# APPENDIX

Due to space constraints, we have included some content in the appendix to complement the main text of FDBKeeper.

## A BACKGROUND

FDBKeeper is designed to be compatible with ZooKeeper metadata and interface definitions. In this section, we first provide a background on ZooKeeper (§A.1), and introduce the basics of FoundationDB (§A.2).

### A.1 ZooKeeper

ZooKeeper [41] is a prominent distributed coordination service. It manages various tasks among multiple nodes in distributed systems (e.g., communication, state synchronization, resource allocation), ensuring system consistency, availability, and reliability while shielding the upper-layer applications from complex consistency protocols.

The hierarchical namespace in ZooKeeper is implemented using a tree structure. This closely resembles the directory tree structure of a file system, with the distinction that ZooKeeper does not differentiate between directory node and file node. Each node has a unique path identifier, with the root node represented by "/" and each child node indicating a hierarchical relationship. Hierarchical namespaces are extensively utilized in distributed systems such as Hadoop, ClickHouse, and Kubernetes to differentiate between various tenants, services, and functions. Specifically, each node in ZooKeeper is called a Znode [13]. Each Znode consists of three parts: data, metadata, and access control list (ACL). The data consists of a binary string with a maximum size of 1 MB. Metadata represents the basic information of the node. ACLs are used for fine-grained control of user access permissions to the node. ZooKeeper also offers a comprehensive API for managing, monitoring, and manipulating these nodes. These definitions are found in ZooKeeper programmer's guide [13].

ZooKeeper guarantees updated linearizability [41] with linearizable writes. On the other hand, ZooKeeper guarantees the FIFO client order [41], in which all requests from a given client are executed in the order in which they were sent. It is important to note that the linearizability in ZooKeeper is asynchronous. They provide a real-time guarantee on the execution of a set of single operations.

### A.2 FoundationDB

FoundationDB (FDB) [68] is a distributed key-value database that provides an ordered and transactional key-value store that combines the flexibility and scalability of NoSQL [29, 43] with ACID transactions. FDB adopts an unbundled architecture [50], consisting of a control plane and a data plane, both of which can be independently scaled out. The control plane is responsible for persisting critical system metadata, such as the configuration of transaction systems, and uses active disk paxos [31] for high availability. The data plane comprises a distributed transaction management system for in-memory transaction processing, a log system that stores write-ahead logs (WALs) for the transaction system, and a separate distributed storage system for storing data and servicing reads.

FDB achieves strict serializability [7, 46, 68] through a combination of optimistic concurrency control (OCC) [38, 44, 45, 60, 62, 63] and multi-version concurrency control (MVCC) [25, 65, 66, 69]. Furthermore, FDB separates the transaction management system (write path) from distributed storage (read path), enabling independent scaling of each. A central research challenge addressed in this paper is to achieve linearizability in ZooKeeper by leveraging the consistency model of FoundationDB (FDB) and fully utilizing its transactional characteristics.

## B WORKFLOW OF CREATING AND UPDATING LEASE

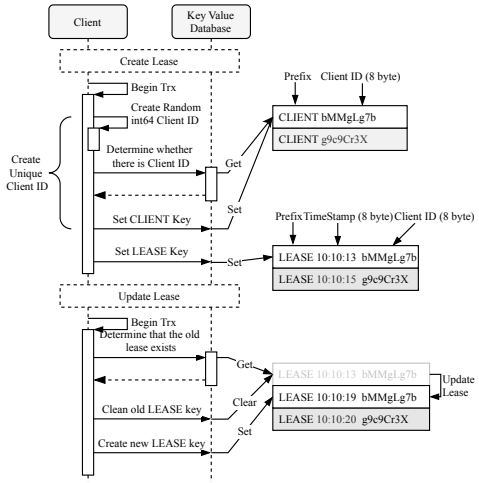


**Figure 17: The creating and updating lease workflow is based on local time.**

Figure 17 illustrates the workflow for creating and updating leases and how the FDBKeeper (i.e., client) interacts with FDB servers. The lease structure includes two parts: the client key, prefixed with "CLIENT", and the lease key, prefixed with "LEASE". Both the "CLIENT" and "LEASE" types are enumerated and can be represented using single-byte numbers.

## C CONTROL FLOW OF FIFO CLIENT ORDER

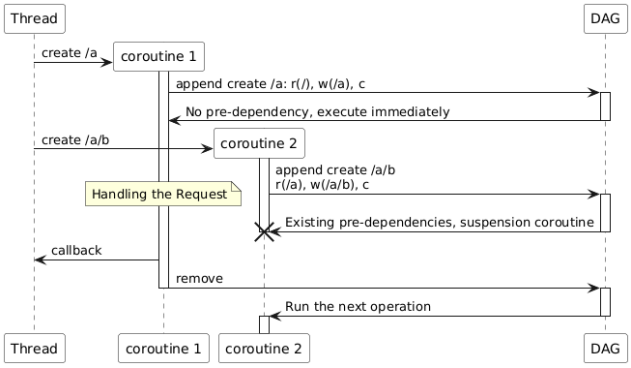


**Figure 18: Control flow for local operations for ensuring FIFO client order.**

Figure 18 illustrates the control flow of local operations. Upon acquiring locks r(/), w(/a), and c initially, the *Create(/a)* operation

executes immediately due to no dependencies. Conversely, *Create(/a/b)* suspends its execution coroutine (i.e., coroutine2) due to dependency. The coroutine2 resumes only after *Create(/a)* commits and its corresponding locks are removed from the DAG.