Leader election best practices

December 26, 2020

Sometimes a single process in a system needs to have special powers, like being the only one that can access a shared resource or assign work to others. To grant a process these powers, the system needs to elect a *leader* among a set of *candidate processes*, which remains in charge until it crashes or becomes otherwise unavailable. When that happens, the remaining processes detect that the leader is no longer available and elect a new one.

Leader election can be implemented without any external dependency like the Raft algorithm does. Doing so is far from trivial, though, and it's best avoided unless you have a very good reason to.

In practice, you can leverage an external linearizable key-value store, like etcd or Zookeeper, which offer abstractions that make it easy to implement leader election. The abstractions span from basic primitives like compare-and-swap to fully-fledged distributed mutexes.

Ideally, the external store needs to offer atomic compare-and-swap and expiration times (TTL) for keys. A compare-and-swap operation updates the value of a key if and only if the value matches the expected one, while an expiration time defines the time to live for a key, after which the key expires and is removed from the store if the lease hasn't been extended. Each process tries to acquire a "lease" by creating a new key with a specific TTL using compare-and-swap. The first process to succeed becomes the leader and remains such until it stops renewing the lease, after which another process can become the leader.

The TTL expiry logic can also be implemented on the client-side, like this locking library for DynamoDB does, but the implementation is more complex, and it still

requires the data store to offer a compare-and-swap operation.

You might think that's enough to guarantee there can't be more than one leader in your application. Unfortunately, that's not the case.

To see why, suppose there are multiple processes that need to update a file on a shared blob store, and you want to guarantee that only a single process at a time can do so to avoid race conditions. To achieve that, you decide to use a distributed mutex - a form of leader election. Each process tries to acquire the lock, and the one that does so successfully reads the file, updates it in memory, and writes it back to the store:

```
if lock.acquire():
    try:
        content = store.read(blob_name)
        new_content = update(content)
        store.write(blob_name, new_content)
    except:
        lock.release()
```

The problem here is that by the time the process writes the content to the store, it might no longer be the leader - a lot might have happened since it was elected. For example, the operating system might have preempted and stopped the process, and several seconds will have passed by the time it's running again. So how can the process ensure that it's still the leader then? It could check one more time before writing to the store, but that doesn't eliminate the race condition, it just makes it less likely.

To avoid this issue, the data store downstream needs to verify that the request has been sent by the current leader. One way to do that is by using a fencing token. A fencing token is a number that increases every time that a distributed lock is acquired - in other words, it's a logical clock. When the leader writes to the store, it passes down the fencing token to it. The store remembers the value of the last token and accepts only writes with a greater value:

```
success, token = lock.acquire()
if success:
    try:
        content = store.read(blob_name)
        new_content = update(content)
        store.write(blob_name, new_content, token)
    except:
        lock.release()
```

This approach adds complexity as the downstream consumer - in our case, the blob store - needs to support fencing tokens. If it doesn't, you are out of luck, and you will have to design your system around the fact that occasionally there will be more than one leader. For example, if there are momentarily two leaders and they both perform the same idempotent operation, no harm is done.

Although having a leader election can simplify the design of a system as it eliminates concurrency, it can become a scaling bottleneck if the number of operations performed by the leader increase to the point where it can no longer keep up. When that happens, you might be forced to re-design the whole system.

Also, having a leader introduces a single point of failure with a large blast radius - if the election process stops working or the leader isn't working as expected, it can bring down the entire system with it.

You can mitigate some of these downsides by introducing partitions and assigning a different leader per partition, but that comes with additional complexity. This is the solution many distributed data stores use.

Before considering the use of a leader, check whether there other are ways of achieving the desired functionality without it. For example, optimistic locking is one way to guarantee mutual exclusion at the cost of wasting some computing power. Or perhaps high availability is not a requirement for your application, in which case having just a single process that occasionally is restarted after failure is not a big deal.

As a rule of thumb, if you must use leader election, you have to minimize the work it

performs and be prepared to occasionally have more than one leader if you can't support fencing tokens end-to-end.

Written by Roberto Vitillo

← How distributed systems fail

Testing and operating distributed systems →