



(/)

← Zur Übersicht (<https://synyx.de/blog/>)

# The struggle with Hazelcast queue persistence

📅 09.06.2017

👤 [Arnold Franke](https://synyx.de/blog/author/franke/)

In this blog I will outline why we used Hazelcast for queueing messages in-memory distributed over a cluster and how we achieved higher resilience by persisting the queue's content. I will explain the pitfalls and difficulties that we encountered and how I constantly switched between praising and condemning Hazelcast.

## The problem to solve

I'm currently working in a project for a large customer data backend. The prod system consists of a load balanced cluster of five VMs each running two Tomcat instances hosting our application. The deployment process performs an A/B switching between the Tomcats on each node to achieve zero downtime. The application has to handle a lot of incoming data and updates and communicates with a lot of external services. At one point we felt the need for a queueing mechanism for two reasons:

1. Enabling controlled asynchronous processing of tasks inside the application. Example: A synchronous user request queues follow-up tasks to be processed later by another part of the

application so the request can deliver the response quicker to the user.

2. Queueing and retrying failed calls to external systems for higher resilience

We gathered the following core requirements for the queueing mechanism:

- Embedded into the application. Using a potentially failing external system would defeat reason 2
- Distributed over the cluster. Due to the nature of our data import mechanism one node creates a lot of tasks and the cluster should work together to process the tasks.
- Resistant to system failure. The queued data is critical – so when one node or even the whole cluster goes down the data should be preserved
- Performance. Due to the amount of processed data the solution has to be fast.
- Low complexity and easy maintainability. “Keep it simple” is a key ambition for everything that we use or build.

After a short evaluation phase these requirements led us to the conclusion that Hazelcast (<https://hazelcast.org/>) might be the solution of our problem. It can be embedded as library, its core feature is distributed data structures like maps and queues, it is known to be lightning fast and easy to use. Also it offers backup- and recovery mechanisms as well as the possibility to implement persistence for the data structures. And it's open source (<https://github.com/hazelcast/hazelcast>), yay!



## The easy part – divide and queue

The first implementation of the Hazelcast queue in our application was a piece of cake. Following the [documentation \(http://docs.hazelcast.org/docs/3.8.2/manual/html-single/index.html\)](http://docs.hazelcast.org/docs/3.8.2/manual/html-single/index.html) we only needed two dependencies in our pom.xml, some properties in our application config and one Spring config class and voilà: The distributed in-memory queue was ready to use in the code just like every other Java BlockingQueue implementation.

In the first tests we realized how great Hazelcast works. Every queued item was available on all nodes in an instant and we could shut down and restart nodes at will without losing data. The only thing that was a bit trickier was to get Hazelcast's network configuration right so the cluster finds its nodes during an A/B deployment without adding nodes that should NOT belong to the cluster.

I will not go into detail on this “easy part” because this blog post should concentrate on the difficulties. All in all we were in awe of Hazelcast's smoothness at this point.

## The hard part – persist the shit out of it

So far, so good. We already managed to meet 90% of our requirements. The last 10% shouldn't be that difficult, right? Pfffff let's just do it!

We wanted to make the data resilient against the improbable event of an outage of the whole cluster. So the data in the cluster should be backed up in some kind of persistence and be recovered when the cluster reboots. Hazelcast offers an abstract solution for this problem, namely the QueueStore interface. You can implement the interface with every persisting technology that you want, add some configuration and all queued data will be mirrored into the data store and be recovered after an eventual downtime.

```
1 public interface QueueStore<T> {  
2  
3     void store(Long key, T value);  
4  
5     void storeAll(Map<Long, T> map);  
6  
7     void delete(Long key);  
8  
9     void deleteAll(Collection<Long> keys);  
10  
11     T load(Long key);  
12  
13     Map<Long, T> loadAll(Collection<Long> keys);  
14  
15     Set<Long> loadAllKeys();  
16 }
```

## Difficulty #1: How to persist?

After the initial euphoria it began to dawn on us that implementing the Queue Store interface obviously meant choosing some technology to persist data (d'uh). Unfortunately Hazelcast does not offer some kind of default implementation that you just roll with if you want to try it out.

Well ok, how about our database? We did not want to do that at this time. We expected it to be slow and our project has a history of database-managed queues that didn't work that well.

The next thing that came in mind was the file system of our application servers. This actually seemed like a viable solution as the queue entries passed to the QueueStore interface are in key-value format and there already are several libraries providing a file-based key-value store.

So the evaluation train departed again and passed several solutions capable of storing key-value pairs in files like Berkeley DB, Map DB, Banana DB(!?) and some others.

In the end the train stopped at ChronicleMap (<https://github.com/OpenHFT/Chronicle-Map>), an off-heap in-memory map that is mirrored to a file and promises consistency and insane speed. The embedded library is developed by a team of professionals and supports file access by multiple JVM instances at the same time, which is crucial for our A/B deployment. Long story short: We implemented the ChronicleMap QueueStore and the first local tests delivered the desired results: A Hazelcast cluster with a huge amount of queued data got shutdown completely and restarted again and the data was still there!

## **Obstacle #2: I am the persistence Master!**

The first test on a production-like system with a cluster of multiple VMs seemed promising. After every simulated cluster downtime the data was still there. But looking closely at the files written by ChronicleMap we noticed a strange thing. Only on one node of the cluster the file size changed, on the other nodes the files got created but stayed on the same size of only a few KB. What was the meaning of this? Why are not all nodes backing up their data? And how was it possible for them to recover their data without the file backup?

After some more research we discovered a sentence in the Hazelcast documentation of the **map** persistence that was missing in the documentation of the **queue** persistence. It says:

//

NOTE: Data store needs to be a centralized system that is accessible from all Hazelcast members. Persistence to a local file system is not supported.

Aaaahrgs! That explained the observed behavior! The cluster assumes that all nodes access the same data store and determines one node to be some kind of persistence master that writes and reads all data from the store for the whole cluster! Further tests showed that it seems pretty unpredictable which node becomes the persistence master. If the nodes are not restarted in exactly the same order after every deployment it could happen that a different node is assigned persistence master and does not recover the data from the previous persistence master. The data would be lost – even without a cluster downtime. To prevent this we had to provide a centralized data store!

The situation was not critical as the persistence implementation had not been merged to master yet – but admittedly we were in some kind of frustration mode at that point and the first reflex was to centralize the ChronicleMap file so we did not have to change the implementation again. After hard negotiations our ops team grudgingly provided us with a test system of multiple VMs all accessing the same file on a NFS share. As expected it worked, but it didn't feel right. We decided to run a long term test with production-like data and to decide afterwards if the solution is good enough to be rolled out on production.

### Stumbling block #3: Configuring the QueueStore

This was only a minor issue but it added up with the uneasy mood that we had about our solution at that point. The Hazelcast queue is configured programmatically via a Spring Configuration Class. During our development we noticed that neither of our configuration changes to the queue persistence seemed to have any effect. It turned out that the QueueStore only accepts strings as configuration parameters which it not obvious at the first glance when using a `java.util.Properties` object to pass the properties, which accepts `Object` as type.

```
1 // no Strings - does not work
2 QueueStoreConfig queueStoreConfig = new QueueStoreConfig();
3 queueStoreConfig.setEnabled(true);
4 Properties properties = new Properties();
5 properties.put("binary", true);
6 properties.put("memory-limit", 0);
7 properties.put("bulk-load", 4L);
8 queueStoreConfig.setProperties(properties);
9 queueStoreConfig.setStoreImplementation(new ChronicleQueueStore());
10
11 // Strings - does work
12 QueueStoreConfig queueStoreConfig = new QueueStoreConfig();
13 queueStoreConfig.setEnabled(true);
14 queueStoreConfig.setProperty("binary", "false");
15 queueStoreConfig.setProperty("memory-limit", "0");
16 queueStoreConfig.setProperty("bulk-load", "4");
17 queueStoreConfig.setStoreImplementation(new ChronicleQueueStore());
```

## Anxiety #4: No transactions – obviously

After testing our solution for a while, more and more scenarios of potential data loss popped into our mind. We've been aware that neither Hazelcast nor ChronicleMap offer some kind of real transac-

tions when writing to the file. Theoretically the persistence master could be killed off during a write operation and the file could be left in an inconsistent state. We tested this scenario with a manually corrupted file and it resulted in the unpleasant situation that the ChronicleMap Spring bean could not be initialized, preventing the creation of the Spring ApplicationContext and consequentially stopping the application startup – not good.

To feel safe about our solution we needed a transactional, central data store. Captain Obvious knocked on the door and said “Helloooo? Database?”.

We reconsidered this option again. The database is transactional, it is centralized and it does not count as external system because it is so essential for our application that when the database is down, the application is down anyway. It still wouldn’t be a database managed queue because the queue still lies in-memory and the queueing mechanism is managed by Hazelcast. The database would be just a backup. Of course performance would be a potential issue but we were ready to give it a try.

So finally we decided to change the QueueStore implementation to persist to a key-value table into our database.

## **WTF #5: It’s a real bug!**

Feeling better with this persistence approach we pushed closer to a production release of the feature. But suddenly we had massive performance difficulties with production-like data. Reading 1000 entries from the queue took several minutes! Was it the fault of the database? Was it really that slow?

After extensive analysis we found out that as a matter of fact it was a bug in the otherwise really robust and stable Hazelcast. When using the `drainTo(numberOfEntries)` method to get data from the



queue the data should be loaded from the persisted data in bulks of a configurable size calling the QueueStore.loadAll(listOfEntries) method once for every bulk. Instead loadAll() got called once for the first bulk and then once for every single following item, resulting in almost the same number of database calls as the number of requested items.

I recently opened an issue (<https://github.com/hazelcast/hazelcast/issues/10621>) for the bug including a small demo project (<https://github.com/indyarni/hazelcastbugdemo>). The Hazelcast team reacted on the same day promising to fix it. One week later the issue was fixed and the fix to be released in the next version 3.9! Kudos to the Hazelcast developers!

Until we are able to use 3.9 we solved the problem with a temporary workaround, setting the bulk size the same as the number of drained entries, resulting in only one bulk loaded at a time and only one database call per drain.

( °□° ) ⌒ ┌──┐ **#6: Going Prod – finally successful?**

Having cleared this last issue and experiencing no further problems or data loss on the test system for weeks we felt confident to merge the persistence solution to master and go live with it. After the release we were relieved to see that it just worked! Seemingly no problems, no data loss, no performance problems – the application became a lot faster and more resilient.

And here comes the “but”: But we sometimes still observe some (5-10) lost items from the in-memory queue after performing a deployment on the cluster. That means that Hazelcast unexpectedly is not always able to synchronize the cluster in time when our deployment performs the A/B switch on one node after another. It is not a critical

problem because the lost items can be manually recovered from the database but obviously we still intend to fix this issue. The cause is possibly this issue (<https://github.com/hazelcast/hazelcast/issues/5444>) that has been fixed in Hazelcast 3.7. Problem is, we rolled out 3.6.3 on production which is incompatible with 3.7 and newer versions, which means to update Hazelcast we would need a cluster downtime.... AAAAAHRG – the story continues 😊

## Final words

Those were only the most dominant of the many challenges we encountered while implementing this solution. Others were e.g. problems with Spring transaction handling when Hazelcast internally opened new threads to call the persistence interface, analyzing different causes of data loss, etc, etc....

After this Odyssey we can draw some conclusions:

- Hazelcast is a great tool! It's fun to work with, the core features work reaaaally well and I would use it again.
- However the non-core functionalities (like queue persistence) require some effort to get them working in a complex environment.
- When using a new tool you should read the documentation carefully and try to understand how it really works!
- If you really want to understand how Hazelcast works it's not enough to read the documentation carefully 😊

**Tags:** cluster (<https://synyx.de/blog/tags/cluster/>)

hazelcast (<https://synyx.de/blog/tags/hazelcast/>)


persistence (<https://synyx.de/blog/tags/persistence/>)


resilience (<https://synyx.de/blog/tags/resilience/>)

## Kontakt

---

synyx GmbH & Co. KG  
Gartenstraße 67  
76135 Karlsruhe, Germany

 +49 721 203823-0

 [info@synyx.de](mailto:info@synyx.de)(mailto:info@synyx.de)

[https://www.instagram.com/synyx/?utm\\_source=ig\\_embed&utm\\_medium=impression\\_source](https://www.instagram.com/synyx/?utm_source=ig_embed&utm_medium=impression_source)  
<https://www.facebook.com/synyx>  
<https://www.linkedin.com/company/synyx>  
<https://www.youtube.com/channel/UC9VAELzS51zB2o2vTYpDSlw>  
<https://www.github.com/synyx>  
<https://www.openstack.org/open-source/>

## Kategorien

---

Administrator Blog (20) (<https://synyx.de/blog/kategorien/administrator-blog/>)

Agile (1)(<https://synyx.de/blog/kategorien/agile/>)

Azubi Blog (19)(<https://synyx.de/blog/kategorien/azubi-blog/>)

Barcamp (2)(<https://synyx.de/blog/kategorien/barcamp/>)

Developer Blog (121)(<https://synyx.de/blog/kategorien/developer-blog/>)

Devoux4kids (4)(<https://synyx.de/blog/kategorien/devoux4kids/>)

Konferenzen (1)(<https://synyx.de/blog/kategorien/konferenzen/>)

Mobile Blog (46)(<https://synyx.de/blog/kategorien/mobile-blog/>)

Open Source Blog (32)(<https://synyx.de/blog/kategorien/open-source-blog/>)

Our apps (11)(<https://synyx.de/blog/kategorien/our-apps/>)

[synyx Blog \(121\)](https://synyx.de/blog/kategorien/synyx-blog/)(<https://synyx.de/blog/kategorien/synyx-blog/>)

[Tutorial \(30\)](https://synyx.de/blog/kategorien/tutorial/)(<https://synyx.de/blog/kategorien/tutorial/>)

## Tags

---

[synyx \(39\)](https://synyx.de/blog/tags/synyx/), (<https://synyx.de/blog/tags/synyx/>)

[android \(33\)](https://synyx.de/blog/tags/android/), (<https://synyx.de/blog/tags/android/>)

[java \(25\)](https://synyx.de/blog/tags/java/), (<https://synyx.de/blog/tags/java/>)

[iphone \(22\)](https://synyx.de/blog/tags/iphone/), (<https://synyx.de/blog/tags/iphone/>)

[conference \(20\)](https://synyx.de/blog/tags/conference/), (<https://synyx.de/blog/tags/conference/>)

[apple \(16\)](https://synyx.de/blog/tags/apple/), (<https://synyx.de/blog/tags/apple/>)

[mobile \(14\)](https://synyx.de/blog/tags/mobile/), (<https://synyx.de/blog/tags/mobile/>)

[spring \(14\)](https://synyx.de/blog/tags/spring/), (<https://synyx.de/blog/tags/spring/>)

[maven \(14\)](https://synyx.de/blog/tags/maven/), (<https://synyx.de/blog/tags/maven/>)

[opencms \(14\)](https://synyx.de/blog/tags/opencms/), (<https://synyx.de/blog/tags/opencms/>)

[konferenz \(13\)](https://synyx.de/blog/tags/konferenz/), (<https://synyx.de/blog/tags/konferenz/>)

[javascript \(13\)](https://synyx.de/blog/tags/javascript/), (<https://synyx.de/blog/tags/javascript/>)

[agile \(11\)](https://synyx.de/blog/tags/agile/), (<https://synyx.de/blog/tags/agile/>)

[scrum \(11\)](https://synyx.de/blog/tags/scrum/), (<https://synyx.de/blog/tags/scrum/>)

[open source \(11\)](https://synyx.de/blog/tags/open-source/), (<https://synyx.de/blog/tags/open-source/>)

[development \(10\)](https://synyx.de/blog/tags/development/), (<https://synyx.de/blog/tags/development/>)

[test \(9\)](https://synyx.de/blog/tags/test/), (<https://synyx.de/blog/tags/test/>)

[ausbildung \(9\)](https://synyx.de/blog/tags/ausbildung/), (<https://synyx.de/blog/tags/ausbildung/>)

[software development \(9\)](https://synyx.de/blog/tags/software-development/), (<https://synyx.de/blog/tags/software-development/>)

[devoxx4kids \(8\)](https://synyx.de/blog/tags/devoxx4kids/)(<https://synyx.de/blog/tags/devoxx4kids/>)

Kontakt (<https://synyx.de/kontakt/>)

Impressum (<https://synyx.de/impressum/>)

Datenschutz (<https://synyx.de/datenschutz/>)

Disclaimer (<https://synyx.de/disclaimer/>)

AGB ([/wp-content/uploads/2019/03/AGB\\_synyx\\_17.pdf](/wp-content/uploads/2019/03/AGB_synyx_17.pdf))

Kommentarrichtlinien (<https://synyx.de/kommentarrichtlinien/>)