ZooKeeper - A Reliable, Scalable Distributed Coordination System

Tuesday, July 15, 2008 at 12:45AM

Todd Hoff in Product, Strategy, backend

ZooKeeper is a *high available and reliable coordination system. Distributed applications use ZooKeeper to store and mediate updates key configuration information. ZooKeeper can be used for leader election, group membership, and configuration maintenance.* In addition ZooKeeper can be used for event notification, locking, and as a priority queue mechanism. It's a sort of central nervous system for distributed systems where the role of the brain is played by the coordination service, axons are the network, processes are the monitored and controlled body parts, and events are the hormones and neurotransmitters used for messaging. Every complex distributed application needs a coordination and orchestration system of some sort, so the ZooKeeper folks at Yahoo decide to build a good one and open source it for everyone to use.

The target market for ZooKeeper are multi-host, multi-process C and Java based systems that operate in a data center. ZooKeeper works using distributed processes to coordinate with each other through a shared hierarchical name space that is modeled after a file system. Data is kept in memory and is backed up to a log for reliability. By using memory ZooKeeper is very fast and can handle the high loads typically seen in chatty coordination protocols across huge numbers of processes. Using a memory based system also mean you are limited to the amount of data that can fit in memory, so it's not useful as a general data store. It's meant to store small bits of configuration information rather than large blobs. Replication is used for scalability and reliability which means it prefers applications that are heavily read based. Typical of hierarchical systems

you can add nodes at any point of a tree, get a list of entries in a tree, get the value associated with an entry, and get notification of when an entry changes or goes away. Using these primitives and a little elbow grease you can construct the higher level services mentioned above.

Why would you ever need a distribute coordination system? It sounds kind of weird. That's more the question I'll be addressing in this post rather than how it works because the slides and the video do a decent job explaining at a high level what ZooKeeper can do. The low level details could use another paper however. Reportedly it uses a version of the famous Paxos Algorithm to keep replicas consistent in the face of the failures most daunting. What's really missing is a motivation showing how you can use a coordination service in your own system and that's what I hope to provide...

Kevin Burton wants to use ZooKeeper to to configure external monitoring systems like Munin and Ganglia for his Spinn3r blog indexing web service. He proposes each service register its presence in a cluster with ZooKeeper under the tree "/services/www." A Munin configuration program will add a ZooKeeper Watch on that node so it will be notified when the list of services under /services/www changes. When the Munin configuration program is notified of a change it reads the service list and automatically regenerates a munin.conf file for the service.

Why not simply use a database? Because of the guarantees ZooKeeper makes about its service:

- Watches are ordered with respect to other events, other watches, and asynchronous replies. The ZooKeeper client libraries ensures that everything is dispatched in order.
- A client will see a watch event for a znode it is watching before seeing

the new data that corresponds to that znode.

- The order of watch events from ZooKeeper corresponds to the order of the updates as seen by the ZooKeeper service.

You can't get these guarantees from an event system plopped on top of a database and these are the sort of guarantees you need in a complex distributed system where connections drop, nodes, fail, retransmits happen, and chaos rules the day. What rules the night is too terrible to contemplate. For example, it's important that a service-up event is seen after the service-down or you may unnecessarily drop revenue producing work because of an event out-of-order issue. Not that I would know anything about this mind you :-)

A weakness of ZooKeeper is the fact that changes happened are dropped: *Because watches are one time triggers and there is latency between getting the event and sending a new request to get a watch you cannot reliably see every change that happens to a node in ZooKeeper. Be prepared to handle the case where the znode changes multiple times between getting the event and setting the watch again. (You may not care, but at least realize it may happen.)*

This means that ZooKeeper is a state based system more than an event system. Watches are set as a side-effect of getting data so you'll always have a valid initial state and on any subsequent change events you'll refresh to get new values. If you want to use events to log when and how something changed, for example, then you can't do that. You would have to include change history in the data itself.

Let's take a look at another example of where ZooKeeper could be useful. Picture a complex backend system running on, let's say, a 100 nodes (maybe a lot less, maybe a lot more) in a data center. For example purposes assume the system is an ad system for serving advertisements to web sites. Ad systems are complex beasts that require a fair bit of

coordination. Imagine all the subsystems needing to run on those 100 nodes: database, monitoring, fraud detectors, beacon servers, web server event log processors, failover servers, customer dashboards, targeting engines, campaign planners, campaign scenario testers, upgrades, installs, media managers, and so on. There's a lot going on.

Now imagine the power in the data center flips and all the machines power on. How do all the processes across all the hosts know what to do? Now imagine everything is up and a few machines go down. How do all the processes know what to do in this situation? This is where a coordination service comes in. A coordination service acts as the backplane over which all these subsystems figure out what they should do relative to all the other subsystems in a product.

How, for example, do the ad servers know which database to use? Clearly there are many options for solving this problem. Using standard DNS naming conventions is one. Configuration files is another. Hard coding is still a favorite. Using a bootstrap service locator service is yet another (assuming you can bootstrap the bootstrap service). Ideally any solution must work just as well during unit testing, system testing, and deployment. In this scenario ZooKeeper acts as the service locator. Each process goes to ZooKeeper and finds out which is the primary database. If a new primary is elected, say because a host fails, then ZooKeeper sends an event that allows everyone dependent on the database to react by getting the new primary database. Having complicated retry logic in application code to fail over to another database server is simply a disaster as every programmer will mess it up in their own way. Using a coordination service nicely deals with the problem of services locating other services in all scenarios. Of course, using a proxy like MySQL Proxy would remove even more application level complexity in dealing with failover and request routing.

How did the database servers decide which role they'll would play in the first place? All the database servers boot up and say "I'm a database server brave and strong, what's my role in life? Am I a primary or a secondary server? Or if I'm a shard what key range do I serve?" If 10 servers are database servers negotiating roles can be a very complicated and error prone process. A declarative approach through configuration files that specify a failover ring in configuration files is a good approach, but it's a pain to get to work in local development and test environments as the machines are always changing. It's easier to let the database servers come up and self organize themselves on initial role election and in failure scenarios. The advantage of this system is that it can run locally on one machine or on a dozen machines in the data center with very little effort. ZooKeeper supports this type of coordination behavior.

Now let's say I want to change the configuration of ad targeting state machines currently running in 40 processes on 40 different hosts. How do I do that? The first approach is no approach. Most systems make it so a new code release has to happen, which is very sloooow. Another approach is a configuration file. Configuration is put in a distribution package and pushed to all nodes. Each process then periodically checks to see if the configuration file has changed and if it has then the new configuration read. That's the basics. Infinite variations can follow. You can have configuration for different subsystems. There's complexity because you have to know what packages are running on which nodes. You have to deal with rollback in case all packages don't push correctly. You have to change the configuration, make a package, test it, then push it to the data center operations team which may take a while to perform the upgrade. It's a slow process. And when you get into changes that impact multiple subsystems it gets even more complicated.

Another approach I've taken is to embed a web server in each process so you can see the metrics and change the configuration for each process on

the fly. While powerful for a single process it's harder to manipulate sets of processes across a data center using this approach.

Using ZooKeeper I can store my state machine definition as a node which is loaded from the static configuration collected from every distribution package in a product. Every process dependent on that node can register as a watcher when they initially read the state machine. When the state machine is updated all dependent entities will get an event that causes them reload the state machine into the process. Simple and straightforward. All processes will eventually get the change and any rebooting processes will pick up the new state machine on initialization. A very cool way to reliably and centrally control a large distributed application.

One caveat is I don't see much activity on the ZooKeeper forum. And the questions that do get asked largely go unanswered. Not a good sign when considering such a key piece of infrastructure.

Another caveat that may not be obvious on first reading is that your application state machine using ZooKeeper will have to be intimately tied to ZooKeeper's state machine. When a ZooKeeper server dies, for example, your application must process that event and reestablish all your watches on a new server. When a watch event comes your application must handle the event and set new watches. The algorithms to carry out higher level operations like locks and queues are driven by multi-step state machines that must be correctly managed by your application. And as ZooKeeper deals with state that is probably stored in your application it's important to worry about thread safety. Callbacks from the ZooKeeper thread could access shared data structures. An Actor model where you dump ZooKeeper events into your own Actor queues could be a very useful application architecture here for synthesizing different state machines in a thread safe manner.

Some Fast Facts

- How data are partitioned across multiple machines? Complete replication in memory. (yes this is limiting)
- How does update happen (interaction across machines)? All updates flow through the master and are considered complete when a quorum confirms the update.
- How does read happen (is getting a stale copy possible) ? Reads go to any member of the cluster. Yes, stale copies can be returned. Typically, these are very fresh, however.
- What is the responsibility of a leader? To assign serial id's to all updates and confirm that a quorum has received the update.
- There are several limitations that stand out in this architecture:

- complete replication limits the total size of data that can be managed using Zookeeper. This is acceptable in some applications, not acceptable in others. In the original domain of Zookeeper (management of configuration and status), it isn't an issue, but zookeeper is good enough to encourage creative misuse where this can become a problem.
- serializing all updates through a single leader can be a performance bottleneck. On the other hand, it is possible to push 50K updates per second through a leader and the associated quorum so this limit is pretty high.
- the data storage model is completely non relational.

These answers were provided by Ted Dunning on the Cloud Computing group.

Related Articles

- An Introduction to ZooKeeper Video (Hadoop and Distributed

Computing at Yahoo!) (PDF)

- ZooKeeper Home, Email List, and Recipes (which has some odd conotations for a Zoo).
- The Chubby Lock Service for Loosely-Coupled Distributed Systems from Google
- Paxos Made Live – An Engineering Perspective by Tushar Chandra, Robert Griesemer, and Joshua Redstone from Google.
- Updates on Open Source Distributed Consensus by Kevin Burton
- Using ZooKeeper to configure External Monitoring Systems by Kevin Burton
- Paxos Made Simple by Leslie Lamport
- Hyperspace - API description of Hyperspace, a Chubby-like service
- Notes on ZooKeeper at the Hadoop Summit by James Hamilton.
