Apache ZooKeeper

Курс «Базы данных»

Щитинин Дмитрий

Computer Science Center

11 ноября 2013 г.

Содержание

Introducing

High-available (decentralized, no SPoF) and high-reliable serivce for

- Naming
- Configuration management
- Synchronization
- Leader election
- Message Queue
- Notification system

Google Chubby

- 2006, Google published a paper on "Chubby"
- Google's primary internal name service
- Common mechanism for MapReduce, GFS, BigTable
  Usages:
  - election a primary from redundant replicas
  - repository for high availabable files

History

- ZooKeeper is the close clone of Chubby:
  - 2006-2008 - Developed at Yahoo![1]
  - 2008-2010 - Moved to Apache as Hadoop subproject
  - Jan 2011-... - Became top-level project of Apache projects system
- Commiters: Yahoo!, Cloudera, Google, Facebook, Microsoft, ...[2]

[1] http://www.sdtimes.com/content/article.aspx?ArticleID=33011

[2] http://zookeeper.apache.org/credits.html

Data Model

- Hierarchical tree of nodes (similar to Google Chubby) called "znode"
- znode can contain either data and subnodes
- Not high-volume data storage (znode data - max 1MB)
- Access by key (seq of znode's names "/"delimieted, like FS paths)
- Watches/Notications of znode changes
- znodes have version counters and other metadata

ACID

- *Atomicy* per node: no partial reads or writes
- No rollbacks
- Sequential *consistency* (clients see a single sequence of operations)
- *Isolation* per node
- *Durable* (commit log + replication)

CAP

- Guarantees consistency (writes are linearizable - have total order)
- May not be available for writes (strict quorum based replication of them)
- Partition Tolerance

Users

3

- Apache: HBase, HDFS, Hadoop MapReduce, Kafka, Solr
- Katta (distributed Lucene indexes)
- Eclipse Communication Framework
- Norbert, LinkedIn, partitioned routing, cluster management
- Neo4j, graph database
- S4, Yahoo, stream processing
- redis_failover (automatic master/slave failover)
- Yandex

[3] `https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy`

Use in HBase

- Leader Election
  Ensure there is at most 1 active master at any time
- Configuration Management
  Store the bootstrap location
- Group Membership
  Discovers servers and notifies about server death

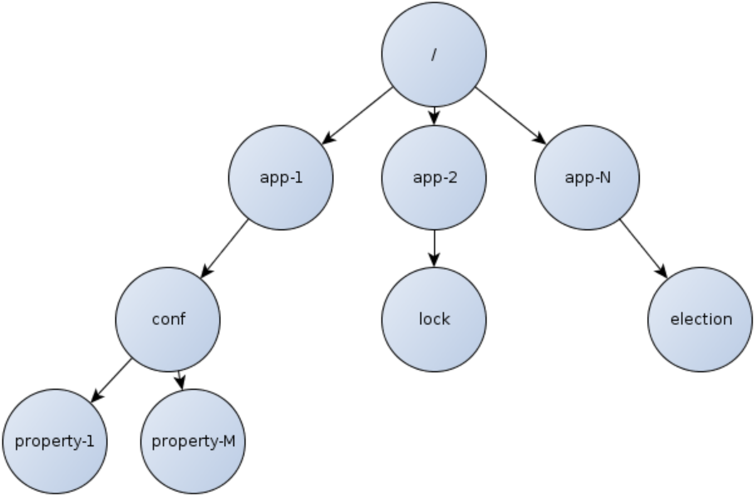
Motivation

- Making up own protocols for coordinate is almost failed
- Distributed system architecture is hard problem
- races, deadlocks, inconsistency, reliability
- ZK provides tools for create correct distributed applications:
    - hard problems already solved
    - no bycile re-inventions
    - using open-source well-tested service and clients

Data Model

- Hierarhical tree of nodes called "znodes"
- znode contains a data and ACL
- znode may contain children znodes
- 1MB of data for any znode
- Atomic data access (reads and writes)
- No append operation
- znodes referenced by path - "/"delimited strings

Data Model



Znode Types

Can be persistent or ephemeral - set at cration time.

- Persistent znodes
  - not tied to a client session
  - deleted explicitly by a client (any according to ACL)
  - may have children
- Ephemeral znodes
  - deleted by ZK when creating client session ends
  - always leaf nodes - may not have subnodes (no even ephemeral)
  - tied to a client session, visible to all clients (according to ACL)
  - ideal for wacth distributed resources availability

Sequential Znodes

- sequential number by ZK is a part of znode name
- sequential flag sets at cration time
- numbers are monotonically increasing
- sequence is maintained by parent znode
- used to impose a global ordering on events
- we will learn how to build distributed lock on top of them

Watches

Allow clients to get notifications when a znode changes

- set by operations on ZK
- triggered by other operations
- triggered once - for multiple notification client reregisters the watch
- used for quick reactions of different changes

Operations

Operation	Description
create	Creates a znode (the parent znode must already exist)
delete	Deletes a znode (the znode must not have any children)
exists	Tests whether a znode exists and retrieves its metadata
getACL, setACL	Gets/sets the ACL for a znode
getChildren	Gets a list of the children of a znode
getData, setData	Gets/sets the data associated with a znode
sync	Synchronizes a client's view of a znode with ZooKeeper

Guarantees

- **Sequential Consistency** - Updates from a client will be applied in the order that they were sent
- **Atomicity** - Updates either succeed or fail. No partial results.
- **Single System Image** - A client will see the same view of the service regardless of the server that it connects to
- **Reliability** - Once an update has been applied, it will persist from that time forward until a client overwrites the update
- **Timeliness** - The clients view of the system is guaranteed to be up-to-date within a certain time bound

Updates

Update operations (*delete*, *setData*) are conditional:

- version number of znode under update is specified
- performs only if version numbers are equal
- so updates are non-blocking

Clients

Core language bindings:

- Java (org.apache.zookeeper:zookeeper:3.4.5)
- C (zookeeper_st, zookeeper_mt libs)

Contrib bindings:

- Perl
- Python
- REST clients

Async & sync

Each binding provides async and sync APIs.

- both have same funtionality
- async is preferred for event-driven programming model
- async API can offer better throughput

Watch triggers

exists, *getChildren*, *getData* may have watches set on them

- triggered by write operations: *create*, *delete*, *setData*
- don't triggered by ACL operations
- watch event generated includes path of modified znode

Operations and triggers

	create znode	create child	delete znode	delete child	set data
exists	NodeCreated		NodeDeleted		NodeData Changed
getData			NodeDeleted		NodeData Chagned
getChildren		NodeChildren Chagned	NodeDeleted	NodeChildren Chagned	

Handling events

- *NodeCreated & NodeDeleted*
  contains modified znode path
- *NodeChildrenChange*
  need to call *getChildren* for retrieve fresh list of children
- *NodeDataChanged*
  need to call *getData* for retrieve fresh znode data

Warning!

State of the znodes may have changed between receiving the watch event and performing the read operation.

ACL

Available authentication types:

- *digest*
  by username and password
- *sasl*
  Kerberos [4] used
- *ip*
  IP address used

[4] http://en.wikipedia.org/wiki/Kerberos_(protocol)

Out of Box usages

Provided directly by API:

- Name Service
- Configuration management

Provided by *create*, *setData*, *getData* methods

Benefits:

- new nodes only need to know how to connect to ZooKeeper and can then download all other configuration information and determine
- application can subscribe to changes in the configuration and apply them quickly

Group Membership

- group is represented by a node (persistent)
- members of the group are *ephemeral* nodes under the group node
- clients call *getChildren()* on group node with watch set
- failed member nodes will be removed automatically by ZK
- *NodeChildrenChanged* notification send to clients

Lock

Obtain the lock:

- define lock node - *create('_locknode_')*
- call *create('_locknode_/lock-', sequential = true, ephemeral = true)*, remember created path as 'path-A'
- call *getChildren('_locknode_')* without setting the watch
- if 'path-A' has lowest seq number suffix from all children client obtains the lock and exits
- (if 'path-A' has not lowest seq number), let 'path-B' has next lowest seq number client calls exists() with the watch set on the 'path-B'
- if *exists()* returns false, go to 2 else wait for a notification for 'path-B' before going to step 2.

Release lock:

- Remove node created at step 1 (i.e. 'path-A')

Note:

- node removal will only cause one client to wake up (each node is watched by exactly one client)
- no polling or timeouts

ReadWrite Lock

Read lock acquire:

- read-A = *create("_locknode_/read- sequential=true, ephemeral=true)*
- *getChildren("_locknode_ watch = false)*
- are there children with path starting with "write-"and having a lower sequence number than 'read-A' ?
  no: acquire the read lock and exit
  yes: let 'write-B' has next lowest seq number and path starts with 'write-'
  call *exists(watch=true)* on 'write-B'
- if *exists()* returns false, go to 2 else wait for a notification for 'write-B' before going to step 2

ReadWrite Lock

Write lock acquire:

- "write-A- *create("_ locknode_ /write- sequential=true, ephemeral=true)*
- *getChildren("_ locknode_ watch =false)*
- are there children with a lower sequence number than the "write-A"?
  no: acquire the write lock and exit
  yes: let 'path-B' has next lowest seq number
  call *exists(watch=true)* on 'path-B'
- if *exists()* returns false, go to step 2 else wait for a notification for 'path-B'

Leader Election

Work path - '_election_'

Leader volunteer behavior:

- $z = create("\_election\_/n\_ sequential{=}true, ephemeral{=}true)$, $z = n\_i$ (i - seq number assigned for created node)
- $C = getChildren("\_election\_")$
  if i is the lowest seq suffix in C then current volunteer becomes a leader
- else watch for changes on "_election_/n_j where j is the largest sequence number and $j < i$

When receive a notification of znode deletion:

- $C = getChildren("\_election\_")$
- if i is the lowest seq suffix in C then current volunteer becomes a leader
- else watch for changes on "_election_/n_j where j is the largest sequence number and $j < i$

Apache Curator

"Curator is a set of Java libraries that make using Apache ZooKeeper much easier"[5]

- initially created by Netflix[6]
- moved too Apache ecosystem

[5] http://curator.apache.org

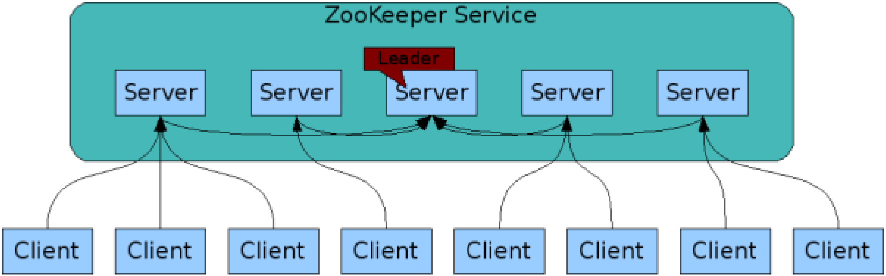
[6] http://en.wikipedia.org/wiki/Netflix

Apache Curator

Consists from:

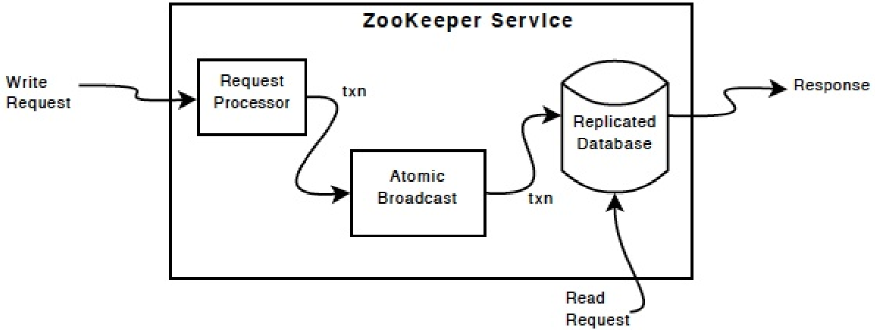
- **Framework** - high-level API that simplifies using ZooKeeper handles the complexity of managing connections to the ZooKeeper cluster and retrying operations
- **Recipes** - Implementations of some of the common ZooKeeper "recipes"
- **Utilities** - Various utilities that are useful when using ZooKeeper
- **Client** - A replacement for ZooKeeper class that takes care of some low-level housekeeping
- **Errors** - How Curator deals with errors, connection issues, recoverable exceptions, etc.
- **Extensions** - Other recipes (ServiceDiscovery)

Physical View



- leader is elected at startup and after leader failures
- clients connect to exactly one server to submit requests
  every server services clients:
  - read requests
  - write requests are forwarded to leader (responses are sent when a majority of followers have persisted the change)

Logical View



Request Processor

- prepares request for execution
- when write request received RP calculates what the state of the system will ater its applying and transforms it into a transaction
- transactions are idempotemt (unlike client request)
- future state must be calculated: there may be outstanding transactions that have not yet been applied to the database

Atomic Broadcast

Atomic broadcast (total order broadcast) - a broadcast messaging protocol that ensures that messages are received reliably and in the same order by all participants.

Special protocol implementation used - Zab

Replicated Database

- in-memory database containing the entire data tree
- each replica has its own copy
- for durability:
  - log updates to disk (replay log (a write-ahead) of committed operations)
  - force writes to be on the disk before they are applied to the in-memory database
  - generate periodic snapshots of the in-memory database

Broadcast algorithms

A broadcast algorithm transmits a message from one process (primary) to all other processes in a network or in a broadcast domain, including the primary. Atomic broadcast satisfies:

- **Validity**
  If a correct process broadcasts a message, then all correct processes will eventually deliver it
- **Uniform Agreement**
  If a process delivers a message, then all correct processes eventually deliver that message
- **Uniform Integrity**
  For any message m, every process delivers m at most once, and only if m was previously broadcast by the sender of m
- **Uniform Total Order**
  If processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'

Paxos

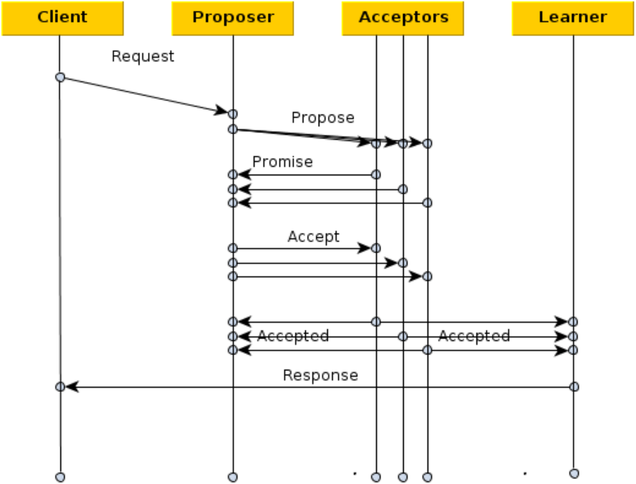
Traditional protocol for solving distributed consensus [7].

[7] http://en.wikipedia.org/wiki/Consensus_(computer_science)

Roles

- Client
  issues a request to the distributed system, and waits for a response
- Acceptor (Voter)
  Act as the fault-tolerant "memory"of the protocol. Acceptors are collected into groups called Quorums.
- Proposer
  Advocates a client request, attempting to convince the Acceptors to agree on it
- Learner
  Execute the request and send a response to the client)
- Leader A distinguished Proposer (called the leader) to make progress.

Basic Paxos


Client
Proposer
Acceptors
Learner
Request
Propose
Promise
Accept
Accepted
Accepted
Response

Why not Paxos

- was not originally intended for atomic broadcasting but consensus protocols can be used for atomic broadcasting
- does not enable multiple outstanding transactions
- does not require FIFO channels for communication, so it tolerates message loss and reordering
- does not support order dependency of proposed values.
  (Problem could be solved by batching multiple transactions into a single proposal and allowing at most one proposal at a time, but this has performance drawbacks)

Crash Recovery

- Peers communicate by message passing. They may crash and recover indefinitely many times.
- Quorum is subset of peers that contains more than half ones

  Any two quorums have a non-empty intersection
- Bidirectional channel for every pair of peers must satisfy:
  - integrity
  - prefix

  ZAB uses TCP (therefore FIFO) channels for communication

Transactions

State changes that the primary propagate ("broadcasts") to the ensemble, and are represented by a pair *(v, z)*:

- *v* - new state
- *z* - transactional identifier - called *zxid*

zxid z of a transaction is a pair *(e, c)*, where:

- *e* - the epoch number of the primary that generated the transaction
- *c* - is an integer acting as a counter

Broadcast protocol

Peer states:

- following
- leading
- election

Whether a peer is a *follower* or a *leader*, it executes three Zab phases:

- discovery
- synchronization
- broadcast

Previous to *Phase 1*, a peer is in state *election*, when it executes a leader election algorithm.

Phase 0: Leader election

- peers have state election
- no specific leader election protocol needs to be employed
- if peer p voted for peer p', then p' is called the prospective leader for p

Phase 1: Discovery

- followers communicate with their prospective leader
- that leader gathers information about the most recent transactions that its followers accepted
- discover the most updated sequence of accepted transactions among a quorum, and to establish a new epoch so that previous leaders cannot commit new proposals

Phase 2: Synchronization

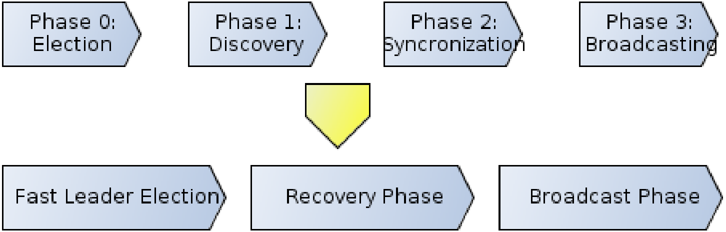
Recovery part of the protocol

- synchronize the replicas in the ensemble using the leader's updated history from the previous phase
- leader communicates with the followers, proposing transactions from its history. Followers acknowledge the proposals if their own history is behind the leader's history.
- when the leader sees acknowledgements from a quorum, it issues a commit message to them.
  At that point, the leader is said to be established and not anymore prospective.

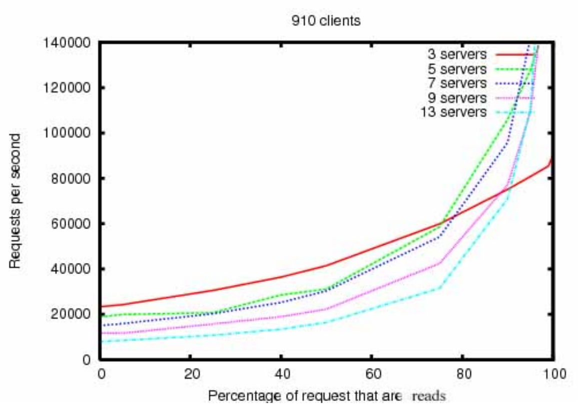
Phase 3: Broadcast

- peers stay in this phase until any crashes occur
- perform broadcast transaction for client write requests

Implemented protocol



Throughput



- running on servers with dual 2Ghz Xeon and two SATA 15K RPM drives
- one drive was used as a dedicated log device
- "servers"indicate the size of the ensemble

Off Screen

- More ZooKeeper usages
- ZooKeeper cluster tuning
- Error handling while native ZooKeeper client usage
- Deep study of Paxos and ZAB

Questions?

Thanks! Questions?