elastic.co

Leader Election, Why Should I Care?

ByKonrad Beiske

14-17 minutes

UPDATE: This article refers to our hosted Elasticsearch offering by an older name, Found. Please note that Found is now known as Elastic Cloud.

Leader election is one of the most tricky things to do in distributed systems. At same time, understanding how a leader is elected and the responsibilities of the leader is key to understanding a distributed system.

Introduction

All distributed systems have to deal with maintaining consistency in one way or another. To simplify, we may divide the strategies into two groups. The ones that attempt to avoid inconsistencies from occurring and the ones that define how to reconcile them. The latter is very powerful for applicable problems, but imposes unacceptable constraints on the data model in others. In this article we will look into one example of the first category and how it copes with network failures.

Why Use a Leader?

Having a node designated as a leader in a distributed system is actually quite similar to the keyword synchronized in Java. The classic example for Java synchronized is when two threads attempt to increment the same integer. For instance, when two threads attempt to deposit $100 each to the same account. Each thread has to read the balance, increment it and write it back. Given a starting balance of $100 and no thread synchronization the following may occur:

- Thread A reads balance ($100)
- Thread B reads balance ($100)
- Thread A calculates new balance ($200)
- Thread B calculates new balance ($200)
- Thread A writes new balance ($200)
- Thread B writes new balance ($200)

Clearly the account started with $100, a total of $200 was deposited, and the final balance should have been $300. However, thread A's deposit is overwritten as thread B has done its calculation on data that is stale by the time it writes its updated balance.

The solution in Java is of course to use the keyword synchronized on the code doing the incrementation and it will make one thread finish its work with the balance before the other gets to read the balance.

If you consider a thread as a node, it's easy to imagine the same problem in a distributed system, but there is no synchronized keyword to the rescue. The reason being that synchronized is implemented using semaphores, and in turn, they are backed by the underlying operating system and the hardware it's running on. But if we look at what synchronized does, we might adapt the concept to a distributed setting. When declaring a block of code as synchronized in Java, it is always synchronized to the monitor of a specific object. The size of this monitor is one. This means that only one thread may be inside the monitor at any given time. When a thread requests the monitor and the monitor is available, it is allowed to enter instantly. If the monitor is occupied, the thread is put in the waiting pool and suspended until the monitor is released.

How can we adapt this to a distributed setting? Well, we do have the option to create as many monitors as we want on each node, but for a given resource that requires protection there may only exist one monitor. In other words, there may only be one monitor for each account balance. This transforms the problem of maintaining data consistency to the problem of ensuring all nodes agree on which node is responsible for implementing the monitor for each balance. In our simple example this could be handled by the leader, and all that remains to be done is to elect a leader.

If you have a peer to peer background like myself you would probably suggest using a distributed hash table rather than a leader to decide which node implements the monitor for each resource. In fact, there are some really great DHT-implementations out there that deal with thousands of nodes leaving and joining every hour. Given that they also work in a heterogenous network with no prior knowledge of underlying network topology, query response times of four to ten message hops is pretty good, but in a grid setting where the nodes are homogenous and the network is fairly stable, we can do better.

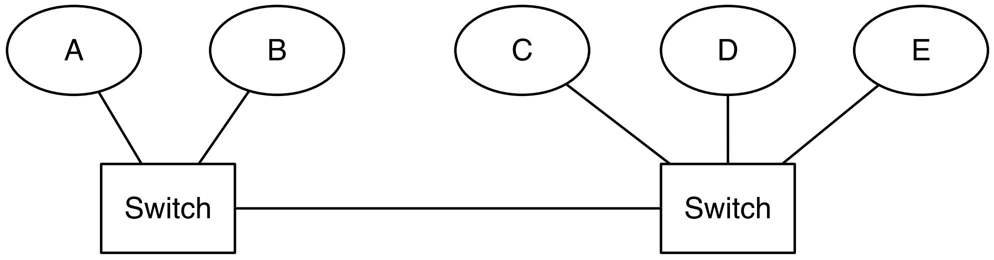
Another simplification in the typical scenario for Elasticsearch is that there are not that many nodes in the cluster. Typically, the number of nodes is far less than the number of connections a single node is capable of maintaining, and a grid environment does not have to deal with nodes joining and leaving that frequently. This is why the leader approach is a better fit for Elasticsearch.

The Zen Way

So we have a leader that regularly pushes the current version of the global cluster state to each node. What happens if the leader goes down? Just because our nodes are fairly reliable does not imply that we can accept a single point of failure. Provided that only the leader node has crashed and all other nodes are still capable of communicating, Elasticsearch will handle this gracefully as the remaining nodes will detect the failure of the leader - or rather the absence of messages from the leader - and initiate leader election. If you are using ZenDiscovery (default) then the process is like this:

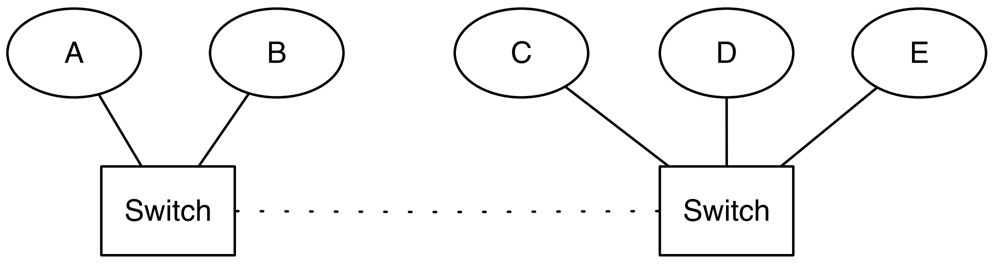
- Each node calculates the lowest known node id and sends a vote for leadership to this node
- If a node receives *sufficiently many votes* and the node also voted for itself, then it takes on the role of leader and starts publishing cluster state.

The definition of *sufficiently many votes* to win an election is what is known as a quorum. In Elasticsearch the quorum size is a configurable parameter. A node in a distributed system is not able to determine whether another node is dead or whether the network is not able to deliver its messages to it, it is therefore common to have a previously agreed upon threshold - the quorum size - to represent the other party's votes. Let's exemplify with the following scenario: A cluster has nodes A, B, C, D, E and quorum size 3. A is the leader. It so happens that A and B is on one network and C, D and E on another. The networks are connected through a link.



Initial network

When this link fails, A and B are still able to communicate with each other, but not with C, D and E. Similarly, on the other network C, D and E may communicate with each other, but not with A and B.



Partitioned network

What happens next is this: nodes C, D and E detect that they no longer have contact with the leader A and subsequently initiate leader election by sending votes to C. Once C has received three votes it takes on the role of leader and starts publishing to C, D and E. On the other network, the leader A detects that it no longer has contact with nodes C, D and E. Leader A calculates that the new cluster size is less than the quorum size and gives up the leader role and effectively stops nodes A and B from responding to queries until the link is restored.

In real life it's unlikely that someone trips over on a crucial network cable, but networks are more complex than my example above, and network partitions are actually not that uncommon. It is not hard to imagine a network split when a system is relying on multiple datacenters; another likely culprit of tricky network errors is a wrongly configured router or switch. As mentioned in the network is reliable, it does occur that network cards do things like dropping all inbound packets while still delivering outbound packets, with the result that a server still sends heartbeats but is unable to service any requests.

Avoid Split Brain

The concept of a quorum size has two immediate advantages. Firstly it simplifies the election process as votes only need to be delivered to the node they're voting for. Secondly and more importantly it is the only way to avoid a split brain. Imagine the same cluster and the same network split, but with a quorum size of two. C would still be elected leader, but A would not give up its leader role. This would result in two leaders, with clusters unknown to each other. This is what is known as a split brain. Both clusters would be accepting read and write operations, and as expected, they would be out of sync. Without human intervention they would probably never recover. Depending on the data model it might be impossible to unify the two data versions, forcing one to simply discard all data on one of the clusters.

Don't Reinvent the Wheel

As you might expect, leader election has been an intriguing topic in academic circles for many years and quite a few smart people have done a great deal of contemplation. If you are more keen on getting your distributed system working than putting a massive effort into research and development, you're probably better off with having a go at implementing a well known algorithm. Some day in the future when you have finally figured out that crazy bug in your system, it's more likely to be just a bug in the implementation than a major design flaw in the algorithm. Of course, the same argument applies to adapting or integrating an existing implementation rather than implementing from scratch, especially if you want one of the more advanced algorithms. This bug report illustrates how tricky it can be to create a good leader election implementation, even when you have a sound algorithm.

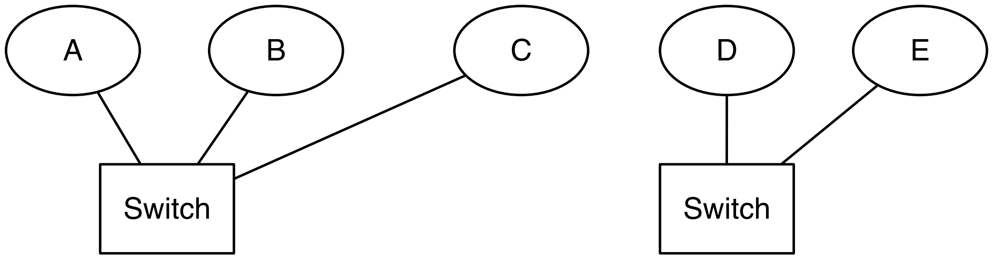
The Bully Algorithm

The bully algorithm is one of the basic algorithms for leader election. It assumes that all nodes are given a unique ID that imposes a total ordering of the nodes. The current leader at any time is the node with the highest id participating in the cluster. The advantage of this algorithm is an easy implementation, but it does not cope well with a scenario whereby the node with the largest id is flaky. Especially in a situation where the node with the largest id tends to be overloaded by the chores of the leader role. Consequently, it will crash as the leader; the node with the second largest id will be elected; and the largest id node will recover - as it's no longer overloaded - and subsequently initiate leader election again, only to be elected and crash yet again. However, in low level hardware interfaces the bully algorithm is quite common. It might be tempting to avoid thrashing by postponing election until the current leader fails, but that will easily lead to a split brain.

Paxos

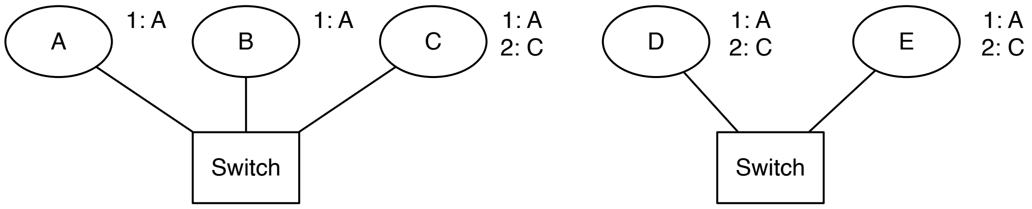
Paxos is actually a much more than a leader election algorithm. Paxos is a family of different protocols for maintaining consensus in a distributed system. I will not go into detail about the varieties of Paxos, but rather discuss the concepts and qualities of Paxos in general. The data model used in Paxos is a growing list of statements where each statement has a unique number. The mathematically proven guarantees of Paxos are that two nodes will never have different statements for the same number, and as long as there is a quorum of nodes participating the algorithm will make progress. This translates to the nodes never being inconsistent and the algorithm never going into deadlock, but it may halt if there are not enough nodes online. These guarantees are in fact quite similar to what we want for a leader election algorithm. We want all nodes participating in the cluster to agree on which node is the leader, and we do not want any other nodes to be able to run off and create their own cluster, resulting in a split brain. Performing leader election with Paxos then becomes as simple as proposing the statement “I am the leader”. If it is accepted, then you are the leader until another node proposes to be the leader. However, this alone is not sufficient to avoid a split brain. In the partitioned network example above the existing leader A will not be notified of the election of Node C on the other switch. There needs to be another criteria to end the leadership. One option is that when initiating leader election the statement used is on the form: “I am leader until ”, but a more preferred option is that the leader is able to detect that it no longer has contact with a quorum and then demotes itself.

Given the previous example of a network partition, let’s imagine that a janitor trips on the network cable to node C and then reconnects the cable to the other switch, resulting in a new partition of the network:



Moved node

The interesting part here is that the quorum now consists of nodes A, B and C instead of the previous C, D and E. If the nodes where running Paxos, their data could look something like this:



Paxos example

Depending on the exact implementation of leader election through Paxos, a number of outcomes are possible, but they all have two properties in common. Nodes D and E will not be able to elect a new leader as they do not have a majority. Nodes A and B will not elect a new leader before learning that node C was elected in round 2. Assuming that the end criterion of each leader term is a timeout, there are two possible outcomes. If node C establishes contact with nodes A and B before its term is ended, it will reestablish a quorum, resume its leader role, and no leader election is necessary. If its term is ended, then either node A, B or C may be elected. Assuming node A is the first to discover node C on the network, it will try to get re-elected as leader for term 2, but node C will reject this as it already has a leader for term 2. Node A then has the option to attempt to be elected for term 3. If it does, it will then also inform node B about node C's time as leader in term 2. Now, what if node C and A try to get elected at the same time? The full answer to that question requires delving into the specifics of Paxos, which out of scope for this article, but the short answer is that election will fail. In fact, the original paxos paper suggests using a leader election algorithm to decide which node should be the one initiating new proposals, but since the Paxos protocol guarantees that there will not be inconsistencies even if there are multiple nodes proposing conflicting statements, this leader election algorithm could be just a simple heuristic to avoid consecutive failed rounds. For instance, a node might decide to wait a randomized amount of time before reattempting to propose a failed statement. This flexibility of Paxos in terms of when and how to do leader election can be a great advantage over the simple bully algorithm, having in mind that in real life there are far more failure modes than a dead network link.

Conclusion

In this article I have tried to give a brief insight into some of the work that has gone into leader election in academic circles and the importance of doing it right. The condensed version is this:

- Quorum size is really important if you care about your data.
- Paxos is really robust, but not as easy to implement.
- It’s better to implement a well known algorithm where the strengths and flaws are known rather than getting a big surprise further down the road. There are certainly plenty of options out there.