**Carlos Alexandro Becker**

# Distributed Locking with Redis

Joinville · 2017-03-04

At ContaAzul, we have several old pieces of code that are still running in production. We are committed to gradually re-implement them in better ways.

One of those parts was our distributed locking mechanism and me and @t-bonatti were up for the job.

## How it was

In normal workloads, we have two servers responsible for running previously scheduled tasks (e.g.: issue an electronic invoice to the government).

An ideal scenario would consist of idempotent services, but, since most of those tasks talk to government services, we are pretty ~~fucking~~ far from an ideal scenario.

We cannot fix the government services' code, so, to avoid calling them several times for the same input (which they don't like, by the way), we had think about mainly two options:

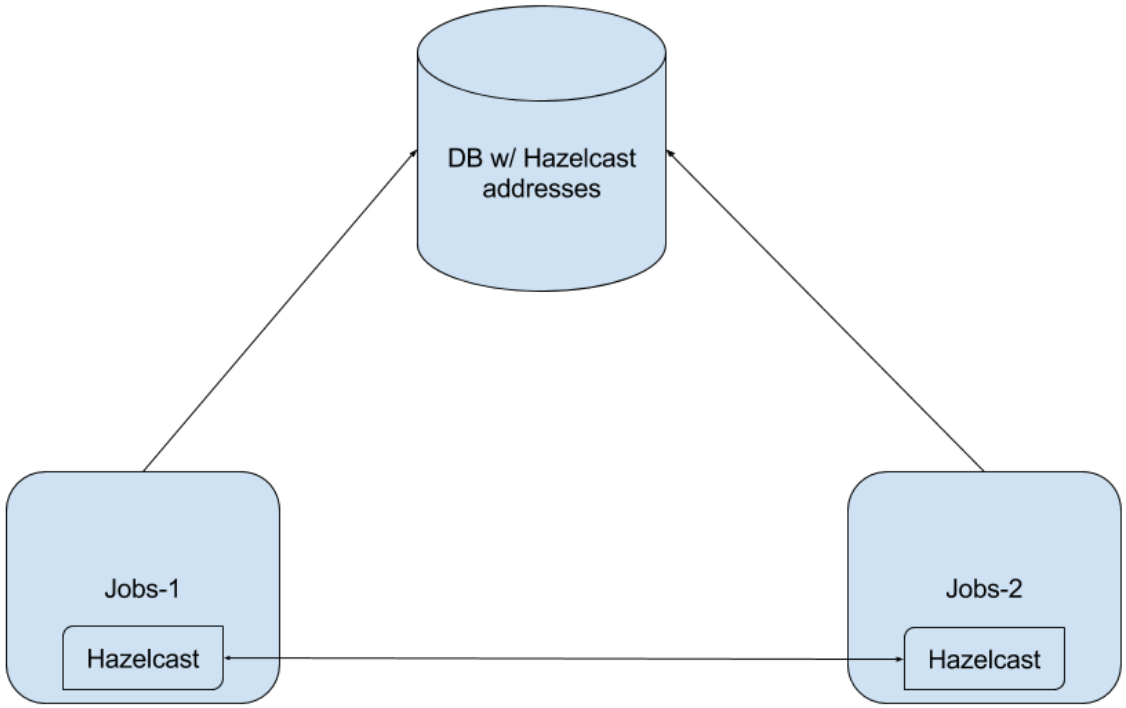
1. Run it all in a single server;
2. Synchronize work, somehow.

One server would probably not scale well, so we decided that synchronizing work between servers was the best way of solving this issue.

At the time, we also decided to use Hazelcast for the job, which seemed reasonable because:

1. It does have a pretty good locking API;
2. It is written in Java, and we are mainly a Java shop, which allowed us to more easily fix issues if needed (and it was).

The architecture was something like this:




Basically, when one of those scheduled tasks servers (let's call them *jobs*) went up, it also starts a Hazelcast node and register itself in a database table.

After that, it reads this same table looking for other nodes, and synchronizes with them.

Finally, in the code, we would basically get a new `ILock` from Hazelcast API and use it, something like this:

```
if (hazelcast.getLock( jobName + ":" + elementId ).tryLock() {
  // do the work
}
```

There was, of course, an API around all this so the developers were just locking things, and may not know exactly how.

This architecture worked for years with very few problems and was used in other applications as well, but still we had our issues with it:

- Lack of proper monitoring (and kind of hard to do that right);
- Sharing resources with the Jobs servers (which may not be considered a good practice);
- Might not work in some cases, like services deployed to AWS BeanStalk (which allows you to open one port per service, so the nodes weren't able to sync);
- Some ugly AWS Security Group rules to allow the connection between machines in the port range that Hazelcast uses (which we bothering us);
- If Hazelcast nodes failed to sync with each other, the distributed lock would not be distributed anymore, causing possible duplicates, and, worst of all, no errors whatsoever.

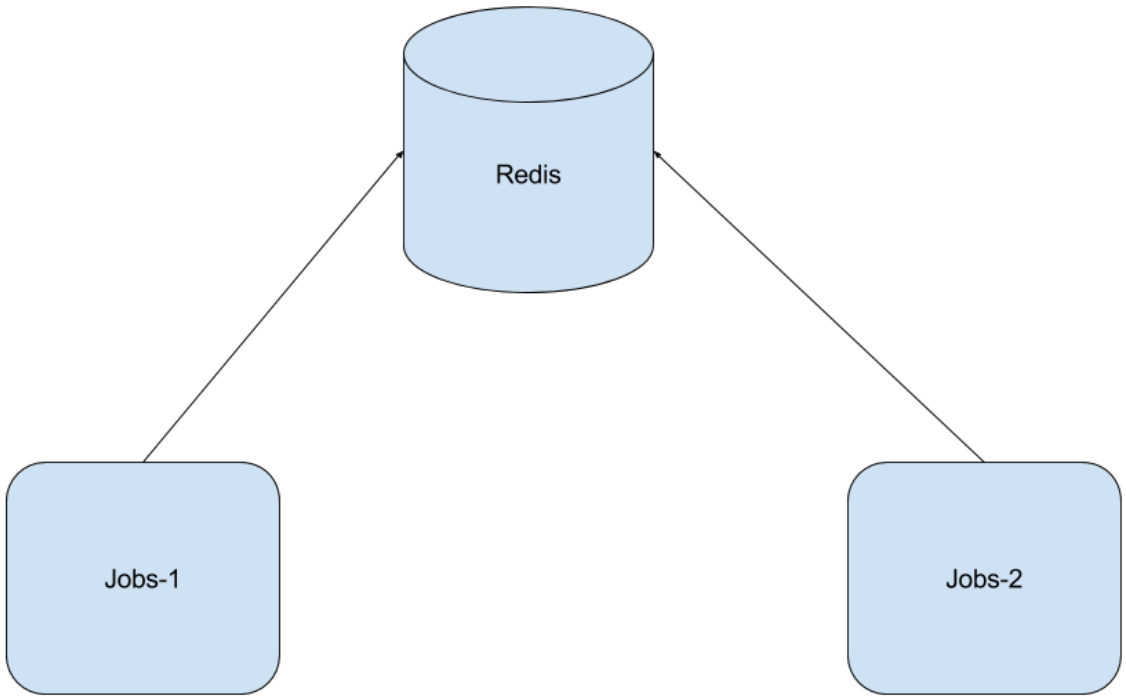
So, we decided to move on and re-implement our distributed locking API.

## The Proposal

The core ideas were to:

- Remove `/.*hazelcast.*/ig`;
- Implement the required interfaces using a Redis back-end;
- Start up an AWS ElastiCache cluster and use it at will.

tl;dr, this:



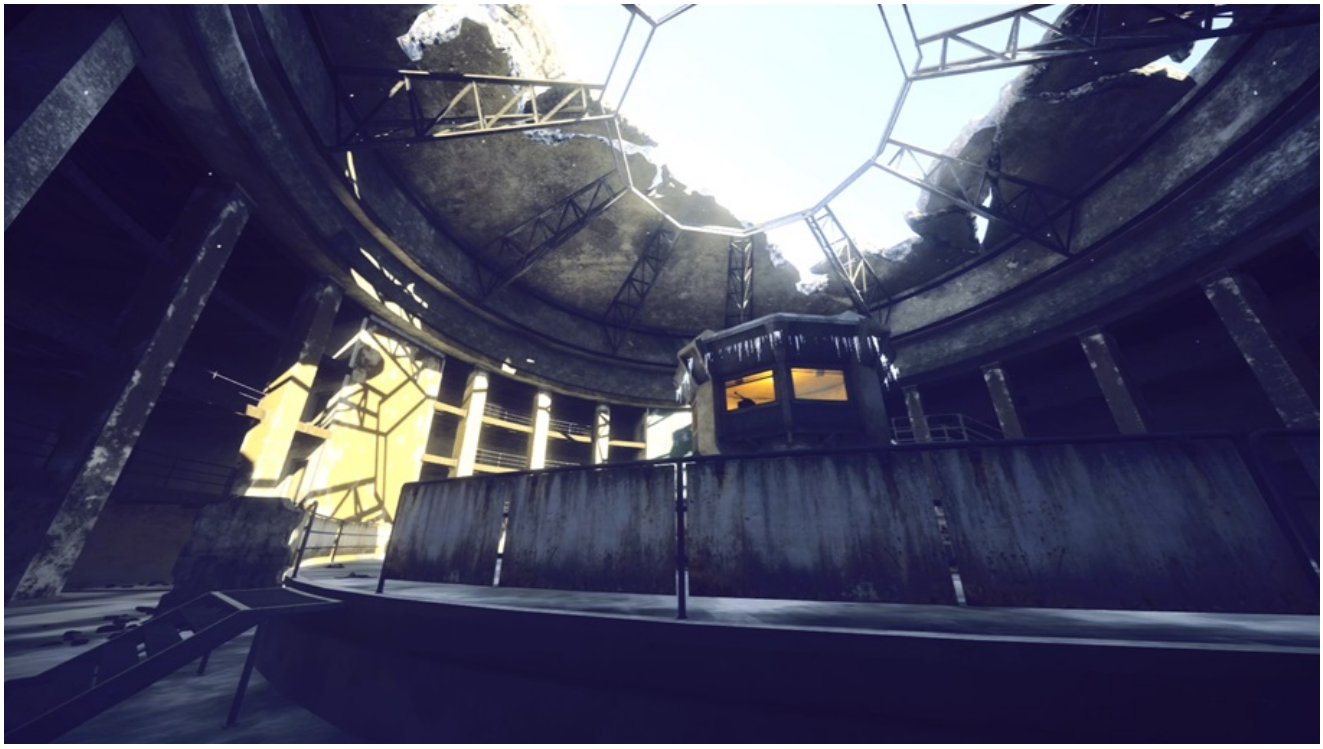

The reasons behind this decision were:

- Resolving the problems of the previous architecture;
- Simplify our actual architecture (and that's a good thing);
- The ElastiCache cluster is supposed to always be up, meaning less stuff for us to worry about;

But, of course, everything have a bad side:

- Redis would now be a depedency of our system (as Hazelcast already was);
- If, for any reason, the Redis cluster goes down, the entire jobs ecosystem simply stop working.

We called this project "*Operation Locker*", which is a very fun Battlefield 4 map:



# Implementation

Our distributed lock API required the implementation of two main interfaces to change its behavior:

`JobLockManager`:

```
public interface JobLockManager {
    <E> boolean lock(Job<E> job, E element);

    <E> void unlock(Job<E> job, E element);
```

```
    <E> void successfullyProcessed(Job<E> job, E element);
}
```

and `JobSemaphore`:

```
public interface JobSemaphore {
    boolean tryAcquire(Job<?> job);

    void release(Job<?> job);
}
```

We looked up several Java Redis libraries, and decided to use Redisson, mostly because it seems more actively developed. Then, we created a JBoss module with it and all its dependencies (after some classloader problems), implemented the required interfaces and put it to test, and, since it worked as expected, we shipped it to production.

After that, we decided to also change all other apps using the previous version of our API. We opened pull requests for all of them, tested in sandbox, and, finally, put them in production. Success!

# Results

We achieved a simplified architecture, reduced a little our time-to-production and improved our monitoring. All that with **zero downtime** and with ~4k less lines of code than before.

# Interesting links

- Distributed locks with Redis

- Distributed Locks using Golang and Redis
- How to do distributed locking
- How to create a distributed lock with Redis?
- Node distributed locking using Redis
- Distributed locks with Redis and Python
- Simplicity: A Prerequisite for Reliability

java contaazul



# Related posts

- Measuring production code coverage with JaCoCo
- Running a Selenium Grid with docker-compose
- I'm Joining TOTVS Labs