ZooKeeper Atomic Broadcast (for Project 2)

10/27/2016
Apache Hadoop

• 2002: Internet Archive search director Doug Cutting and UW grad student Mike Carafella set out to build a better open-source search engine.
  – Project called Nutch
• 2003: They though it was going well
  – Could run on a few machines and index hundreds of millions of pages (the web now has over a trillion pages)
• 2003 – 2004: Google’s GFS and MapReduce papers are published
  – Cutting and Carafella realize the Google framework approach was much better
  – Began to build something similar
Apache Hadoop (2)

• 2006: Cutting goes to work for Yahoo! (Carafella to academia)
  – Spins storage and processing part of Nutch into open source Apache project – Hadoop
  – Hadoop was the name of Cutting’s son’s stuffed elephant

• 2009: Cutting leaves Yahoo! to co-found Cloudera
  – Business to support Hadoop

• https://hadoop.apache.org
What is Hadoop today?

• “The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.

• The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.

• It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

• Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.”

http://hadoop.apache.org/
ZooKeeper

• “ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.”

• ZooKeeper provides “wait-free data objects” with “one-at-a-time” access to each object

• ZooKeeper must be highly available
  – “centralized” service actually consists of multiple servers
  – Each server stores a replica of the ZooKeeper data
Zookeeper Data Model and API

• Zookeeper has a file system data model

• Client API:
  – create(path, data, flags);
  – delete(path, version);
  – exists(path, watch);
  – getData(path, watch);
  – setData(path, data, version);
  – getChildren(path, watch);
  – sync(path);
Other ZooKeeper Features

• 2 types of files (znodes)
  – Regular znodes: explicitly created and deleted
  – Ephemeral znodes: deleted when creator’s session ends

• Watches
  – Clients can elect to receive notifications of changes to a file or directory
Example: Using ZooKeeper for Locks

• Application needs lock on resource to guarantee exclusive access
• Simplest approach: use lock file
  – Lock is a znode
• To acquire lock, client tries to create (ephemeral) lock file
  – Client deletes lock file when done with resource
  – Lock file is also deleted if client crashes
• Client cannot create lock file if it already exists
  – Means some other client has the lock
  – Client creates watch on lock file so it can try again when file is deleted (lock is released)
Example: Using ZooKeeper for Group Membership

- Group membership: application requires a list of all processes that are currently participating in it.
- Znode directory use to keep track of members
- When client joins, creates ephemeral file in directory
- When client leaves, file is deleted
- Application can get member list by listing all files in the directory.
Zookeeper System Architecture

Image Credit: ebook - Zookeeper: Distributed Process Coordination from O'Reilly
Reading and Writing to ZooKeeper

• A client accesses ZooKeeper by creating a session with a ZooKeeper server
  – Opens a TCP connection
• Read operations are performed by reading data on that server
• All write operations are funneled through primary (leader) replica
  – Server forwards write request to primary server
  – Primary uses ZooKeeper atomic broadcast algorithm to propagate writes to all servers
ZooKeeper for Project 2

• For Project 2, you will implement a replicated file system that provides the same API as in Project 1.

• You will use Zookeeper Atomic Broadcast (zab) to update the state at all servers.

• You do not need to implement any other features from ZooKeeper.
Zab System Model

- N processes (servers), complete graph
- Servers have stable storage
- Servers may crash and recovery (repeatedly)
- Reliable, FIFO communication channels
  - Implemented at network layer (TCP)

- Single server acts as leader (primary replica), broadcasts all state change messages to followers
  - Leader may change
- Leader also plays role of a follower
Zab Architecture

Primary Replica

Replica

Replica

PO Atomic Broadcast (Zab)

abcast

abdeliver

abdeliver

abdeliver
Messages in Zab

• If leader (primary) crashes, new leader is elected.
  – At most one leader is active at a time.
  – Infinite sequence of primary processes: \( \rho_1 \rho_2 \ldots \rho_e \ldots \)
    • e is called the \textit{epoch}
• A state change message is called a \textit{transaction}
  – \((v,z)\) where \(v\) is value and \(z\) is transaction id (zxid)
  – \(zxid = (epoch, counter)\)
  – \(zxid\)’s are totally ordered
• When new leader takes over
  – It increments epoch
  – It starts counter at 0, and increments before each broadcast
Properties of Zab

• **Integrity**: If some server delivers $(v,z)$, then some server has broadcast $(v,z)$.

• **Total order**: If some server delivers $(v,z)$ before $(v’,z’)$, then any server that delivers $(v’,z’)$ must also deliver $(v,z)$ before $(v’,z’)$.

• **Agreement**: If some server $p_i$ delivers $(v,z)$ and some server $p_j$ delivers $(v’,z’)$, then either $p_i$ delivers $(v’,z’)$ or $p_j$ delivers $(v,z)$. 
Properties of Zab (2)

• **Local primary order:** If a primary broadcasts \((v,z)\) before \((v',z')\), then a process that delivers \((v',z')\) must have delivered \((v,z)\) before \((v',z')\).

• **Global primary order:** If primary \(\rho_i\) broadcasts \((v,z)\) and primary \(\rho_j > \rho_i\) broadcasts \((v',z')\), then if a process delivers \((v,z)\) and \((v',z')\) it delivers \((v,z)\) before \((v',z')\).

• **Primary integrity:** If a primary \(\rho_i\) broadcasts \((v,z)\) and some process delivers \((v',z')\) broadcast by \(\rho_j < \rho_i\), then \(\rho_i\) must have delivered \((v',z')\) before broadcasting \((v,z)\)
Replica State

- Each replica stores
  - A log of all transactions, called a **history**
    - The history is stored on disk (stable storage)
    - For Project 2, this will be the sequence of create, append, and delete operations
  - The file system
    - The file system is stored in memory
    - For Project 2, this can be a Map of (name, contents) pairs
- Goal is for all replicas to have same history and file system
  - Some replicas may be slightly behind
Replica State (2)

- When a replica crashes and recovers, it can recreate its file system from the history and a list of transactions it missed
  - Need to make sure each transaction is executed only once when creating/updating file system
  - You must figure out how to do this for Project 2

- This recovery approach is not scalable – slow recovery time
  - Can reduce time through checkpointing
  - You do not need to implement checkpointing for Project 2
Zookeeper Atomic Broadcast

• Phase 0: Leader Election
  – Decide which server is the leader (primary)
• Phase 1: Discovery
  – The leader learns the most up-to-date state (history)
• Phase 2: Synchronization
  – The leader brings all followers’ histories up to date
• Phase 3: Broadcast
  – The leader broadcasts transactions to all followers
  – Broadcast is like Two-Phase Commit with FIFO Reliable Broadcast
Phase 3: Broadcast

1. On invocation of abcast(v) at leader
   generate transaction (v,z)
   
   \[ z = (\text{epoch, counter}) \text{ where counter is incremented between each proposal} \]
   
   send proposal (v,z) to all followers

2. When follower receives proposal (v,z)
   append (v,z) to history (write to disk)
   send ack to leader

3. When leader receives ack from majority of followers (all servers)
   send commit (v,z) to all followers

4. When follower receives commit (v,z)
   buffer (v,z) until all preceding transactions are delivered,
   then deliver (v,z)
Follower crashes

• When a follower crashes:
  leader detects crash, but does not need to do anything about it
• One caveat:
  – leader requires acks from majority of all followers before sending commit
  – System is blocked if majority of followers are down
• Your Project 2 implementation should handle this gracefully
  – Retry proposal until majority are up?
  – Let client know system is unavailable?
Follower Recovery

• When a follower recovers, it finds out who the leader is
  • by holding an election
    It then connects to the leader
• On learning about a new follower, the leader adds it to the set of servers to broadcast to.
• Leader also sends its history to the new follower to bring the follower up to date

• Follower may know about transactions that leader does not know about
  – E.g., old leader crashes before sending all proposals, but follower crashed after receiving proposal
• The follower should remove such proposals from its history
Zookeeper Atomic Broadcast

- **Phase 0: Leader Election**
  - Decide which server is the leader (primary)
- **Phase 1: Discovery**
  - The leader learns the most up-to-date state
- **Phase 2: Synchronization**
  - The leader brings all followers up to date
- **Phase 3: Broadcast**
  - The leader broadcasts transactions to all followers
  - Broadcast is like Two-Phase Commit with FIFO Reliable broadcast
Leader crash and election (Phases 0 and 1)

• When leader crash is detected, followers elect new leader
• New leader must learn about all transactions that have been delivered
  – Leader must have most up-to-date history
• Can achieve this by electing new leader that has most up to date history
• For Project 2, use the Bully Algorithm to elect a new leader
  – A server’s “process id” is the largest zxid in a server’s history, with some mechanism for tie-breaking
Zookeeper Atomic Broadcast

- **Phase 0: Leader Election**
  - Decide which server is the leader (primary)
- **Phase 1: Discovery**
  - The leader learns the most up-to-date state
- **Phase 2: Synchronization**
  - The leader brings all followers up to date
- **Phase 3: Broadcast**
  - The leader broadcasts transactions to all followers
  - Broadcast is like Two-Phase Commit with FIFO Reliable broadcast
Phase 2: Synchronization

• Before beginning the broadcast phase, the leader must bring all followers up to date:
  • Can be achieved by sending leader’s entire history to each follower

• Use 2PC approach
  – Leader sends history to all followers
  – Each follower sends ack to leader (records history)
  – On receiving ack from majority of followers, leader sends commit to all followers
  – On receiving commit, follower delivers all transactions in history in order of zxid
Implementation Details

• Each follower has a TCP socket open with the leader
  – Not one socket connection per message

• Server failure can be detected by socket going down (for Project 2)
  – In actual Zookeeper, this is too slow, so it uses heartbeats
Implementation Details (3)

• Clients can read from any server
  – Server state may not be up-to-date
  – That is ok

• Write operations (create, append, delete) can only be executed when
  – A majority of followers are active
  – The algorithm is in the broadcast phase

• Your code should gracefully handle client write requests when the system is not in a state to handle them
  – Queue request until later?
  – Give nice error message?
Implementation Details (2)

- Zookeeper atomic broadcast defines how transaction messages are disseminated
- It does not define how your file system is kept up to date
- You must design an algorithm for this