Everything I know about distributed locks

Davide Cerbo Follow
Oct 15, 2019 · 4 min read ★

Locking, often, isn't a good idea, and trying to lock something in a distributed environment may be more dangerous. But sometimes, we need to keep this risk and try to use a distributed lock for two main reasons:

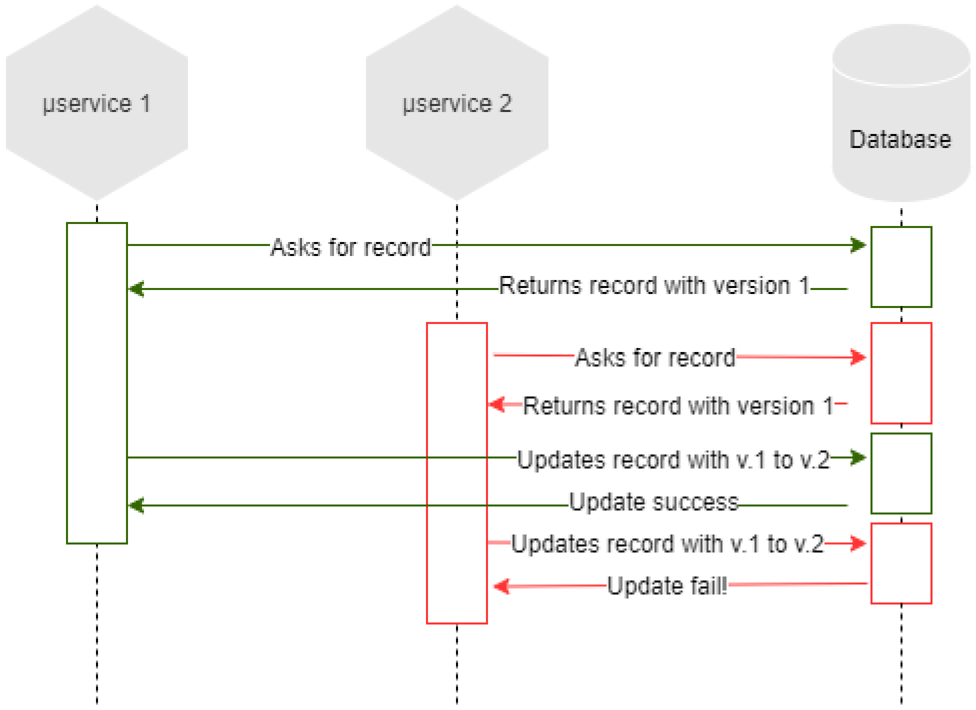
- **Efficiency:** a lock can save our software from performing unuseful work more times than it is really needed, like triggering a timer twice.

- **Correctness:** a lock can prevent the concurrent processes of the same data, avoiding data corruption, data loss, inconsistency and so on.

We have two kinds of locks:

- **Optimistic**: instead of blocking something potentially dangerous happens, we continue anyway, in the hope that everything will be ok.
- **Pessimistic**: block access to the resource before operating on it, and we release the lock at the end.

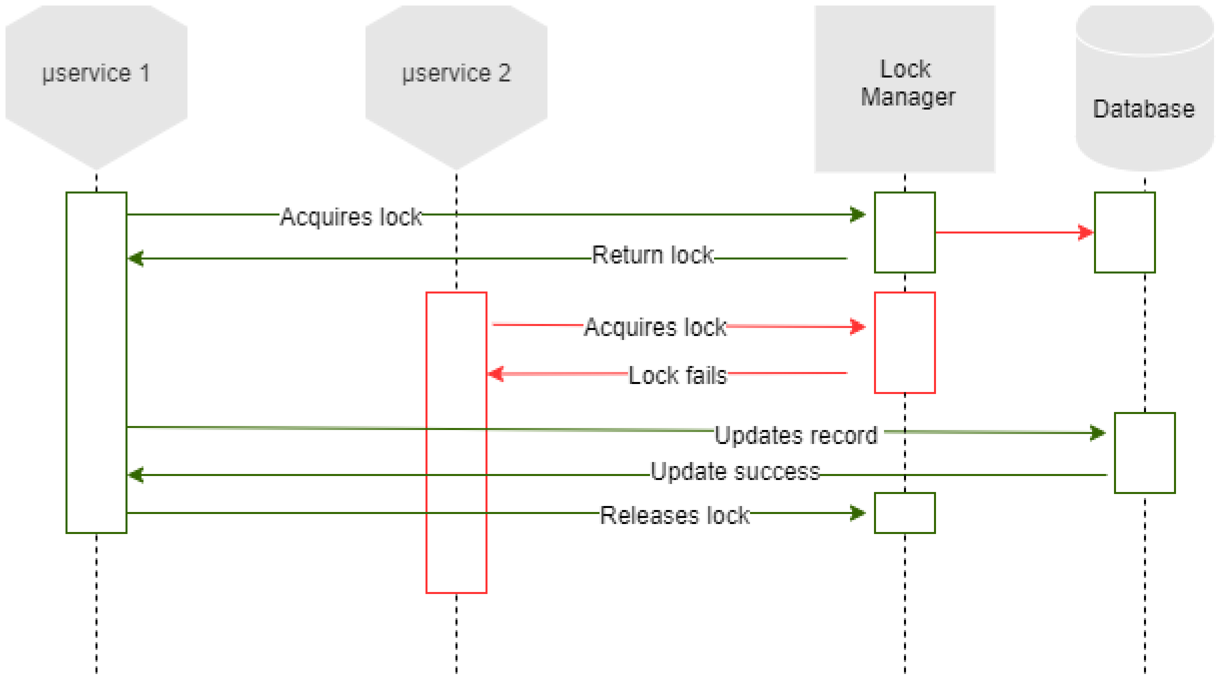
To use **optimistic** lock we usually use a version field on the database record we have to handle, and when we update it we check if the data we read has the same version of the data we are writing.


µservice 1
µservice 2
Database
Asks for record
Returns record with version 1
Asks for record
Returns record with version 1
Updates record with v.1 to v.2
Update success
Updates record with v.1 to v.2
Update fail!

Optimistic lock sequence diagram

Database access libraries, like Hibernate, usually provide facilities to use an optimistic lock.

The **pessimistic lock** instead will rely on an external system that will hold the lock for our microservices.



Pessimistic lock sequence diagram

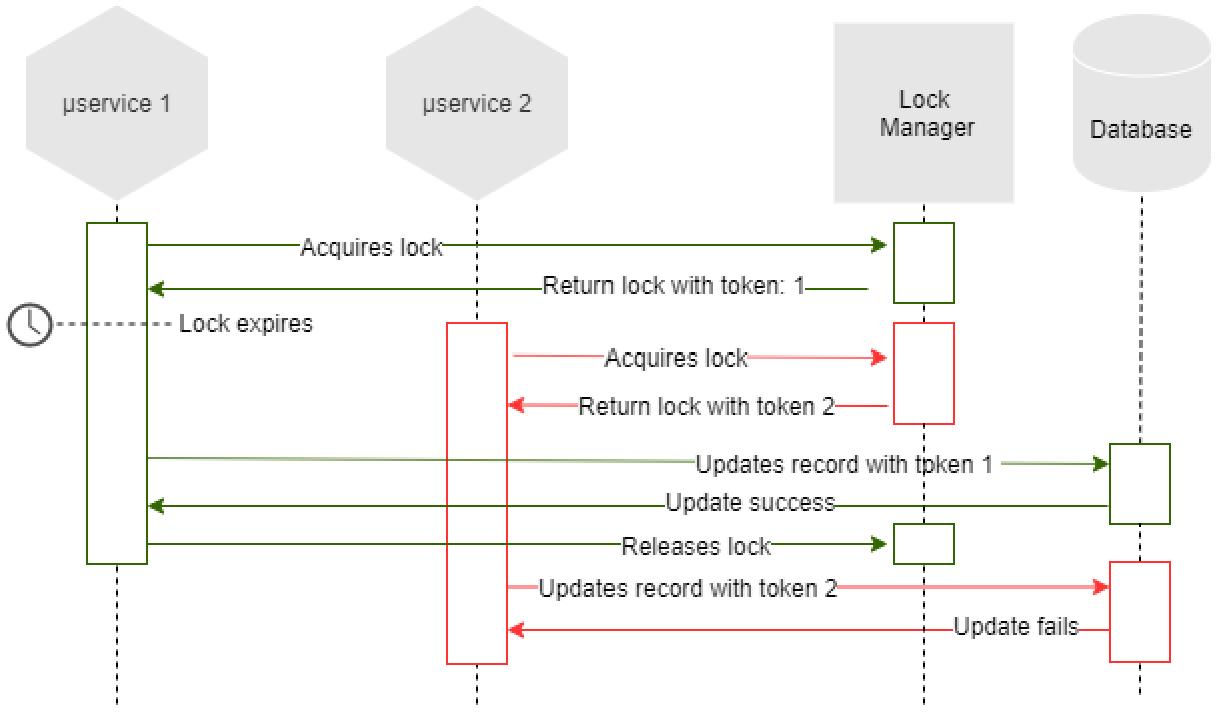
As for optimistic lock, database access libraries, like Hibernate usually provide facilities, but in a distributed scenario we would use more specific solutions that use to implement more complex algorithms like:

- Redis, using libraries that implements lock algorithm like ShedLock, and Redisson. The first one provides lock implementation using also other systems like MongoDB, DynamoDB, and more.
- Zookeeper, provides some recipes about locking.
- Hazelcast, offers a lock system based on his CP subsystem.

Implementing a pessimistic lock we have a big issue, what happened if the lock owner doesn't release it? If the lock owner dies? The lock will be held forever and we could be in a **deadlock**. To prevent this issue we will set an **expiration time** on the lock, so the lock will be **auto-released**.

But if the time **expires before the task handled by the owner isn't yet finished**, another microservice can acquire the lock, and both lock holders can now release the lock causing inconsistency. Remember, no timer assumption can be reliable in asynchronous networks.

We need to use a **fencing token** which is incremented each time a microservice acquires a lock. This token must be passed to the lock manager when we release the lock, so if the first owner releases the lock before the second owner, the system will refuse the second lock release. Depending on implementation we can also decide to let win the second lock owner.



Fencing token sequence diagram

As you can see in the image above the database has an active role in this scenario.

In the end, I have another question in my mind: **is it better to have one node, with a replica in case of disaster, or is it better to have a cluster of nodes?** The answer is: it depends :)

If we are looking to Redis, we should be careful of what shines and maybe better to avoid running 5 Redis servers and checking for a majority to acquire your lock, when a Redis server with a replica may be enough and, above all, maybe better because the lock will be faster and it will be cleaner where the lock is. About this, I suggest you read the Martin Kleppmann blog.

But also in general, when we are talking about lock it is better to have only one node in charge of a particular lock, then that more nodes in order to prevent the **split-brain** issue. If we would like to split the load we can elect a leader for a specific set of lock identifiers and contact always the same node. Some systems like Hezelcast and Zookeeper work using a **leader election**, but, take attention, their protocol cannot work in some network scenario, or the configuration may be hard (i.g. AWS BeanStalk), so a simple **Redis** server, also with its **cluster and data sharding** configuration, could be easier to set up and manage.

Resources

This is a fast overview of what you need to know about a distributed lock, but I strongly suggest you read some useful and more detailed blog posts:

- https://hazelcast.com/blog/long-live-distributed-locks/
- https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html
- https://carlosbecker.com/posts/distributed-locks-redis
- https://redis.io/topics/distlock

Redis Locks Distributed Systems Microservices Software Architecture