Overview

Before being made durable, FDB transactions must flow through several processes in an FDB cluster. This document gives a brief, high-level overview of each step in the commit pipeline. It focuses on the FDB7 implementation. For a more detailed overview of the commit pipeline, see this video.

Step 1: Client

Before a transaction is committed, all mutations are cached locally. The NativeAPI exposes a `Transaction` class which stores a vector of cached mutations. Bindings use a higher-level `ReadYourWritesTransaction` class, which also stores cached mutations in a tree that can be used to serve future read requests from the same transaction. In addition to storing vector of mutations, read- and write-conflict sets are also generated by each mutation.

Once the client calls `Transaction::commit`, a `CommitTransactionRequest` is sent to a commit proxy.

Step 2: CommitProxy

`CommitTransactionRequest`s are batched on the `CommitProxy`. Once a commit batch is ready, it is forwarded to the `commitBatch` actor, which coordinates the rest of the commit pipeline. The `commitBatch` actor first sends a `GetCommitVersionRequest` to the master server.

Step 3: MasterServer

The master server is in charge of distributing read versions and commit versions. Versions monotonically increase at a rate of approximately 1 million versions per second. The master server sends back a commit version via a `GetCommitVersionReply`.

Step 4: Back to CommitProxy

Once the commit proxy has received a commit version, it has all of the information it needs to send the commit batch to the resolvers for conflict resolution. In particular, the commit version and the read versions and read- and write-conflict sets of all transactions in a batch are sent to the resolvers in a `ResolveTransactionBatchRequest`.

Step 5: Resolver

The resolver maintains an in-memory skip-list storing the versioned write-conflict sets of all transactions committed in the last 5 seconds. This skip

list is used to detect transactions in the batch that must be aborted with `not_committed` errors due to serializability violations. Furthermore, transactions with read versions more than 5 seconds old must also be rejected with `transaction_too_old` errors.

In addition to resolving conflicts, resolvers are also responsible for saving a log of mutations to the transaction state store. This log is used by proxies to maintain a consistent view of the transaction state store.

Resolvers send a `ResolveTransactionBatchReply` reply back to the commit proxy, containing both information about which transactions must be committed or aborted and information about which transaction state store mutations from other proxies must be applied locally.

Step 6: Back to CommitProxy again

When the commit proxy receives a `ResolveTransactionBatchReply`, it must apply the transaction state store mutations to its own local copy of the transaction state store. Using the transaction state store, it must then determine how all mutations in the set of committed transactions should be tagged. These mutations are then sent to the transaction log system through the `ILogSystem` interface.

Step 7: Transaction Log System

Full details of the transaction log system are outside of the scope of this page. When running with replication factor `n`, each mutation will typically be sent to `n` transaction logs, and it will only be acknowledged as successful if all `n` transaction logs report being able to make the mutation durable. Asynchronously (outside of the critical commit path), storage servers pull these mutations from transaction logs.

Step 8: Back to CommitProxy again

When the commit proxy has heard back from the transaction log system, it reports the commit version to the master server with a `ReportRawCommittedVersionRequest`.

Step 9: Back to MasterServer

The master then updates its local register of the latest committed version, and all future read versions given out must be at least this large. This is part of the way FoundationDB provides read-after-commit consistency. The master acknowledges to the proxy that it has learned about the reported committed version.

Step 10: Back to CommitProxy again

The proxy is then ready to send replies back to the committing clients. Replies either consist of a commit version for successful commits or an error for unsuccessful (or maybe successful) commits.

Step 11: Back to the Client

For successfully committed transactions, clients have access to the commit version. For unsuccessful (or maybe-successful) commits, clients can retry transactions depending on the type of error received.