10 APPENDIX

10.1 Proof of Linearizability in FDBKeeper

Theorem 10.1 (Linearizability of FDBKeeper). *FDBKeeper guarantees the linearizability of operations by combining the strict serializability of FDB with client-side locking schemes.*

Proof. We prove this theorem by constructing a global linear history and demonstrating that it satisfies the properties of linearizability. This proof formalizes FDBKeeper's three-type client-side locking mechanism and dependency-based execution model, showing how it implements the same two ordering guarantees as ZooKeeper: cross-client linearizability and FIFO client order.

10.1.1 Notation and Definitions.

Basic Sets and Elements.

- $\mathcal{O}$: Set of all operations
- C: Set of all clients
- $\mathcal{P}$: Set of all paths (e.g., ZNode paths)

Operation Properties.

- $o, o_i, o_j \in \mathcal{O}$: Operations
- $T_s(o), T_e(o) \in \mathbb{N}$: Start and end times of operation o, measured by a global clock
- $\pi(o) \in \mathcal{P}$: Path on which operation o operates

Order Relations.

- $<_{rt}$: Real-time partial order, where $o_i <_{rt} o_j$ if $T_e(o_i) < T_s(o_j)$ (i.e., o_i completes before o_j begins)
- $<_c$: Client FIFO partial order, where $o_i <_c o_j$ if o_i and o_j are issued by the same client and o_i is issued before o_j
- $<_H$: Total order constructed by FDBKeeper, represented as $H = \langle o_1, o_2, \ldots, o_n \rangle$, where $o_i <_H o_j$ if $i < j$
- $S = \langle t_1, t_2, \ldots, t_m \rangle$: FDB transaction sequence, where t_i represents a transaction
- $\sigma : \mathcal{O} \to S$: Mapping from operations to their corresponding FDB transactions, i.e., $\sigma(o)$ is the transaction containing operation o

Locking Mechanism Definitions. FDBKeeper uses three types of locks to control operation execution order:

- $RL(p, c)$: Shared read lock held by client c on path p
  - This is a shared lock; multiple clients can hold read locks on the same path simultaneously
  - If operation o satisfies $\tau(o) =$ READ and $\gamma(o) = c$, it requires read locks on $\pi(o)$ and all its ancestor paths
- $WL(p, c)$: Exclusive write lock held by client c on path p
  - This is an exclusive lock; only one client can hold a write lock on a path at a time
  - If operation o satisfies $\tau(o) =$ WRITE and $\gamma(o) = c$, it requires a write lock on $\pi(o)$ and read locks on all its ancestor paths
- $CL(c)$: Commit lock for client c
  - Implemented as a FIFO queue to ensure operations from the same client are committed in the order they were issued
  - All operations from client c must acquire and release this lock in sequence
- D_c: Directed Acyclic Graph (DAG) of operation dependencies for client c
  - An edge $o_i \to o_j$ indicates that operation o_j must wait for operation o_i to release at least one lock
  - This graph is used to avoid deadlocks and ensure operations execute in the correct order
- $locks(o)$: Set of all locks required by operation o, including read locks, write locks, and the commit lock

Client Queue.

- $Q_c = \langle q_1, q_2, \ldots \rangle$: Queue of operations from client c, ordered by their issue time

10.1.2 Construction of Global Linear History H. We now construct FDBKeeper's global linear history H, which reflects the execution order of all operations in the system. This construction shows how FDBKeeper combines FDB's strict serializability capability with its own locking mechanism to achieve linearizability guarantees equivalent to ZooKeeper.

I. Cross-Client Linearizability via FDB Strict Serializability. In FDBKeeper, each operation is encapsulated as an FDB transaction, forming the transaction sequence $S = \langle t_1, t_2, \ldots, t_m \rangle$. FDB uses a single sequencer to assign a globally unique commit number to each transaction, ensuring strict serializability. This is similar to how ZooKeeper's Zab protocol achieves linearizability for cross-client updates.

Formally, if the end time of operation o_i is earlier than the start time of operation o_j (i.e., $T_e(\sigma(o_i)) < T_s(\sigma(o_j))$), then o_i's transaction $\sigma(o_i)$ must precede o_j's transaction $\sigma(o_j)$ in the FDB transaction sequence S.

II. FIFO Client Order via Client-Side Locking. FDB itself does not guarantee that operations from the same client are executed in the order they were issued. For example, if a client sequentially sends `Create(/a)`, `Create(/a/b)`, and `Create(/a/b/c)`, FDB does not guarantee they will execute in this order, potentially causing later operations to fail due to dependency violations.

FDBKeeper addresses this issue through its three-type client-side locking mechanism:

(1) **Read Locks (RL)**: Shared locks that allow multiple operations to read the same path simultaneously
(2) **Write Locks (WL)**: Exclusive locks that ensure serialized write access to paths
(3) **Commit Locks (CL)**: FIFO queues that ensure operations from the same client are committed in the order they were issued

For example, consider the operation sequences `Create(/a)`, `Create(/a/b)`, and `Create(/a/b/c)`:

- `Create(/a)` requires a read lock on the root path "/" and a write lock on "/a"
- `Create(/a/b)` requires read locks on "/" and "/a", and a write lock on "/a/b"
- The dependency graph D_c ensures that `Create(/a/b)` executes only after `Create(/a)` completes

We define H as the sequence of operations derived from the FDB transaction sequence S, where if $S[i] = \sigma(o)$ then $H[i] = o$. The commit lock mechanism ensures that operations from the same

client appear in H in the order they were issued, i.e., according to $<_c$.

10.1.3 Verification of Linearizability Properties. We now prove that the constructed global linear history H satisfies all properties of linearizability.

I. Existence and Well-Definedness of Global Order H.

Proposition 10.2. *H is a well-defined total order on O.*

Proof. The FDB transaction sequence S is a total order because FDB's sequencer assigns a unique commit number to each transaction. Since each transaction t_i maps to a unique operation o_i (in our model, each transaction contains one operation), H inherits the total ordering property of S.

For client c, the commit lock $CL(c)$ ensures that the commitment order of its operations matches their issue order $<_c$, while the dependency graph D_c ensures that hierarchical dependencies (e.g., `Create(/a)` $\rightarrow$ `Create(/a/b)`) are respected.

Therefore, $H = \langle o_1, o_2, \ldots, o_n \rangle$ forms a total order on all committed operations. □

II. Real-Time Order Guarantee (Cross-Client Linearizability).

Proposition 10.3. $\forall o_i, o_j \in O : o_i <_{rt} o_j \Rightarrow o_i <_H o_j$

Proof. Assume $o_i <_{rt} o_j$, meaning operation o_i completes before operation o_j begins in real time ($T_e(o_i) < T_s(o_j)$).

Since each operation is encapsulated as an FDB transaction, we have:

- $T_e(\sigma(o_i)) = T_e(o_i)$ (transaction end time equals operation end time)
- $T_s(\sigma(o_j)) = T_s(o_j)$ (transaction start time equals operation start time)

Therefore, $T_e(\sigma(o_i)) < T_s(\sigma(o_j))$, meaning transaction $\sigma(o_i)$ completes before transaction $\sigma(o_j)$ begins.

According to FDB's strict serializability property, when transaction $\sigma(o_i)$ completes before transaction $\sigma(o_j)$ begins, $\sigma(o_i)$ must precede $\sigma(o_j)$ in the transaction sequence S. Suppose $\sigma(o_i) = t_p$ and $\sigma(o_j) = t_q$, then $p < q$ in S.

By the construction of H, we have $H[p] = o_i$, $H[q] = o_j$, and $p < q$, which means $o_i <_H o_j$.

This proves that H respects the real-time order, corresponding to ZooKeeper's cross-client linearizability guarantee. □

III. Client FIFO Order Guarantee.

Proposition 10.4. $\forall o_i, o_j \in O : o_i <_c o_j \Rightarrow o_i <_H o_j$

Proof. Consider two operations o_i and o_j from the same client c, where $o_i <_c o_j$ (meaning o_i was issued before o_j).

Let the operation queue for client c be $Q_c = \langle q_1, q_2, \ldots \rangle$, where $q_i = o_i$ and $q_j = o_j$, with $i < j$ (positions in the queue).

FDBKeeper's commit lock $CL(c)$ is implemented as a FIFO queue, ensuring that operations acquire the lock in the order they appear in Q_c. Specifically, operation q_k must commit and release $CL(c)$ before operation q_{k+1} can acquire it, leading to $T_e(q_k) < T_s(q_{k+1})$.

Additionally, the dependency graph D_c enforces operation dependencies. For instance, if q_i creates path "/a" and q_j creates "/a/b", there exists an edge $q_i \rightarrow q_j$ in D_c, ensuring that q_j must wait for q_i to complete.

By induction from i to $j-1$, we can prove that $T_e(o_i) < T_s(o_j)$, meaning $o_i <_{rt} o_j$. According to the real-time order guarantee proven in the previous section, this implies $o_i <_H o_j$.

This demonstrates that FDBKeeper implements the same client FIFO order guarantee as ZooKeeper, addressing FDB's lack of native FIFO support. □

10.1.4 Conclusion. FDBKeeper successfully implements the same two basic guarantees as ZooKeeper:

- **Cross-Client Linearizability**: Through FDB's single sequencer and strict serializability mechanism
- **Client FIFO Order**: Through its three-type client-side locking mechanism (read locks RL, write locks WL, commit locks CL) and dependency graph D_c

These two mechanisms together ensure the linearizability of operations, even though FDB itself does not directly support client FIFO order. Therefore, Theorem 10.1 holds: FDBKeeper guarantees the linearizability of operations by combining the strict serializability of FDB with client-side locking schemes. □

10.2 Linearizability with Batch Processing

Theorem 10.5 (Linearizability with Batch Processing). *FDBKeeper maintains linearizability even with batch processing optimizations.*

Proof. We show that both batch processing optimization strategies preserve linearizability by demonstrating that we can construct a valid linearizable history for the optimized system. We build upon the linearizability proof of Theorem 10.1.

Let us recall the key elements from the original proof:

- O: Set of all operations
- S: FDB's transaction sequence
- H: Global history derived from S
- $<_{rt}$: Real-time order, where $o_i <_{rt} o_j$ if $T_e(o_i) < T_s(o_j)$
- $<_c$: Client FIFO order within a client
- $<_H$: Total order in history H

In this optimization, N write operations $O_{\text{batch}} = \{o_1, o_2, \ldots, o_N\}$ from the same client c are merged into a single FDB transaction t_m. According to the FDBKeeper implementation, transaction batching is specifically designed to operate only on operations from the same client, as stated in the optimization description: "Configure N to merge multiple write operations into a single transaction." This is enforced by the client's commit lock mechanism.

I. Constructing a Modified History H'.

Let S' be the sequence of FDB transactions where batched operations are represented by a single transaction. We construct a new history H' as follows:

- For operations not in any batch, they appear in H' in the same position as in H
- For operations in a batch O_{batch}, we place them consecutively in H' at the position where their batch transaction t_m appears in S', preserving their original order from $<_c$

Formally, we define $\beta : O \to \mathbb{N}$ as a function that maps each operation to its position in H'.

II. Atomicity vs. Ordering.

Although we place batch operations consecutively in H' (which is a logical construct for our proof), it's crucial to understand that from an external observer's perspective, all operations in a batch appear to execute atomically at the point when the transaction commits. The ordering within the batch is only visible in the final state and results, not in the execution timeline. This maintains the atomic property of transactions while allowing us to reason about the order of effects.

Example 1: Consider operations o_1, o_2, o_3 from client c that create nodes "/a", "/a/b", and "/a/b/c" respectively. When batched in transaction t_m, externally they appear to happen atomically at the commit point of t_m. In H', we place them as $\langle o_1, o_2, o_3 \rangle$ to reflect their logical order, though they are executed in a single transaction.

III. Proving H' Respects Real-time Order.

We must show $\forall o_i, o_j \in O : o_i <_{rt} o_j \Rightarrow o_i <_{H'} o_j$

Case 1: Neither o_i nor o_j is in a batch.

- In this case, their positions in H' correspond to their transaction positions in S', which respects real-time order due to FDB's strict serializability.

Case 2: o_i is in batch O_{batch} but o_j is not.

- If $o_i <_{rt} o_j$, then the entire batch transaction t_m completes before o_j begins, so $t_m <_{S'} t_j$. By construction of H', all operations in O_{batch} appear before o_j in H'.

Case 3: o_i is not in a batch but o_j is in batch O_{batch}.

- If $o_i <_{rt} o_j$, then o_i completes before any operation in the batch begins. By FDB's strict serializability, $t_i <_{S'} t_m$, so o_i appears before all operations in O_{batch} in H'.

Case 4: o_i and o_j are in the same batch O_{batch}.

- As established earlier, FDBKeeper's transaction batching only combines operations from the same client. Therefore, o_i and o_j must come from the same client, so either $o_i <_c o_j$ or $o_j <_c o_i$.
- If $o_i <_c o_j$, then our construction of H' places o_i before o_j within the consecutive sequence representing the batch.
- This preserves the logical dependency between them, while still maintaining their atomic execution from an external observer's perspective.

Case 5: o_i and o_j are in different batches O_{batch1} and O_{batch2}.

- If $o_i <_{rt} o_j$, then the entire transaction t_{m1} for O_{batch1} completes before t_{m2} for O_{batch2} begins.
- By FDB's strict serializability, $t_{m1} <_{S'} t_{m2}$, so all operations in O_{batch1} appear before all operations in O_{batch2} in H'.

IV. Proving H' Preserves Client FIFO Order.

We must show $\forall o_i, o_j \in O : o_i <_c o_j \Rightarrow o_i <_{H'} o_j$

If $o_i <_c o_j$, then either:

- They are in different transactions, and the commit lock $CL(c)$ ensures o_i transaction commits before o_j transaction begins, so $o_i <_{H'} o_j$.
- They are in the same batch, and our construction of H' preserves their client order, placing o_i before o_j in the consecutive sequence.

For the batched operations, all operations in the batch are executed atomically in a single transaction, which ensures their combined effect is consistent with their sequential execution in order of $<_c$. The atomicity of FDB transactions guarantees no partial effects are visible.

In this optimization, certain operation pairs are merged into a single operation based on heuristic rules:

1. Create+Get → CreateOrGet
2. Create+Create → Single Create

Case 1: Create+Get → CreateOrGet

Let o_c be a Create operation and o_g be a Get operation on the same path, with $o_c <_c o_g$. These are merged into a single operation o_{cg} (CreateOrGet).

We need to show that replacing $\{o_c, o_g\}$ with o_{cg} in the history preserves linearizability:

- o_{cg} acquires the same locks that o_c and o_g would have acquired (write lock for path and read locks for ancestors).
- The results returned by o_{cg} are identical to what would be returned by sequential execution of o_c followed by o_g.
- For any other operation o_k:
  - If $o_k <_{rt} o_c$, then $o_k <_{rt} o_{cg}$
  - If $o_g <_{rt} o_k$, then $o_{cg} <_{rt} o_k$
  - If operation o_k is concurrent with either o_c or o_g, we need to consider lock interactions:
    - If o_k is a write operation affecting the same path, it will contend for the write lock with o_{cg}.
    - The lock manager ensures that either o_{cg} acquires the lock first and completes before o_k can proceed ($o_{cg} <_H o_k$), or o_k acquires the lock first and completes before o_{cg} can proceed ($o_k <_H o_{cg}$).
    - This lock-based serialization ensures a consistent ordering between o_{cg} and any conflicting concurrent operation.

Case 2: Create+Create → Single Create (from same client)

Let o_{c1} and o_{c2} be two Create operations for the same path from the *same client* c. Under this optimization:

- When a client issues multiple Create operations for the same path (e.g., due to retries or application logic), only the first operation o_{c1} executes normally.
- Subsequent Create operations o_{c2} for the same path are detected by FDBKeeper.
- Instead of executing a separate transaction, o_{c2} immediately fails with a "node already exists" response without requiring a transaction to the database.

Since o_{c1} and o_{c2} are from the same client, the client FIFO order relation applies: $o_{c1} <_c o_{c2}$. This optimization maintains linearizability because:

- The merged behavior results in the same external outcome as if o_{c1} completed before o_{c2} started.
- The client still sees the results in the correct order: first operation succeeds, subsequent operations fail with "node exists."
- This is consistent with a sequential execution where $o_{c1} <_H o_{c2}$.

Example 1: A client issues Create("/a") twice in succession. The first operation o_{c1} executes normally. When the second operation o_{c2} is processed, FDBKeeper detects that it's attempting to create the same path and immediately returns "node already exists" without executing another transaction. This behavior is identical to the non-optimized case where both operations execute as separate transactions, but improves performance by avoiding unnecessary transactions.

Example 2: Consider operations o_1, o_2, o_3, o_4 from client c creating nodes "/a", "/a/b", "/a/b/c", and "/a/d" respectively. When batched in transaction t_m, they appear to happen atomically at t_m's commit point. In H', we place them as $\langle o_1, o_2, o_3, o_4 \rangle$ to reflect their logical order, though they are executed in a single transaction.

V. Combining Both Optimizations.

When both optimization strategies are applied, they interact in the following way:

- Operation lock merging happens first, converting certain operation pairs (like Create+Get) into single composite operations.
- Transaction merging then potentially batches multiple operations (including these already-merged composite operations) from the same client into a single transaction.

To prove that their combination preserves linearizability, we need to show that the composition of these optimizations doesn't introduce any behaviors that violate linearizability requirements.

For any operations o_i and o_j where $o_i <_{rt} o_j$:

- If o_i is part of a merged operation (e.g., CreateOrGet), the merged operation completes no earlier than o_i would have.
- If o_j is part of a merged operation, the merged operation starts no later than o_j would have.
- If either operation is part of a batch transaction, the batch transaction respects the real-time ordering with other transactions.

This ensures that real-time ordering is preserved even when both optimizations are applied. The client FIFO ordering is similarly preserved by the commit lock mechanism, regardless of which optimizations are applied.

The batch processing optimizations in FDBKeeper modify the execution of operations but preserve the essential properties required for linearizability:

- Real-time ordering is respected across all operations
- Client FIFO order is maintained

Therefore, FDBKeeper maintains linearizability even with batch processing optimizations. □