BookKeeper overview

by

Table of contents

1. BookKeeper overview

This document explains basic concepts of BookKeeper. We start by discussing the basic elements of BookKeeper, and next we discuss how they work together.

1.1. Basic elements

BookKeeper uses four basic elements:

- **Ledger**: A ledger is a sequence of entries, and each entry is a sequence of bytes. Entries are written sequentially to a ledger and at most once. Consequently, ledgers have an append-only semantics;
- **BookKeeper client**: A client runs along with a BookKeeper application, and it enables applications to execute operations on ledgers, such as creating a ledger and writing to it;
- **Bookie**: A bookie is a BookKeeper storage server. Bookies store the content of ledgers. For any given ledger L, we call an *ensemble* the group of bookies storing the content of L. For performance, we store on each bookie of an ensemble only a fragment of a ledger. That is, we stripe when writing entries to a ledger such that each entry is written to sub-group of bookies of the ensemble.
- **Metadata storage service**: BookKeeper requires a metadata storage service to store information related to ledgers and available bookies. We currently use ZooKeeper for such a task.

1.2. In slightly more detail...

BookKeeper implements highly available logs, and it has been designed with write-ahead logging in mind. Besides high availability due to the replicated nature of the service, it provides high throughput due to striping. As we write entries in a subset of bookies of an ensemble and rotate writes across available quorums, we are able to increase throughput with the number of servers for both reads and writes. Scalability is a property that is possible to achieve in this case due to the use of quorums. Other replication techniques, such as state-machine replication, do not enable such a property.

An application first creates a ledger before writing to bookies through a local BookKeeper client instance. To create a ledger, an application has to specify which kind of ledger it wants to use: self-verifying or generic. Self-verifying includes a digest on every entry, which enables a reduction on the degree of replication. Generic ledgers do not store a digest along with entries at the cost of using more bookies.

Upon creating a ledger, a BookKeeper clients writes metadata about the ledger to

ZooKeeper. A given client first creates a znode named "L" as a child of "/ledger" with the SEQUENCE flag. ZooKeeper consequently assigns a unique sequence number to the node, naming the node "/Lx", where x is the sequence number assigned. We use this sequence number as the identifier of the ledger. This identifier is necessary when opening a ledger. We also store the ensemble composition so that readers know which set of bookies of access for a given ledger.

Each ledger currently has a single writer. This writer has to execute a close ledger operation before any other client can read from it. If the writer of a ledger does not close a ledger properly because, for example, it has crashed before having the opportunity of closing the ledger, then the next client that tries to open a ledger executes an procedure to recover it. As closing a ledger consists essentially of writing the last entry written to a ledger to ZooKeeper, the recovery procedure simply finds the last entry written correctly and writes it to ZooKeeper in the form of a close znode as a child of "/Lx", where x is the identifier of the ledger.

Note that currently this recovery procedure is executed automatically upon trying to open a ledger and no explicit action is necessary. Although two clients may try to recover a ledger concurrently, only one will succeed, the first one that is able to create the close znode for the ledger.