Cloud and Cluster Data Management

FOUNDATIONDB: A DISTRIBUTED UNBUNDLED TRANSACTIONAL KEY VALUE STORE

Motivation

- **Cloud services** must be fault tolerant and highly available
- Provide sufficiently strong semantics and flexible data models
  - For rapid application development
- They must scale to
  - Billions of users
  - Petabytes or exabytes of data
  - Millions of requests/second
- NoSQL systems emerged in response to these requirements
- But to scale, these systems scarified transactional semantics
  - Provided eventual consistency
  - Developers must reason about interleaving of updates from concurrent operations
- NoSQL systems retrofitted ACID transactions in recent years
  - Cassandra, MongoDB, CouchBase are now support some form of ACID

FoundationDB (FDB)

- “Foundational set of building blocks required to build higher-level distributed systems”
- FDB is an ordered transactional, key-value store
- natively supports multi-key strictly serializable transactions across its entire key space.
- Defaults to strictly serializable trxns, but allows for relaxing these semantics.
- Modular approach:
  - Highly scalable transactional storage engine
  - No schema management, no data model, no query language, no secondary indices or many other features found in traditional transactional dbs
- Allows users to implement advanced features:
  - Consistent secondary indices, Referential integrity checks,…

Why FDB is popular?

- Focuses on the “lower half” of database
- Stateless applications developed on top can provide various data models and other capabilities
- Apps that traditionally require different types of storage systems can instead leverage FDB
- Example layers built on top of FDB:
  - Record Layer: provides a relational database capabilities
  - JanusGraph: a graph database
  - CouchDB: rebuilt as a layer over FDB
- Provides a deterministic database simulation framework
  - Simulates network of interacting processes + variety of disk, process, NT, and request-level failures and recoveries.
  - Makes FDB stable and allows developers to introduces new features in a rapid cadence.

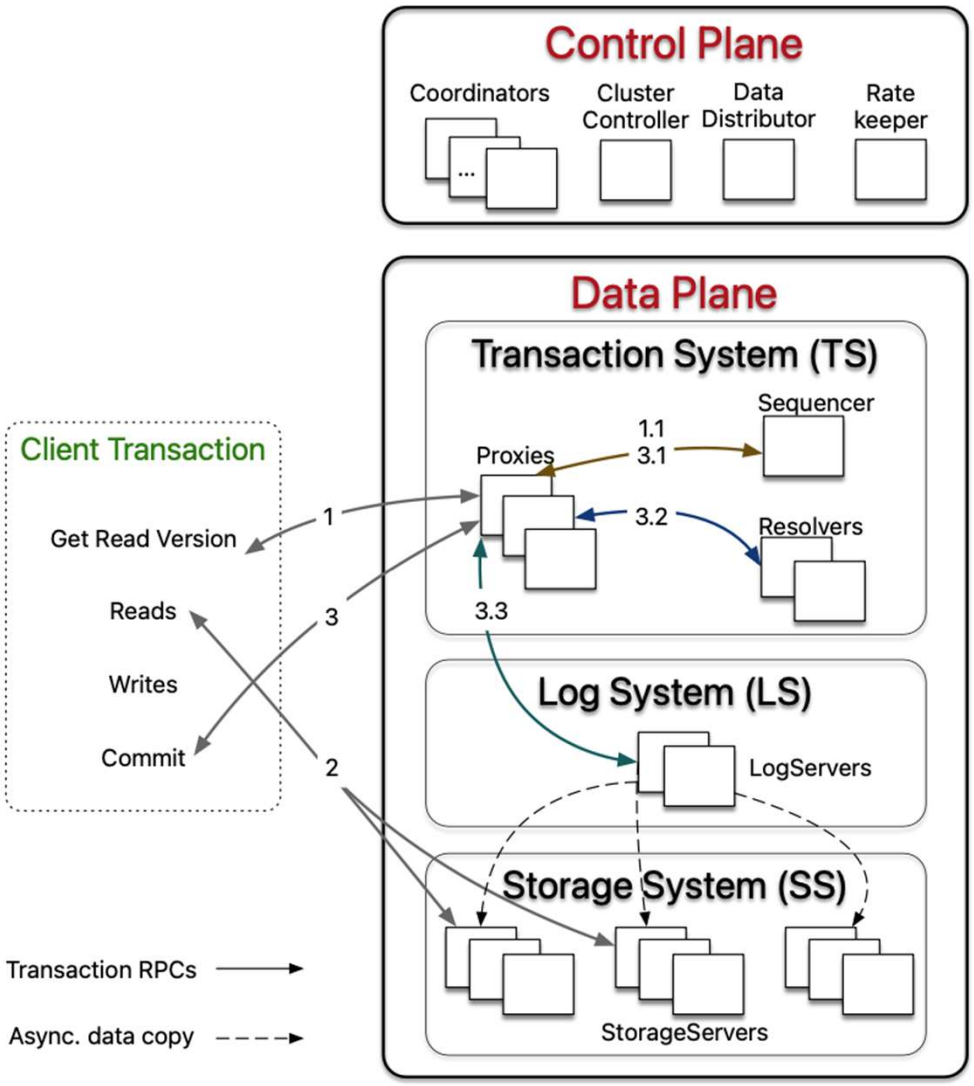
FDB Design Principles

- Divide-and-Conquer:
  - FDB decouples the TMS (write path) from the distributed storage (read path) [scale independently]
  - In TMS, processes are assigned various roles: timestamp management, accepting commits, conflict detection, logging.
  - Cluster-wide orchestrating tasks (Load control, balancing and failure recovery) are divided and serviced by heterogeneous roles
- Make failure a common case:
  - Handle all failures through the same recovery path:
    - TS proactively shuts down when it detects a failure ➔ all failures handling is reduced to a single operation
- Fail fast and recover fast:
  - FDB strives to reduces MTTR (Mean-Time-To-Recovery) = time to detect failure + shut down TMS + recover (<5s in production cluster)
- Simulate testing:
  - Randomized, deterministic simulation framework for testing the correctness of its distributed database
  - Expose deep bugs and boost developer productivity and code quality.

System Interface

- Single key:
  - get(): returns the value of a single key
  - set(): writes the value of a key
- Key Ranges:
  - getRange(): returns a sorted list of keys and their values within the given range
  - clear(): deletes all keys-value pairs within a range or starting with a certain key prefix
- Mutations updates a snapshot of the DB at a certain version, only applied to the db when the traxn commits
- Writes (set() and clear()) buffered by the client until final commit()
- Offers read-your-writes by combining lookup results from DB with uncommitted writes of the trxn
- Size limits: Key ➔ 10 KB, Values ➔ 100KB, trxn size ➔ 10MB

FDB Architecture



- **Control Plane:**
  - responsible for persisting critical system metadata: configuration of trxn systems, on Coordinators.
- **Data Plane:**
  - FDB uses an unbundled arch.
    - Distributed TS: performs in-memory trxn processing
    - Log system (LS): stores Write-Ahead-Log (WAL) for TS
    - Storage system (SS): for storing data and servicing reads

Figure Credit: "FoundationDB: A Distributed Unbundled Transactional Key Value Store, by Zhou J, et al. 2021"

Control Plane

- Coordinators Form disk Paxos group and elect a ClusterController (CC)
- CC monitors all servers in the cluster and recruits:

1. **Sequencer**: assigns read and commit versions to trxns
2. **DataDistributor**: monitors failures and balances data among StorageServers
3. **Ratekeeper**: provides overload protection for the cluster

- Sequencer, DataDistributor and Ratekeeper are re-recruited if they fail or crash

> "Disk Paxos Group: a reliable distributed system with a network of processors and disks. [It] maintains consistency in the presence of arbitrary non-Byzantine faults. Progress can be guaranteed as long as a majority of the disks are available, even if all processors but one have failed."
> *Credit: "Disk Paxos" By Eli Gafni & Leslie Lamport*

Data Plane

- TS process trxns and consists of Sequencer, Proxies and Resolvers (all stateless)
  - Sequencer: assigns a read version and a commit version to each trxn and recruits Proxies, Resolvers and LogServers
  - Proxies: provide MVCC read versions to clients and orchestrate trxn commits
  - Resolvers: check for conflicts between trxns
- LS contains a set of LogServers
  - LogServers: act as replicated, sharded, distributed persistent queues, each queue stores WAL data for a StorageServer
- SS has several StorageServers serve client reads
  - Each StorageServer stores a set of data shards (contiguous key ranges)
  - StorageServers are the majority of processes in the system
  - Uses a modified version of SQLite as a storage engine

Read-Write Separation and Scaling

- Processes are assigned different roles (e.g., Coordinators, StorageServers, Sequencer)
- FDB scales by increasing the number of processes for each role
- This separates scaling of reads from writes
  - Reads scale linearly to with the number of StorageServers
  - Writes (trxn commits) scale by adding more Porxies, Resolvers and LogServers in TS and LS.
- Singletons (ClusterController and Sequencer) and Coordinators are not performance Bottlenecks
  - they only perform limited metadata operations

Bootstrapping

- FDB has no external service dependency
- All users' data and most metadata are stored in StorageServers
- StorageServers metadata is persisted in LogServers
- Config of LS (info about LogSevers) are stored in all Coordinators
- Bootstrapping Process:
  1. Coordinators (using disk Paxos group) attempt to become the ClusterController (if none exists)
  2. The elected ClusterController recruits a new Sequencer
  3. The sequencer reads config of old LS stored in Coordinators and spawns a new TS and LS
  4. Proxies recover system metadata (info about all StorageServers) from the old LS.
  5. Sequencer waits until the new TS finishes recovery then writes the new LS config to all Coordinators

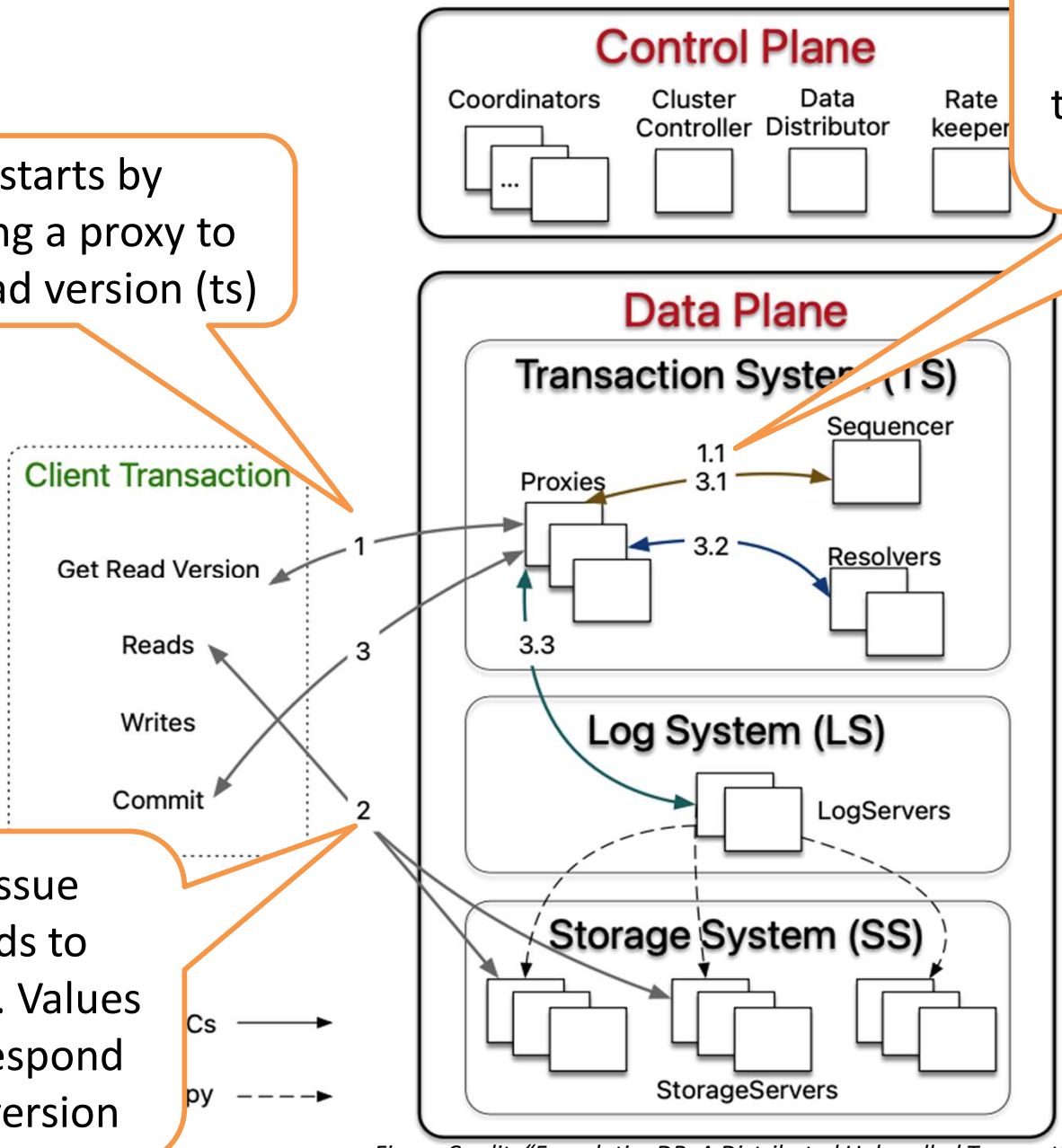
Reconfiguration

- When TS or LS fails, or DB config changes, reconfiguration process resets the configuration of the TS (brings its config to a clean state)
- The sequencer process is terminated if any of the Proxies, Resolvers, or LogServers fails or db config changes
- ClusterController detects the Sequencer failure event
- Then recruits a new Sequencer, which follows the same Bootstrapping process we discussed, to spawns the new TS and LS instance
- Trxn processing is divided into epochs: each represents a generation of the trxn management system with its unique sequencer process

Transaction Management: End-to-end Trxn Processing (Read)

Trxn starts by contacting a proxy to obtain read version (ts)

Proxy asks Sequencer for read version (no less than prev. issued trxn commit version) & send it back to client



Client may issue multiple reads to StorageServers. Values returned correspond to their read version

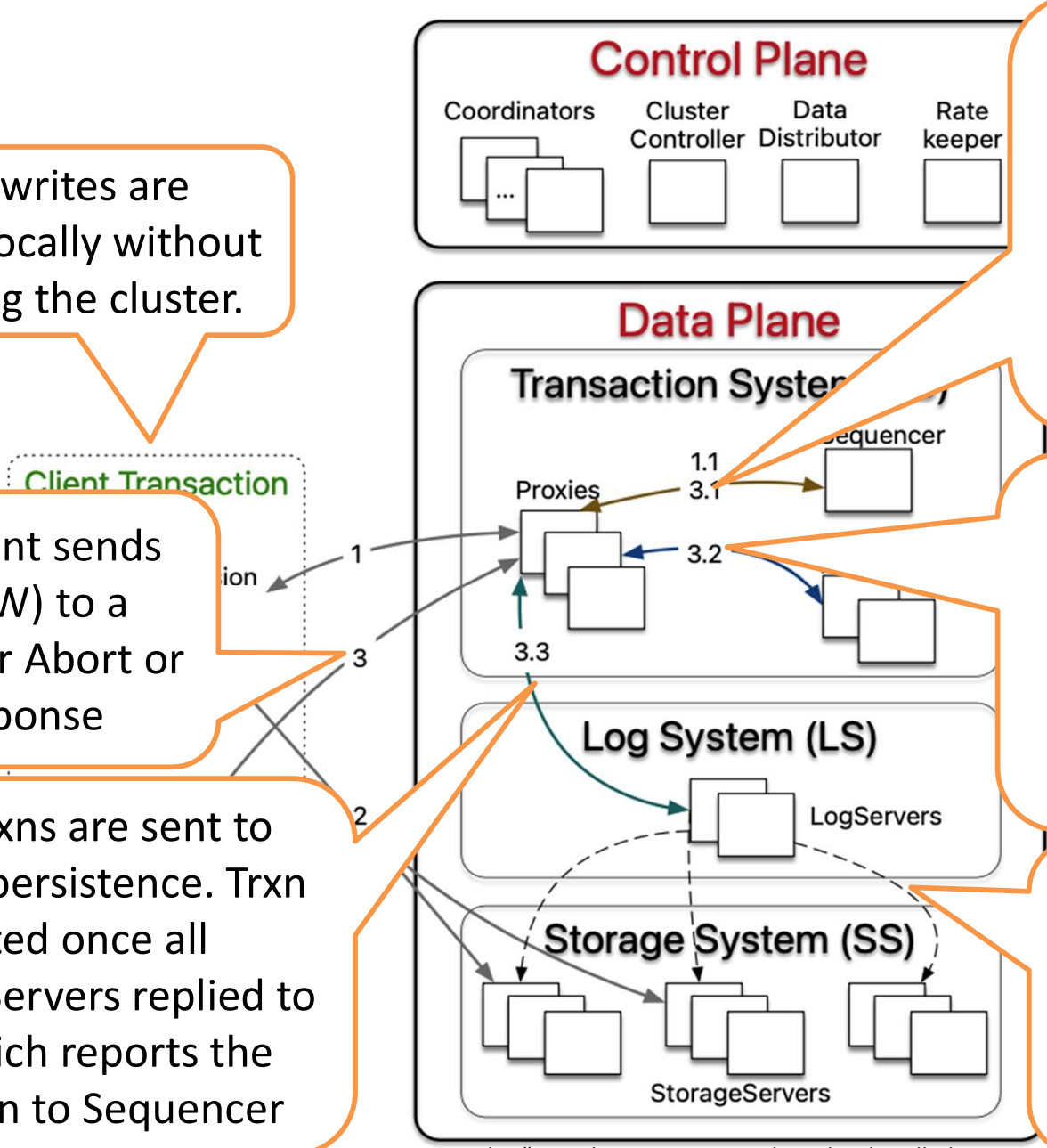
Figure Credit: "FoundationDB: A Distributed Unbundled Transactional Key Value Store, by Zhou J, et al. 2021"

Transaction Management: End-to-end Trxn Processing (Write/Commit)

Client writes are buffered locally without contacting the cluster.

At Commit, client sends trxn data(R&W) to a proxy & waits for Abort or Commit response

Committed trxns are sent to LogServers for persistence. Trxn is committed once all designated LogServers replied to the Proxy, which reports the commit version to Sequencer



Proxy asks Sequencer for commit version (no less than prev. issued trxn read/commit versions), advance its version & send trxn info to range-partitioned Resolver

Resolver: implements optimistic con. ctrl. by checking for R/W conflicts. Trxn proceeds if no conflict. Marked as aborted, otherwise

StorageServers pull mutation log from LogServers and apply committed updates to disk

Figure Credit: "FoundationDB: A Distributed Unbundled Transactional Key Value Store, by Zhou J, et al. 2021"

Participation Question

How is Multi-version Concurrency Control different from lock-based (2PL) concurrency control?

What is the difference between Optimistic Concurrency Control and Two-Phase Locking Protocol?

Read-Only and Snapshot Reads in FDB

- FDB supports read-only transactions and snapshot reads
- Read-only trxns in FDB are serializable (happens at the read version) and Performant (MVCC)
- Client can commit these trxns locally without contacting DB
- Snapshot reads relax the isolation property of a trxn by reducing conflicts
  - i.e., concurrent writes will not conflict with snapshot reads