PartitionedZooKeeper

Partitioned Zookeeper

10,000 ft view

Our main goal is to enable write throughput scalability through partitions in ZooKeeper. The overall idea is to split a set of zookeeper servers into overlapping ensembles, and have each partition handle a portion of the ZooKeeper state. By having distinct ensembles handling different portions of the state, we end up relaxing the ordering guarantees that we have with plain ZooKeeper. To overcome this problem, we provide an abstraction that we call "containers". Containers are subtrees of znodes which require that all update operations are ordered as with plain ZooKeeper.

Containers

Containers are subtrees of znodes, and the root of a container is not necessarily the root of a ZooKeeper tree. Upon the creation of a node, we state whether to create a new container for that node or not.

Changes to the API

The only change we envision is an extra parameter on create that tells whether to create a new container or not for the new node. All other operations should be the same.

Internal changes

Requirements

Internally, we will require more changes:

1. **Routing**: We need a mechanism to route requests to the correct ensemble. We can perform it in a distributed fashion, as with DHTs, or we can have one ZooKeeper ensemble responsible for mapping prefixes to ensembles;
2. **Containers**: A ZooKeeper server with this approach has to store and handle requests for a set of containers. Such a set may contain containers from different partitions. Handling different subsets for different partitions does not necessarily imply having multiple instances of the ZooKeeper server on a single machine because containers are disjoint by definition and they can be operated upon in parallel. We just have to make sure that we identify a container correctly when executing a ZooKeeper operation;
3. **Failure and Recovery**: Upon failure of a server, the immediate neighbor of that server should take over the position of that server. It might be necessary to transfer new containers to the server that is taking over.

Architecture

A proposed architecture is as follows. We have one ZooKeeper ensemble called **main**, which is dedicated to keeping metadata of other ensembles and of containers. All other servers map to one or more ensembles. The ensembles are the ones keeping containers and executing operations on them. Main has three branches:

1. "/ensembles": Contains a set of children mapping ZooKeeper servers to ensembles;
2. "/containers": Contains a set of children that maps containers to ensembles;
3. "/servers": A map from server to set of ensembles.

The clients only need to know the servers of the main ensemble to connect to everyone else. For example, "/subtree/apps/yaview" has the name of the ensemble that stores "/apps/yaview" and "/ensemble/A" has the list of servers that make up ensemble A. Assuming data stored in the main ensemble is relatively static and small, such an ensemble does not have to process a pretty heavy workload.

With respect to sessions, there are two options in this architecture:

1. Clients have a session with the main ensemble, and each server in main has a session with each of the other ensembles. Operations of clients connected to the same server and to the same server are therefore serialized;
2. Clients read the state of main, and create a session directly with the ensemble holding the container of interest.

Changes to consistency guarantees

In the above architecture, we do not provide any order guarantee across containers. That is, two update operations targetting nodes in different containers can be executed in any order. For the same container, the guarantees are the same as with plain ZooKeeper, including the order of watch notifications.

Proxies

Partitioning has the potential of solving the scalability issue if the traffic is roughly balanced across the ensembles. If there is a hot spot, however, partitioning can only help if containers can be further partitioned or if we can move containers around. If we can't perform neither of these due to application constraints, one proposed solution is to use connection proxies to enable a large number of connections to the same subspace of nodes, thus enabling more connected clients. It is not clear, though, how to make throughput of operations scalable, and there seems to be a limit imposed by replication. We would have to relax the consistency of even a single partition to enable hot spots to scale.
An additional possibility to increasing th