FoundationDB: A Distributed Unbundled Transactional Key Value Store

March 11, 2025

1 Introduction

FDB can tolerate f failures with only $f + 1$ (rather than $2f + 1$) replicas.

2 Design

2.1 Design Principles

- **Divide-and-Conquer/Separation of concerns**. FDB decouples the transaction management system (write path) from the distributed storage (read path) and scales them independently.
- **Make failure a common case**. Instead of fixing all possible failure scenarios, the transaction system proactively shuts down when it detects a failure. As a result, all failure handling is reduced to a single recovery operation, which becomes a common and well-tested code path. Such error handling strategy is desirable as long as the recovery is quick, and pays dividends by simplifying the normal transaction processing.
- **Fail fast and recover fast**. To improve availability, FDB strives to minimize Mean-Time-To-Recovery (MTTR), which includes the time to detect a failure, proactively shut down the transaction management system, and recover.
- **Simulation testing**. FDB relies on a randomized, deterministic simulation framework for testing the correctness of its distributed database.

2.2 System Interface

- `get()`
- `set()`
- `gerRange()`
- `clear()` deletes all kv pairs within a range or starting with a certain key prefix

An FDB transaction observes and modifies a snapshot of the database at a certain version and changes are applied to the underlying database only when the transaction commits. A transaction's writes (i.e., `set()` and `clear()` calls) are buffered by the FDB client until the final `commit()` call, and read-your-write semantics are preserved by combining results from database look-ups with uncommitted writes of the transaction.

Key and value sizes are limited to 10 KB and 100 KB respectively for better performance. Transaction size is limited to 10 MB, including the size of all written keys and values as well as the size of all keys in read or write conflict ranges that are explicitly specified.

2.3 Architecture

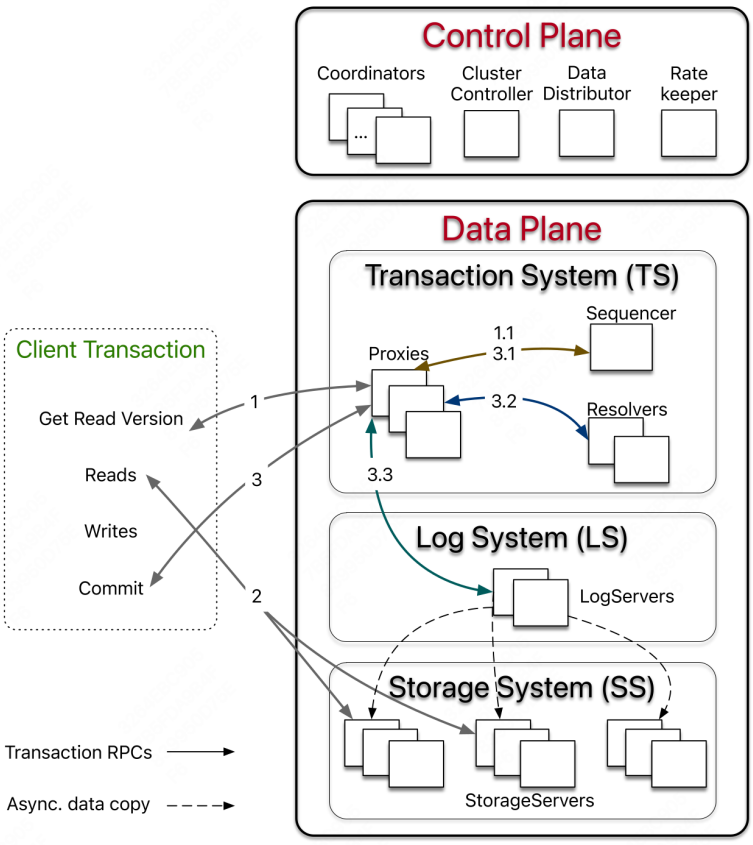

Figure 1: The architecture and the transaction processing of FDB

2.3.1 Control Plane

The control plane is responsible for persisting critical system metadata, i.e., the configuration of transaction systems, on Coordinators. These Coordinators form a disk Paxos group and select a singleton `ClusterController`. The `ClusterController` monitors all servers in the cluster and recruits three singleton processes, `Sequencer`, `DataDistributor`, and `Ratekeeper`, which are re-recruited if they fail or crash. The `Sequencer` assigns read and commit versions to transactions. The `DataDistributor` is responsible for monitoring failures and balancing data among `StorageServers`. `Ratekeeper` provides overload protection for the cluster.

2.3.2 Data Plane

FDB targets OLTP workloads that are read-mostly, read and write a small set of keys, have low contention, and require scalability. FDB chooses an unbundled architecture: a distributed transaction management system (TS) performs in-memory transaction processing, a log system (LS) stores Write-Ahead-Log (WAL) for TS, and a separate distributed storage system (SS) is used for storing data and servicing reads. The TS provides transaction processing and consists of a `Sequencer`, `Proxies`, and `Resolvers`, all of which are stateless processes. The LS contains a set of `LogServers` and the SS has a number of `StorageServers`.

- The `Sequencer` assigns a read version and a commit version to each transaction and, for historical reasons, also recruits `Proxies`, `Resolvers`, and `LogServers`.
- `Proxies` offer MVCC read versions to clients and orchestrate transaction commits.
- `Resolvers` check for conflicts between transactions.
- `LogServers` act as replicated, sharded, distributed persistent queues, where each queue stores WAL data for a `StorageServer`.

The SS consists of a number of `StorageServers` for serving client reads, where each `StorageServer` stores a set of data shards, i.e., contiguous key ranges. `StorageServers` are the majority of processes in the system, and together they form a distributed B-tree. Currently, the storage engine on each `StorageServer` is a modified version of SQLite, with enhancements that make range clears faster, defer deletion to a background task, and add support for asynchronous programming.

Why SQLite? Why not RocksDB? Two fold:

1. According to the thread: They do not work with the simulation testing framework as they require thread pools because they do synchronous IO. The simulation testing framework requires that every part of the database be able to run in a single thread.
2. FDB's scenario has more range reads. B+ tree outperforms LSM tree.

But in FDB 8.0, rocksdb will be supported. (Guess c++20's coroutine makes this work?)

2.3.3 Read-Write Separation and Scaling

FDB's design is decoupled; processes are assigned different roles (e.g., `Coordinators`, `StorageServers`, `Sequencer`), and the database scales by expanding the number of processes for each role. This separates the scaling of client reads from client writes (i.e., transaction commits).

- Because clients directly issue reads to sharded `StorageServers`, reads scale linearly with the number of `StorageServers`.
- Writes are scaled by adding more processes to `Proxies`, `Resolvers`, and `LogServers` in TS and LS. For this reason, MVCC data is stored in the SS.
- The singletons (e.g., `ClusterController` and `Sequencer`) and `Coordinators` on the control plane are not performance bottlenecks, because they only perform limited metadata operations.

2.3.4 Bootstrapping

FDB has no external dependency on other services. All user data and most of the system metadata (keys that start with `0xFF` prefix) are stored in `StorageServers`. The metadata about `StorageServers` is persisted in `LogServers`, and the configuration of LS (i.e., information about `LogServers`) is stored in all `Coordinators`.

1. Using Coordinators as a disk Paxos group, servers attempt to become the `ClusterController` if one does not exist.
2. The newly elected `ClusterController` recruits a new `Sequencer`
3. The `sequencer` reads the configuration of old LS stored in `Coordinators` and spawns a new TS and LS.
4. From the old LS, `Proxies` recover system metadata, including information about all `StorageServers`.
5. The `Sequencer` waits until the new TS finishes recovery, and then writes the new LS configuration to all `Coordinators`.

At this time, the new transaction system becomes ready to accept client transactions.

2.3.5 Reconfiguration

Whenever there is a failure in the TS or LS, or a database configuration change, a reconfiguration process brings the transaction management system to a new configuration, i.e., a clean state. Specifically, the `Sequencer` process monitors the health of `Proxies`, `Resolvers`, and `LogServers`. If any one of the monitored processes fails or the database configuration changes, the `Sequencer` process terminates. The `ClusterController` will detect the `Sequencer` failure event, then recruit a new Sequencer, which follows the above bootstrapping process to spawn the new TS and LS instance. In this way, transaction processing is divided into epochs, where each epoch represents a generation of the transaction management system with its unique Sequencer process.

2.4 Transaction Management

2.4.1 End-to-end Transaction Processing

1. A client transaction starts by contacting one of the `Proxies` to obtain a read version (i.e., a timestamp).

2. The `Proxy` then asks the `Sequencer` for a read version that is guaranteed to be no less than any previously issued transaction commit version, and this read version is sent back to the client.

3. Then the client may issue multiple reads to `StorageServers` and obtain values at that specific read version.

Client writes are buffered locally without contacting the cluster. At commit time, the client sends the transaction data, including the read and write sets (i.e., key ranges), to one of the `Proxies` and waits for a commit or abort response from the `Proxy`. If the transaction cannot commit, the client may choose to restart the transaction from the beginning again.

A `Proxy` commits a client transaction in three steps.

1. The `Proxy` contacts the `Sequencer` to obtain a commit version that is larger than any existing read versions or commit versions. The `Sequencer` chooses the commit version by advancing it at a rate of one million versions per second.

2. the `Proxy` sends the transaction information to range-partitioned `Resolvers`, which implement FDB's optimistic concurrency control by checking

for read-write conflicts. If all Resolvers return with no conflict, the transaction can proceed to the final commit stage. Otherwise, the Proxy marks the transaction as aborted.

3. committed transactions are sent to a set of `LogServers` for persistence. A transaction is considered committed after all designated `LogServers` have replied to the `Proxy`, which reports the committed version to the `Sequencer` (to ensure that later transactions' read versions are after this commit) and then replies to the client. At the same time, `StorageServers` continuously pull mutation logs from `LogServers` and apply committed updates to disks.

In addition to the above read-write transactions, FDB also supports read-only transactions and snapshot reads. A read-only transaction in FDB is both serializable (happens at the read version) and performant (thanks to the MVCC), and the client can commit these transactions locally without contacting the database. This is particularly important because the majority of transactions are read-only. Snapshot reads in FDB selectively relax the isolation property of a transaction by reducing conflicts, i.e., concurrent writes will not conflict with snapshot reads.

2.4.2 Support Strict Serializability

FDB implements Serializable Snapshot Isolation (SSI) by combining OCC with MVCC. Recall that a transaction T_x gets both its read version and commit version from `Sequencer`, where the read version is guaranteed to be no less than any committed version when T_x starts and the commit version is larger than any existing read or commit versions.

This commit version defines a serial history for transactions and serves as Log Sequence Number (LSN). Because T_x observes the results of all previous committed transactions, FDB achieves strict serializability. To ensure there is no gaps between LSNs, the `Sequencer` returns the previous commit version (i.e., previous LSN) with commit version.

A `Proxy` sends both LSN and previous LSN to `Resolvers` and `LogServers` so that they can serially process transactions in the order of LSNs. Similarly, `StorageServers` pull log data from `LogServers` in increasing LSNs as well.

Algorithm 1: Check conflicts for transaction T_x.

Require: *lastCommit*: a map of key range → last commit version

```
1 for each range ∈ R_r do
2     ranges = lastCommit.intersect(range)
3     for each r ∈ ranges do
4         if lastCommit[r] > T_x.readVersion then
5             return abort;
  // commit path
6 for each range ∈ R_w do
7     lastCommit[range] = T_x.commitVersion;
8 return commit;
```

Algorithm 2.4.2 illustrates the lock-free conflict detection algorithm on `Resolvers`. Specifically, each `Resolver` maintains a history of $lastCommit$ recently modified key ranges by committed transactions, and their corresponding commit versions. The commit request for T_x comprises two sets: a set of modified key ranges R_w, and a set of read key ranges R_r, where a single key is converted to a single key range. The read set is checked against the modified key ranges of concurrent committed transactions (line 1—5), which prevents phantom reads. If there are no read-write conflicts, `Resolvers` admit the transaction for commit and update the list of modified key ranges with the write set (line 6—7). For snapshot reads, they are not included in the set R_r . In practice, $lastCommit$ is represented as a version-augmented probabilistic SkipList.

The entire key space is divided among `Resolvers` so that the above read-write conflict detection algorithm may be performed in parallel. A transaction can commit only when all `Resolvers` admit the transaction. Otherwise, the transaction is aborted.

It is possible that an aborted transaction is admitted by a subset of `Resolvers`, and they have already updated their history of $lastCommit$, which may cause other transactions to conflict (i.e., a false positive). In practice, this has not been an issue for our production workloads:

- transactions' key ranges usually fall into one Resolver.
- Additionally, because the modified keys expire after the MVCC window, the false positives are limited to only happen within the short MVCC window time (i.e., 5 seconds).

- the key ranges of `Resolvers` are dynamically adjusted to balance their loads.

The OCC design of FDB avoids the complicated logic of acquiring and releasing (logical) locks, which greatly simplifies interactions between the TS and the SS. The price paid for this simplification is to keep the recent commit history in `Resolvers`. Another drawback is not guaranteeing that transactions will commit, a challenge for OCC. Because of the nature of our multi-tenant production workload, the transaction conflict rate is very low (less than 1%) and OCC works well. If a conflict happens, the client can simply restart the transaction.

2.4.3 Logging Protocol

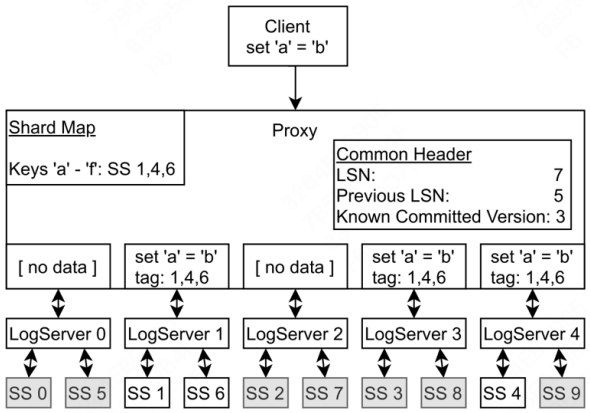

Figure 2: Proxy writes a client mutation to `LogServers` after sequencing and resolution. Later, the mutation will be asynchronously replicated to `StorageServers`.

After a `Proxy` decides to commit a transaction, the log message is broadcast to all `LogServers`. As illustrated in Figure 2.4.3, the Proxy first consults its in-memory shard map to determine the `StorageServers` responsible for the modified key range. Then the Proxy attaches StorageServer tags 1, 4, and 6 to the mutation, where each tag has a preferred LogServer for storage.

In this example, tags 1 and 6 have the same preferred `LogServer`. Note the mutation is only sent to the preferred LogServers (1 and 4) and an additional LogServer 3 to meet the replication requirements. All other `LogServers` receive an empty message body. The log message header includes both LSN and the previous LSN obtained from the Sequencer, as well as the known

committed version (KCV) of this Proxy. LogServers reply to the Proxy once the log data is made durable, and the Proxy updates its KCV to the LSN if all replica LogServers have replied and this LSN is larger than the current KCV.

Shipping the redo log from the LS to the SS is not a part of the commit path and is performed in the background. In FDB, StorageServers aggressively fetch redo logs from LogServers before they are durable on the LS, allowing very low latency for serving multi-version reads.

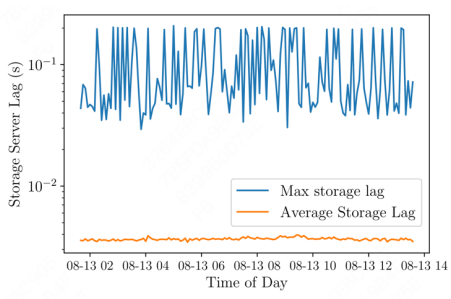

Figure 2: The lag from StorageServers to LogServers

Figure 2 shows the time lag between StorageServers and LogServers in one of our production clusters for a 12-hour period, where the 99.9 percentile of the average and maximum delay is 3.96 ms and 208.6 ms, respectively. Because this lag is small, when client read requests reach StorageServers, the requested version (i.e., the latest committed data) is usually already available. If due to a small delay the data is not available to read at a StorageServer replica, the client either waits for the data to become available or issues a second request to another replica. If both reads timed out, the client gets a retryable error to restart the transaction.

Because the log data is already durable on LogServers, StorageServers can buffer updates in memory and only persist batches of data to disks with a longer delay, thus improving I/O efficiency by coalescing the updates. Aggressively pulling redo logs from LogServers means that semi-committed updates, i.e., operations in transactions that are aborted during recovery (e.g., due to LogServer failure), need to be rolled back.

2.4.4 Transaction System Recovery

In FDB, `StorageServers` always pull logs from `LogServers` and apply them in the background, which essentially decouples redo log processing from the recovery. The recovery process starts by detecting a failure, recruits

a new transaction system, and ends when old LogServers are no longer needed. The new transaction system can even accept transactions before all the data on old LogServers is processed, because the recovery only needs to find out the end of redo log and re-applying the log is performed asynchronously by StorageServers.

For each epoch, the Sequencer executes recovery in several steps.

1. the Sequencer reads the previous transaction system states (i.e. configurations of the transaction system) from Coordinators and locks the coordinated states to prevent another Sequencer process from recovering at the same time.

2. the Sequencer recovers previous transaction system states, including the information about all older LogServers, stops these LogServers from accepting transactions, and recruits a new set of Proxies, Resolvers, and LogServers.

3. After previous LogServers are stopped and a new transaction system is recruited, the Sequencer then writes the coordinated states with current transaction system information.

4. Finally, the Sequencer accepts new transaction commits.

Because Proxies and Resolvers are stateless, their recoveries have no extra work. In contrast, LogServers save the logs of committed transactions, and we need to ensure a ll previously committed transactions are durable and retrievable by StorageServers. That is, for any transactions that the Proxies may have sent back a commit response, their logs are persisted in multiple LogServers satisfying the configured replication degree.

The essence of the recovery of old LogServers is to determine the end of redo log, i.e., a Recovery Version (RV). Rolling back undo log is essentially discarding any data after RV in the old LogServers and StorageServers. Figure 2.4.4 illustrates how RV is determined by the Sequencer.

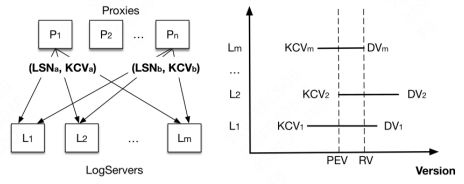

Figure 4: An illustration of RV and PEV. On the left, a Proxy sends redo logs to LogServers with a KCV and the LSN, and LogServers keep the maximum KCV received. On the right, recovery uses the maximum of KCVs and the minimum of DVs on a set of LogServers as PEV and RV, respectively.

Recall that a `Proxy` request to `LogServers` piggybacks its KCV, the maximum LSN that this Proxy has committed. Each LogServer keeps the maximum KCV received and a Durable Version (DV), which is the maximum persisted LSN. During a recovery, the `Sequencer` attempts to stop all m old `LogServers`, where each response contains the DV and KCV on that `LogServer`. Assume the replication degree for LogServers is k. Once the `Sequencer` has received more than $m - k$ replies , the `Sequencer` knows the previous epoch has committed transactions up to the maximum of all KCVs, which becomes the previous epoch's end version (PEV). All data before this version has been fully replicated.

For current epoch, its start version is PEV+1 and the `Sequencer` chooses the minimum of all DVs to be the RV. Logs in the range of $[PEV + 1, RV]$ are copied from previous epoch's `LogServers` to the current ones, for healing the replication degree in case of `LogServer` failures. The overhead of copying this range is very small because it only contains a few seconds' log data.

When `Sequencer` accepts new transactions, the first is a special recovery transaction that informs `StorageServers` the RV so that they can roll back any data larger than RV. The current FDB storage engine consists of an unversioned SQLite B-tree and in-memory multi-versioned redo log data. Only mutations leaving the MVCC window (i.e., committed data) are written to SQLite. The rollback is simply discarding in-memory multi-versioned data in `StorageServers`. Then `StorageServers` pull any data larger than version PEV from new `LogServers`.

2.5 Replication

FDB uses a combination of various replication strategies for differ ent data to tolerate f failures:

- *Metadata replication*. System metadata of the control plane is stored on Coordinators using Active Disk Paxos. As long as a quorum (i.e., majority) of Coordinators are live, this metadata can be recovered. Why choose this version of Paxos?

- *Log replication*. When a Proxy writes logs to LogServers, each sharded log record is synchronously replicated on $k = f + 1$ `LogServers`. Only when all k have replied with successful persistence can the Proxy send back the commit response to the client. Failure of a LogServer results in a transaction system recovery.

- *Storage replication*. Every shard, i.e., a key range, is asynchronously replicated to $k = f + 1$ StorageServers, which is called a **team**. A `StorageServer` usually hosts a number of shards so that its data is evenly distributed across many teams. A failure of a `StorageServer` triggers `DataDistributor` to move data from teams containing the failed process to other healthy teams

Why do FoundationDB need to consider this?
What if we want to split the storage server

Note the storage team abstraction is more sophisticated than the Copyset policy. Copyset reduces the chance of data loss during simultaneous process failures by assigning shards to a limited number of possible k-process groups. Otherwise, any k-process failure can cause a higher probability of data loss. In our deployment, teams need to consider multiple dimensions: each replica group needs to satisfy several constraints at the same time. For instance, a cluster can have a number of hosts and each host runs multiple processes. In this case, a failure can happen at the host level, affecting many processes. Thus, a replica group cannot place two processes on the same host. More generally, the placement needs to ensure at most one process in a replica group can be placed in a fault domain, e.g., racks or availability zones in a cloud environment.

To solve the above problem, we designed a hierarchical replication policy to reduce the chance of data loss during simultaneous failures. Specifically, we construct the replica set at both host and process levels and ensure that each process group belongs to a host group that satisfies the fault domain requirement. This policy has the benefits that data loss can only happen when all hosts in a selected host group fail simultaneously; that is, when we experience concurrent failures in multiple fault domains. Otherwise, each team is guaranteed to have at least one process live and there is no data loss if any one of the fault domains remains available..

2.6 Other Optimizations

- **Transaction batching**

- **Atomic operations**. FDB supports atomic operations such as atomic add, bitwise "and" operation, compare-and-clear, and set-versionstamp.

3 Geo-replication and failover

The main challenge of providing high availability during region failures is the trade-off of performance and consistency. Synchronous cross-region replication provides strong consistency, but pays the cost of high latency. Conversely, asynchronous replication reduces latency by only persisting in the primary region, but may lose data when performing a region failover. FDB can be configured to perform either synchronous or asynchronous cross-region replication. However, there is a third possibility that leverages multiple availability zones within the same region, and provides a high level of failure independence, notwithstanding the unlikely event of a complete region outage.

Our design

1. always avoids cross-region write latencies, as for asynchronous replication
2. provides full transaction durability, like synchronous replication, so long as there is no simultaneous failure of multiple availability zones in a region,
3. can do rapid and completely automatic failover between regions,
4. can be manually failed-over with the same guarantees as asynchronous replication (providing A, C, and I of ACID but potentially exhibiting a Durability failure) in the unlikely case of a simultaneous total region failure
5. only requires full replicas of the database in the primary and secondary regions' main availability zones, not multiple replicas per region. The rest of this section is dedicated to this design.

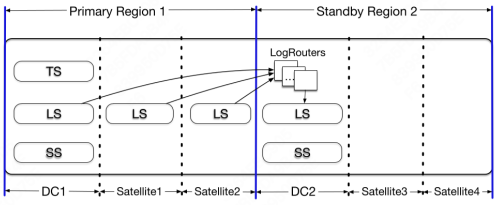

Figure 5: A two-region replication setup for an FDB cluster. Both regions have a data center and two satellite sites.

Figure 3 illustrates the layout of a two-region replication of a cluster. Both regions have a data center (DC) as well as one or more satellite sites. Satellites are located in close proximity to the DC (in the same region) but are failure independent. The resource requirements from satellites are insignificant as they only need to store log replicas (i.e., a suffix of the redo logs), while data centers host LS, SS, and (when primary) the TS. Control plane replicas (i.e., coordinators) are deployed across three or more failure domains (in some deployments utilizing an additional region), usually with at least 9 replicas. Relying on majority quorums allows the control plane to tolerate one site (data center/satellite) failure and an additional replica failure.

A typical deployment configuration is illustrated in Figure 3, depicting two regions with a data center and two satellites in each region.

- One of the data centers (DC1), configured with a higher priority compared to DC2, is designated as the primary (its region is denoted as the primary region, accordingly) and contains the full TS, LS, and SS
- DC2 in the secondary region has replicas of data with its own LS and SS
- Reads can be served from storage replicas at both primary and secondary data centers (consistent reads do require obtaining a read version from the primary data center).
- All client writes are forwarded to the primary region and processed by Proxies in DC1, then synchronously persisted onto LogServers in DC1 and one or both satellite sites in the primary region (depending on the configuration), avoiding the cross-region WAN latency.

  The updates are then asynchronously replicated to DC2, where they are stored on multiple LS servers and eventually spread out to multiple StorageServers.
- LogRouters implement a special type of FDB role that facilitates cross-region data transfer. They were created to avoid redundant cross-region transfers of the same information. Instead, LogRouters transfer each log entry across WAN only once, and then deliver it to all relevant LS servers locally in DC2.
- The cluster automatically fails-over to the secondary region if the primary data center becomes unavailable. Satellite failures could, in some cases, also result in a fail-over, but this decision is currently manual.

- When the fail-over happens, DC2 might not have a suffix of the log, which it proceeds to recover from the remaining log server in the primary region.

Next, we discuss several alternative satellite configurations which provide different levels of fault-tolerance. Satellite configuration can be specified per region. Each satellite is given a static priority, which is considered relatively to other satellites in the same region. FDB is usually configured to store multiple log replicas at each location. Three main alternatives are supported:

1. synchronously storing updates on all log replicas at the satellite with the highest priority in the region. In this case, if the satellite fails, another satellite with the next priority is recruited for the task

2. synchronously storing updates on all replicas of two satellites with the highest priorities in the region. In this case, if a satellite fails, it can be similarly replaced with a different satellite of lower priority, or, if none available, fall back to option (1) of using a single satellite. In either case, the secondary region isn't impacted, as it can continue to pull updates from remaining LogServers in the primary region.

3. Similar to option (2) but FDB only waits for one of the two satellites to make the mutations durable before considering a commit successful.

In all cases, if no satellites are available, only the LogServers in DC1 are used. With option 1 and 3, a single site (data center or satellite) failure can be tolerated, in addition to one or more LogServer failures (since the remaining locations have multiple log replicas). With option 2, two site failures in addition to one or more LogServer failures can be tolerated. In options 1 and 2, however, commit latency is sensitive to the tail network latencies between the primary data center and its satellites, which means that option 3 is usually faster. The choice ultimately depends on the number of available satellite locations, their connectivity to the data center and the desired level of fault tolerance and availability.

When DC1 in the primary region suddenly becomes unavailable, the cluster (with the help of Coordinators) detects the failure and starts a new transaction management system in DC2. New LogServers are recruited from satellites in the secondary region, in accordance with the region's replication policy. During recovery, LogRouters in DC2 may need to fetch the last few seconds' data from primary satellites, which, due to the asynchronous

replication, may not have made it to DC2 prior to the failover. After the recovery, if the failures in Region 1 are healed and its replication policy can again be met, the cluster will automatically fail-back to have DC1 as the primary data center due to its higher priority. Alternatively, a different secondary region can be recruited.

4 Simulation Testing

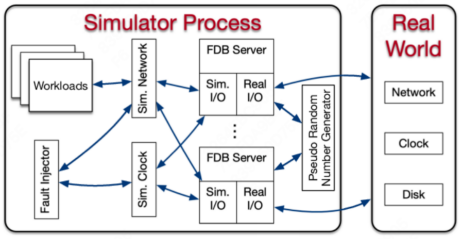

Figure 6: The FDB deterministic simulator.

- **Deterministic simulator**. FDB was built from the ground up to make this testing approach possible. All database code is deterministic; accordingly multithreaded concurrency is avoided (instead, one database node is deployed per core). Figure 4 illustrates the simulator process of FDB, where all sources of nondeterminism and communication are abstracted, including network, disk, time, and pseudo random number generator.

  FDB is written in Flow, a novel syntactic extension to C++ adding async/await-like concurrency primitives. Flow provides the Actor programming model that abstracts various actions of the FDB server process into a number of actors that are scheduled by the Flow runtime library.

  The simulator process is able to spawn multiple FDB servers that communicate with each other through a simulated network in a single discrete-event simulation. The production implementation is a simple shim to the relevant system calls.

  The simulator runs multiple workloads (also written in Flow) that communicate with simulated FDB servers through the simulated network. These workloads include fault injection instructions, mock applications, database configuration changes, and direct internal database

functionality invocations. Workloads are composable to exercise various features and are reused to construct comprehensive test cases.

5 Problems

6 References