A

Leader election in distributed systems

ARCHITECTURE | LEVEL 300

Leader election is the simple idea of giving one thing (a process, host, thread, object, or human) in a distributed system some special powers. Those special powers could include the ability to assign work, the ability to modify a piece of data, or even the responsibility of handling all requests in the system.

Leader election is a powerful tool for improving efficiency, reducing coordination, simplifying architectures, and reducing operations. On the other hand, leader election can introduce new failure modes and scaling bottlenecks. In addition, leader election may make it more difficult for you to evaluate the correctness of a system.

Because of these complications, we carefully consider other options before implementing leader election. For data processing and workflows, workflow services like AWS Step Functions can achieve many of the same benefits as leader election and avoid many of its risks. For other systems, we often implement idempotent APIs, optimistic locking, and other patterns that make a single leader unnecessary.

In this article, I discuss some of the pros and cons of leader election in general and how Amazon approaches leader election in our distributed systems, including insights into leader failure.

Advantages and disadvantages of leader election

Leader election is a common pattern in distributed systems because it has some significant advantages:

• A single leader makes systems easier for humans to think about. It puts all the concurrency in the system into a single place, reduces partial failure modes, and adds a single place to look for logs and metrics.
• A single leader can work more efficiently. It can often simply inform other systems about changes, rather than building consensus about the changes to be made.
• Single leaders can easily offer clients consistency because they can see and control all the changes made to the state of the system.
• A single leader can improve performance or reduce cost by providing a single consistent cache of data which can be used every time.
• Writing software for a single leader may be easier than other approaches like quorum. The single leader doesn't need to consider that other systems may be working on the same state at the same time.

Leader election also has some considerable downsides:

• A single leader is a single point of failure. If the system fails to detect or fix a bad leader, the whole system can be unavailable.
• A single leader means a single point of scaling, both in data size and request rate. When a leader-elected system needs to grow beyond a single leader, it requires a complete re-architecture.
• A leader is a single point of trust. If a leader is doing the wrong work with nobody checking it, it can quickly cause problems across the entire system. A bad leader has a high blast radius.
• Partial deployments may be hard to apply in leader-elected systems. Many software safety practices at Amazon depend on partial deployments, such as one-box, A-B testing, blue/green deployment, and incremental deployment with automatic rollback.

Many of these disadvantages can be mitigated by carefully choosing the leader's scope. How much of the system or data does the leader own? A common pattern here is sharding. Each item of data still belongs to a single leader, but the whole system contains many

leaders. This is the fundamental design approach behind Amazon DynamoDB (DynamoDB), Amazon Elastic Block Store (Amazon EBS), Amazon Elastic File System (Amazon EFS), and many other systems at Amazon. Sharding has its own downsides, though. Specifically, more design complexity and the need to carefully think through how to shard data.

How Amazon elects a leader

There are many ways to elect a leader, ranging from algorithms like Paxos, to software like Apache ZooKeeper, to custom hardware, to leases. Leases are the most widely used leader election mechanism at Amazon. Leases are relatively straightforward to understand and implement, and they offer built-in fault tolerance. Leases work by having a single database that stores the current leader. Then, the lease requires that the leader *heartbeat* periodically to show that it's still the leader. If the existing leader fails to heartbeat after some time, other leader candidates can try to take over.

We avoid depending on time in distributed systems, even with the great time sync feature in Amazon Elastic Compute Cloud (Amazon EC2). It's difficult to ensure that system clocks across a cluster are synchronized enough to depend on that synchronization for ordering or coordinating distributed operations. In Amazon, distributed systems only use time for human consumption. Leases depend on time. However, they depend only on local elapsed time *duration*, rather than a wall-clock time that is synchronized and needs to be agreed upon by multiple servers.

The DynamoDB lock client source code offers examples and details related to leader election. However, we've found that, although leases and locks are conceptually simple, correct implementation can be subtle. Implementations require an understanding of how a server is measuring local time duration. For example, if a server or library that

measures time thought that time jumped backwards occasionally, it would break the assumptions about time durations that are built into leases. Durations sidestep problems with global clock synchronization that cause servers to stop agreeing on what time it is, from leap seconds to local clock drift over time from sustained high CPU utilization.

A larger issue for leases, and all kinds of distributed locks, is making sure that the leader is only doing work while it holds the lock. Ensuring that the leader holds the lock is actually fairly hard. We've found it important to ensure that a leader on a slow or lossy network doesn't believe that it's holding locks longer than it really is. Similarly, garbage collection pauses between a lock being checked and work being done can lead to incorrect behavior. In practice, hardening against these issues is often the biggest challenge.

DynamoDB and ZooKeeper offer simple lease-based locking clients which provide fault-tolerant leader election. Unless there are particular needs, we prefer these clients as we think they provide the easiest and most tested way to implement leader election. Amazon teams prefer to avoid creating a custom leader election implementation. Instead, we favor existing clients that are well-tested and battle-hardened.

Examples of systems using leader election at Amazon
Leader election is a widely deployed pattern across Amazon. For example:

• Almost all systems using traditional relational database management systems (RDBMSs) rely on leader election to pick a leader database which handles all writes, and sometimes, all reads. In these systems, election may be automated, but it's frequently done manually by a human operator.
• Amazon EBS distributes reads and writes for a volume over many storage servers. To ensure consistency, it uses leader election to elect primaries for each area of the volume, which order the reads and

writes. If that primary fails, follower copies steps in using the same leader election mechanism. In Amazon EBS, leader election ensures consistency, while improving performance by avoiding coordination on the data plane. DynamoDB, Amazon Quantum Ledger Database (Amazon QLDB), and Amazon Kinesis (Kinesis) use similar approaches for the same reason.
• The Kinesis Client Library (KCL) uses leases to ensure that each Kinesis shard is processed by one owner, making it easy to do scale-out processing of Kinesis streams.

What happens when the leader fails?

Another thing to think about carefully is what happens to a leader's work when it fails. If a leader fails during a task, how does the new leader complete the task? If a leader fails before making its work durable, is the system still correct? Many kinds of systems have separate "make work durable" and "tell others it is complete" steps. At Amazon, our systems always do the former before the latter (or tolerate data loss). Again, *idempotency* is useful here. It allows the new leader to confidently redrive work that the outgoing leader may have partially completed or completed but didn't tell others about.

To tolerate failures, Amazon distributed systems don't have a single leader. Instead, leadership is a property that passes from server to server, or process to process. In distributed systems, it's not possible to guarantee that there is exactly one leader in the system. Instead, there can mostly be one leader, and there can be either zero leaders or two leaders during failures.

How we choose the system's behavior on leader failure depends on what happens in a system when there are two leaders. Systems that perform work which is *idempotent* can often tolerate two leaders with minimal loss of efficiency. With two leaders, systems can achieve higher availability and choose weaker leader election approaches.

Systems that absolutely need at most one leader are harder to build than multiple leader systems. The leader election system must always be correct and consistent. And, it must ensure that the outgoing leader is deposed before a new leader is elected, which is harder than it seems. In distributed systems, it's often difficult to know if a system has failed or whether it's simply continuing to do work in some other network partition. At Amazon, we ensure that any leader-elected system handles this edge case.

Best practices for leader election

At Amazon, we follow these best practices for leader election:

• Check the remaining lease time (or lock status in general) frequently, and especially before initiating any operation that has side-effects beyond the leader itself.
• Consider that slow networking, timeouts, retries, and garbage collection pauses can cause the remaining lease time to expire before the code expects it to.
• Avoid heartbeating leases in a background thread. This can cause correctness issues if the thread can't interrupt the code when the lease expires or the heartbeating thread dies. Availability issues can occur if the work thread dies or stops while the heartbeating thread holds on to the lease.
• Have reliable metrics that show how much work a leader can do versus how much it is doing now. Review these metrics often, and make sure that there are plans for scaling in advance of running out of capacity.
• Make it easy to find which host is the current leader and which host was the leader at any given time. Keep an audit trail or log of leadership changes.
• Model and formally verify the correctness of distributed algorithms using tools like TLA+. This catches subtle, difficult to observe, and rare

bugs that can creep in when an application assumes too much about the guarantees provided by the leader election protocol.

Conclusion

Leader election is a powerful tool used in systems across Amazon to help make our systems fault-tolerant and easier to operate. However, when we use leader election, we're careful to consider the guarantees that each leader election protocol provides, and more importantly, doesn't provide.

Amazon systems often use leader election to ensure there is fault-tolerance built in. When systems use leader election to ensure that at least one server is processing a task, they use separate mechanisms for maintaining correctness in the face of multiple concurrent leaders. For example, they might use an underlying database to make sure that if two leaders think they are both holding a lease, they don't interfere with each other. Rather than making assumptions about the guarantees provided by a lease implementation, we focus on correctness in these systems, often with modeling with techniques like TLA+.

Despite its nuances, leader election remains a useful tool in our distributed systems toolkit at Amazon, alongside patterns like idempotency and optimistic locking.

Further reading

For more information about how leases work, see: