Distributed locks with Redis

Distributed locks are a very useful primitive in many environments where different processes must operate with shared resources in a mutually exclusive way.

There are a number of libraries and blog posts describing how to implement a DLM (Distributed Lock Manager) with Redis, but every library uses a different approach, and many use a simple approach with lower guarantees compared to what can be achieved with slightly more complex designs.

This page is an attempt to provide a more canonical algorithm to implement distributed locks with Redis. We propose an algorithm, called **Redlock**, which implements a DLM which we believe to be safer than the vanilla single instance approach. We hope that the community will analyze it, provide feedback, and use it as a starting point for the implementations or more complex or alternative designs.

Implementations

Before describing the algorithm, here are a few links to implementations already available that can be used for reference.

- Redlock-rb (Ruby implementation). There is also a fork of Redlock-rb that adds a gem for easy distribution and perhaps more.
- Redlock-py (Python implementation).
- Aioredlock (Asyncio Python implementation).
- Redlock-php (PHP implementation).
- PHPRedisMutex (further PHP implementation)
- cheprasov/php-redis-lock (PHP library for locks)
- Redsync.go (Go implementation).
- Redisson (Java implementation).
- Redis::DistLock (Perl implementation).
- Redlock-cpp (C++ implementation).
- Redlock-cs (C#/.NET implementation).
- RedLock.net (C#/.NET implementation). Includes async and lock extension support.
- ScarletLock (C# .NET implementation with configurable datastore)
- node-redlock (NodeJS implementation). Includes support for lock extension.

Safety and Liveness guarantees

We are going to model our design with just three properties that, from our point of view, are the minimum guarantees needed to use distributed locks in an effective way.

1. Safety property: Mutual exclusion. At any given moment, only one client can hold a lock.
2. Liveness property A: Deadlock free. Eventually it is always possible to acquire a lock, even if the client that locked a resource crashes or gets partitioned.
3. Liveness property B: Fault tolerance. As long as the majority of Redis nodes are up, clients are able to acquire and release locks.

Why failover-based implementations are not enough

To understand what we want to improve, let's analyze the current state of affairs with most Redis-based distributed lock libraries.

The simplest way to use Redis to lock a resource is to create a key in an instance. The key is usually created with a limited time to live, using the Redis expires feature, so that eventually it will get released (property 2 in our list). When the client needs to release the resource, it deletes the key.

Superficially this works well, but there is a problem: this is a single point of failure in our architecture. What happens if the Redis master goes down? Well, let's add a slave! And use it if the master is unavailable. This is unfortunately not viable. By doing so we can't implement our safety property of mutual exclusion, because Redis replication is asynchronous.

There is an obvious race condition with this model:

1. Client A acquires the lock in the master.
2. The master crashes before the write to the key is transmitted to the slave.
3. The slave gets promoted to master.
4. Client B acquires the lock to the same resource A already holds a lock for. **SAFETY VIOLATION!**

Sometimes it is perfectly fine that under special circumstances, like during a failure, multiple clients can hold the lock at the same time. If this is the case, you can use your replication based solution. Otherwise we suggest to implement the solution described in this document.

Correct implementation with a single instance

Before trying to overcome the limitation of the single instance setup described above, let's check how to do it correctly in this simple case, since this is actually a viable solution in applications where a race condition from time to time is acceptable, and because locking into a single instance is the foundation we'll use for the distributed algorithm described here.

To acquire the lock, the way to go is the following:

```
SET resource_name my_random_value NX PX 30000
```

The command will set the key only if it does not already exist (NX option), with an expire of 30000 milliseconds (PX option). The key is set to a value "my*random*value". This value must be unique across all clients and all lock requests.

Basically the random value is used in order to release the lock in a safe way, with a script that tells Redis: remove the key only if it exists and the value stored at the key is exactly the one I expect to be. This is accomplished by the following Lua script:

```
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

This is important in order to avoid removing a lock that was created by another client. For example a client may acquire the lock, get blocked in some operation for longer than the lock validity time (the time at which the key will expire), and later remove the lock, that was already acquired by some other client. Using just DEL is not safe as a client may remove the lock of another client. With the above script instead every lock is "signed" with a random string, so the lock will be removed only if it is still the one that was set by the client trying to remove it.

What should this random string be? I assume it's 20 bytes from /dev/urandom, but you can find cheaper ways to make it unique enough for your tasks. For example a safe pick is to seed RC4 with /dev/urandom, and generate a pseudo random stream from that. A simpler solution is to use a combination of unix time with microseconds resolution, concatenating it with a client ID, it is not as safe, but probably up to the task in most environments.

The time we use as the key time to live, is called the "lock validity time". It is both the auto release time, and the time the client has in order to perform the operation required before another client may be able to acquire the lock again, without technically violating the mutual exclusion guarantee, which is only limited to a given window of time from the moment the lock is acquired.

So now we have a good way to acquire and release the lock. The system, reasoning about a non-distributed system composed of a single, always available, instance, is safe. Let's extend the concept to a distributed system where we don't have such guarantees.

The Redlock algorithm

In the distributed version of the algorithm we assume we have N Redis masters. Those nodes are totally independent, so we don't use replication or any other implicit coordination system. We already described how to acquire and release the lock safely in a single instance. We take for granted that the algorithm will use this method to acquire and release the lock in a single instance. In our examples we set N=5, which is a reasonable value, so we need to run 5 Redis masters on different computers or virtual machines in order to ensure that they'll fail in a mostly independent way.

In order to acquire the lock, the client performs the following operations:

1. It gets the current time in milliseconds.
2. It tries to acquire the lock in all the N instances sequentially, using the same key name and random value in all the instances. During step 2, when setting the lock in each instance, the client uses a timeout which is small compared to the total lock auto-release time in order to acquire it. For example if the auto-release time is 10 seconds, the timeout could be in the ~ 5-50 milliseconds range. This prevents the client from remaining blocked for a long time trying to talk with a Redis node which is down: if an instance is not available, we should try to talk with the next instance ASAP.

3. The client computes how much time elapsed in order to acquire the lock, by subtracting from the current time the timestamp obtained in step 1. If and only if the client was able to acquire the lock in the majority of the instances (at least 3), and the total time elapsed to acquire the lock is less than lock validity time, the lock is considered to be acquired.
4. If the lock was acquired, its validity time is considered to be the initial validity time minus the time elapsed, as computed in step 3.
5. If the client failed to acquire the lock for some reason (either it was not able to lock N/2+1 instances or the validity time is negative), it will try to unlock all the instances (even the instances it believed it was not able to lock).

Is the algorithm asynchronous?

The algorithm relies on the assumption that while there is no synchronized clock across the processes, still the local time in every process flows approximately at the same rate, with an error which is small compared to the auto-release time of the lock. This assumption closely resembles a real-world computer: every computer has a local clock and we can usually rely on different computers to have a clock drift which is small.

At this point we need to better specify our mutual exclusion rule: it is guaranteed only as long as the client holding the lock will terminate its work within the lock validity time (as obtained in step 3), minus some time (just a few milliseconds in order to compensate for clock drift between processes).

For more information about similar systems requiring a bound *clock drift*, this paper is an interesting reference: Leases: an efficient fault-tolerant mechanism for distributed file cache consistency.

Retry on failure

When a client is unable to acquire the lock, it should try again after a random delay in order to try to desynchronize multiple clients trying to acquire the lock for the same resource at the same time (this may result in a split brain condition where nobody wins). Also the faster a client tries to acquire the lock in the majority of Redis instances, the smaller the window for a split brain condition (and the need for a retry), so ideally the client should try to send the SET commands to the N instances at the same time using multiplexing.

It is worth stressing how important it is for clients that fail to acquire the majority of locks, to release the (partially) acquired locks ASAP, so that there is no need to wait for key expiry in order for the lock to be acquired again (however if a network partition happens and the client is no longer able to communicate with the Redis instances, there is an availability penalty to pay as it waits for key expiration).

Releasing the lock

Releasing the lock is simple and involves just releasing the lock in all instances, whether or not the client believes it was able to successfully lock a given instance.

Safety arguments

Is the algorithm safe? We can try to understand what happens in different scenarios.

To start let's assume that a client is able to acquire the lock in the majority of instances. All the instances will contain a key with the same time to live. However, the key was set at different times, so the keys will also expire at different times. But if the first key was set at worst at time T1 (the time we sample before contacting the first server) and the last key was set at worst at time T2 (the time we obtained the reply from the last server), we are sure that the first key to expire in the set will exist for at least `MIN_VALIDITY=TTL-(T2-T1)-CLOCK_DRIFT`. All the other keys will expire later, so we are sure that the keys will be simultaneously set for at least this time.

During the time that the majority of keys are set, another client will not be able to acquire the lock, since N/2+1 SET NX operations can't succeed if N/2+1 keys already exist. So if a lock was acquired, it is not possible to re-acquire it at the same time (violating the mutual exclusion property).

However we want to also make sure that multiple clients trying to acquire the lock at the same time can't simultaneously succeed.

If a client locked the majority of instances using a time near, or greater, than the lock maximum validity time (the TTL we use for SET basically), it will consider the lock invalid and will unlock the instances, so we only need to consider the case where a client was able to lock the majority of instances in a time which is less than the validity time. In this case for the argument already expressed above, for `MIN_VALIDITY` no client should be able to re-acquire the lock. So multiple clients will be able to lock N/2+1 instances at the same time (with "time" being the end of Step 2) only when the time to lock the majority was greater than the TTL time, making the lock invalid.

Are you able to provide a formal proof of safety, point to existing algorithms that are similar, or find a bug? That would be greatly appreciated.

Liveness arguments

The system liveness is based on three main features:

1. The auto release of the lock (since keys expire): eventually keys are available again to be locked.
2. The fact that clients, usually, will cooperate removing the locks when the lock was not acquired, or when the lock was acquired and the work terminated, making it likely that we don't have to wait for keys to expire to re-acquire the lock.
3. The fact that when a client needs to retry a lock, it waits a time which is comparably greater than the time needed to acquire the majority of locks, in order to probabilistically make split brain conditions during resource contention unlikely.

However, we pay an availability penalty equal to TTL time on network partitions, so if there are continuous partitions, we can pay this penalty indefinitely. This happens every time a client acquires a lock and gets partitioned away before being able to remove the lock.

Basically if there are infinite continuous network partitions, the system may become not available for an infinite amount of time.

Performance, crash-recovery and fsync

Many users using Redis as a lock server need high performance in terms of both latency to acquire and release a lock, and number of acquire / release operations that it is possible to perform per second. In order to meet this requirement, the strategy to talk with the N Redis servers to reduce latency is definitely multiplexing (or poor man's multiplexing, which is, putting the socket in non-blocking mode, send all the commands, and read all the commands later, assuming that the RTT between the client and each instance is similar).

However there is another consideration to do about persistence if we want to target a crash-recovery system model.

Basically to see the problem here, let's assume we configure Redis without persistence at all. A client acquires the lock in 3 of 5 instances. One of the instances where the client was able to acquire the lock is restarted, at this point there are again 3 instances that we can lock for the same resource, and another client can lock it again, violating the safety property of exclusivity of lock.

If we enable AOF persistence, things will improve quite a bit. For example we can upgrade a server by sending SHUTDOWN and restarting it. Because Redis expires are semantically implemented so that virtually the time still elapses when the server is off, all our requirements are fine. However everything is fine as long as it is a clean shutdown. What about a power outage? If Redis is configured, as by default, to fsync on disk every second, it is possible that after a restart our key is missing. In theory, if we want to guarantee the lock safety in the face of any kind of instance restart, we need to enable fsync=always in the persistence setting. This in turn will totally ruin performances to the same level of CP systems that are traditionally used to implement distributed locks in a safe way.

However things are better than what they look like at a first glance. Basically the algorithm safety is retained as long as when an instance restarts after a crash, it no longer participates to any **currently active** lock, so that the set of currently active locks when the instance restarts, were all obtained by locking instances other than the one which is rejoining the system.

To guarantee this we just need to make an instance, after a crash, unavailable for at least a bit more than the max TTL we use, which is, the time needed for all the keys about the locks that existed when the instance crashed, to become invalid and be automatically released.

Using *delayed restarts* it is basically possible to achieve safety even without any kind of Redis persistence available, however note that this may translate into an availability penalty. For example if a majority of instances crash, the system will become globally unavailable for TTL (here globally means that no resource at all will be lockable during this time).

Making the algorithm more reliable: Extending the lock

If the work performed by clients is composed of small steps, it is possible to use smaller lock validity times by default, and extend the algorithm implementing a lock extension mechanism. Basically the client, if in the middle of the computation while the lock validity is approaching a low value, may extend the lock by sending a Lua script to all the instances that extends the TTL of the key if the key exists and its value is still the random value the client assigned when the lock was acquired.

The client should only consider the lock re-acquired if it was able to extend the lock into the majority of instances, and within the validity time (basically the algorithm to use is very similar to the one used when acquiring the lock).

However this does not technically change the algorithm, so the maximum number of lock reacquisition attempts should be limited, otherwise one of the liveness properties is violated.

Want to help?

If you are into distributed systems, it would be great to have your opinion / analysis. Also reference implementations in other languages could be great.

Thanks in advance!

Analysis of Redlock

1. Martin Kleppmann analyzed Redlock here. I disagree with the analysis and posted my reply to his analysis here.