Building Replicated State Machines with Apache BookKeeper

Majordodo Usecase

Enrico Olivelli
Development Manager @Diennea

enrico.olivelli@diennea.com
eolivelli@gmail.com
Twitter: @eolivelli
https://www.linkedin.com/in/enrico-olivelli-984b7874

Agenda

Replicated State Machines
Apache BookKeeper Overview
Majordodo Overview
Majordodo and BookKeeper

Diennea and BookKeeper
Community updates
Q&A

Replicated State Machines

The problem:

Implement an high-available service without using any shared facility, such as databases or shared disks

Replicate State Machine approach:
- Execute independent copies (replicas) of the service on independent servers
- Replay the sequence of inputs to each replica
- Expect to observe the same sequence of state transitions on each replica
- Expect from each server to respond to clients with the same output

Main issue:

Each replica must see all the same entries, in the same order

Apache BookKeeper comes to help

Apache BookKeeper is a replicated log service which resolves our main issue.

- BookKeeper manages 'ledgers'
- A ledger is an ordered sequence of entries, like a database transaction log
- You can write to a ledger only once
- Only one client can write to a ledger
- Many clients can read from a ledger, even during writes
- Ability to 'tail' the log
- BookKeeper gives enough guarantees on this sequence of entries to implement the ordered sequence of inputs for a replicated state machine

BookKeeper guarantees

- Every entry which has been acked will be available for reads
- Only one client can write to a ledger
- Entries are immutable
- If an entry as been read from a client it will be readable from other clients
- Fencing: ability to kick out a writer

Replica state machines using a shared log

Consider a simple Key-Value store, such as a Java Map.

We want a **shared consistent view** of the Map.

Each replica will have a consistent view of the Map at a given point in time (position on the log)

It is possibile that the contents of the Map differ if the replicas have not reached the same position on the log

A **leader** is elected from the set of replica (using ZookKeeper for instance), other replicas will be **followers**.

The leader can update the Map:

1) write the change to the log (BookKeeper)
2) apply the change to memory

Followers tail the log and replay each change to the local copy of the Map.

What happens when the leader is lost or a network partition occours ?

Leader recovery and fencing

When leadership is lost a new leader is to be elected

Issue:

If the leader continues to act as a leader, and applies changes to its local copy of the Map then its view will diverge from the view of the other replicas

Usually leader election, for instance using ZookKeeper is not enough to prevent this situation.

Fencing is the solution

When a new leader is elected it first opens the log with the “fence” option: now the old leader is really kicked off and it is not able to write new entries, and so it will change its local Map

BookKeeper architecture

- Thick client
- Servers (Bookie) only store data
- ZooKeeper used for Bookie discovery, coordination and metadata management

Operations are driven directly by the client, which coordinates reads and writes.

Writing to a Ledger

Writes are driven by the writer of the ledger:

- Creates Ledger metadata
- Discovers and chooses Bookies (customizablwith e EnsemblePlacementPolicy)
- Writes data directly to Bookies
- Waits for ACKs
- Configurable durability/consistency (write quorum/ack quorum/ensablesize parameters)
- Last-Add-Confirmed (LAC) protocol

Interesting properties:

- No single point of failure
- No shared device
- Scalability (unique bottleneck can be ZooKeeper)
- Asynchronous API for every operation (open/read/write/delete/close)

Reading from a ledger

Reads are controlled by the client:

- Discover location of bookie data using ZooKeeper directly
- Read data directly from Bookies

Fencing: recover a ledger opened from writes

- mark the ledger as 'fenced' on metadata
- notify fencing to bookies
- recover last-add-confirmed (LAC) entry

Bookie internals

Bookie components:

- Journal: local Bookie log
- Ledger Storage Manager: store interleaved ledger data
- Garbage Collector Thread: free up disk space
- Auditor: handle ledger under-replication
- Cache for last entries: speed up “tailing reads”
- Index: speeds up random access reads
- Netty network server (TCP, local-transport, SSL is coming)

Majordodo Overview

- Majordodo is a Distributed Resource Manager
- Executes **Tasks** on a set of machines (**Workers**)
- Peculiarities:
  - Multi-tenancy
  - Low latency
  - Many (millions) concurrent tasks
  - at-most-once, at-least-once semantics
  - Dynamic priority and resource allocation to users even after task submission
- Usage example at MagNews:

  100 million tasks per day on a 100 machine cluster with a single broker

Majordodo Features

- Users submit tasks
- A Broker locates available resources on the cluster and assign the task to a worker
- Broker takes into account a dynamic priority assigned to the user and the type of task
- Resource usage by a task is a variable, it depends on:
  - User and resources assigned to it (databases, RAM, distributed storage, CPU...)
  - Type of task
- Each worker offers its CPU and other resources (datasources, disks., CPU, memory..)
- Broker handles failover of crashing workers or tasks execution errors
- A deadline can be set on each task
- REST Client API: tasks can be submitted from any language
- Transactions: batch of tasks can be submitted in transaction
- Slots: a sort of distributed lock facility
- Dynamic code deployment from Java Clients without Workers restart
- Embeddable in other JVM (both Brokers and Workers)

Majordodo replication

Majordodo Broker is a replicated state machine

- Task status, Transactions and Workers Status is the “state” of the machine
- One broker is leader, other brokers are followers. Fast fail of the leader in case of crash
- Follower brokers are used to keep the service always up
- ZookKeeper is used for Leader-Election and Broker discovery
- BookKeeper is the shared log

You cannot keep your ledgers forever, issues:

- Coordination of ledger deletion

- Taking snapshots of local state

- Adding a new broker

Snapshots and Ledgers deletion

Majordodo deletes ledgers after a configurable period of time.

Every X minutes each broker saves a copy of the state on local disk.

Two cases:

- a follower is tailing the log: no problem
- a follower boots and local snapshot is too old (or is absent at all)

If it it not possible to recover state from BookKeeper a fresh snapshot is downloaded from the leader.

Bad edge case: no follower is in synch for a very long time and the leader crashes and disk is lost.

Diennea Projects on BookKeeper

OpenSource projects:

- Majordodo: Distributed Resource Manager
  - https://github.com/diennea/majordodo
- HerdDB: Distributed SQL Database
  - https://github.com/diennea/herddb

MagNews platform:

- Distributed transaction log

Community Update

Known projects built on BookKeeper

- Twitter Distributed Log: Manhattan, Pub/Sub, DeferredRPC
- Yahoo Cloud Messaging
- Salesforce Distributed Store
- Huawei – HDFS NameNode
- HubSpot – WAL
- Diennea – Majordodo/HerdDB/MagNews TX log

Community

- 9 PMC members
- 11 Commiters
- 20-25 active members
- 5 Enterprises actively using/contributing

See our Wiki and WebSite

http://bookkeeper.apache.org

https://cwiki.apache.org/confluence/display/BOOKKEEPER