BP-14 Relax Durability

Enrico Olivelli, JV Jujjuri, Sijie Guo

NOTE: This is a combination of proposal [1] [2].

Motivation

Apache BookKeeper is a distributed log storage. It works very hard to make every ***Entry*** durable and it works great for many use cases. There are many other use cases which doesn't need strong durability guarantees at the ***Entry*** boundary and would love to trade it for better throughput. Some of these use cases are described as below:

Close-to-Open consistency

There are multiple applications which need durability and consistency at Ledger boundaries. These applications treat entire Ledger as one unit, and a failure at any point treats the entire ledger write failure. The main focus of these applications is to achieve highest available throughput. For example, database bulk load is one of such use cases.

Raw data streaming:

When using ledgers as pure streams of data it would be useful to have an API similar to the FileChannel API, which lets the user to force sync at given points and not at every single "write" call.

Single disk layout:

Performance degradation happens when using single-disk-layout (storing indexes, data and journal on one same disk). Reducing the number of "fsync" operations helps in getting an overall good throughput and latency. In some use cases, it is acceptable to compensate the risk of not doing "fsync" and leveraging replication for durability.

This proposal is proposing the changes for achieving relax durability. These changes are summarized as below:

- *Introduce LedgerType: different types of ledgers have different durability level.*
- *The durability level only controls when an addEntry request is acknowledged.*

- *No changes to addEntry API (no mixing sync and no-sync addEntry operations on same ledger). AddEntry Request will be changed to send ledger type to bookies, which informs the durability level.*
- *No changes to LAC protocol:*
    - *LAC is only advanced when all entries before LAC has been successfully replicated and store at persistent medium.*
    - *ReadEntry can only read entries up to LAC*
- *ReadUnconfirmed can be used for reading beyond LAC until LAP (LastAddPushed). Entries between LAC and LAP are not guaranteed to be readable by the system. The application manages their correctness if they want to use ReadUnconfirmed.*
- *Introduce Sync API for explicitly sync written entries.*

The details of changes are listed in Section "Proposed Changes".

Proposed Changes

Durability Level

We will introduce "Durability Level" in this BP for supporting relaxed durability. These durability levels are:

- PD: Persistent Durability (default)
- VD: Volatile Durability

PD: Persistent Durability. The acknowledgement of `addEntry` request happens after an entry is successfully **replicated** and committed to **persistent storage**. (fsync is enforced)

VD: Volatile Durability. The acknowledgement of `addEntry` request happens after an entry is successfully **replicated** and committed to **volatile storage / memory**. (fsync is not enforced)

PD is the existing and default durability support in BookKeeper. VD is the one we will introduce in this BP.

If we have different durability levels,

- What happen to LAC protocol?
- How do we expose `durability level` to users?

Changes to LAC protocol

“Durability Level” only controls when an `addEnrtry` request is acknowledged. It ***DOES NOT*** and ***WILL NOT*** change existing LAC semantic.

That means:

- LAC is an entry id, which all entries until LAC are successfully replicated and committed to the persistent storage.
- LAC is only advanced when all entries before LAC are already successfully replicated and committed to the persistent storage.
- All the entries util LAC must readable all the time.

It also means, ***NO CHANGES*** to existing readEntries semantic:

- readEntries can only read entries up to LAC

It also means, ***NO CHANGES*** to existing recovery procedure.

The only change that we need to make is ***how do we advance LAC when receiving write responses***, because write responses can not be used as confirm acknowledges any more.

We will describe the LAC advance protocol at the end of this section.

Changes to Public API

We don’t expose `durability level` directly to end users at this moment[1]. The durability level will be implied by **LedgerType**.

Ledger Type

A ledger type is provided when creating a ledger. The ledger type will be persisted as part of ledger metadata and remaining unchanged during the ledger’s lifecycle (until it is deleted). The ledger type will be used as an indicator for durability level, and also as an indicator for storage level in future (e.g. bypassing journal).

The current supported ledger types are:

- PD_JOURNAL : (durability-level = PD)

[1] “Durability Level” is a concept about the persistent behavior. Durability can be achieved by either using journal or without journal. If we expose “durability level”, what does it mean if we create a ledger with bypassing journal and using PD level? It is going to be confusing. Durability level should be implicitly implied by the ledger type, which the implementation of this ledger type knows what durability level it provides.

- *VD_JOURNAL : (durability-level = VD, using journal)*[2]

How does the new API look like?[3]

```
CreateBuilder builder = bk.createBuilder();
…
CompletableFuture<LedgerHandle> createFuture = builder.ensembleSize(...)
        .writeQuorumSize(..)
        …..
        .ledgerType(LedgerType.PD_JOURNAL)
        .apply();

LedgerHandle lh = createFuture.get();
...
```

Ensemble changes are not allowed or disabled when ledgerType is not `LedgerType.PD_JOURNAL` for simplicity.

Sync API

A new sync API will added to `LedgerHandle` to enforce persisting entries. It will flush out any inflight entries and force fsyncing them at the bookie storage. It will return the last entry id that has been successfully replicated and committed to persistent storage.

```
Interface SyncCallback {
        Void syncComplete(int rc, LedgerHandle lh, long lastAddSynced, Object ctx);
}

void asyncSync(SyncCallback callback, Object ctx);
long sync();
```

why we return last synced entry id in the sync call? Instead of waiting on committing all the entries until LAP. Because it has a simple and deterministic semantic: 1) we don't need to reply on ordering for sync call; 2) we don't need to maintain unsynced entries for retries, because the inflight entries can be gone due to hardware failure when we issue fsyncing.

Changes to Wire protocol

[2] In future, we can have "VD_BYPASS_JOURNAL" for ledgers will be stored bypassing journals.
[3] The API is using the new style introduced in BP-15.

- Add flags to AddEntryRequest: the flag indicates the durability level.
- Add a `LastAddSynced` field in AddEntryResponse for piggyback `LastAddSynced` to the client. (we will explain how it is used in Section "LAC Advanced Protocol")
- Introduce SyncRequest/SyncResponse

Changes to Bookies

- For addEntry with durability level VD, the journal callback will be happen once the entry has been written to filesystem page cache.
- Introduce a sync cursor for ledgers with durability level VD, the sync cursor is comprised of:
    1. A 'LastAddSynced' entry id - all the entries before this 'LastAddSynced' entry are already synced locally here (or confirmed in the ensemble)
    2. A list of entry id ranges - the entries are already synced locally. The existence of this list is to handle out of order entries.

Update Sync Cursor

- When an entry is fsync'd for whatever reason, it updates sync cursor
    - If entry_id == LastAddSynced + 1, LastAddSynced = entry_id
    - If entry_id > LastAddSynced + 1, insert it into the list of entry id ranges.
- On receiving an `addEntry` request, it will also try to use LAC piggybacked in this entry to update cursor. Because LAC means all the entries have been successfully confirmed in the cluster.
    - If LAC > LastAddSynced, LastAddSynced = LAC.

LAC Advance Protocol

How do we advance LAC?

On Bookies

- On responding add responses, it will piggyback the `LastAddSynced` value in the sync cursor in the add responses.

On Clients

- The LastAddConfirmed can be calculated in following way:
    - Sort the ensemble by its `LastAddSynced` in ascending order
    - LastAddConfirmed = max(ensemble[0..(write_quorum_size - ack_quorum_size)])

For example: A, B, C is the ensemble, their LastAddSynced are 1, 2, 3.

If ack_quorum_size == 1, so LAC will be max(1, 2, 3) = 3
If ack_quorum_size == 2, so LAC will be max(1, 2) = 2
If ack_quorum_size == 3, so LAC will be max(1) = 1

Rejected Alternatives

No changes to the LAC Protocol:

The simplest possibility is not to change the behaviour of the writer in case of unsynced entries, but this will break the LAC protocol as readers may fail to read entries which are within the LastAddConfirmed range, or anyway reads won't be repeatable (entries may disappear even they have been already read once)

Bookie-wide option on the journal to disable 'sync':

Same as above, entries will disappear even if they have been read once.

Sending two acknowledge message back to the client:

An idea is to send back to the client an early acknowledge in case of no-sync and then a final message on the real sync .
This option is to be excluded because it will not reduce the number of required sync on the journal and it will make the protocol more complicated: a new flow of messages must be sent from to the bookie to the client, as the addEntry won't follow the simple request-response scheme, an addEntry will have two response messages.

Maintain different semantics for relaxed durability ledgers:

We could change the LAC Protocol and maintain a different semantic for ledger which leverage no-sync writes, for instance we can relax the constraint that an entry is read once it must be always readable. This could happen if we are advancing the LastAddConfirmed pointer even in case of no-sync writes.

Future Works

This sections covers the items beyond the scope of this BP. They will be addressed in future tasks.

Bypass the journal

We keep `bypassing journal` out of this BP to keep the discussion focused. The `bypassing journal` can be implemented as:

- Introduce a new ledger type
- Bookies use `ledger type` for:
    - Telling where to store the entries
    - How to sync entries when a `sync` request is received.

One EntryLogger per Ledger

This can be accomplished by adding a new ledger type. Whether we should combine `bypass the journal` and `one entrylogger per ledger` into one ledger type will be a separate discussion.

Allow ensemble changes for ledgers with no-sync entries

In order to simplify the work we are going to disable ensemble changes in presence of no-sync entries.

Allow no-sync writes on striping ledgers

At the moment striping ledgers, the ones with ensemble size > write quorum size, will not be able to require no-sync. This case will be covered in future works.

References

[1] Enrico's Proposal:
https://docs.google.com/document/d/1JLYO3K3tZ5PJGmyS0YK_-NW8VOUUgUWVBmswCUOG158/edit#heading=h.q2rewiqndr5v
[2] JV's Proposal:
https://docs.google.com/document/d/1g1eBcVVCZrTG8YZliZP0LVqvWpq432ODEghrGVQ4d4Q/edit