Achieving Consensus in a Cluster – Part 2

This lesson continues the discussion about achieving consensus in a cluster.

We'll cover the following

- Popular algorithms that both distributed systems and the decentralized systems implement to reach a consensus
- Leader election
- Distributed node coordination with Zookeeper

Popular algorithms that both distributed systems and the decentralized systems implement to reach a consensus

When developing distributed or decentralized systems, there are several algorithms that can be used to achieve a consensus among the nodes in the cluster.

First off, if you are not clear on the difference between the terms distributed and decentralized, I've written an article on it on my blog. Do check it out.

A few of the popular and widely used consensus algorithms are:

- *Byzantine Fault Tolerance algorithm*
- *Paxos algorithm*
- *Raft algorithm*
- *Proof of Work*
- *Proof of Stake*
- *Proof of Burn*
- *Proof of Authority*

Google's distributed lock service *Chubby* uses the *Paxos consensus algorithm* to keep the nodes in consensus. It uses a replicated database to store the lock-related information. The replication enables the service to be available when the nodes fail in the cluster. The replication database is written on top of a fault-tolerant log

layer that uses the Paxos consensus algorithm.

Bitcoin uses the *Proof of Work* algorithm to maintain a consensus among the nodes in its peer to peer network. Different cryptocurrencies use different consensus algorithms to get the desired behavior from their network.

Discussions on how these algorithms work are beyond the scope of this course. Unless you intend to write distributed systems like *Google Big Table, Google File System, Spanner*, and so on from the bare bones, it's rare that you would be developing these systems from scratch in your day to day work as an application developer. This is because these systems are complex; they achieve perfection after years of testing, deployment in production, continual patching, and etc. Writing these products is time-consuming and resource-intensive.

One does not simply write them for fun. Large scale internet services like *Facebook, Google, and Uber* build these systems incrementally over long periods of time. These are then deployed internally in production, continually monitored, tested, and patched. When their product appears stable, these companies release it as open-source, and some may even choose to offer them as a managed service, as Google Cloud does.

The cloud products that *Google Cloud* offers are the same that Google uses internally to run its services. Everyone prefers to use these open-sourced, distributed systems to write their service as opposed to building them from the bare bones and then writing their service on it.

When using these services, we can be certain that we won't stumble against any major design-related or security issues in the long run. Since these companies, which handle billions of users every month, first use these services internally before releasing them to the public, there is a trust factor that comes along with these products.

> ***Recommended read***: How do Google Services store petabyte exabyte scale data?

If you are interested in reading about how these distributed systems work behind the scenes, what the underlying architecture is, and so on. I recommend going through the whitepapers of Google Cloud BigTable, Google File System, and

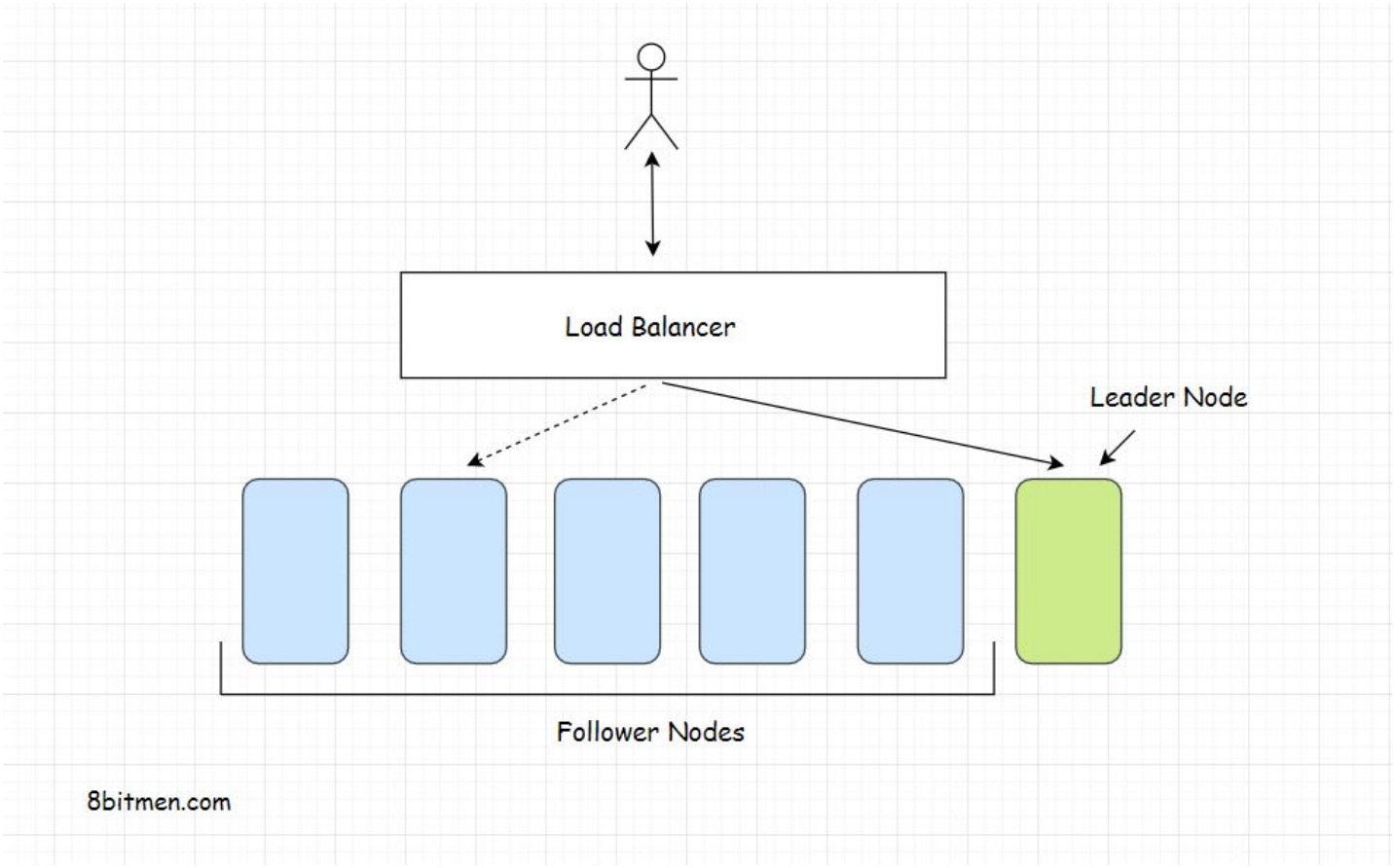
MillWheel.

Alright!! Now, it's time to talk about *leader election*.

Previously in this chapter, I brought up that the cluster elects a leader among the nodes to facilitate smooth organized consistent communication. Let's find out how the leader is generally elected in a cluster.

Leader election

A leader node among all the other nodes in a cluster is a privileged one. It has the power to assign work to the other nodes, distribute tasks to them, modify data, and so on. It is given the onus of facilitating cluster coordination efficiently.



Different algorithms have different strategies to elect a leader. I'll give you a general idea of how a leader is elected in a cluster.

The consensus algorithm keeps the cluster functional as long as the majority (*51%*) of the nodes are running. When a new node is added to the cluster, it joins as a *follower node*. It waits for the heartbeat message from the leader of the cluster to acknowledge it's onboarding.

In case it doesn't receive the message from the leader within a stipulated time, it becomes eligible to become the leader. Now, the new node broadcasts a message to

all the other nodes in the cluster that it wants to become the leader. Based on a set of rules, the other nodes in the cluster may accept or reject the proposal.

If the majority of the nodes accept the proposal, the newly joined node becomes the leader. Only the leader has the right to write updates to the cluster log.

If you are interested in delving into detail. You can read about the Raft Consensus algorithm here and the Paxos algorithm here.

When developing our system, we can write our own implementation of selecting a leader in a cluster, or we can use a library dedicatedly written to handle these cluster management tasks for us, like *Apache Zookeeper*.

Distributed node coordination with Zookeeper

Zookeeper is an open-source, widely used, and battle-tested distributed node coordination service. It takes care of all the cluster management tasks, such as assigning unique IDs to thousands of instances and facilitating coordination between them, for us. The service also offers distributed synchronization features, like locks, queues, leader election, and so on.

It is used by most of the large-scale services to share configuration information across the cluster (click here for details.) *Yahoo* uses *Zookeeper* for leader election, config management, sharding, locking, and etc. *Zynga* uses *Zookeeper* for the same. Meanwhile, *Rackspace* uses the service to implement distributed locking.

You can read more on how Zookeeper works behind the scenes, here.

By now, I believe you have a good understanding of the fundamentals of *clustering*. I'll discuss cluster management and monitoring later in this course when I discuss *containers, virtual machines, Docker*, and *Kubernetes*.

In the next chapter, let's take a look at how the cloud deploys our applications across the globe in different data centers.

But first, you'll take a short quiz in the next lesson.