ZooKeeper: Wait-free Coordination for Internet-Scale Systems

Key Takeaways

- Zookeeper is a service for coordinating processes of distributed applications.
- Zookeeper has very simple API which is effectively get and set of a data node, and distributed applications can be built upon these API such as configuration broadcast, leader election, and distributed read/write locks.
- Zookeeper is wait-free as it allows a client to watch for the update of a data node, and get notified when a data node gets an update. Therefore, there is no need to Zookeeper client to wait for certain node to be created or removed. Instead, it just wait for the signal that such creation or removal is success. This allows the client to send multiple unrelated requests at the same time, with improved performance.
- Zookeeper handles the consistency by relaxing the consistency requirement into linearizable write order guarantee and FIFO client order guarantee. The former guarantees all requests that update the state of ZooKeeper are serializable and respect precedence, and the latter guarantees that all requests from a given client are executed in the order that they were sent by the client. Such guarantees are the basis for ensuring the applications have the correct behavior, while still allowing high performance of the system.
- Zookeeper gets high performance in workload that mostly consists of read operations. When there are more write operations, the performance quickly deteoriate because the write operation needs an agreement protocal which requires broadcasting messages between all servers to reach the agreement. The broadcast is the bottleneck of Zookeeper performance.

Structure Overview

- Zookeeper uses a hierachy of data nodes called z-nodes, and these nodes are organized by namespace. In other words, the z-nodes are just like directories and files in a normal file system.
- The z-node is mainly designed for storage of meta-data used for coordination of clients, instead of for general data storage.
- The z-node also has meta-data of itself such as timestamp and version counters to keep track of changes to z-node.
- There are two types of z-nodes: the regular and the ephemeral type. Here ephemeral means transient and short lived.
- Clients create and delete the regular z-nodes explicitly by sending requests, and the ephemeral z-nodes are created explicitly, and can be removed either explicitly or automatically by the system when the connection session with the client terminates.
- When a new node is created, there is a sequential flag associated with the new node. The flag of a child node is guaranteed to be larger than its parent node.
- The change and monitor of node by clients is realized by subscription and notification: the client sends a watch flag with its read operation on a node, then in additionl to the completion of normal read operation, the server also promises it will notify the client next time the node data is changed. The notification is done only once then the watch flag is reset.
- Zookeeper has the following API: create(path, data, flags) for creating z-node, delete(path, version) for deleting z-node, exists(path, watch) for checking z-node existence, getData(path, watch) for read data from z-node, setData(path, data, version) for write data to z-node, getChildren(path, watch) for getting names of child z-nodes, sync(path) to wait for all updates pending at the start of the operation to propagate to the z-node.

Guarantees

- Zookeeper has the two basic ordering guarantees: linearizable writes and FIFO client order.
- Linearizable writes order guarantee means all requests that update the state of ZooKeeper are serializable and respect precedence.

- FIFO client order guarantee means all requests from a given client are executed in the order that they were sent by the client.
- The above two guarantees can be used to have configuration change for the system. The leader designates one node as ready node, which is used as signal to clients that they can use the configuration nodes for their own configuration. The clients will use these configurations only if they see the ready node. At the start of configuration change, the leader deletes the ready node before changing configuration, hence the clients will stop using the configuration from the configuration nodes. Then the leader creates the ready node again, so the clients will use the configuration from configuration nodes again.
- Due to the linearizable writes order guarantee, the clients will see the order of operation the same as the order of operation from the leader. Therefore, they will always see the ready node disappear, the configuration finishes changes, then the ready node appear again. Therefore, when the clients see the ready node, the configuration change must have been completed, so they will not use partially changed configurations.
- If a client checks the status of the ready node and gets OK then reads the configuration from configuration nodes, it is possible that the ready node disappears after the OK message then the configuration nodes are being changed and should not be read. However, the client could set a watch flag on the ready node then read the configuration nodes. Due to the FIFO client order guarantee, the change of status of ready node will reach the client first if it appears before it receives the configuration node data, hence the client can simply discard the configuration node data.
- Zookeeper has the guarantee that if a majority of ZooKeeper servers are active and communicating the service will be available, and if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a majority of servers is eventually able to recover.

Applications

- Configuration management: the configuration is stored in a single z-node. The clients get the configuration by reading the z-node and get notified about the configuration change by watching the z-node.
- Barrier: this means all clients need to be ready at a barrier then the process can continue. This is done by having a barrier node, and each ready client writes to the node and watch for its change. When the client are notified of the change, they proceed if the node information says everyone is ready.
- Group membership: Each group is represented by a group node. When a group member within the group starts, it creates a ephemeral child node under the group node. To find all members of a group, just list all childs of the group. When the group member is dead, the ephemeral node is automatically removed.
- Locks: The lock can be represented by a lock node. If a lock node does not exist, then this means the lock is not held and the client can create the lock node and operates on such files. Otherwise it watches the lock node and get notified when the lock node is released by another node. The lock can be optimized by implementing read-write lock in a similar way.
- The Fetching Service (FS): FS is the crawling performed by Yahoo search engine. It has a master providing the fetchers with configurations, and the fetchers write back information about their status and health. Here Zookeeper is used to manage configuration metadata and leader election.
- Katta: Katta is a distributed indexer that uses Zookeeper for coordination. Katta uses Zookeeper to track the status of master and slaves, handle master failure, and track and progagates configuration change to slaves.
- Yahoo Message Broker (YMB): YMB uses Zookeeper to manage distribution of topics of article to clients, and failure handling.

Implementation

- Upon receiving a request, a server prepares it for execution. If such a request requires coordination among the servers such as a write requests, then they use an agreement protocol such as Paxos with atomic broadcast, then finally servers commit changes to the ZooKeeper database fully replicated across all servers.

- The replicated database is an in-memory database.
- The size of each z-node is 1MB by default.
- Logs and periodic snapshots are used for recovery of in-memory database.
- In the stage where a server prepares a request, the server converts the transaction into a state transformation by calculating using current state information, so that the request becomes idempotent which means it does not harm to apply the same transaction multiple times.
- The idempotent transaction is forwarded to the leader of the agreement protocal, then the leader broadcasts the transaction to all followers within the agreement protocal.
- The broadcast protocal guarantees that changes broadcast by a leader are delivered in the order they were sent and all changes from previous leaders are delivered to an established leader before it broadcasts its own changes.
- Zookeeper uses snapshot to capture the state of the system, so that we do not need to apply each request in the log for recovery after failure.
- When a client requests a read, it connects to one of the many servers, and the read operation is done locally at the in-memory database replicate of this server. This local read is important for the high performance for read operations.
- The local read might get stale data. Therefore, the client calls sync followed by read to guarantee a given read returns the latest updated value.
- The request from the client is tagged with an ID called zxid which corresponds to the last transaction seen by the server. If the client connects to a new server, that new server ensures that its view of the ZooKeeper data is at least as recent as the view of the client by checking the last zxid of the client against its last zxid. If the client has a more recent view than the server, the server does not reestablish the session with the client until the server has caught up.

Performance

- The write throughput is less than the read throughput. This is because the write has to be broadcast to all machines, and the write must be logged before the result is returned to the client.
- When there are more servers, the throughput for a workload with high percentage of read becomes higher, while the throughput for a workload with high percentage of write becomes lower due to the atomic broadcast which puts more loads when the number of servers is larger.
- The atomic broadcast is the bottleneck for the performance of Zookeeper.
- If followers fail and recover quickly, then ZooKeeper is able to sustain a high throughput despite the failure.
- The leader election algorithm is able to recover fast enough to prevent throughput from dropping substantially, with less than 200ms downtime.
- The throughput of ZooKeeper is more than 3 times higher than the published throughput of Chubby.