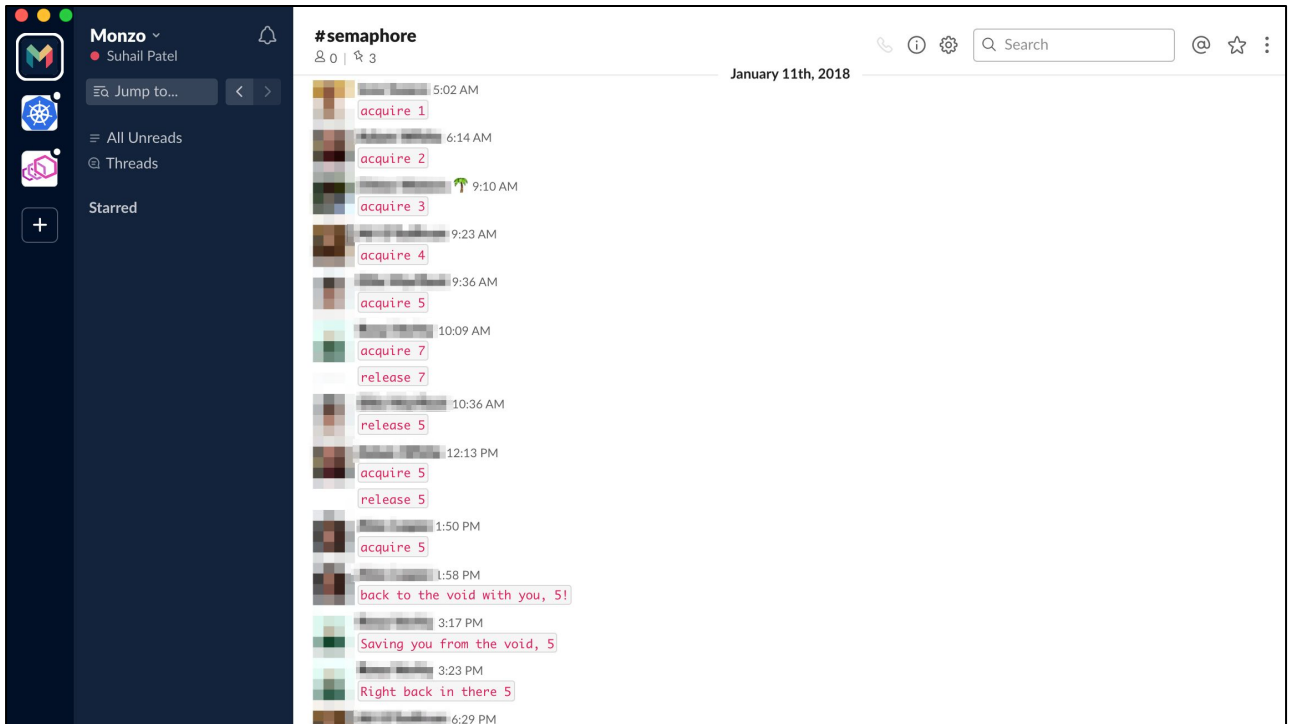


Eventually Consistent Service Discovery

@suhailpatel - SRECon 2019

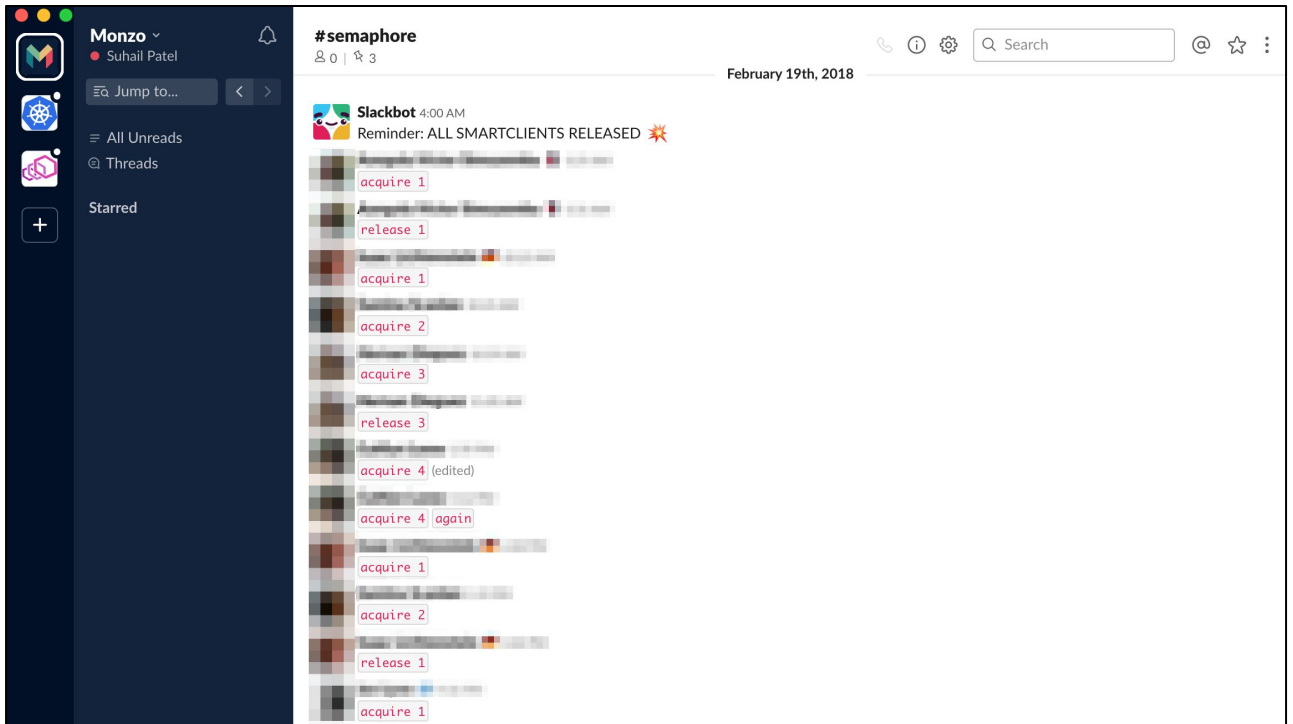


So the other day, I was browsing through our company Slack and found that we used to have an old channel called Semaphore. As someone who's interested in locking systems and especially distributed locking, it sounded intriguing!

Reading more into it, the purpose of this channel was to coordinate access to 10 remote access machines used to access banking data for our customer operations in the really early days. Staff members would go in to this channel, declare to acquire a lock with a number from 1 to 10 and use the corresponding machine.

This was necessary because for various boring reasons that were totally not our company's fault, these machines had single access use only, if you logged in whilst someone else logged in, you would kick the other person out and terminate any queries they were running.

People were pretty fast in acquiring these verbal locks but often forgot to release them. You'd get a deadlock and have to beg for someone to release their locks. There has to be a better way right?



Yep you guessed it, an early morning 4AM auto release of all locks via a Slackbot reminder. If you were running something which crossed the 4AM boundary, you better be sure to quickly re-acquire your verbal lock before someone else takes it.

I just love the chaos involved around this entire system. Happily, we've grown a lot and gotten rid of the need of those 10 machines

Eventually Consistent Service Discovery

@suhailpatel - SRECon 2019

Now on to the main programme. We're going to spend the next 20 minutes or so talking about service discovery and how most of the time, you should really consider having systems which do service discovery in an eventually consistent manner (and also some of the pitfalls)



Hi, i'm Suhail

I'm an Engineer at Monzo on the Platform squad.
We help build the base so other engineers can
ship their services and applications.

Email: **hi@suhailpatel.com**

Twitter: **@suhailpatel**

Hello, my name is Suhail and i'm an Engineer on the Platform Squad at Monzo

We build the foundations so other engineers can ship their services and applications without having to worry whether their service or database is provisioned or there's enough capacity in the cluster.

Most of the time, it involves debugging things like Kubernetes, Cassandra, Envoy and other distributed systems



Monzo is a fully licensed and regulated bank in the UK (and recently launched in the US). We have 3 million plus customers who trust us with their money. Our mission is to make money work for you.

We also have these really nice hot coral cards which look mighty cool! They don't exactly glow in the dark but are the next best thing because of the fluorescent looking hot coral colour

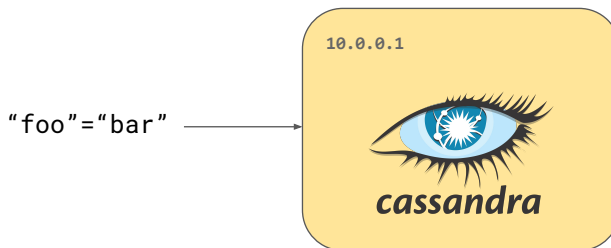
Service Discovery?



The inspiration of this talk came from some discussions about always defaulting systems to strongly consistent service discovery. It makes systems easier to understand and debug if every component has the same view all the time

The goal here is to talk about the kinds of consensus systems out there and discuss the various approaches you can take, what works with thousands and possibly millions of clients

Service Discovery?



Let's take a step back, what do I mean by service discovery?

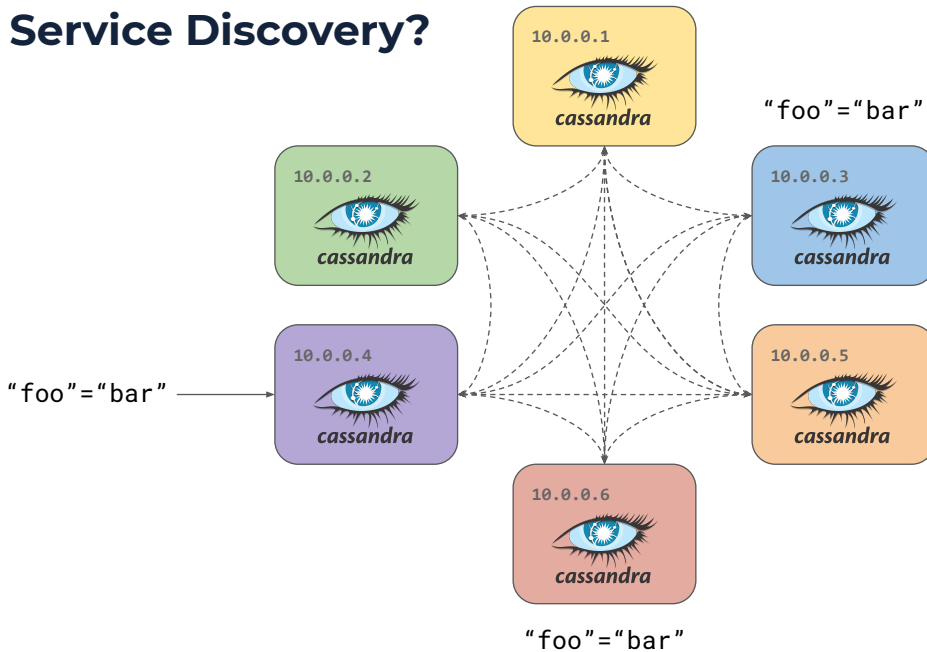
Say you have a database like Cassandra.

When you are first running it with just one node, it's running solo. You read your data from this one node and write your data to this one node.

You'll give it a fixed IP and a deterministic port and because it's just one node, there's nothing to really discover from Cassandra's point of view. All your data is in one canonical place and all decisions are made on this one node in isolation

All is good in this simple world.

Service Discovery?



A single node isn't typically how you run a data store like Cassandra.

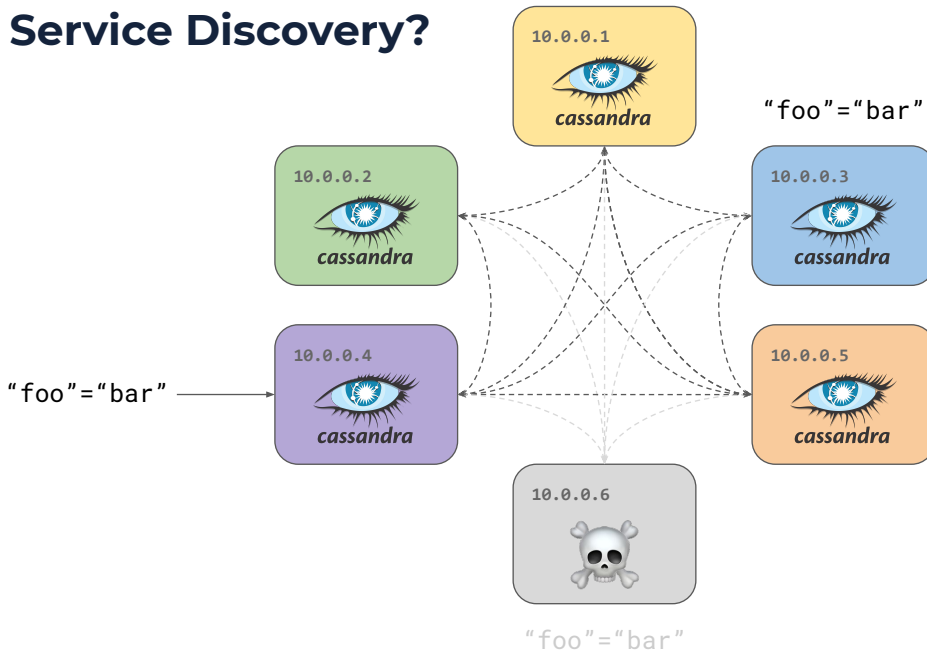
Typically you run Cassandra in a cluster of multiple nodes to take advantage of things like high availability and replication. You don't want a single node failure to cause your entire database to go down and lose all your data.

Each of these Cassandra nodes needs to know what other Cassandra nodes are to form the cluster. This involves exchanging things like IP address of where it lives and whether it's up and other metadata in order to communicate.

What it boils down to is for a set of members, how you automatically detect which ones are present and alive and able to serve requests or conduct an action.

This is what I mean by service discovery. You might have heard of it also referred to as a membership protocol

Service Discovery?



What if one of the nodes in our cluster fails or goes down all of a sudden?

Service discovery often encapsulates things like active health checking or heartbeating to detect if a node fails or part of a system degrades leading to loss of performance.

This is so when you actually go to make a request to another node in the system, you can make a more informed decision and improve your likelihood of getting a response in a timely manner by not picking nodes that are down or have really high latency.

Strong Consistency



<https://etcd.io/>

So what does strong consistency mean and how is it typically implemented?

I'm going to explain it by talking a little bit about etcd and raft.

Etcd is a very popular open source distributed key value store. It's used for a wide variety of reasons for storing and retrieving data. If you have a Kubernetes deployment, you'll be using etcd for storing all of your cluster data such as which pods exist in your deployments.

Strong Consistency



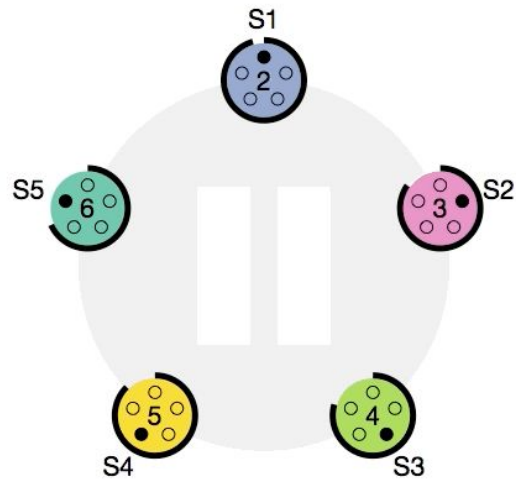
<https://raft.github.io>

Under the hood, etcd uses an algorithm called Raft to implement its distributed consensus, a fancy way of saying agreement between multiple nodes.

Raft is actually an acronym for 'Replicated and Fault Tolerant'. The goal was to develop an algorithm for agreement between multiple nodes which is easily understandable and thus easier to implement compared to predecessor algorithms like Paxos

I only found out that the Raft project has a logo whilst preparing this talk. Not often you see an algorithm with a logo

Strong Consistency



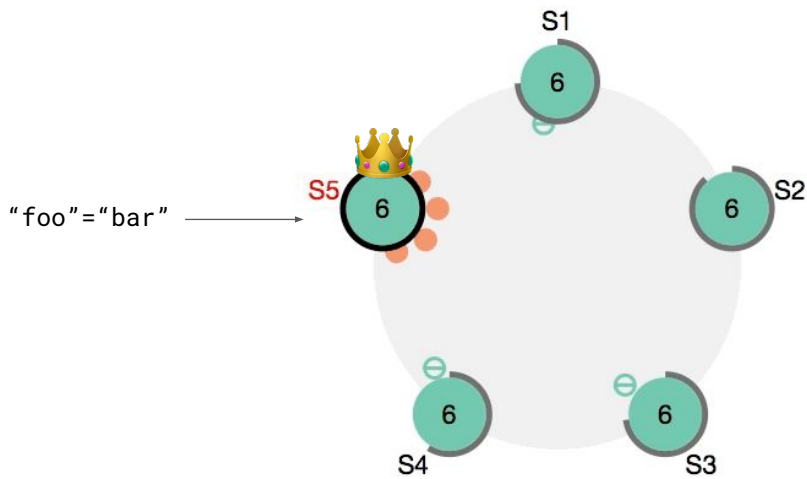
<https://raft.github.io>

In Raft, nodes start off by electing a leader. In this animation, you can see nodes exchanging messages to become the leader. Any node can propose to become a leader by putting itself forward as a candidate and pushing out a message to garner votes.

A majority of the nodes must vote agree to support this candidate for leader. If a node has agreed to supporting a candidate for leadership, it can't change it's vote in that particular race term. Unlike most human politics nowadays, computers play by the rules

In this example, eventually, node S5 gains the leadership and starts it's reign as supreme overload

Strong Consistency

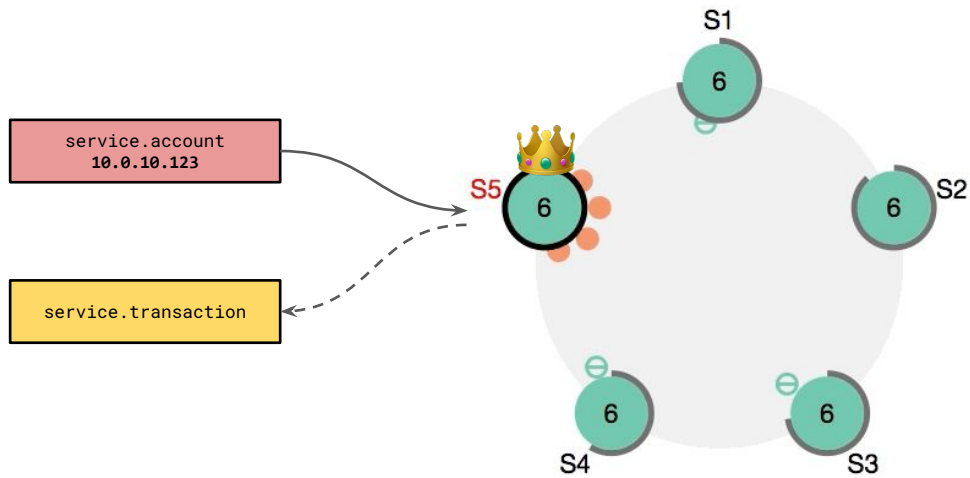


Now say you are a client application and you want to write a value foo equal to bar.

All operations go through the leader. The leader which in this case is S5 will reach out to all nodes to commit this change and only accept it in a scenario where the majority agree to commit this write for foo.

The leader does writes to all nodes in this cluster for redundancy. If the leader fails or drops out, another will contest it's reign and pick up where the leader left off. All actions which require consistency which includes reads and writes go through the leader.

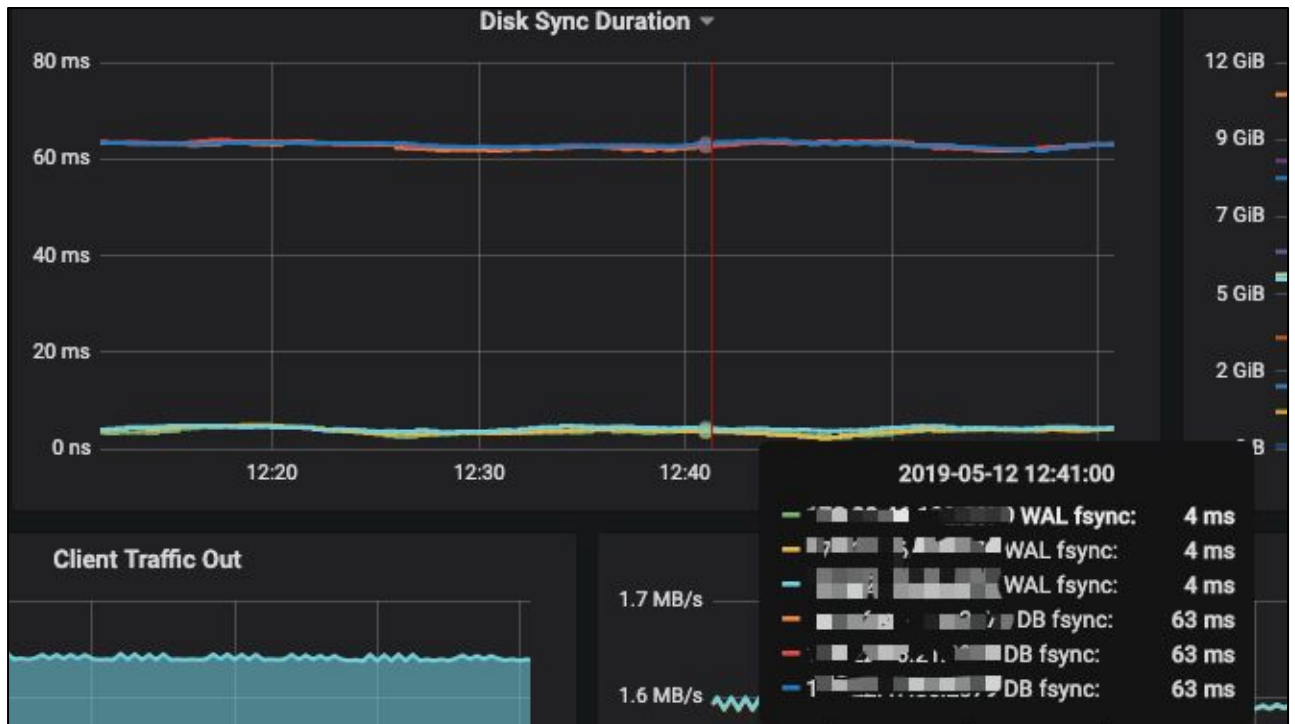
Strong Consistency



Let's put this into perspective of service discovery. Every change goes through this leader write process. Say a replica of an account service in red spins up, it can register itself with its IP address and other metadata, it can heartbeat at a certain interval.

Another service such as the transaction service in yellow can query for events and get the current state of registrations and heartbeats via the leader. It can know that this replica of the account service exists and is available at a certain IP and last checked in at a certain timestamp. Some Raft implementations add a watcher on top so your application can get notified of every state change in linear order.

This kind of system sounds ideal. All your clients can query this system and consistently get the same view as nodes go in and out. It's easy to debug because everything has the same linear consistent view of the world.



Things start to get a bit hairy when you have lots of clients registering and unregistering and lots of other clients watching as things are going on. Each node needs to commit each write to a log. To do this durably and reliably, you can't just keep this log buffered in memory because otherwise a machine failure could lose data. Each write needs to be committed and acknowledged to a hard disk.

Those running Raft based systems like etcd obsess over low level metrics like hard disk commit duration because an increase in hard disk write latency can drastically reduce write performance. There's also a limitation in how many concurrent writes you can do bound by how fast your disks are.

In the read path, again the leader will have larger CPU usage compared to the rest of the cluster if you are doing a large volume of reads. The other nodes in the cluster are there for redundancy, they aren't really useful for serving data. Now of course, depending on the implementation, you can optimise this

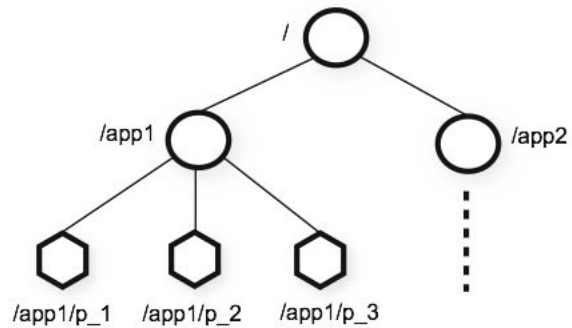
Sequential Consistency



Let's talk a little bit about Sequential Consistency, this is one rung down from Strong Consistency.

To really explain it, i'm going to use examples from ZooKeeper. For those who haven't worked with ZooKeeper, it dubs itself as a 'open source distributed coordination server'. It's good for keeping metadata and configuration in a distributed fashion, similar to what we saw with etcd.

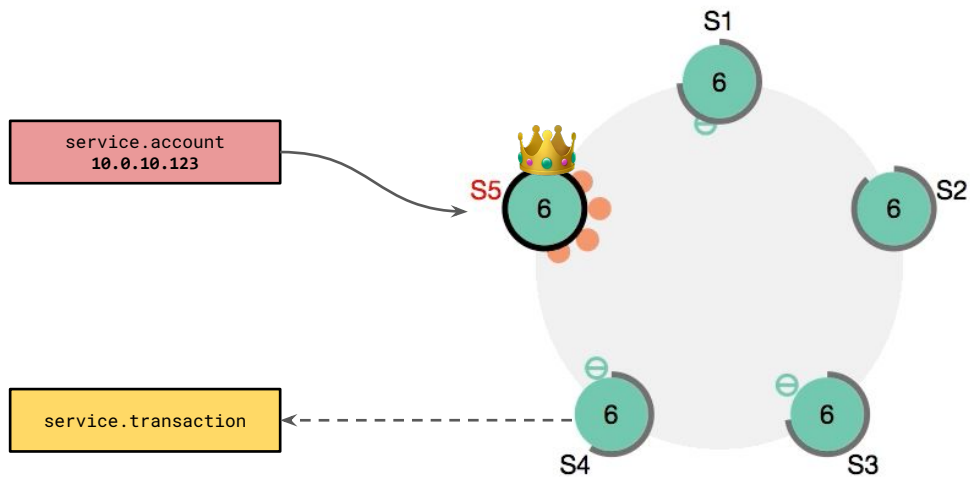
Sequential Consistency



<https://zookeeper.apache.org/doc/r3.1.2/zookeeperOver.html>

ZooKeeper is pretty neat because it exposes itself with an API similar to a file system hierarchy with folders and levels. You can do the equivalent of `ls` on a path such as `ls /app1` to list all the items beneath it. This provides a lot of flexibility in querying.

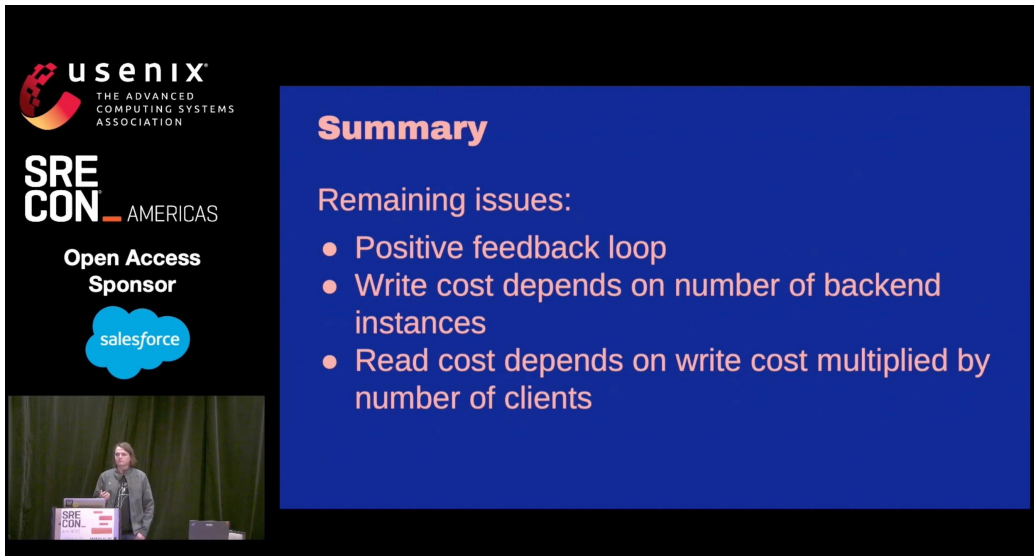
Sequential Consistency



Similar to Raft and Etcd, a leader is elected and writes go through the leader via Zookeeper's Atomic Broadcast protocol. Where ZooKeeper is slightly different is reads can be served completely through any node without direct interaction with the leader. The entire protocol is built on a model of state machine transitions

This is achieved by the leader writing to quorum. This does mean for a small period of time, you can have some nodes serve stale data if they haven't caught up with all the writes. It's called sequential consistency because no matter which node you read from, an update won't be presented out of order.

Say two events happen sequentially across time, it's guaranteed that a node will have observed the first event if the same node has the second event.



usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

SRE
CON AMERICAS

Open Access
Sponsor

salesforce

Summary

Remaining issues:

- Positive feedback loop
- Write cost depends on number of backend instances
- Read cost depends on write cost multiplied by number of clients

<https://www.usenix.org/conference/srecon19americas/presentation/nigmatullin>

Again with ZooKeeper, running a cluster of ZooKeeper instances means you have a central repository with a leader which you need to operate and optimise. Every state change needs to be stored and tracked.

ZooKeeper provides plenty of knobs and consistency options to allow you to do this with various tradeoffs. Dropbox has a great talk from a previous SRECon on the challenges of service discovery at scale using Zookeeper with millions of clients and tens of thousands of state changes a second.

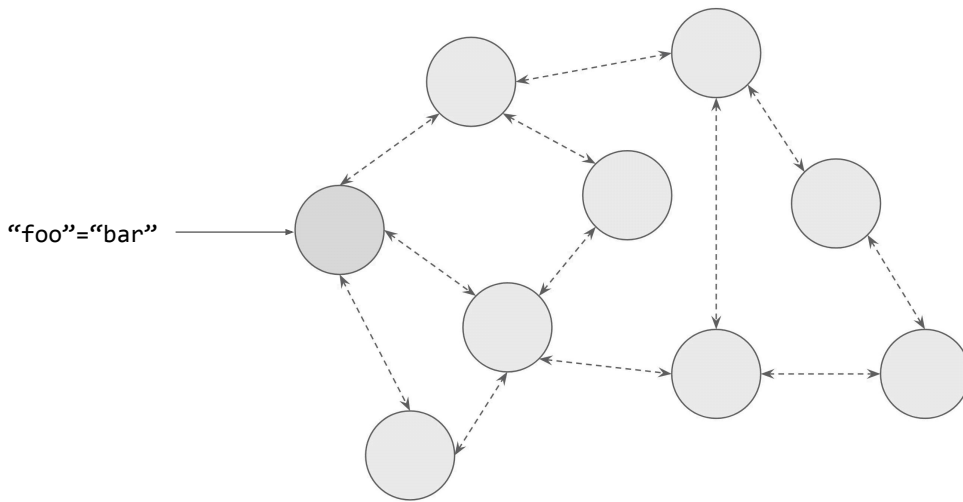


We've spent some time talking about strongly consistent systems, now let's give into eventually consistent systems to see how they work and can scale

Has anyone heard of or played this game called Plague before? It's a video game where you develop a virus and your goal is to level it up and have it infect the entire world before the clever doctors can find a cure and fight back. There's also a board game called Pandemic which is similar but you are on the side of finding a cure in the face of global disaster

This is relevant here because some of the most widely used eventually consistent systems are built on gossip and infection style protocols

Gossip Protocols

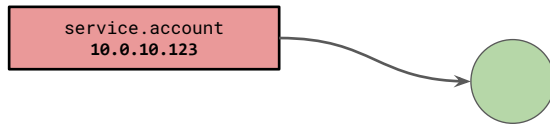


In a gossip style system, a node would typically be connected to a handful of other nodes. When a change comes in or when you want to propagate a message, you will tell your peer nodes who will in turn pass it on to their peer nodes, thus gossiping that change across all the nodes.

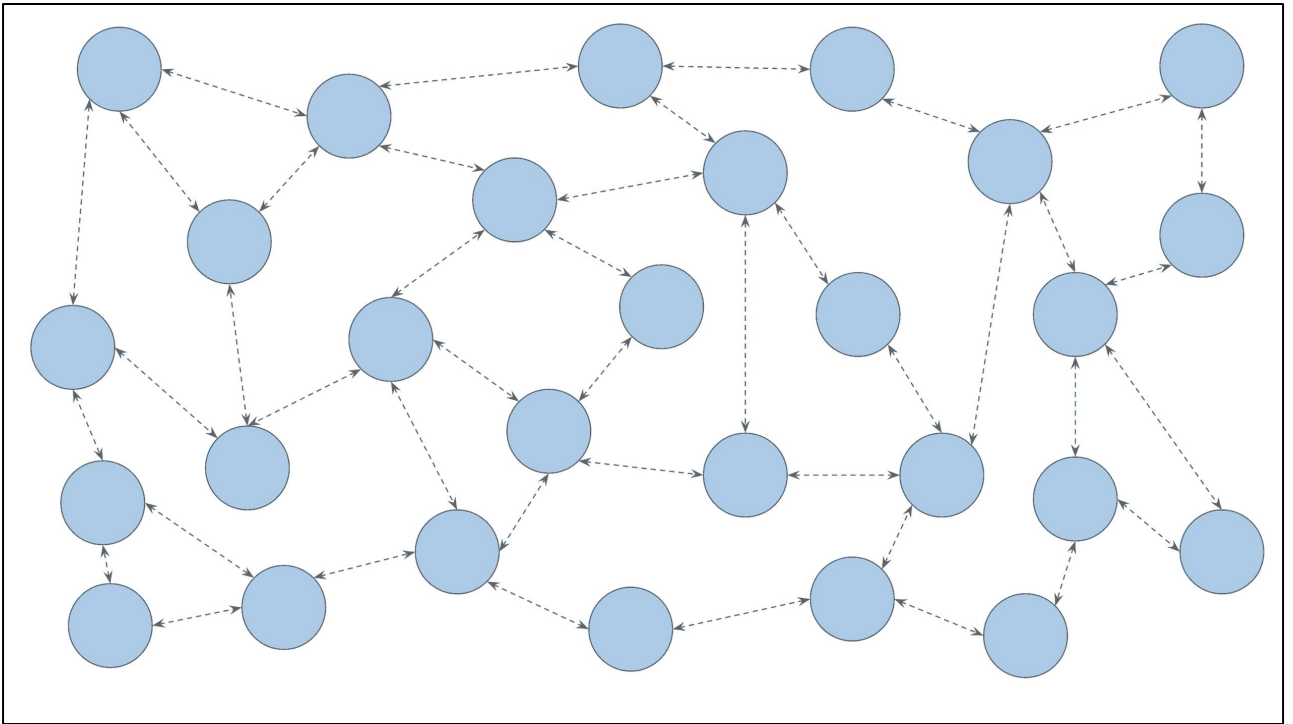
Depending on the implementation, you might have a fixed set of alive peers or you may pick a random subset of nodes in the group per message.

Gossip messages will typically be sent to peers at a time interval and can be used for communicating a range of things. By communicating at a fixed interval, you also implicitly get liveness checks of each node included in the protocol which is nice.

Gossip Protocols



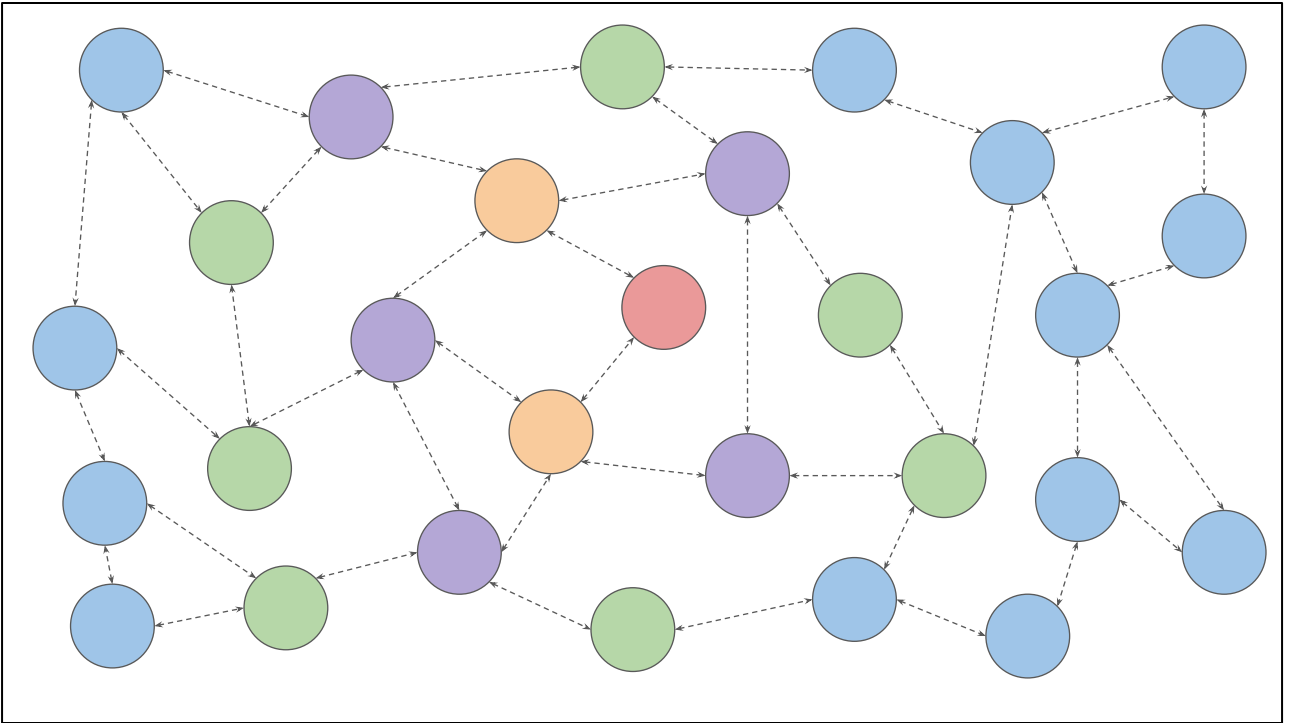
In the case of service discovery, this approach feels really scalable. Think of this orb as an node of our service. It's going to connect to the other peer nodes and tell everyone that it's alive and available to serve requests at a particular IP



It starts in the middle connecting to handful of peers and the arrival and information of this service propagates across the cluster

You can keep adding and removing nodes since each node is only concerned about a handful of others within it's vicinity. Messages will eventually propagate through the entire system. Because messages are distributed to multiple peers and spread, there is implicit replication. If one of the nodes here disappeared, messages would still be replicated to all the other nodes due to the multiple connectivity.

Everyone now knows that this account service is alive and available to serve requests as the green state has propagated everywhere.

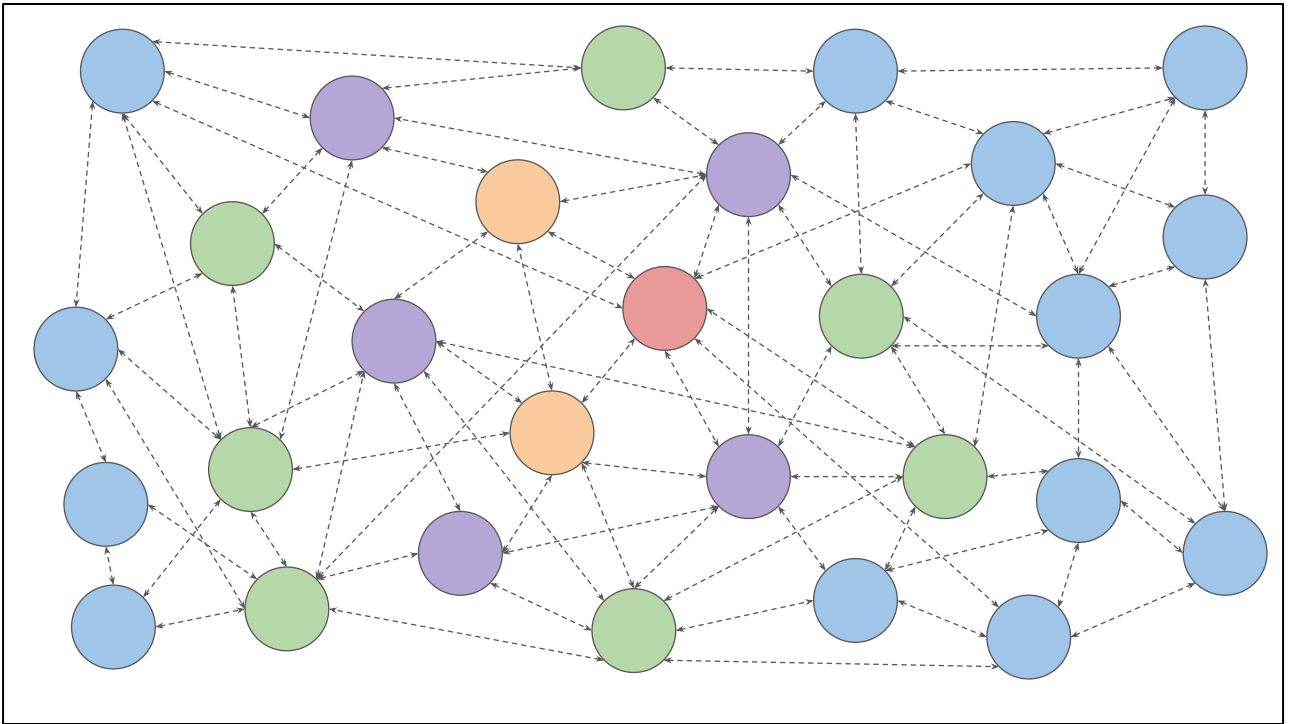


As you add more and more nodes however, you run into two key issues.

Firstly, getting a message out to all the nodes requires a lot of hops for the state to propagate through fully. If you gossip changes at a fixed interval, it can take some time before every other node in the system is aware. Even when you gossip really frequently, you will have bottlenecks with network jitter, packet loss and propagation delay which slow you down.

That means your time to converge can't always be bounded or even fully controlled unless you have the perfect environment with perfect networking equipment. How we'd all love that!

If you are doing a lot of changes rapidly, you might get into a position where any one particular state never gets replicated to all nodes because it takes too long to propagate a state before the next one is generated and starts being gossiped to all nodes.



You can try and tackle this by increasing the number of connections to peer nodes. This increases the likelihood of being infected earlier on.

Here's where you run into the second issue, each of these connections increases the communication chatter back and forth. Any node can be infected with a state by any one of it's connections but all nodes are going to try and gossip state or heartbeats with it's peers constantly. This can be costly in network bandwidth and overhead.

Assume that every node was connected to every other node in the system. That would be the absolute best case because a change in state could be passed on in a maximum of one hop. If we had 100 nodes all connected to each other and gossiping every 10 milliseconds, that'd generate over a million messages per second. The number of messages grows quadratically as we add more nodes.

SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol

Abhinandan Das, Indranil Gupta, Ashish Motivala*
Dept. of Computer Science, Cornell University
Ithaca NY 14853 USA
{asdas,gupta,ashish}@cs.cornell.edu

Abstract

Several distributed peer-to-peer applications require weakly-consistent knowledge of process group membership information at all participating processes. SWIM is a generic software module that offers this service for large-scale process groups. The SWIM effort is motivated by the unscalability of traditional heart-beating protocols, which either impose network loads that grow quadratically with group size, or compromise response times or false positive frequency w.r.t. detecting process crashes. This paper re-

1. Introduction

*As you swim lazily through the milieu,
The secrets of the world will infect you.*

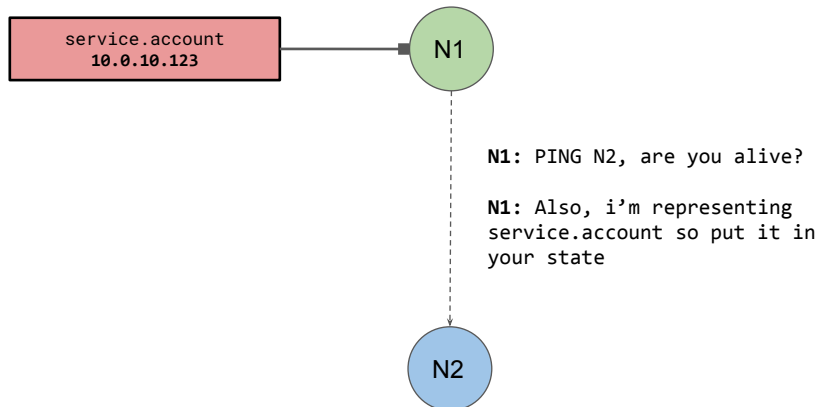
Several large-scale peer-to-peer distributed process groups running over the Internet rely on a distributed membership maintenance sub-system. Examples of existing middleware systems that utilize a membership protocol include reliable multicast [3, 11], and epidemic-style information dissemination [4, 8, 13]. These protocols in turn find use in applications such as distributed databases that need to reconcile re-

https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/SWIM.pdf

Luckily for us, some really smart folks have been thinking about these sorts of problems for a long time. There is a whole suite of papers assessing the performance of group membership and the almost quadratic like growth in gossip propagation overhead.

This paper on Scalable Weakly Consistent Infection Style Process Group Membership (or SWIM for short) is quite novel in some of it's approaches in optimising messaging bandwidth and improving detection of failures. There is a whole set of mathematical models presented in the paper where you can optimise network usage based on your tolerance for failure.

(Scalable) Gossip Protocols

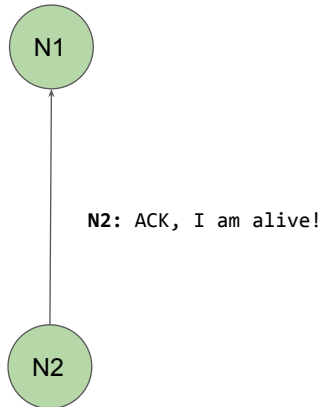


https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/SWIM.pdf

In practice, each heartbeat also propagates information about the cluster state. This means bandwidth is used more efficiently by combining health checks to connected peers whilst also gossiping information about changes in the state. A node will send periodic pings randomly to other nodes in the cluster

In this example, Node 1 is checking to see if Node 2 is alive but also takes the opportunity to inform Node 2 of an addition of this instance of the account service.

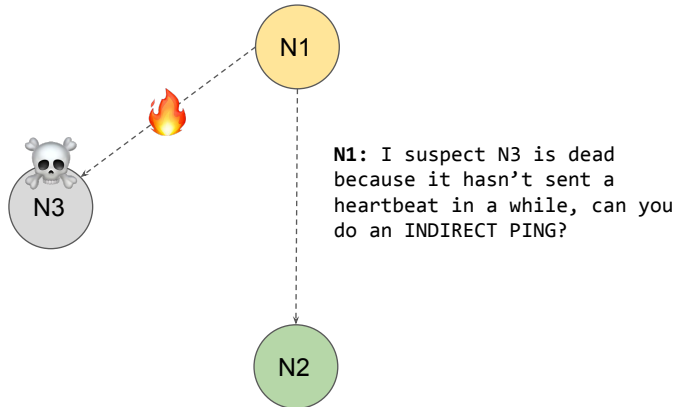
(Scalable) Gossip Protocols



https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/SWIM.pdf

Node 2 replies that it is indeed alive and processes the update in the state. We've managed to propagate the state change and get a heartbeat in one round trip of messaging

(Scalable) Gossip Protocols

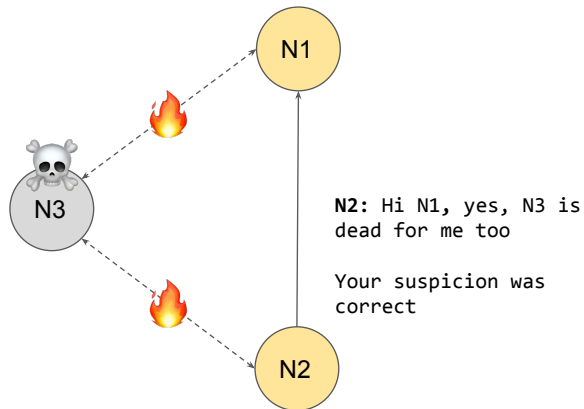


https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/SWIM.pdf

The SWIM paper also gives a solution to making failure detection more robust by gossiping suspicion that another node is dead rather than outright declaring that a node is dead.

In this example, Node 1 thinks Node 3 is dead and tells Node 2 by requesting an indirect ping.

(Scalable) Gossip Protocols

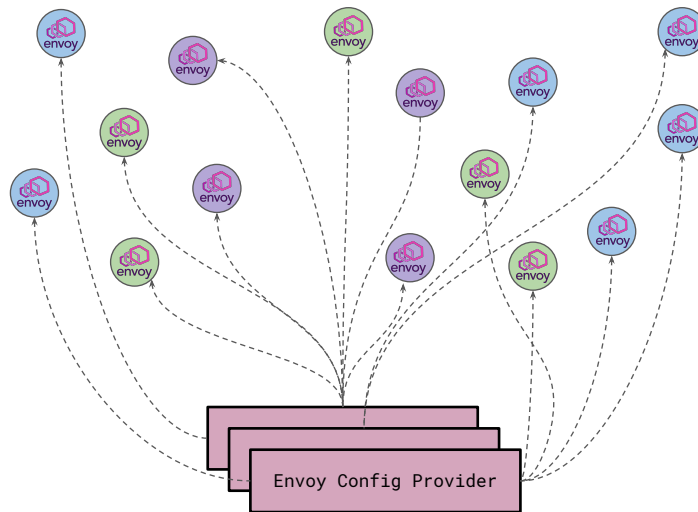


https://www.cs.cornell.edu/projects/Quicksilver/public_pdfs/SWIM.pdf

By arousing a suspicion from Node 1 to Node 2, Node 2 can do its own independent checks and confirm or deny that Node 3 is dead.

This confirmation reduces the rate of false positives, however, it does increase the lag time in marking a node as dead. In most implementations, you can tweak how much confirmation you want from indirect pings before you truly mark a node as dead and remove it from your list of active nodes.

Eventual Consistency



So that's all well and good. If you are going with a eventual consistent system, you can also go simple and have a pull based configuration where your clients pull service discovery configuration on a timer from a central registry, here dubbed the config provider.

This config provider can be implemented in many ways, it can just be eventually consistent by gossiping just like any other nodes in the system or be backed by something strongly consistent.

This is how Envoy Proxy works. Envoy is a service networking proxy that you put in front of your services. It's goal is to abstract pulling discovery configuration, load balancing and retries and circuit breaking and a whole heap of other networking functions into a robust lightweight proxy. You typically run a copy of Envoy alongside each instance of your service

Envoy pulls it's dynamic configuration such as endpoints and routes from a config provider you specify which can provide that config. If you are running lots of Envoy instances, those will converge to the latest configuration eventually, not all at once and there isn't any ordering in which updates go out first.

Eventual Consistency

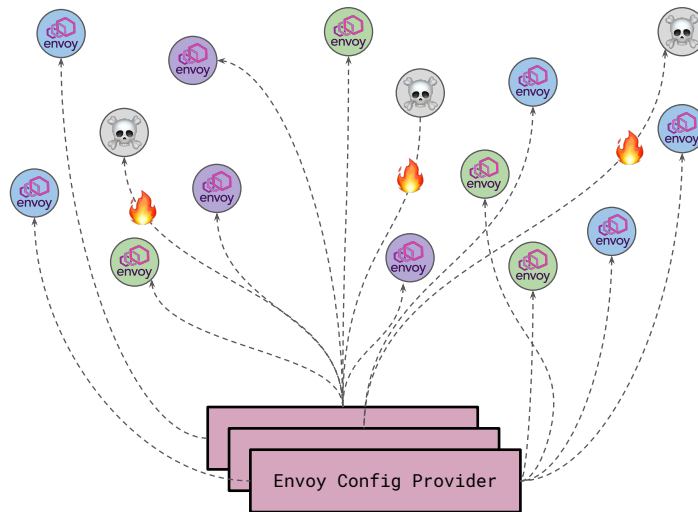


Say you have a service running and serving traffic at Version 1. It's now time to release a new version of your service, version 2.

You aren't going to cut over 100% of your traffic instantly to the new version and need all callers to flip over to nodes service version 2. If you are running multiple instances of each version for high availability, not all instances of version 2 are going to start at precisely the same time. There is some lag time, even if it's a few seconds where you run both versions in parallel or do some sort of rolling deployment

This means for an eventually consistent service discovery system, in practice it will in pretty much converge within that lag time that you take to do your deployment.

Eventual Consistency



What happens if something fails or there is a lag in getting all the latest information? The system to a large extent just keeps humming along because in practice, you have things like retries and load balancing and independent detection of constantly failing nodes also an integral part of the system.

In strongly consistent systems, even things like a critical a node failure won't be picked up immediately, typically you will do a few probes before you declare a node to be dead. There is always some lag time. If the node to node communication layer embraces this failure scenario, the impact in practice can be reduced to zero or very minimal impact.

At Monzo, we have over 1300 microservices and thousands of instances of these various services running at any one time on top of Kubernetes. We see Envoy Proxy converge typically in seconds and even that's bound on how fast we can pump configuration out via the provider. This works great pretty much all the time alongside rolling deployments.

Summary

Need Agreement?

- A strongly consistent system may be ideal for this use case

Scalability?

- Eventual consistency will work as long as you acknowledge in your applications that it's not always perfect

In summary, the takeaway is with service discovery, you can go with the level of consistency you are comfortable with and scales to your requirements.

If you need to prioritise agreement on the state of your cluster, especially across a small number of nodes, a strongly consistent backed store and propagation will work great for you

For the vast majority of service discovery use cases, you shouldn't discount eventual consistency propagation because in practice, pretty much all changes to service discovery state are eventual. Rollouts are eventual, changing endpoints happens eventually, health check failures have some lag time to confirm, pretty much nothing about service discovery needs to be 100% instant.

By understanding how these various service discovery mechanisms work conceptually, you can make better informed decisions in other parts of your systems such as deployment pipelines and node to node communication logic.

As always, run tests and simulations in your environment. The majority of these protocols have tried and tested robust implementations and sometimes entire systems like etcd and zookeeper to build upon. You have systems like Consul by Hashicorp which runs on Raft or Apache Curator which runs on top

of ZooKeeper.



As always, run tests and simulations in your environment based on your use case. The majority of these protocols have tried and tested robust implementations and sometimes entire systems like etcd and zookeeper to build upon.

This is our office dog Bingo after a hard day of work testing the scalability of cardboard boxes. Be like Bingo and run tests and simulations

Thanks!

Email: hi@su hailpatel.com

Twitter: [@su hailpatel](https://twitter.com/su hailpatel)

And that's about it. Thank you for listening and again a big thanks to everyone at SRECon taking the time to organise.

If you have any feedback, please feel free to email or tweet at me.

Otherwise, i'm happy to take a few questions from the audience