Eureka! Why You Shouldn't Use ZooKeeper for Service Discovery

Knewton Follow
Dec 15, 2014 · 6 min read

Written by Peter Kelley

Many companies use ZooKeeper for service discovery. At Knewton, we believe this is a fundamentally flawed approach. In this article, I will walk through our failures with ZooKeeper, tell you why you shouldn't be using it for service discovery, and explain why Eureka is a better solution.

Remember What You're Building On

Let's back up. It's important to first discuss what your target environment is *before* deciding what software to use or how to build your own. In the cloud, resiliency to equipment and network failure has to be a primary concern. When you're running your software across a large number of replaceable pieces of hardware, it's inevitable that one of them will fail at some point. At Knewton, we run on AWS, and we've seen many different types of failure. You have to design your systems expecting failure. Other companies on AWS agree (there are whole books written on the topic). You have to anticipate box failure, high latency, and network partitions—and build resiliency against them into your system.

Don't assume your environment is the same as others. Sure, if you're managing your own datacenters, you quite possibly could be putting in the time and money to minimize hardware failures and network partitions. But cloud environments like AWS make a different trade-off. You are going to have these issues, and you had better prepare for them.

Failures with ZooKeeper

ZooKeeper is a great software project. It is mature, has a large community supporting it, and is used by many teams in production. It's just the wrong solution to the problem of service discovery.

In CAP terms, ZooKeeper is CP, meaning that it's *consistent* in the face of partitions, not *available*. For many things that ZooKeeper does, this is a necessary trade-off. Since ZooKeeper is first and foremost a coordination service, having an eventually consistent design (being AP) would be a horrible design decision. Its core consensus algorithm, Zab, is therefore all about consistency.

For coordination, that's great. But for service discovery it's better to have information that may contain falsehoods than to have no information at all. It is much better to know what servers were available for a given service five minutes ago than to have no idea what things looked like due to a transient network partition. The guarantees that ZooKeeper makes for coordination are the wrong ones for service discovery, and it hurts you to have them.

ZooKeeper simply doesn't handle network partitions the right way for service discovery. Like other types of failure in the cloud, partitions actually happen. It is best to be as prepared as possible. But—as outlined in a Jepsen post on ZooKeeper and the ZooKeeper website—in ZooKeeper, clients of the nodes that are part of the partition that can't reach quorum lose communication with ZooKeeper and their service discovery mechanism altogether.

It's possible to supplement ZooKeeper with client-side caching or other techniques to alleviate certain failure conditions. Companies like Pinterest and Airbnb have done this. On the surface, this appears to fix things. In particular, client-side caching helps ensure that if any or all clients lose contact with the ZooKeeper cluster, they can fall back to their cache. But even here there are situations where the client won't get all the discovery information that could be available. If quorum is lost altogether, or the cluster partitions and the client happens to be connected to nodes that are not part of quorum but still healthy, the client's status will be lost even to those other clients communicating with those *same* healthy ZooKeeper nodes.

More fundamentally, supplementing ZooKeeper, a consistent system, with optimistic caching is attempting to make ZooKeeper more *available*. ZooKeeper is meant to be *consistent*. This gives you neither: you have bolted a system that wants to be AP on top of a system that is CP. This is fundamentally the wrong approach. A service discovery system should be designed for availability from the start.

Even ignoring CAP tradeoffs, setting up and maintaining ZooKeeper correctly is hard. Mistakes are so common that projects have been developed just to mitigate them. They exist for the clients and even the ZooKeeper servers themselves. Because ZooKeeper is so hard to use correctly, many of our failures at Knewton were a direct result of our misuse of ZooKeeper. Some things appear simple but are actually easy to get wrong: for example, reestablishing watchers correctly, handling session and exceptions in clients, and managing memory on the ZK boxes. Then there are actual ZooKeeper issues we hit, like ZOOKEEPER-1159 and ZOOKEEPER-1576. We even saw leadership election fail in production. These types of issues happen because of the guarantees ZooKeeper needs to make. It needs to manage things like sessions and connections, but because they aren't needed for service discovery, they hurt more than they help.

Making the Right Guarantees: Success with Eureka

We switched to Eureka, an open-source service discovery solution developed by Netflix. Eureka is built for availability and resiliency, two primary pillars of development at Netflix. They just can't stop talking about it—and for good reason. Since the switch, we haven't had a single service-discovery-related production outage. We acknowledged that in a cloud environment you are guaranteed failure, and it is absolutely critical to have a service discovery system that can survive it.

First, if a single server dies, Eureka doesn't have to hold any type of election; clients automatically switch to contacting a new Eureka server. Eureka servers in this case will also accept the missing Eureka server back when it reappears, but will only *merge* any registrations it has. There's no risk of a revived server blowing out the entire service registry. Eureka's even designed to handle broader partitions with zero downtime. In the case of a partition, each Eureka server will continue to accept new registrations and publish them to be read by any clients that can reach it. This ensures that new services coming online can still make themselves available to any clients on the same side of the partition.

But Eureka goes beyond these. In normal operation, Eureka has a built-in concept of service heartbeats to prevent stale data: if a service doesn't phone home often enough, then Eureka will remove the entry from the service registry. (This is similar to what people typically build with ZooKeeper and ephemeral nodes.) This is a great feature, but could be dangerous in the case of partitions: clients might lose services that were actually still up, but partitioned from the Eureka server. Thankfully, Netflix thought of this: if a Eureka server loses connections with too many clients too quickly, it will enter "self-preservation mode" and stop expiring leases. New services can register, but "dead" ones will be kept, just in case a client might still be able to contact them. When the partition mends, Eureka will exit self-preservation mode. Again, holding on to good and bad data is better than losing any of the good data, so this scheme works beautifully in practice.

Lastly, Eureka caches on the client side. So even if every last Eureka server goes down, or there is a partition where a client can't talk to any of the Eureka servers, then the service registry still won't be lost. Even in this worst-case scenario, your service will still likely be able to look up and talk to other services. It is important to note that client-side caching is appropriate here. Because all healthy Eureka servers must be unresponsive to resort to it, we know there is no chance of new and better information possibly being reachable.

Eureka makes the right set of guarantees for service discovery. There are no equivalents for leadership election or transaction logs. There is less for you to get wrong and less that Eureka has to do right. Because Eureka is built explicitly for service discovery, it provides a client library that provides functionality such as service heartbeats, service health checks, automatic publishing, and refreshing caches. With ZooKeeper, you would have to implement all of these things yourself. Eureka's library is built using open-source code that everyone sees and uses. That's better than a client library that only you and two other people have seen the code for.

The Eureka servers are also infinitely easier to manage. To replace nodes, you just remove one and add another under the same EIP. It has a clear and concise website that provides a visual representation of all your services and their health. I can't tell you how great it is to glance at a web page and see exactly what services are running or suffering issues. Eureka even provides a REST API, allowing for easy integration with other potential uses and querying mechanisms.

Conclusion

The biggest takeaways here are to remember what hardware you are building on and to only solve the problems you have to solve. Using Eureka provides both of these for us at Knewton. Cloud platforms are unreliable and Eureka is designed to handle their unique challenges. Service discovery needs to be as available and resilient as possible, and Eureka is designed to be just that.

Additional Resources

Netflix Shares Cloud Load Balancing And Failover Tool: Eureka!

Thread on ZooKeeper Vs Eureka

Why Not Use Curator/Zookeeper as a Service Registry?

A Gotcha When Using ZooKeeper Ephemeral Nodes

Introducing Curator—The Netflix ZooKeeper Library