Distributed lock

Use cases

- Efficiency: Taking a lock saves you from unnecessarily doing the same work twice (e.g. some expensive computation).
    - e.g. If the lock fails and two nodes end up doing the same piece of work, the result is a minor increase in cost (you end up paying 5 cents more to AWS than you otherwise would have)
    - e.g. SNS scenarios: A minor inconvenience (e.g. a user ends up getting the same email notification twice).
- Correctness: Taking a lock prevents concurrent processes from stepping on each others' toes and messing up the state of your system. If the lock fails and two nodes concurrently work on the same piece of data, the result is a corrupted file, data loss, permanent inconsistency, the wrong dose of a drug administered to a patient, or some other serious problem.

Requirementss

- Exclusive
- Avoid deadlock
- High available
- Reentrant

AP model - Redis

Internals

* Distributed lock needs to serialize the processing of different events. Redis is based on a single thread, which serialize different events in nature.

Efficiency implementation

Acquire lock

* Before Redis version 2.6.12, set and expire are two separate commands
  * Deadlock if SETNX succeed but EXPIRE fails

```
// return 1 if success; return 0 otherwise
SETNX Key Value   (key=lock id, value=currentTime + timeout)

// set expiration time
EXPIRE Key seconds

// Execute multiple commands
MULTI
EXEC
```

* Additional parameters could be passed to redis SET command (version 2.6.12)
  * SET resource_name my_random_value NX PX value

Release lock

* DELETE

Pros

* Lock is stored in memory. No need to access disk

Cons

Limited use cases

* Only applicable for efficiency use cases, not for correctness use cases.
  * Efficiency
    * You could use a single Redis instance, of course you will drop

    some locks if the power suddenly goes out on your Redis node, or something else goes wrong. But if you're only using the locks as an efficiency optimization, and the crashes don't happen too often, that's no big deal. This "no big deal" scenario is where Redis shines. At least if you're relying on a single Redis instance, it is clear to everyone who looks at the system that the locks are approximate, and only to be used for non-critical purposes.
  - Add on top of the single application case, you could use master-slave setup for high availability.
- Correctness
  - A simple master - slave setup won't work. Think about the following scenario:
    a. Client A writes an entry A to master.
    b. Master dies before the asynchronous replication of the write operation reaches slave.
    c. The slave becomes the master
    d. Client B writes the same entry A to original salve (current master)
    e. Now A and B share the same lock.
  - You will need to rely on Redlock. However, there are some concerns about it. To summarize:
    - Redlock does not have any facility to generate fencing tokens. And it is not straightforward to repurpose Redlock for generating fencing tokens.
      - Relying on expiration time to avoid deadlock is not reliable.
        - What if the lock owner dies? The lock will be held forever and we could be in a deadlock. To prevent this issue Redis will set an expiration time on the lock, so the lock will be auto-released. However, if the time expires before the task handled by the owner isn't yet finish, another microservice can acquire the lock, and both lock holders can now release the lock causing inconsistency.
        - A fencing token needed to be used to avoid race conditions. Please see this post for details.

        - Redlock depends on a lot of timing assumptions
            a. All Redis nodes hold keys for approximately the right length of time before expiring
            b. The network delay is small compared to the expiry duration
            c. Process pauses are much shorter than the expiry duration

Link to history on RedLock

- Typical failures causing failures of distributed locks
- What Redlock tries to solve?
    - 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:
        - Client A acquires the lock in the master.
        - The master crashes before the write to the key is transmitted to the slave.
        - The slave gets promoted to master.
        - Client B acquires the lock to the same resource A already holds a lock for. SAFETY VIOLATION!
- How to implement distributed lock with Redis, an algorithm called RedLock
    - How to implement it in a single instance case
    - How to extend the single instance algorithm to cluster
- A hot debate on the security perspective of RedLock algorithm.

CP model

Comparison

- Comparison between different ways to implement distributed lock
  - From the perspective of understanding difficulty (from low to high)
    - Database > Caching > Zookeeper
  - From the perspective of complexity of implementation (from low to high)
    - Zookeeper > Cache > Database
  - From a performance perspective (from high to low)
    - Cache > Zookeeper > = database
  - From the point of view of reliability (from high to low)
    - Zookeeper > Cache > Database

Database

Approach

- Create a row within database. When there are multiple requests against the same record, only one will succeed.

```
SELECT stock FROM tb_product where product_id=#{product_id};
UPDATE tb_product SET stock=stock-#{num} WHERE product_id=#
{product_id} AND stock=#{stock};
```

Pros and Cons

- Suitable for low concurrency scenarios
- Low performance because needs to access database

Zookeeper

Algorithm

- Consistency algorithm: ZAB algorithm
- To build the lock, we'll create a persistent znode that will serve as the parent. Clients wishing to obtain the lock will create sequential, ephemeral child znodes under the parent znode. The lock is owned by the client process whose child znode has the lowest sequence number. In Figure 2,

there are three children of the lock-node and child-1 owns the lock at this point in time, since it has the lowest sequence number. After child-1 is removed, the lock is relinquished and then the client who owns child-2 owns the lock, and so on.

- The algorithm for clients to determine if they own the lock is straightforward, on the surface anyway. A client creates a new sequential ephemeral znode under the parent lock znode. The client then gets the children of the lock node and sets a watch on the lock node. If the child znode that the client created has the lowest sequence number, then the lock is acquired, and it can perform whatever actions are necessary with the resource that the lock is protecting. If the child znode it created does not have the lowest sequence number, then wait for the watch to trigger a watch event, then perform the same logic of getting the children, setting a watch, and checking for lock acquisition via the lowest sequence number. The client continues this process until the lock is acquired.
- Reference: https://nofluffjuststuff.com/blog/scott_leberknight/2013/07/distributed_coordination_with_zookeeper_part_5_building_a_distributed_lock

Design considerations:

- How would the client know that it successfully created the child znode if there is a partial failure (e.g. due to connection loss) during znode creation
  - The solution is to embed the client ZooKeeper session IDs in the child znode names, for example child--; a failed-over client that retains the same session (and thus session ID) can easily determine if the child znode was created by looking for its session ID amongst the child znodes.
- How to avoid herd effect?
  - In our earlier algorithm, every client sets a watch on the parent lock znode. But this has the potential to create a "herd effect" - if every client is watching the parent znode, then every client is notified when any changes are made to the children, regardless of whether a client would be able to own the lock. If there are a small number of clients this probably doesn't matter, but if there are a large number it has the potential for a spike in network traffic. For example, the client owning child-9 need only watch the child immediately preceding it, which is most likely child-8 but could be an earlier child if the 8th child znode

somehow died. Then, notifications are sent only to the client that can actually take ownership of the lock.

Implementation

- https://time.geekbang.org/course/detail/100034201-119499

Pros and Cons

- Reliable
- Need to create ephemeral nodes which are not as efficient

etcd

Operations

1. business logic layer apply for lock by providing (key, ttl)
2. etcd will generate uuid, and write (key, uuid, ttl) into etcd
3. etcd will check whether the key already exist. If no, then write it inside.
4. After getting the lock, the heartbeat thread starts and heartbeat duration is ttl/3. It will compare and swap uuid to refresh lock

```
// acquire lock
curl http://127.0.0.1:2379/v2/keys/foo -XPUT -d value=bar -d
ttl=5 prevExist=false

// renew lock based on CAS
curl http://127.0.0.1; 2379/v2/keys/foo?prevValue=prev_uuid -
XPUT -d ttl=5 -d refresh=true -d prevExist=true

// delete lock
curl http://10.10.0.21:2379/v2/keys/foo?prevValue=prev_uuid -
XDELETE
```

Limitations

1. Todo: to add more detail "baiwan"