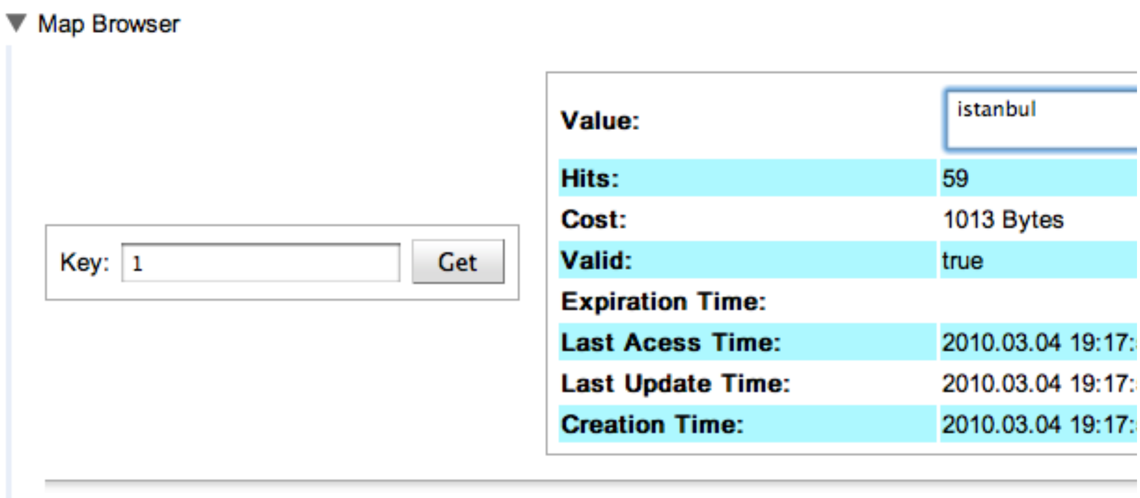
You also can see total number of locked entries on each member and total number of threads at all members waiting for those entries.

Times Details

This table displays several timing details on each member.

***Map Browser**

Enables you to perform get operation on the map. Gives the details of the entry. To browse the map the key of your map have to be String. See the picture for more details.



4. Monitoring with JMX

- Add the following system properties to enable [jmx agent]
    - -Dcom.sun.management.jmxremote
    - -Dcom.sun.management.jmxremote.port=_portNo_ (to specify jmx port) *optional*
    - -Dcom.sun.management.jmxremote.authenticate=false (to disable jmx auth) *optional*
- Enable Hazelcast property *hazelcast.jmx*
    - using Hazelcast configuration (api, xml, spring)
    - or set system property -Dhazelcast.jmx=true
- Use jconsole, jvisualvm (with mbean plugin) or another jmx-compliant monitoring tool.

Following attributes can be monitored:

- Cluster
    - config
    - group name
    - count of members and their addresses (host:port)
    - operations: restart, shutdown cluster
- Member
    - inet address
    - port
    - super client state
- Statistics
    - count of instances
    - number of instances created, destroyed since startup
    - max instances created, destroyed per second
- AtomicNumber
    - name
    - actual value
    - operations: add, set, compareAndSet, reset

- List, Set
  - name
  - size
  - items (as strings)
  - operations: clear, reset statistics
- Map
  - name
  - size
  - operations: clear
- Queue
  - name
  - size
  - received and served items
  - operations: clear, reset statistics
- Topic
  - name
  - number of messages dispatched since creation, in last second
  - max messages dispatched per second

5. Cluster Utilities

5.1. Cluster Interface

Hazelcast allows you to register for membership events to get notified when members added or removed. You can also get the set of cluster members.

```
import com.hazelcast.core.*;

Cluster cluster = Hazelcast.getCluster();
cluster.addMembershipListener(new MembershipListener(){
        public void memberAdded(MembershipEvent membersipEvent) {
                System.out.println("MemberAdded " + membersipEvent);
        }
```

```
        public void memberRemoved(MembershipEvent membersipEvent) {
                System.out.println("MemberRemoved " + membersipEvent);
        }
});

Member localMember  = cluster.getLocalMember();
System.out.println ("my inetAddress= " + localMember.getInetAddress());

Set setMembers  = cluster.getMembers();
for (Member member : setMembers) {
        System.out.println ("isLocalMember " + member.localMember());
        System.out.println ("member.inetaddress " + member.getInetAddress());
        System.out.println ("member.port " + member.getPort());
}
```

5.2. Cluster-wide Id Generator

Hazelcast IdGenerator creates cluster-wide unique IDs. Generated IDs are long type primitive values between 0 and `Long.MAX_VALUE` . Id generation occurs almost at the speed of `AtomicLong.incrementAndGet()` . Generated IDs are unique during the life cycle of the cluster. If the entire cluster is restarted, IDs start from 0 again.

```
import com.hazelcast.core.IdGenerator;
import com.hazelcast.core.Hazelcast;

IdGenerator idGenerator = Hazelcast.getIdGenerator("customer-ids");
long id = idGenerator.newId();
```

5.3. Super Client

Super Clients are members with no storage. If `-Dhazelcast.super.client=true` JVM parameter is set, then the JVM will join the cluster as a 'super client' which will not be a 'data partition' (no data on that node) but will have super fast access to the cluster just like any regular member does.

6. Transactions

6.1. Transaction Interface

Hazelcast can be used in transactional context. Basically start a transaction, work with queues, maps, sets and do other things then commit/rollback in one shot.

```
import java.util.Queue;
```

```
import java.util.Map;
import java.util.Set;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.Transaction;

Queue queue = Hazelcast.getQueue("myqueue");
Map map     = Hazelcast.getMap  ("mymap");
Set set     = Hazelcast.getSet  ("myset");

Transaction txn = Hazelcast.getTransaction();
txn.begin();
try {
        Object obj = queue.poll();
        //process obj
        map.put ("1", "value1");
        set.add ("value");
        //do other things..
        txn.commit();
}catch (Throwable t)  {
        txn.rollback();
}
```

Isolation is always `READ_COMMITTED` . If you are in a transaction, you can read the data in your transaction and the data that is already committed and if not in a transaction, you can only read the committed data. Implementation is different for queue and map/set. For queue operations (offer,poll), offered and/or polled objects are copied to the next member in order to safely commit/rollback. For map/set, Hazelcast first acquires the locks for the write operations (put, remove) and holds the differences (what is added/removed/updated) locally for each transaction. When transaction is set to commit, Hazelcast will release the locks and apply the differences. When rolling back, Hazelcast will simply releases the locks and discard the differences. Transaction instance is attached to the current thread and each Hazelcast operation checks if the current thread holds a transaction, if so, operation will be transaction aware. When transaction is committed, rolled back or timed out, it will be detached from the thread holding it.

6.2. J2EE Integration

Hazelcast can be integrated into J2EE containers via Hazelcast Resource Adapter (hazelcast-ra.rar). After proper configuration, Hazelcast can participate in standard J2EE transactions.

```
<%@page import="javax.resource.ResourceException" %>
<%@page import="javax.transaction.*" %>
<%@page import="javax.naming.*" %>
<%@page import="javax.resource.cci.*" %>
<%@page import="java.util.*" %>
<%@page import="com.hazelcast.core.Hazelcast" %>

<%
UserTransaction txn = null;
Connection conn = null;
Queue queue = Hazelcast.getQueue ("default");
Map map     = Hazelcast.getMap   ("default");
Set set     = Hazelcast.getSet   ("default");
```

```
List list   = Hazelcast.getList  ("default");

try {
        Context context = new InitialContext();
        txn = (UserTransaction) context.lookup("java:comp/UserTransaction");
        txn.begin();

        ConnectionFactory cf = (ConnectionFactory) context.lookup ("java:comp/env/HazelcastCF");
        conn = cf.getConnection();

        queue.offer("newitem");
        map.put ("1", "value1");
        set.add ("item1");
        list.add ("listitem1");

        txn.commit();
} catch (Throwable e) {
        if (txn != null) {
                try{
                        txn.rollback();
                }catch (Exception ix) {ix.printStackTrace();};
        }
        e.printStackTrace();
} finally {
        if (conn != null) {
                try{
                        conn.close();
                }catch (Exception ignored) {};
        }
}
%>
```

6.2.1. Resource Adapter Configuration

Deploying and configuring Hazelcast resource adapter is no different than any other resource adapter since it is a standard JCA resource adapter but resource adapter installation and configuration is container specific, so please consult your J2EE vendor documentation for details. Most common steps are:

1. Add the `hazelcast.jar` to container's classpath. Usually there is a lib directory that is loaded automatically by the container on startup.
2. Deploy `hazelcast-ra.rar`. Usually there is a some kind of deploy directory. Name of the directory varies by container.
3. Make container specific configurations when/after deploying `hazelcast-ra.rar`. Besides container specific configurations, JNDI name for Hazelcast resource is set.
4. Configure your application to use the Hazelcast resource. Updating `web.xml` and/or `ejb-jar.xml` to let container know that your application will use the Hazelcast resource and define the resource reference.
5. Make container specific application configuration to specify JNDI name used for the resource in the application.

6.2.2. Sample Glassfish v3 Web Application Configuration

1. Place the `hazelcast-<version>.jar` into `GLASSFISH_HOME/glassfish/domains/domain1/lib/ext/` directory.
2. Place the `hazelcast-ra-<version>.rar` into `GLASSFISH_HOME/glassfish/domains/domain1/autodeploy/` directory
3. Add the following lines to the `web.xml` file.

```
<resource-ref>
  <res-ref-name>HazelcastCF</res-ref-name>
  <res-type>com.hazelcast.jca.ConnectionFactoryImpl</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Notice that we didn't have to put `sun-ra.xml` into the rar file because it comes with the `hazelcast-ra-<version>.rar` file already.

If Hazelcast resource is used from EJBs, you should configure `ejb-jar.xml` for resource reference and `JNDI` definitions, just like we did for `web.xml`.

6.2.3. Sample JBoss Web Application Configuration

- Place the `hazelcast-<version>.jar` into `JBOSS_HOME/server/deploy/default/lib` directory.
- Place the `hazelcast-ra-<version>.rar` into `JBOSS_HOME/server/deploy/default/deploy` directory
- Create a `hazelcast-ds.xml` at `JBOSS_HOME/server/deploy/default/deploy` directory containing the following content. Make sure to set the `rar-name` element to `hazelcast-ra-<version>.rar`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE connection-factories
    PUBLIC "-//JBoss//DTD JBOSS JCA Config 1.5//EN"
    "http://www.jboss.org/j2ee/dtd/jboss-ds_1_5.dtd">
<connection-factories>
   <tx-connection-factory>
          <local-transaction/>
          <track-connection-by-tx>true</track-connection-by-tx>
        <jndi-name>HazelcastCF</jndi-name>
        <rar-name>hazelcast-ra-<version>.rar</rar-name>
        <connection-definition>
             javax.resource.cci.ConnectionFactory
        </connection-definition>
    </tx-connection-factory>
</connection-factories>
```

- Add the following lines to the `web.xml` file.

```
<resource-ref>
  <res-ref-name>HazelcastCF</res-ref-name>
  <res-type>com.hazelcast.jca.ConnectionFactoryImpl</res-type>
```

```
    <res-auth>Container</res-auth>
  </resource-ref>
```

- Add the following lines to the `jboss-web.xml` file.

```
<resource-ref>
  <res-ref-name>HazelcastCF</res-ref-name>
  <jndi-name>java:HazelcastCF</jndi-name>
</resource-ref>
```

If Hazelcast resource is used from EJBs, you should configure `ejb-jar.xml` and `jboss.xml` for resource reference and `JNDI` definitions.

7. Distributed Executor Service

One of the coolest new futures of Java 1.5 is the Executor framework, which allows you to asynchronously execute your tasks, logical units of works, such as database query, complex calculation, image rendering etc. So one nice way of executing such tasks would be running them asynchronously and doing other things meanwhile. When ready, get the result and move on. If execution of the task takes longer than expected, you may consider canceling the task execution. In Java Executor framework, tasks are implemented as `java.util.concurrent.Callable` and `java.util.Runnable`.

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

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

       public Echo() {
       }

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

       public String call() {
               return Hazelcast.getCluster().getLocalMember().toString() + ":"
                               + input;
       }
}
```

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

Executing a task by using executor framework is very straight forward. Simply obtain a `ExecutorService` instance, generally via `Executors` and submit the task which returns a `Future`. After executing task, you don't have to wait for execution to complete, you can process other things and when ready use the future object to retrieve the result as show in code below.

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

7.1. Distributed Execution

Distributed executor service is a distributed implementation of java.util.concurrent.ExecutorService. It allows you to execute your code in cluster. In this chapter, all the code samples are based on the Echo class above. Please note that Echo class is `Serializable` .

You can ask Hazelcast to execute your code (`Runnable`, `Callable`):

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

```
import com.hazelcast.core.Member;
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.MultiTask;
import com.hazelcast.core.DistributedTask;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.FutureTask;
import java.util.concurrent.Future;
import java.util.Set;
public void echoOnTheMember(String input, Member member) throws Exception {
       FutureTask<String> task = new DistributedTask<String>(new Echo(input), member);
       ExecutorService executorService = Hazelcast.getExecutorService();
       executorService.execute(task);
       String echoResult = task.get();
}
public void echoOnTheMemberOwningTheKey(String input, Object key) throws Exception {
       FutureTask<String> task = new DistributedTask<String>(new Echo(input), key);
       ExecutorService executorService = Hazelcast.getExecutorService();
       executorService.execute(task);
       String echoResult = task.get();
}
public void echoOnSomewhere(String input) throws Exception {
       ExecutorService executorService = Hazelcast.getExecutorService();
       Future<String> task = executorService.submit(new Echo(input));
       String echoResult = task.get();
}
public void echoOnMembers(String input, Set<Member> members) throws Exception {
       MultiTask<String> task = new MultiTask<String>(new Echo(input), members);
       ExecutorService executorService = Hazelcast.getExecutorService();
       executorService.execute(task);
       Collection<String> results = task.get();
}
```

Note that you can obtain the set of cluster members via `Hazelcast.getCluster().getMembers()` call.

You can also extend the `MultiTask` class to override `set(V result)`, `setException(Throwable exception)`, `done()` methods for custom behaviour.

Just like `java.util.concurrent.FutureTask.get()` , `MultiTask.get()` will throw `java.util.concurrent.ExecutionException` if any of the executions throws exception.

7.2. Key based Distributed Executions

Hazelcast has a standard way of finding out which member owns/manages each key object. Following operations will be routed to the same member, since all of them are operating based on the same key, "key1".

```
Hazelcast.getMap("mapa").put("key1", value);
Hazelcast.getMap("mapb").get("key1");
Hazelcast.getMap("mapc").remove("key1");
// since map names are different, operation will be manipulating
// different entries, but the operation will take place on the
// same member since the keys ("key1") are the same
Hazelcast.getLock ("key1").lock();
// lock operation will still execute on the same member of the cluster
// since the key ("key1") is same
Hazelcast.getExecutorService().execute(new DistributedTask(runnable, "key1"));
// distributed execution will execute the 'runnable' on the same member
// since "key1" is passed as the key.
```

Let say you have a customers map where customerId is the key and the customer object is the value. and customer object contains the customer's orders. and let say you want to remove one of the orders of a customer and return the number of remaining orders.

Here is how you would normally do it:

```
public static int removeOrder(long customerId, long orderId) throws Exception {
                IMap<Long, Customer> mapCustomers = Hazelcast.getMap("customers");
                mapCustomers.lock (customerId);
                Customer customer = mapCustomers. get(customerId);
                customer.removeOrder (orderId);
                mapCustomers.put(customerId, customer);
                mapCustomers.unlock(customerId);
                return customer.getOrderCount();
}
```

There are couple of things we should consider:

1. There are four distributed operations there.. lock, get, put, unlock.. Can we reduce the number of distributed operations?
2. Customer object may not be that big but can we not have to pass that object through the wire? Notice that, we are actually passing customer object through the wire twice; get and put.

So instead, why not moving the computation over to the member (JVM) where your customer data actually is.

Here is how you can do this with distributed executor service:

1. Send a `Callable` task to the member owning the key, clusterId.
2. `Callable` does the deletion of the order right there and returns with the remaining order count.
3. Upon completion of the `Callable` task, return the result (remaining order count).

Plus you do not have to wait until the the task complete; since distributed executions are asynchronous, you can do other things meanwhile.

here is a sample code:

```
public static int removeOrder(long customerId, long orderId) throws Exception {
          ExecutorService es = Hazelcast.getExecutorService();
          FutureTask<Integer> task = new DistributedTask<Integer>(new OrderDeletionTask(customerId, orc
          es.execute(task);
          int remainingOrders = task.get();
          return remainingOrders;
}
public static class OrderDeletionTask implements Callable<Integer>, Serializable {
          private static final long serialVersionUID = 1L;
          private long customerId;
          private long orderId;
                 public OrderDeletionTask() {
          }
          public OrderDeletionTask(long customerId, long orderId) {
                 super();
                 this.customerId = customerId;
                 this.orderId = orderId;
          }
          public Integer call () {
                 IMap<Long, Customer> mapCustomers = Hazelcast.getMap("customers");
                 mapCustomers.lock (customerId);
                 Customer customer = mapCustomers. get(customerId);
                 customer.removeOrder (orderId);
                 mapCustomers.put(customerId, customer);
                 mapCustomers.unlock(customerId);
                 return customer.getOrderCount();
          }
}
```

Benefits of doing the same operation with `DistributedTask` based on the key are:

1. Only one distributed execution (`es.execute(task)`), instead of four.
2. Less data sent over the wire.
3. Since lock/update/unlock cycle is done locally (local to the customer data), lock duration for the `Customer` entry is much less so enabling higher concurrency.

7.3. Execution Cancellation

What if the code you execute in cluster takes longer than acceptable. If you cannot stop/cancel that task it will keep eating your resources. Standard Java executor framework solves this problem with by introducing `cancel()` api and 'encouraging' us to code and design for cancellations, which is highly ignored part of software development.

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

    public Fibonacci() {
    }

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

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

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

The callable class above calculates the fibonacci number for a given number. In the calculate method, we are checking to see if the current thread is interrupted so that code can be responsive to cancellations once the execution started.

Following `fib()` method submits the Fibonacci calculation task for number 'n' and waits maximum 3 seconds for result. If the execution doesn't complete in 3 seconds, `future.get()` will throw `TimeoutException` and upon catching it we interruptibly cancel the execution for saving some CPU cycles.

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

`fib(20)` will probably will take less than 3 seconds but `fib(50)` will take way longer. (This is not the example for writing better fibonacci calculation code but for showing how to cancel a running execution that takes too long.)

`future.cancel(false)` can only cancel execution before it is running (executing) but `future.cancel(true)` can interrupt running executions if your code is able to handle the interruption. So if you are willing to be able to cancel already running task then your task has to be designed to handle interruption.

If `calculate (int n)` method didn't have if `(Thread.currentThread().isInterrupted())` line, then you wouldn't be able to cancel the execution after it started.

7.4. Execution Callback

`ExecutionCallback` allows you to asynchronously get notified when the execution is done. When implementing `ExecutionCallback.done(Future)` method, you can check if the task is already cancelled.

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

ExecutorService es = Hazelcast.getExecutorService();
DistributedTask<String> task = new DistributedTask<String>(new Fibonacci<Long>(10));
task.setExecutionCallback(new ExecutionCallback<Long> () {
        public void done (Future<Long> future) {
                try {
                        if (! future.isCancelled()) {
                                System.out.println("Fibonacci calculation result = " + future.get());
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
});
es.execute(task);
```

You could have achieved the same results by extending `DistributedTask` and overriding the `DistributedTask.done()` method.

```
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.DistributedTask;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;

ExecutorService es = Hazelcast.getExecutorService();
es.execute(new DistributedTask<String>(new Fibonacci<Long>(10)) {
        public void done () {
                try {
                        if (! isCancelled()) {
                                System.out.println("Fibonacci calculation result = " + get());
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
});
```

8. Http Session Clustering with HazelcastWM

- Target application or web server should support Java 1.5+
- Target application or web server should support Servlet 2.3+ spec
- Session objects that needs to be clustered have to be Serializable

To cluster your web application, all you have to do is to use `clusterWebapp.bat` or `clusterWebapp.sh` scripts that comes with the Hazelcast distribution.

e.g

`clusterWebapp.bat|sh` will not modify the war/ear files passed in. Instead, it will create a new war/ear file and name it as `clustered-<app-file-name/>`.

You can pass multiple war and ear files. Besides if you pass `-apps-sharing-sessions` as an argument then your applications will be able to share the same sessions. If `-apps-sharing-sessions` argument is used, `hazelcast.jar` file will not be placed into the war/ear files so you need to add `hazelcast.jar` into your application server's classpath.

8.1. How It Works

To understand how Hazelcast achieves clustering your application, we will go over what clusterWebapp. bat|sh script does and what happens at runtime. This could be a summary of how HazelcastWM works but some parts of the process are skipped for brevity.

Hazelcast Webapp Manager will read your web.xml files and foreach web.xml file

- Adds a HazelcastWM Filter
- Removes all listener definitions and puts them into the HazelcastWM Filter definition so that HazelcastWM will pass all corresponding events to those listeners.
- Adds a HazelcastWM listener that implements `ServletContextListener` and `ServletContextAttributeListener` interfaces. HazelcastWM listener will listen for the actual servlet container events and deligates them to the application's listeners.
- Replaces each Servlet definition with Hazelcast servlet wrapper.
- Puts `<%@ page extends="com.hazelcast.web.JspWrapper" %>` at the top of every JSP.

When a request hits the servlet container, servlet container will pass the request to HazelcastWM Filter and it will first find out if the request is already associated with a valid session. Then it wraps the request and response objects and fires

`ServletRequestEvent` if necessary and pass the wrapped request/response through the filter chain by invoking `chain.doFilter(wrappedRequest, wrappedResponse)`.

Throughout the chain, request might go through servlets, jsps, other filters. During the processing of the request, as attributes are added/removed to/from the request, HazelcastWM will fire corresponding events. When `request.getSession(true)` is called, then Hazelcast `RequestWrapper` object will return a `HttpSession` instance created by HazelcastWM. Servlet container never creates or know anything about the sessions. HazelcastWM will manage the sessions instead.As attributes added/removed to/from the session, HazelcastWM will fire corresponding events, dictated by the Servlet spec.

When request is forwarded to resource or include a response from another resource via `RequestDispatcher`, HazelcastWM will wrap the dispatcher and when the wrapped dispatcher calls forward or include, it will hand container's original request/response objects over to the container's original dispatcher. That way container can work with its own request/response object, not with the wrapped ones.

After the chain is completed, request will come back to the filter again. Filter will check if request is in a valid session again. If it is then it will go over the Serializable session attributes and persist them on the Hazelcast cluster.

When requested is dispatched, container may not have to call the filters again. So HazelcastWM makes sure that the resource (JSP or Servlet) that the request is forwarded or included will get the original request/response, not the wrapped request/response. This is the reason why all application JSP and Servlets are also wrapped by HazelcastWM. That way when the `service(ServletRequest req, ServletResponse res)` method called on the wrapper servlet or JSP, it will delegate that call to your application JSP and Servlets with the wrapped request/request objects.

For the containers that support Servlet 2.4 or higher, this is no need to wrap JSP and Servlets as Servlet 2.4 spec let us tell container that it should go through filter when dispatching by using the `<dispatcher>FORWARD</dispatcher>` and `<dispatcher>INCLUDE</dispatcher>` elements when configuring filters.

During that life-cycle of the application, HazelcastWM will collect statistics like response times, session count etc. and pass these metrics onto `SnapshotListener`s in the form of `SnapshotEvent`. Also note that regardless of the number of jsps and/or servlets a request goes through via `RequestDispatcher.forward` or `RequestDispatcher.include` calls, it is counted as a single request.

8.2. Snapshot Events

Hazelcast Webapp Manager (HazelcastWM) lets you get statistics on your running application so that you can monitor the health of your application. HazelcastWM collects statistics like number of created/destroyed sessions, average session data size, number of requests, average response time, min. and max. response times for a given time interval (x seconds). So it is like taking a snapshot of the running application every x seconds and passing it to the listeners as a SnapshotEvent . Here is a sample servlet that listens for snapshot events.

```
public class MonitoringServlet extends HttpServlet implements SnapshotListener {
                public void init(ServletConfig config) throws ServletException {
                super.init(config);
                ServletContext servletContext = config.getServletConfig();
                Context hazelcastContext = (Context) servletContext.getAttribute(Context.ATTRIBUTE_NAME)
                hazelcastContext.addSnapshotListener(this);
        }
        public void service(ServletRequest req, ServletResponse res) throws IOException, ServletExceptic
                //do something here
        }
        public void handleSnapshot (SnapshotEvent snapshot) {
                System.out.println("===== Here are the metrics of the last 30 seconds =====");
                System.out.println(snapshot);
                System.out.println("ave.      response time : " + snapshot.getAverageResponseTime());
                System.out.println("min.      response time : " + snapshot.getMinResponseTime());
                System.out.println("max.      response time : " + snapshot.getMaxResponseTime());
                System.out.println("# of           requests : " + snapshot.getNumberOfRequests());
                System.out.println("# of       new sessions : " + snapshot.getCreatedSessionCount());
                System.out.println("# of destroyed sessions : " + snapshot.getDestroyedSessionCount());
                System.out.println("=======================================================");
        }
}
```

8.3. Tested Web Containers

Hazelcast Webapp Manager is designed to work with any application server that supports Servlet spec. 2.3 or higher and constantly tested on following containers:

- Tomcat5.0/5.5/6.0
- Jetty 5.1.12/5.1.14/6.1.10/6.1.11
- Weblogic 9.0/10
- Websphere 6.1
- GlassfishV3
- Caucho Resin 3.0.26

Any application server that uses Tomcat or Jetty as its container such as JBoss and Geronimo should work just fine.

9. Encryption

Hazelcast allows you to encrypt entire socket level communication among all Hazelcast members. Encription is based on Java Cryptography Architecture and both symmetric and asymmetric encryption are supported.

In symmetric encryption, each node uses the same key, so the key is shared.

Here is a sample configuration for symmetric encryption:

```
<hazelcast>
    ...
    <network>
        ...
        <!--
            Make sure to set enabled=true
            Make sure this configuration is exactly the same on
            all members
        -->
        <symmetric-encryption enabled="true">
            <!--
               encryption algorithm such as
               DES/ECB/PKCS5Padding,
               PBEWithMD5AndDES,
               Blowfish,
               DESede
            -->
            <algorithm>PBEWithMD5AndDES</algorithm>
            <!-- salt value to use when generating the secret key -->
            <salt>thesalt</salt>
            <!-- pass phrase to use when generating the secret key -->
            <password>thepass</password>
            <!-- iteration count to use when generating the secret key -->
            <iteration-count>19</iteration-count>
        </symmetric-encryption>
    </network>
    ...
</hazelcast>
```

In asymmetric encryption, public and private key pair is used. Data is encrypted with one of these keys and decrypted with the other.

The idea is that each node has to have its own private key and other trusted members' public key. So that means, for each member, we should do the followings:

- Pick a unique name for the member. We will use the name as the key alias. Let's name them as member1, member2...memberN.
- Generate the keystore and the private key for the member1. `keytool -genkey -alias member1 -keyalg RSA -keypass thekeypass -keystore keystore -storetype JKS` Remember all the parameters you used here because you will need this information when you configure asymmetric-encryption in your hazelcast.xml file.
- Create a public certificate file so that we can add it to the other members' keystore `keytool -export -alias member1 -keypass thekeypass -storepass thestorepass -keystore keystore -rfc -file member1.cer`
- Now take all the other members' public certificates, and add (import) them into member1's keystore

You should repeat these steps for each trusted member in your cluster.

Here is a sample configuration for asymmetric encryption:

```
<hazelcast>
    ...
    <network>
        ...
        <!--
            Make sure to set enabled=true
        -->
        <asymmetric-encryption enabled="true">
            <!-- encryption algorithm -->
            <algorithm>RSA/NONE/PKCS1PADDING</algorithm>
            <!-- private key password -->
            <keyPassword>thekeypass</keyPassword>
            <!-- private key alias -->
            <keyAlias>member1</keyAlias>
            <!-- key store type -->
            <storeType>JKS</storeType>
            <!-- key store password -->
            <storePassword>thestorepass</storePassword>
            <!-- path to the key store -->
            <storePath>keystore</storePath>
        </asymmetric-encryption>
    </network>
    ...
</hazelcast>
```

10. Configuration

Before configuring Hazelcast, please try to work with default configuration to see if it works for you. Default should be just fine for most of the users. If not, then consider custom configuration for your environment.

Hazelcast will look into two places for the configuration file, if there is any:

- System property Hazelcast will first check if "hazelcast.config" system property is set to a file path. Example: `-Dhazelcast.config=C:/myhazelcast.xml`.
- Classpath or inside `hazelcast.jar` If config file is not set as a system property, Hazelcast will use the `hazelcast.xml` that `hazelcast.jar` contains. You may prefer to update the hazelcast.xml file located inside the `hazelcast.jar`.

If Hazelcast doesn't find any config file, it will happily start with default configuration.

```
<hazelcast>
    <group>
        <name>dev</name>
        <password>dev-pass</password>
    </group>
    <network>
        <port auto-increment="true">5701</port>
        <join>
            <multicast enabled="true">
                <multicast-group>224.2.2.3</multicast-group>
                <multicast-port>54327</multicast-port>
            </multicast>
            <tcp-ip enabled="false">
                <interface>192.168.1.2</interface>
```

```
            </tcp-ip>
        </join>
        <interfaces enabled="false">
            <interface>10.3.17.*</interface>
        </interfaces>
    </network>
    <executor-service>
        <core-pool-size>16</core-pool-size>
        <max-pool-size>64</max-pool-size>
        <keep-alive-seconds>60</keep-alive-seconds>
    </executor-service>
    <queue name="default">
        <!--
            Maximum size of the queue. When a JVM's local queue size reaches the maximum,
            all put/offer operations will get blocked until the queue size
            of the JVM goes down below the maximum.
            Any integer between 0 and Integer.MAX_VALUE. 0 means
            Integer.MAX_VALUE. Default is 0.
        -->
        <max-size-per-jvm>10000</max-size-per-jvm>
        <!--
            Maximum number of seconds for each item to stay in the queue. Items that are
            not consumed in <time-to-live-seconds> will automatically
            get evicted from the queue.
            Any integer between 0 and Integer.MAX_VALUE. 0 means
            infinite. Default is 0.
        -->
        <time-to-live-seconds>0</time-to-live-seconds>
    </queue>
    <map name="default">
        <!--
            Number of backups. If 1 is set as the backup-count for example,
            then all entries of the map will be copied to another JVM for
            fail-safety. Valid numbers are 0 (no backup), 1, 2, 3.
        -->
        <backup-count>1</backup-count>
        <!--
            Valid values are:
            NONE (no eviction),
            LRU (Least Recently Used),
            LFU (Least Frequiently Used).
            NONE is the default.
        -->
        <eviction-policy>NONE</eviction-policy>
        <!--
            Maximum size of the map. When max size is reached,
            map is evicted based on the policy defined.
            Any integer between 0 and Integer.MAX_VALUE. 0 means
            Integer.MAX_VALUE. Default is 0.
        -->
        <max-size>0</max-size>
        <!--
            When max. size is reached, specified percentage of
            the map will be evicted. Any integer between 0 and 100.
            If 25 is set for example, 25% of the entries will
            get evicted.
        -->
        <eviction-percentage>25</eviction-percentage>
    </map>
</hazelcast>
```

10.1. Configuring Hazelcast for full TCP/IP cluster

If multicast is not prefered way of discovery for your environment, then you can configure Hazelcast for full TCP/IP cluster. As configuration below shows, while `enable` attribute of `multicast` is set to false, `tcp-ip` has to be set to true. For the none-multicast option, all or subset of cluster members' hostnames and/or ip addreses must be listed. Note that all of the cluster members don't have to be listed there but at least one of them has to be active in cluster when a new member joins.

```
<hazelcast>
        ...
        <network>
                <port auto-increment="true">5701</port>
                <join>
                        <multicast enabled="false">
                                <multicast-group>224.2.2.3</multicast-group>
                                <multicast-port>54327</multicast-port>
                        </multicast>
                        <tcp-ip enabled="true">
                                <hostname>machine1</hostname>
                                <hostname>machine2</hostname>
                                <hostname>machine3:5799</hostname>
                                <interface>192.168.1.0-7</interface>
                                <interface>192.168.1.21</interface>
                        </tcp-ip>
                </join>
                ...
        </network>
        ...
</hazelcast>
```

10.2. Creating Separate Clusters

By specifying group-name and group-password, you can separate your clusters in a simple way. dev group, production group, test group, app-a group etc... a jvm can only participate in one group and it only joins to its own group, does not mess with others.

```
<hazelcast>
        <group>
                <name>dev</name>
                <password>dev-pass</password>
        </group>
        ...

</hazelcast>
```

10.3. Specifying network interfaces

You can also specify which network interfaces that Hazelcast should use. Servers mostly have more than one network interface so you may want to list the valid IPs.

Range characters ('`*`' and '-') can be used for simplicity. So 10.3.10.`*`, for instance, refers to IPs between 10.3.10.0 and 10.3.10.255. Interface 10.3.10.4-18 refers to IPs between 10.3.10.4 and 10.3.10.18 (4 and 18 included). If network interface configuration is enabled (disabled by default) and if Hazelcast cannot find an matching interface, then it will print a message on console and won't start on that node.

```
<hazelcast>
        ...
        <network>
                ....
                <interfaces enabled="true">
                        <interface>10.3.16.*</interface>
                        <interface>10.3.10.4-18</interface>
                        <interface>192.168.1.3</interface>
                </interfaces>
        </network>
        ...
</hazelcast>
```

10.4. Network Partitioning (Split-Brain Syndrome)

Imagine that you have 10-node cluster and for some reason the network is divided into two in a way that 4 servers cannot see the other 6. As a result you ended up having two separate clusters; 4-node cluster and 6-node cluster. Members in each sub-cluster are thinking that the other nodes are dead even though they are not. This situation is called Network Partitioning (aka Split-Brain Syndrome).

Since it is a network failure, there is no way to avoid it programatically and your application will run as two separate independent clusters but we should be able answer the following questions: "What will happen after the network failure is fixed and connectivity is restored between these two clusters? Will these two clusters merge into one again? If they do, how are the data conflicts resolved, because you might end up having two different values for the same key in the same map?"

Here is how Hazelcast deals with it:

1. The oldest member of the cluster checks if there is another cluster with the same group-name and group-password in the network.

2. If the oldest member founds such cluster, then figures out which cluster should merge to the other.

3. Each member of the merging cluster will do the followings

    - pause (`HazelcastInstance.getLifecycleService().pause()`)

    - take locally owned map entries

    - close all its network connections (detach from its cluster)

- join to the new cluster
- send merge request for each its locally owned map entry
- resume (`HazelcastInstance.getLifecycleService().resume()`)

So each member of the merging cluster is actually rejoining to the new cluster and sending merge request for each its locally owned map entry.

Q: Which cluster will merge into the other?

A. Smaller cluster will merge into the bigger one. If they have equal number of members then a hashing algorithm determines the merging cluster.

Q. Each cluster may have different versions of the same key in the same map. How is the conflict resolved?

A. Destination cluster will decide how to handle merging entry based on the `MergePolicy` set for that map. There are built-in merge policies such as `hz.NO_MERGE, hz.ADD_NEW_ENTRY and hz.LATEST_UPDATE` but you can develop your own merge policy by implementing `com.hazelcast.merge.MergePolicy`. You should register your custom merge policy in the configuration so that Hazelcast can find it by name.

```
public interface MergePolicy {
    /**
      Returns the value of the entry after the merge
      of entries with the same key. Returning value can be
      You should consider the case where existingEntry is null.

      @param mapName       name of the map
      @param mergingEntry  entry merging into the destination cluster
      @param existingEntry existing entry in the destination cluster
      @return final value of the entry. If returns null then no change on the entry.
     */
    Object merge(String mapName, MapEntry mergingEntry, MapEntry existingEntry);
}
```

Here is how merge policies are registered and specified per map.

```
<hazelcast>
    ...
        <map name="default">
            <backup-count>1</backup-count>
            <eviction-policy>NONE</eviction-policy>
            <max-size>0</max-size>
            <eviction-percentage>25</eviction-percentage>
            <!--
                While recovering from split-brain (network partitioning),
                map entries in the small cluster will merge into the bigger cluster
                based on the policy set here. When an entry merge into the
                cluster, there might an existing entry with the same key already.
                Values of these entries might be different for that same key.
                Which value should be set for the key? Conflict is resolved by
                the policy set here. Default policy is hz.ADD_NEW_ENTRY
                There are built-in merge policies such as
                hz.NO_MERGE      ; no entry will merge.
                hz.ADD_NEW_ENTRY ; entry will be added if the merging entry's key
                                   doesn't exist in the cluster.
```

```
                hz.HIGHER_HITS   ; entry with the higher hits wins.
                hz.LATEST_UPDATE ; entry with the latest update wins.
            -->
            <merge-policy>MY_MERGE_POLICY</merge-policy>
        </map>
        <merge-policies>
            <map-merge-policy name="MY_MERGE_POLICY">
                <class-name>com.acme.MyOwnMergePolicy</class-name>
            </map-merge-policy>
        </merge-policies>
    ...
</hazelcast>
```

11. Hibernate Second Level Cache

Hazelcast provides distributed second level cache for your Hibernate entities, collections and queries. Hazelcast has two implementations of Hibernate 2nd level cache, one for hibernate-pre-3.3 and one for hibernate-3.3.x versions. With release of hibernate-3.3.x, Hibernate now has a different api for 2nd level cache, although they have not mentioned this in their public documentation clearly.

Hibernate 2nd Level Cache Documentation

In your Hibernate configuration file (ex: hibernate.cfg.xml), add these properties;

- To enable use of second level cache

```
<property name="hibernate.cache.use_second_level_cache">true</property>
```

- To enable use of query cache

```
<property name="hibernate.cache.use_query_cache">true</property>
```

- And to force minimal puts into cache

```
<property name="hibernate.cache.use_minimal_puts">true</property>
```

- To configure Hazelcast for Hibernate, it is enough to put configuration file named `hazelcast.xml` into root of your classpath. If Hazelcast can not find `hazelcast.xml` then it will use default configuration from hazelcast.jar.

- You can define custom named Hazelcast configuration xml file with one of these Hibernate configuration properties.

```
<property name="hibernate.cache.provider_configuration_file_resource_path">hazelcast-custom-config.xml
```

or

```
<property name="hibernate.cache.hazelcast.configuration_file_path">hazelcast-custom-config.xml</proper
```

- You can set up Hazelcast to connect cluster as Super Client. Super Client is a member of the cluster, it has socket connection to every member in the cluster and it knows where the data, but does not contain any data.

```
<property name="hibernate.cache.hazelcast.use_super_client">true</property>
```

- You can set up Hazelcast to connect cluster as Native Client. Native client is not member and it connects to one of the cluster members and delegates all cluster wide operations to it. When the relied cluster member dies, client will transparently switch to another live member. *(Native Client property takes precedence over Super Client property.)*

```
<property name="hibernate.cache.hazelcast.use_native_client">true</property>
```

To setup Native Client properly, you should add Hazelcast **group-name**, **group-password** and **cluster member hosts** properties. Member hosts are comma-seperated addresses. Additionally you can add port number at the end of each address.

```
<property name="hibernate.cache.hazelcast.native_client_hosts">10.34.22.15,10.34.16.43:5702,127.0.0.1:
<property name="hibernate.cache.hazelcast.native_client_group">dev</property>
<property name="hibernate.cache.hazelcast.native_client_password">dev-pass</property>
```

To use Native Client you should add `hazelcast-client-<version>.jar` *into your classpath.*

Read more about NativeClient & SuperClient

- If you are using one of Hibernate pre-3.3 version, add following property.

```
<property name="hibernate.cache.provider_class">com.hazelcast.hibernate.provider.HazelcastCacheProvide
```

- If you are using Hibernate 3.3.x (or newer) version, you can choose to use either configuration property above (Hibernate has a built-in bridge to use old-style cache implementations) or following property.

```
<property name="hibernate.cache.region.factory_class">com.hazelcast.hibernate.HazelcastCacheRegionFact
```

Hazelcast creates a seperate distributed map for each Hibernate cache region. So these regions can be configured easily via Hazelcast map configuration. You can define **backup**, **eviction**, **TTL** and **Near Cache** properties.

1. Backup Configuration
2. Eviction And TTL Configuration
3. Near Cache Configuration

   Hibernate has 4 cache concurrency strategies: *read-only*, *read-write*, *nonstrict-read-write* and *transactional*. But Hibernate does not forces cache providers to support all strategies. And Hazelcast supports first three (**read-only**, **read-write**, **nonstrict-read-write**) of these four strategies. Hazelcast has not support for *transactional* strategy yet.

   - If you are using xml based class configurations, you should add a *cache* element into your configuration with *usage* attribute with one of *read-only*, *read-write*, *nonstrict-read-write*.

```
<class name="eg.Immutable" mutable="false">
  <cache usage="read-only"/>
  ....
</class>
<class name="eg.Cat" .... >
```

```
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

- If you are using Hibernate-Annotations then you can add *class-cache* or *collection-cache* element into your Hibernate configuration file with *usage* attribute with one of *read only*, *read/write*, *nonstrict read/write*.

```
<class-cache usage="read-only" class="eg.Immutable"/>
<class-cache usage="read-write" class="eg.Cat"/>
<collection-cache collection="eg.Cat.kittens" usage="read-write"/>
```

OR

- Alternatively, you can put Hibernate Annotation's *@Cache* annotation on your entities and collections.

```
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Cat implements Serializable {
...
}
```

And now last thing you should be aware of is to drop `hazelcast-hibernate-<version>.jar` into your classpath.

12. Spring Integration

#summary Hazelcast Spring integration

You can declare Hazelcast beans for Spring context using *beans* namespace (default spring *beans* namespace) as well to declare hazelcast maps, queues and others.

Hazelcast-Spring integration requires either hazelcast-spring jar or hazelcast-all jar in the classpath.

```
<bean id="instance" class="com.hazelcast.core.Hazelcast" factory-method="newHazelcastInstance">
    <constructor-arg>
        <bean class="com.hazelcast.config.Config">
            <property name="groupConfig">
                <bean class="com.hazelcast.config.GroupConfig">
                    <property name="name" value="dev"/>
                    <property name="password" value="pwd"/>
                </bean>
            </property>
            <!-- and so on ... -->
        </bean>
    </constructor-arg>
</bean>
<bean id="map" factory-bean="instance" factory-method="getMap">
    <constructor-arg value="map"/>
</bean>
```

Hazelcast has Spring integration (requires version 2.5 or greater) since 1.9.1 using *hazelcast* namespace.

- Add namespace *xmlns:hz*="*http://www.hazelcast.com/schema/config*" to beans tag in context file:
- Use *hz* namespace shortcuts to declare cluster, its items and so on.

After that you can configure Hazelcast instance (node):

```
<hz:hazelcast id="instance">
        <hz:config>
            <hz:group name="dev" password="password"/>
            <hz:properties>
                <hz:property name="hazelcast.merge.first.run.delay.seconds">5</hz:property>
                <hz:property name="hazelcast.merge.next.run.delay.seconds">5</hz:property>
            </hz:properties>
            <hz:network port="5701" port-auto-increment="false">
                <hz:join>
                    <hz:multicast enabled="false"
                                  multicast-group="224.2.2.3"
                                  multicast-port="54327"/>
                    <hz:tcp-ip enabled="true">
                        <hz:members>10.10.1.2, 10.10.1.3</hz:members>
                    </hz:tcp-ip>
                </hz:join>
            </hz:network>
            <hz:map name="map"
                    backup-count="2"
                    max-size="0"
                    eviction-percentage="30"
                    read-backup-data="true"
                    cache-value="true"
                    eviction-policy="NONE"
                    merge-policy="hz.ADD_NEW_ENTRY"/>
        </hz:config>
    </hz:hazelcast>
```

As of version hazelcast 1.9.3, you can easily configure map-store and near-cache too. (For map-store you should set either *class-name* or *implementation* attribute.)

```
 <hz:config>
          <hz:map name="map1">
                   <hz:near-cache time-to-live-seconds="0" max-idle-seconds="60"
                      eviction-policy="LRU" max-size="5000"  invalidate-on-change="true"/>
                   <hz:map-store enabled="true" class-name="com.foo.DummyStore"
                       write-delay-seconds="0"/>
          </hz:map>
                    <hz:map name="map2">
                   <hz:map-store enabled="true" implementation="dummyMapStore"
                       write-delay-seconds="0"/>
          </hz:map>
           <bean id="dummyMapStore" class="com.foo.DummyStore" />
</hz:config>
```

It's possible to use placeholders instead of concrete values.

For instance, use property file *app-default.properties* for group configuration:

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
```

```
        <list>
            <value>classpath:/app-default.properties</value>
        </list>
    </property>
</bean>
<hz:hazelcast id="instance">
    <hz:config>
        <hz:group
            name="${cluster.group.name}"
            password="${cluster.group.password}"/>
        <!-- ... -->
    </hz:config>
</hz:hazelcast>
```

Similar for client

```
<hz:client id="client"
    group-name="${cluster.group.name}" group-password="${cluster.group.password}">
    <hz:members>10.10.1.2:5701, 10.10.1.3:5701</hz:members>
</hz:client>
```

You can declare beans for the following Hazelcast objects:

- map
- multiMap
- queue
- topic
- set
- list
- executorService
- idGenerator
- atomicNumber

Example:

```
    <hz:map id="map" instance-ref="instance" name="map"/>
    <hz:multiMap id="multiMap" instance-ref="instance" name="multiMap"/>
    <hz:queue id="queue" instance-ref="instance" name="queue"/>
    <hz:topic id="topic" instance-ref="instance" name="topic"/>
    <hz:set id="set" instance-ref="instance" name="set"/>
    <hz:list id="list" instance-ref="instance" name="list"/>
    <hz:executorService id="executorService" instance-ref="instance" name="executorService"/>
    <hz:idGenerator id="idGenerator" instance-ref="instance" name="idGenerator"/>
    <hz:atomicNumber id="atomicNumber" instance-ref="instance" name="atomicNumber"/>
```

13. Clients

There are currently three ways to connect to a running Hazelcast cluster:

13.1. Native Client

Native Client enables you to do all Hazelcast operations without being a member of the cluster. It connects to one of the cluster members and delegates all cluster wide operations to it. When the relied cluster member dies, client will transparently switch to another live member.

There can be hundreds, even thousands of clients connected to the cluster.

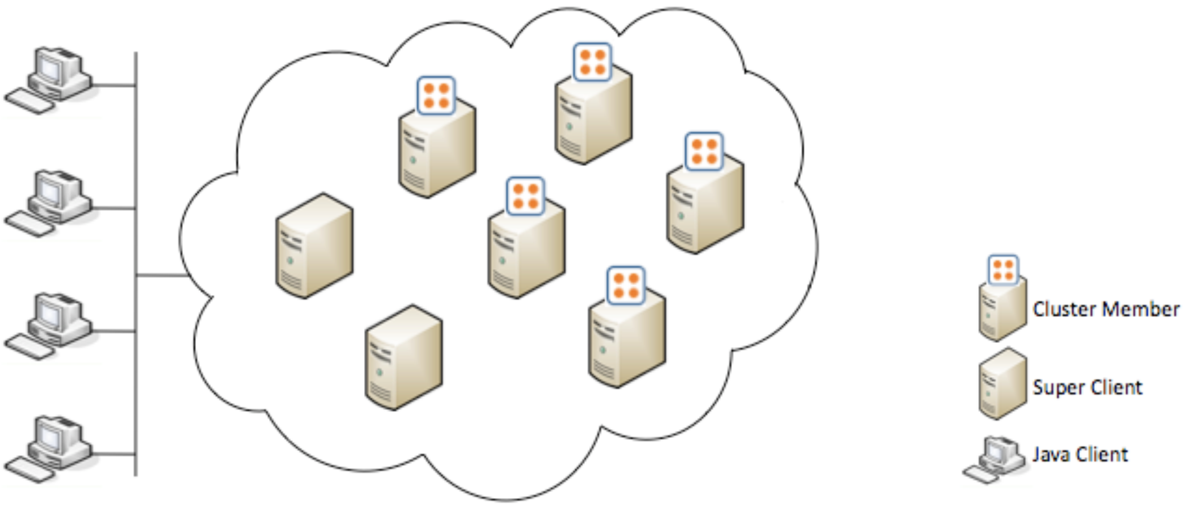
Imagine a trading application where all the trading data stored and managed in a 10 node Hazelcast cluster. Swing/Web applications at traders' desktops can use Native Java Client to access and modify the data in the Hazelcast cluster.

Currently Hazelcast has Native Java Client available. The next client implementation will be CSharp.

Super Client vs. Native Client

Super Client is a member of the cluster, it has socket connection to every member in the cluster and it knows where the data is so it will get to the data much faster. But Super Client has the clustering overhead and it must be on the same data center even on the same RAC. However Native client is not member and relies on one of the cluster members. Native Clients can be anywhere in the LAN or WAN. It scales much better and overhead is quite less. So if your clients are less than Hazelcast nodes then Super client can be an option; otherwise definitely try Native Client. As a rule of thumb: Try Native client first, if it doesn't perform well enough for you, then consider Super client.

The following picture describes the clients connecting to Hazelcast Cluster. Note the difference between Super Client and Java Client



13.1.1. Java Client

You can do almost all hazelcast operations with Java Client. It already implements the same interface. You must include hazelcast-client.jar into your classpath.

```
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.client.HazelcastClient;

import java.util.Map;
import java.util.Collection;


HazelcastInstance client = HazelcastClient.newHazelcastClient("dev", "dev-pass", "10.90.0.1", "10.9(
//All Cluster Operations that you can do with ordinary HazelcastInstance
Map<String, Customer> mapCustomers = client.getMap("customers");
mapCustomers.put("1", new Customer("Joe", "Smith"));
mapCustomers.put("2", new Customer("Ali", "Selam"));
mapCustomers.put("3", new Customer("Avi", "Noyan"));

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

13.1.2. CSharp Client

CSharp client is not available yet.

13.2. Memcache Client

A Memcache client written in any language can talk directly to Hazelcast cluster.

No additional configuration is required. Here is an example:

Let's say your cluster's members are:

```
Members [5] {
    Member [10.20.17.1:5701]
    Member [10.20.17.2:5701]
    Member [10.20.17.4:5701]
    Member [10.20.17.3:5701]
    Member [10.20.17.5:5701]
 }
```

And you have a PHP application that uses PHP Memcache client to cache things in Hazelcast. All you need to do is have your PHP memcache client connect to one of these members. It doesn't matter which member the client connects to because Hazelcast cluster looks as one giant machine (Single System Image).

PHP client code sample:

```
<?php
    $memcache = new Memcache;
    $memcache->connect('10.20.17.1', 5701) or die ("Could not connect");
    $memcache->set('key1','value1',0,3600);
    $get_result = $memcache->get('key1'); //retrieve your data
    var_dump($get_result); //show it
?>
```

Notice that memcache client is connecting to `10.20.17.1` and using port `5701`.

Java client code sample with SpyMemcached client:

```
MemcachedClient client = new MemcachedClient(AddrUtil.getAddresses("10.20.17.1:5701 10.20.17.2:5701"));
client.set("key1", 3600, "value1");
System.out.println(client.get("key1"));
```

An entry written with a memcache client can be read by another memcache client written in another language.

13.3. Rest Client

Let's say your cluster's members are:

```
Members [5] {
    Member [10.20.17.1:5701]
    Member [10.20.17.2:5701]
    Member [10.20.17.4:5701]
    Member [10.20.17.3:5701]
    Member [10.20.17.5:5701]
 }
```

And you have a distributed map named 'stocks'. You can put a new `key1/value1` entry into this map by issuing `HTTP POST` call to `http://10.20.17.1:5701/hazelcast/rest/maps`

`/stocks/key1` URL. Your http post call's content body should contain the value (value1). You can retrieve this entry via `HTTP GET` call to `http://10.20.17.1:5701/hazelcast/rest/maps/stocks/key1`. You can also retrieve this entry from another member such as `http://10.20.17.3:5701/hazelcast/rest/maps/stocks/key1`.

RESTful access is provided through any member of your cluster. So you can even put an HTTP load-balancer in-front of your cluster members for load-balancing and fault-tolerance.

Now go ahead and install a REST plugin for your browser and explore further.

Hazelcast also stores the mime-type of your `POST` request if it contains any. So if, for example, you post binary of an image file and set the mime-type of the `HTTP POST` request to `image/jpeg` then this mime-type will be part of the response of your `HTTP GET` request for that entry.

Let's say you also have a task queue named 'tasks'. You can offer a new item into the queue via HTTP POST and take and item from the queue via HTTP DELETE.

HTTP POST http://10.20.17.1:5701/hazelcast/rest/queues/tasks <CONTENT>

means

```
Hazelcast.getQueue("tasks").offer(<CONTENT>);
```

and

HTTP DELETE http://10.20.17.1:5701/hazelcast/rest/queues/tasks/3

means

```
Hazelcast.getQueue("tasks").poll(3, SECONDS);
```

Note that you will have to handle the failures on REST polls as there is no transactional guarantee.

14. Internals

14.1. Internals 1: Threads

In this section, we will go over the Hazelcast's internal threads, the client threads executing Hazelcast API and how these threads work together in Hazelcast. When developing Hazelcast, you should know which thread will execute your code, which variables are local to that thread, and how you should interact with other threads.

1. Client Threads:

Client threads are your threads, user's application threads, and or user's application/web server's threads that are executing Hazelcast API. User's threads that are client to Hazelcast. For example, `Hazelcast.getQueue("myqueue"), map.put(key, value), set.size()` calls are initiated and finalized in the client threads. Serialization of the objects also happens in the client threads. This also eliminates the problems associated with classloaders. Client threads initiate the calls, serialize the objects into Hazelcast `com.hazelcast.nio.Data` object which holds a `java.nio.ByteBuffer`. `Data` object is the binary representation of the application objects (key, value, item, callable objects). All Hazelcast threads such as `ServiceThread`, `InThread` and `OutThread` work with `Data` objects; they don't know anything about the actual application objects. When the calls are finalized, if the return type is an object, `Data` object is returned to the client thread and client thread then will deserialize the `Data` (binary representation) back to the application objects.

For each client thread, there is a `com.hazelcast.impl.ThreadContext` thread-local instance attached which contains thread context information such as transaction.

2. `ServiceThread`:

`ServiceThread`, implemented at `com.hazelcast.impl.ClusterService`, is the most significant thread in Hazelcast. Almost all none-IO operations happens in this thread. `ServiceThread` serves to local client calls and the calls from other members.

If you look at the `ClusterService` class you will see it is constantly dequeueing its queue and processing local and remote events. `ClusterService` queue receives local events in the form of `com.hazelcast.impl.BaseManager.Processable` instances and remote events in the form of `com.hazelcast.nio.PacketQueue.Packet` instances from `InThread`.

All distributed data structures (queue, map, set) are eventually modified in this thread so there is -no- synchronization needed when data structures are accessed/modified.

It is important to understand that client threads initiates/finalizes the calls, in/out threads handles the socket read/writes and `ServiceThread` does the actually manipulation of the data structures. There is no other threads allowed to touch the data structures (maps, queues) so that there is no need to protect the data structures from multithread access: no synchronization when accessing data structures. It may sound inefficient to allow only one thread to do all data structure updates but here is the logic behind it (Please note that numbers given here are not exact but should give you an idea.): If there is only one member (no IO), `ServiceThread` utilization will be about 95% and it will process between 30K and 120K operations per second, depending on the server. As the number of members in the cluster increases, IO Threads will be using more CPU and eventually `ServiceThread` will go down to 35% CPU utilization so `ServiceThread` will process between 10K and 40K operations per second. `ServiceThread` CPU utilization will be at about 35% regardless of the size of the cluster. (The only thing that can affect that would be the network utilization.) This also means that total number of operations processed by the entire cluster will be between N*10K and N*40K; N being the number of nodes in the cluster. About half of these operations will be backup operations (assuming one backup) so client threads will realize between N*5K and N*20K operations per second in total. Since there is only one thread accessing the data structures, increase in the number of nodes doesn't create any contention so access to the data structures will be always at the same speed. This is very important for Hazelcast's scalability.

This also makes writing code super easy because significant portion of the code is actually single-threaded so it is less error-prone and easily maintainable.

No synchronization or long running jobs are allowed in this thread. All operations running in this thread have to complete in microseconds.

3. `InThread` and `OutThread`:

Hazelcast separates reads and writes by having two separate threads; one for reading, and the other for writing. Eache IO thread uses its own `NIO` selector instance. `InThread` handles `OP_ACCEPT` and `OP_READ` socket operations while `OutThread` handles `OP_CONNECT` and `OP_WRITE` operations.

Each thread has its queue that they dequeue to register these operations with their selectors so operation registrations and operation processing happens in the same threads.

`InThread`'s runnable is the `com.hazelcast.nio.InSelector` and `OutThread`'s runnable is the `com.hazelcast.nio.OutSelector`. They both extends `SelectorBase` which constantly processes its registration queue ('selectorQueue') and the selectedKeys.

Members are connected to each other via `TCP/IP`. If there are N number of members in the cluster then there will be `N*(N-1)` connection end point and `(N*(N-1))/2` connections. There can be only one connection between two members, meaning, if m2 creates a connection to m1, m1 doesn't create another connection to m2 and the rule here is that new members connect to the older members.

If you look at the `com.hazelcast.nio.Connection`, you will see that each connection is representing a socket channel and has `com.hazelcast.nio.ReadHandler` and `WriteHandler` instances which are attached to the socket channel's `OP_READ` and `OP_WRITE` operation selectionKeys respectively. When `InSelector` selects `OP_READ` selection key (when this operation is ready for the selector), `InSelector` will get the attached `ReadHandler` instance from the selectionKey and call its `ReadHandler.handle()` method. Same for the `OutSelector`. When `OutSelector` selects `OP_WRITE` selection key (when this operation is ready for the selector), `OutSelector` will get the attached `WriteHandler` instance from the selectionKey and call its `WriteHandler.handle()` method.

When `ServiceThread` wants to send an Invocation instance to a member, it will

1. get the Connection for this member by calling `com.hazelcast.nio.ConnectionManager.get().getConnection(address)`
2. check if the connection is live; Connection.live()
3. if live, it will get the `WriteHandler` instance of the Connection
4. enqueue the invocation into the `WriteHandler`'s queue
5. and add registration task into `OutSelector`'s queue, if necessary

6. `OutSelector` processes the `OP_WRITE` operation registration with its selector
7. when the selector is ready for the `OP_WRITE` operation, `OutSelector` will get the `WriteHandler` instance from selectionKey and call its `WriteHandler.handle()`.

see `com.hazelcast.impl.BaseManager.send(Packet, Address)`. see `com.hazelcast.nio.SelectorBase.run()`.

Connections are always registered/interested for `OP_READ` operations. When `InSelector` is ready for reading from a socket channel, it will get the `ReadHandler` instance from the selectionKey and call its handle() method. handle() method will read Invocation instances from the underlying socket and when an Invocation instance is fully read, it will enqueue it into `ServiceThread`'s (`ClusterService` class) queue to be processed.

4. `MulticastThread`:

If multicast discovery is enabled (this is the default), and node is the master (oldest member) in the cluster then `MulticastThread` is started to listen for join requests from the new members. When it receives join request (`com.hazelcast.nio.MulticastService.JoinInfo` class), it will check if the node is allowed to join, if so, it will send its address to the sender so that the sender node can create a `TCP/IP` connection to the master and send a `JoinRequest`.

5. Executor Threads:

Each node employs a local `ExecutorService` threads which handle the event listener calls and distributed executions. Number of core and max threads can be configured.

14.2. Internals 2: Serialization

All your distributed objects such as your key and value objects, objects you offer into distributed queue and your distributed callable/runnable objects have to be `Serializable`.

Hazelcast serializes all your objects into an instance of `com.hazelcast.nio.Data`. `Data` is the binary representation of an object and it holds list of `java.nio.ByteBuffer` instances which are reused. When Hazelcast serializes an object into `Data`, it first checks whether the object is an instance of well-known, optimizable object such as `String, Long, Integer, byte[], ByteBuffer, Date`. If not, it then checks whether the object is an instance of `com.hazelcast.nio.DataSerializable`. If not, Hazelcast uses standard java serialization to convert the object into binary format. See `com.hazelcast.nio.Serializer` for details.

So for faster serialization, Hazelcast recommends to use of `String`, `Long`, `Integer`, `byte[]` objects or to implement `com.hazelcast.nio.DataSerializable` interface. Here is an example of a class implementing `com.hazelcast.nio.DataSerializable` interface.

```
public class Address implements com.hazelcast.nio.DataSerializable {
        private String street;
        private int zipCode;
        private String city;
```

```
        private String state;

        public Address() {}

        //getters setters..

        public void writeData(DataOutput out) throws IOException {
                out.writeUTF(street);
                out.writeInt(zipCode);
                out.writeUTF(city);
                out.writeUTF(state);
        }

        public void readData (DataInput in) throws IOException {
                street  = in.readUTF();
                zipCode = in.readInt();
                city    = in.readUTF();
                state   = in.readUTF();
        }
}
```

Lets take a look at another example which is encapsulating a `DataSerializable` field.

```
public class Employee implements com.hazelcast.nio.DataSerializable {
        private String firstName;
        private String lastName;
        private int age;
        private double salary;
        private Address address; //address itself is DataSerializable

        public Employee() {}

        //getters setters..

        public void writeData(DataOutput out) throws IOException {
                out.writeUTF(firstName);
                out.writeUTF(lastName);
                out.writeInt(age);
                out.writeDouble (salary);
                address.writeData (out);
        }

        public void readData (DataInput in) throws IOException {
                firstName = in.readUTF();
                lastName  = in.readUTF();
                age       = in.readInt();
                salary    = in.readDouble();
                address   = new Address();
                // since Address is DataSerializable let it read its own internal state
                address.readData (in);
        }
}
```

As you can see, since `address` field itself is `DataSerializable`, it is calling `address.writeData(out)` when writing and `address.readData(in)` when reading.

Caution

Hazelcast serialization is done on the user thread and it assumes that there will be only one object serialization at a time. So putting any Hazelcast operation that will require to serialize anything else will brake the serialization. For Example: Putting

```
Hazelcast.getMap("anyMap").put("key", "dummy value");
```

line in readData or writeData methods will brake the serialization. If you have to perform such an operation, at least it should be performed in another thread which will force the serialization to take on different thread.

14.3. Internals 3: Cluster Membership

It is important to note that Hazelcast is a peer to peer clustering so there is no 'master' kind of server in Hazelcast. Every member in the cluster is equal and has the same rights and responsibilities.

When a node starts up, it will check to see if there is already a cluster in the network. There are two ways to find this out:

- Multicast discovery: If multicast discovery is enabled (that is the defualt) then the node will send a join request in the form of a multicast datagram packet.
- Unicast discovery: if multicast discovery is disabled and `TCP/IP` join is enabled then the node will try to connect to he IPs defined in the `hazelcast.xml` configuration file. If it can successfully connect to at least one node, then it will send a join request through the `TCP/IP` connection.

If there is no existing node, then the node will be the first member of the cluster. If multicast is enabled then it will start a multicast listener so that it can respond to incoming join requests. Otherwise it will listen for join request coming via `TCP/IP`.

If there is an existing cluster already, then the oldest member in the cluster will receive the join request and check if the request is for the right group. If so, the oldest member in the cluster will start the join process.

In the join process, the oldest member will:

- send the new member list to all members
- tell members to sync data in order to balance the data load

Every member in the cluster has the same member list in the same order. First member is the oldest member so if the oldest member dies, second member in the list becomes the first member in the list and the new oldest member.

See `com.hazelcast.impl.Node` and `com.hazelcast.impl.ClusterManager` for details.

Q. If, let say 50+, nodes are trying to join the cluster at the same time, are they going to join the cluster one by one?

No. As soon as the oldest member receives the first valid join request, it will wait 5 seconds for others to join so that it can join multiple members in one shot. If there is no new node willing to join for the next 5 seconds, then oldest member will start the join process. If a member leaves the cluster though, because of a JVM crash for example, cluster will immediately take action and oldest member will start the data recovery process.

14.4. Internals 4: Distributed Map

Hazelcast distributed map is a peer to peer, partitioned implementation so entries put into the map will be almost evenly partitioned onto the existing members. Entries are partitioned according to their keys.

Every key is owned by a member. So every key-aware operation, such as `put`, `remove`, `get` is routed to the member owning the key.

Q. How does Hazelcast determine the owner of a key?

Hazelcast creates fixed number of virtual partitions (blocks). Partition count is set to `271` by default. Each key falls into one of these partitions. Each partition is owned/managed by a member. Oldest member of the cluster will assign the ownerships of the partitions and let every member know who owns which partitions. So at any given time, each member knows the owner member of a each partition. Hazelcast will convert your key object to `com.hazelcast.nio.Data` then calculate the partition of the owner: `partition-of-the-key = hash(keyData) % PARTITION_COUNT`. Since each member(JVM) knows the owner of each partition, each member can find out which member owns the key.

Q. Can I get the owner of a key?

Yes. Use Partition API to get the partition that your key falls into and then get the owner of that partition. Note that owner of the partition can change over time as new members join or existing members leave the cluster.

```
PartitionService partitionService = Hazelcast.getPartitionService();
Partition partition = partitionService.getPartition(key);
Member ownerMember = partition.getOwner();
```

Locally owned entries can be obtained by calling `map.localKeySet()`.

Q. What happens when a new member joins?

Just like any other member in the cluster, the oldest member also knows who owns which partition and what the oldest member knows is always right. The oldest member is also responsible for redistributing the partition ownerships when a new member joins. Since there is new member, oldest member will take ownership of some of the partitions and give them to the new member. It will try to move the least amount of data possible. New ownership information of all partitions is then sent to all members.

Notice that the new ownership information may not reach each member at the same time and the cluster never stops responding to user map operations even during joins so if a member routes the operation to a wrong member, target member will tell the caller to `re-do` the operation.

If a member's partition is given to the new member, then the member will send all entries of that partition to the new member (Migrating the entries). Eventually every member in the cluster will own almost same number of partitions, and almost same number of entries. Also eventually every member will know the owner of each partition (and each key).

You can listener for migration events. `MigrationEvent` contains the `partitionId`, `oldOwner`, and `newOwner` information.

```
PartitionService partitionService = Hazelcast.getPartitionService();
partitionService.addMigrationListener(new MigrationListener () {

     public void migrationStarted(MigrationEvent migrationEvent) {
         System.out.println(migrationEvent);
     }

     public void migrationCompleted(MigrationEvent migrationEvent) {
         System.out.println(migrationEvent);
     }
});
```

Q. How about distributed set and list?

Both distributed set and list are implemented on top of distributed map. The underlying distributed map doesn't hold value; it only knows the key. Items added to both list and set are treated as keys. Unlike distributed set, since distributed list can have duplicate items, if an existing item is added again, `copyCount` of the entry (`com.hazelcast.impl.ConcurrentMapManager.Record`) is incremented. Also note that index based methods of distributed list, such as `List.get(index)` and `List.indexOf(Object)`, are not supported because it is too costly to keep distributed indexes of list items so it is not worth implementing.

Check out the `com.hazelcast.impl.ConcurrentMapManager` class for the implementation. As you will see, the implementation is lock-free because `ConcurrentMapManager` is a singleton and processed by only one thread, the `ServiceThread`.

15. Miscellaneous

15.1. Common Gotchas

Hazelcast is the distributed implementation of several structures that exist in Java. Most of the time it behaves as you expect. However there are some design choices in Hazelcast that violate some contracts. This page will list those violations.

1. **equals() and hashCode() methods for the objects stored in Hazelcast** When you store a key, value in a distributed Map, Hazelcast serializes the key and value and stores the byte array version of them in local ConcurrentHashMaps. And this ConcurrentHashMap uses the equals and hashCode methods of byte array version of your key. So it does not take into account the actual equals and hashCode implementations of your objects. So it is important that you choose your keys in a proper way. Implementing the equals and hashCode is not enough, it is also important that the object is always serialized into the same byte array. All primitive types, like; String, Long, Integer and etc. are good candidates for keys to use in Hazelcast. An unsorted Set is an example of a very bad candidate because Java Serialization may serialize the same unsorted set in two different byte arrays.

Note that the distributed Set and List stores its entries as the keys in a distributed Map. So the notes above apply to the objects you store in Set and List.

15.2. Testing Cluster

Hazelcast allows you to create more than one member on the same JVM. Each member is called `HazelcastInstance` and each will have its own configuration, socket and threads, so you can treat them as totally separate members. This enables us to write and run cluster unit tests on single JVM. As you can use this feature for creating separate members different applications running on the same JVM (imagine running multiple webapps on the same JVM), you can also use this feature for testing Hazelcast cluster.

Let's say you want to test if two members have the same size of a map.

```
@Test
 public void testTwoMemberMapSizes() {
     // start the first member
     HazelcastInstance h1 = Hazelcast.newHazelcastInstance(null);
     // get the map and put 1000 entries
     Map map1 = h1.getMap("testmap");
     for (int i = 0; i < 1000; i++) {
         map1.put(i, "value" + i);
     }
     // check the map size
     assertEquals(1000, map1.size());
     // start the second member
     HazelcastInstance h2 = Hazelcast.newHazelcastInstance(null);
     // get the same map from the second member
     Map map2 = h2.getMap("testmap");
     // check the size of map2
     assertEquals(1000, map2.size());
     // check the size of map1 again
     assertEquals(1000, map1.size());
 }
```

In the test above, everything happened in the same thread. When developing multi-threaded test, coordination of the thread executions has to be carefully handled. Usage of `CountDownLatch` for thread coordination is highly recommended. You can certainly use other things. Here is an example where we need to listen for messages and make sure that we got these messages:

```
@Test
public void testTopic() {
    // start two member cluster
    HazelcastInstance h1 = Hazelcast.newHazelcastInstance(null);
    HazelcastInstance h2 = Hazelcast.newHazelcastInstance(null);
    String topicName = "TestMessages";
    // get a topic from the first member and add a messageListener
    ITopic<String> topic1 = h1.getTopic(topicName);
    final CountDownLatch latch1 = new CountDownLatch(1);
    topic1.addMessageListener(new MessageListener() {
        public void onMessage(Object msg) {
            assertEquals("Test1", msg);
            latch1.countDown();
        }
    });
    // get a topic from the second member and add a messageListener
    ITopic<String> topic2 = h2.getTopic(topicName);
    final CountDownLatch latch2 = new CountDownLatch(2);
    topic2.addMessageListener(new MessageListener() {
        public void onMessage(Object msg) {
            assertEquals("Test1", msg);
            latch2.countDown();
        }
    });
    // publish the first message, both should receive this
    topic1.publish("Test1");
    // shutdown the first member
    h1.shutdown();
    // publish the second message, second member's topic should receive this
    topic2.publish("Test1");
    try {
        // assert that the first member's topic got the message
        assertTrue(latch1.await(5, TimeUnit.SECONDS));
        // assert that the second members' topic got two messages
        assertTrue(latch2.await(5, TimeUnit.SECONDS));
    } catch (InterruptedException ignored) {
    }
}
```

You can surely start Hazelcast members with different configuration. Let's say we want to test if Hazelcast `SuperClient` can shutdown fine.

```
@Test(timeout = 60000)
public void shutdownSuperClient() {
    // first config for normal cluster member
    Config c1 = new XmlConfigBuilder().build();
    c1.setPortAutoIncrement(false);
    c1.setPort(5709);
    // second config for super client
    Config c2 = new XmlConfigBuilder().build();
    c2.setPortAutoIncrement(false);
    c2.setPort(5710);
    // make sure to super client = true
    c2.setSuperClient(true);
    // start the normal member with c1
    HazelcastInstance hNormal = Hazelcast.newHazelcastInstance(c1);
    // start the super client with different configuration c2
```

```
        HazelcastInstance hSuper = Hazelcast.newHazelcastInstance(c2);
        hNormal.getMap("default").put("1", "first");
        assert hSuper.getMap("default").get("1").equals("first");
        hNormal.shutdown();
        hSuper.shutdown();
    }
```

Also remember to call `Hazelcast.shutdownAll()` after each test case to make sure that there is no other running member left from the previous tests.

```
    @After
    public void cleanup() throws Exception {
        Hazelcast.shutdownAll();
    }
```

Need more info? check out existing tests.

15.3. Planned Features

Planned Features

Random order of planned features.

- Native C# Client
- Native C++ Client
- Ready-to-go Hazelcast Cache Server Image for Amazon EC2
- Symmetric Encryption support for Java Client
- Continuous query (events based on given criteria)
- Distributed `java.util.concurrent.DelayQueue` implementation.
- Distributed `java.util.concurrent.CountDownLatch` implementation.
- Cluster-wide receive ordering for topics.
- Security (JAAS).
- Distributed Tree implementation.
- Distributed Tuple implementation.
- Call interceptors for modifying the request or the response.
- Built-in file based storage.

History of existing features is available at Release Notes.

15.4. Release Notes

Please see, Todo page for planned features.

1.9.3

- Re-implementation of distributed queue.
    - Configurable backup-count and synchronous backup.
    - Persistence support based on backing MapStore
    - Auto-recovery from backing MapStore on startup.
- Re-implementation of distributed list supporting index based operations.
- New distributed semaphore implementation.
- Optimized `IMap.putAll` for much faster bulk writes.
- New `IMap.getAll` for bulk reads which is calling `MapLoader.loadAll` if necessary.
- New `IMap.tryLockAndGet` and `IMap.putAndUnlock` API
- New `IMap.putTransient` API for storing only in-memory.
- New `IMap.addLocalEntryListener()` for listening locally owned entry events.
- New `IMap.flush()` for flushing the dirty entries into MapStore.
- New `MapLoader.getAllKeys` API for auto-pre-populating the map when cluster starts.
- Support for min. initial cluster size to enable equally partitioned start.
- Graceful shutdown.
- Faster dead-member detection.

1.9

- Memcache interface support. Memcache clients written in any language can access Hazelcast cluster.
- RESTful access support. http://<ip>:5701/hazelcast/rest/maps/mymap/key1
- Split-brain (network partitioning) handling
- New LifecycleService API to restart, pause Hazelcast instances and listen for the lifecycle events.
- New asynchronous put and get support for IMap via IMap.asyncPut() and IMap.asyncGet()
- New AtomicNumber API; distributed implementation of

java.util.concurrent.atomic.AtomicLong

- So many bug fixes.

1.8.4

- Significant performance gain for multi-core servers. Higher CPU utilization and lower latency.
- Reduced the cost of map entries by 50%.
- Better thread management. No more idle threads.
- Queue Statistics API and the queue statistics panel on the Monitoring Tool.
- Monitoring Tool enhancements. More responsive and robust.
- Distribution contains hazelcast-all-<version>.jar to simplify jar dependency.
- So many bug fixes.

1.8.3

- Bug fixes
- Sorted index optimization for map queries.

1.8.2

- A major bug fix
- Minor optimizations

1.8.1

- Hazelcast Cluster Monitoring Tool (see the hazelcast-monitor-1.8.1.war in the distro)
- New Partition API. Partition and key owner, migration listeners.
- New IMap.lockMap() API.
- New Multicast+TCP/IP join feature. Try multicast first, if not found, try tcp/ip.
- New Hazelcast.getExecutorService(name) API. Have separate named ExecutorServices. Do not let your big tasks blocking your small ones.
- New Logging API. Build your own logging. or simply use Log4j or get logs as LogEvents.
- New MapStatistics API. Get statistics for your Map operations and entries.
- HazelcastClient automatically updates the member list. no need to pass all members.

- Ability to start the cluster members evenly partitioned. so no migration.
- So many bug fixes and enhancements.
- There are some minor Config API change. Just make sure to re-compile.

1.8

- Java clients for accessing the cluster remotely. (C# is next)
- Distributed Query for maps. Both Criteria API and SQL support.
- Near cache for distributed maps.
- TTL (time-to-live) for each individual map entry.
  - IMap.put(key,value, ttl, timeunit)
  - IMap.putIfAbsent(key,value, ttl, timeunit)
- Many bug fixes.

1.7.1

- Multiple Hazelcast members on the same JVM. New `HazelcastInstance` API.
- Better API based configuration support.
- Many performance optimizations. Fastest Hazelcast ever!
- Smoother data migration enables better response times during joins.
- Many bug fixes.

1.7

- Persistence via Loader/Store interface for distributed map.
- Socket level encryption. Both symmetric and asymmetric encryption supported.
- New JMX support. (many thanks to Marco)
- New Hibernate second level cache provider (many thanks to Leo)
- Instance events for getting notified when a data structure instance (map, queue, topic etc.) is created or destroyed.
- Eviction listener. `EntryListener.entryEvicted(EntryEvent)`
- Fully 'maven'ized.
- Modularized...
  - hazelcast (core library)

  - hazelcast-wm (http session clustering tool)
  - hazelcast-ra (JCA adaptor)
  - hazelcast-hibernate (hibernate cache provider)

1.6

- Support for synchronous backups and configurable backup-count for maps.
- Eviction support. Timed eviction for queues. LRU, LFU and time based eviction for maps.
- Statistics/history for entries. create/update time, number of hits, cost. see `IMap.getMapEntry(key)`
- `MultiMap` implementation. similar to google-collections and apache-common-collections `MultiMap` but distributed and thread-safe.
- Being able to `destroy()` the data structures when not needed anymore.
- Being able to Hazelcast.shutdown() the local member.
- Get the list of all data structure instances via `Hazelcast.getInstances()`.

1.5

- Major internal refactoring
- Full implementation of `java.util.concurrent.BlockingQueue`. Now queues can have configurable capacity limits.
- Super Clients: Members with no storage. If `-Dhazelcast.super.client=true` JVM parameter is set, that JVM will join the cluster as a 'super client' which will not be a 'data partition' (no data on that node) but will have super fast access to the cluster just like any regular member does.
- Http Session sharing support for Hazelcast Web Manager. Different webapps can share the same sessions.
- Ability to separate clusters by creating groups. [ConfigGroup]
- `java.util.logging` support.

1.4

- Add, remove and update events for queue, map, set and list
- Distributed Topic for pub/sub messaging
- Integration with J2EE transactions via JCA complaint resource adapter
- ExecutionCallback interface for distributed tasks
- Cluster-wide unique id generator

1.3

- Transactional Distributed Queue, Map, Set and List

1.2

- Distributed Executor Service
- Multi member executions
- Key based execution routing
- Task cancellation support

1.1

- Session Clustering with Hazelcast Webapp Manager
- Full TCP/IP clustering support

1.0

- Distributed implementation of java.util.{Queue,Map,Set,List}
- Distributed implementation of java.util.concurrency.Lock
- Cluster Membership Events

Open Source Community: Get Involved!

Hazelcast is released under Apache License and the project is hosted at Google Code. It can be freely used in commercial or non-commercial applications.

- Project Site
- Mail Group
- Issue Tracker
- Browse Source Code