Overview

Upgrading FoundationDB can be a challenging process. FDB has an internal wire protocol for communication between server processes that is not guaranteed to be stable across versions. Patch releases for the same minor version are protocol-compatible, but different minor versions are not protocol-compatible. This means that when you are doing a minor version upgrade, you need to upgrade all of the processes at once, because the old and new processes will be unable to communicate with each other. `fdbcli` uses the same wire protocol, so you will need to use a version of `fdbcli` that matches the version of FDB that is running at the time.

Additionally, clients must have a client library that is protocol-compatible with the database in order to make a connection. To avoid client outages during upgrades, you must install both the old and new client libraries, using FDB's multi-version library feature to load both library versions at the same time.

Despite these challenges, it is possible to build a safe, zero-downtime upgrade process for FoundationDB. This document will describe that process, using an upgrade from 6.1.12 to 6.2.8 as an example. This process assumes that you are running `fdbserver` through `fdbmonitor`, and that you have the capability to install new binaries and new config files into the environment where your processes are running.

Upgrade Process

The high-level upgrade process is:

1. Install the new fdbserver binaries alongside the old binaries, with each binary in a path that contains its version. For instance, you might have the old binary at `/usr/bin/fdb/6.1.12/fdbserver`, and the new binary at `/usr/bin/fdb/6.2.8/fdbserver`.
2. Update the monitor conf to change the fdbserver path to `/usr/bin/fdb/6.2.8/fdbserver`.
3. Using the CLI at version 6.1.12, run the command `kill; kill all; status`.
4. Using the CLI at version 6.2.8, connect to the database and confirm that the cluster is healthy.

Handling Client Upgrades

To ensure that clients remain connected during the upgrade, you should use the multi-version client. The recommended process for managing client libraries is:

1. Install version 6.2.8 in a special folder for multi-version clients. For instance, `/var/lib/fdb-multiversion/libfdbc_6.2.8.so`. You should include the version in the filename for the multiversion libraries to make sure you can support as many as you need to have, and to help with debugging.

2. Set the `FDB_NETWORK_OPTION_EXTERNAL_CLIENT_DIRECTORY` environment variable to `/var/lib/fdb-multiversion`.
3. Bounce the client application.
4. Use the JSON status from the database to confirm that all clients have compatible protocol versions. You can get this client information in `cluster.clients.supported_versions`. That will hold a list of every version supported by any connected client of the database. Each version entry will hold the client version, the protocol version, and the list of clients that are using that client version. You can get the protocol version for the new version of FDB by running `/usr/bin/fdb/6.2.8/fdbcli --version`. To confirm that the clients are ready for the upgrade, check that for every client address that exists for any client version, there exists an entry under a client version whose protocol version matches the new version.
5. Run the server upgrade steps above.
6. Once the database is running on the new version, you can update the clients to use `6.2.8` as the main client library version, and remove any older client libraries that you no longer need.

Steps 1 through 3 can be done at any point before the upgrade of the server. You may want to have your client applications include new versions of the FDB client library as part of there normal build and deployment process, so that you can decouple the upgrades of the clients and the servers. It is generally safe to have clients use multiple client libraries, and if you encounter any issues with that it may be easier to debug them as part of the normal process for updating the client application.

Upgrading fdbmonitor

The upgrade process above does not restart fdbmonitor, so it will continue running at the old version. This is generally not a problem, since fdbmonitor does not change with every release, but you may want to get it running on the new version for the sake of consistency in your configuration. Once you have the database running at the new version, you can upgrade fdbmonitor as a follow-on task. You should note that restarting fdbmonitor will also restart fdbserver, and depending on how you are upgrading fdbmonitor it may take longer for the processes to come back up. You may need to do a rolling bounce of your fdbmonitor processes to make sure that you maintain availability.

Other Binaries

The fdbbackup and fdbdr binaries also must be protocol-compatible with the running version of the database. The process for upgrading those binaries will depend on your infrastructure and your orchestration tooling. You should be able to run and upgrade those processes through the same process you would use for any other application. This will create a gap between when the database is upgraded and when the backup and DR binaries are upgraded. This will produce a temporary lag in backup and DR. Once all of the components are running on the same version, the backup and DR will catch up.

Additional Notes

To ensure that fdbmonitor does not kill the old processes too soon, you should set `kill_on_configuration_change=false` in your monitor conf file.

If fdbserver processes restart for organic reasons between steps 2 and 3 in the upgrade, they will not be able to connect to the rest of the cluster. If this happens to a single process, then you should be able to kill the remaining processes through the CLI, and the process that restarted early will be able to connect. If this happens to enough processes, it can take the database unavailable, and you won't be able to kill processes through the CLI. If this happens, you can restart all of the fdbmonitor processes to bring everything up on the new version. We recommend minimizing the gap between steps 2 and 3 to help mitigate this risk.

This process of installing new binaries while the process is still running can present additional challenges in containerized environment, but it is still possible, as long as the deployment system allows making changes to running containers. While this can violate goals of container immutability, it is only necessary during the upgrade itself. Once the upgrade is complete, you can roll out the new version of the container image through a rolling bounce, through the fdbmonitor upgrade process described above. We have implemented a process like this in our Kubernetes Operator. ## Components in FoundationDB

The FoundationDB architecture chooses a decoupled design, where processes are assigned different heterogeneous roles (e.g., Coordinators, Storage Servers, Master). Scaling the database is achieved by horizontally expanding the number of processes for separate roles:

Coordinators

All clients and servers connect to a FoundationDB cluster with a cluster file, which contains the IP:PORT of the coordinators. Both the clients and servers use the coordinators to connect with the cluster controller. The servers will attempt to become the cluster controller if one does not exist, and register with the cluster controller once one has been elected. Clients use the cluster controller to keep an up-to-date list of proxies.

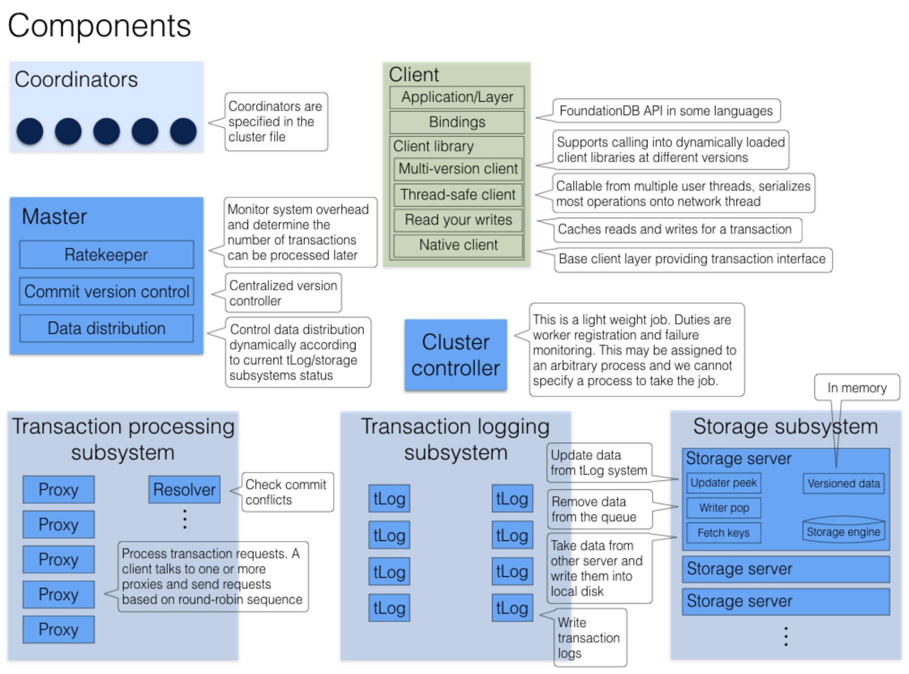
Cluster Controller

The cluster controller is a singleton elected by a majority of coordinators. It is the entry point for all processes in the cluster. It is responsible for determining when a process has failed, telling processes which roles they should become, and passing system information between all of the processes.

Master

The master is responsible for coordinating the transition of the write sub-system from one generation to the next. The write sub-system includes the master, proxies, resolvers, and transaction logs. The three roles are treated as a unit, and if any of them fail, we will recruit a replacement for all three roles. The master provides the commit versions for batches of the mutations to the proxies.

Historically, Ratekeeper and Data Distributor are coupled with Master on the same process. Since 6.2, both have become a singleton in the cluster. The life time is no longer tied with Master.



Components
Coordinators
Coordinators are specified in the cluster file
Client
Application/Layer
Bindings
Client library
Multi-version client
Thread-safe client
Read your writes
Native client
FoundationDB API in some languages
Supports calling into dynamically loaded client libraries at different versions
Callable from multiple user threads, serializes most operations onto network thread
Caches reads and writes for a transaction
Base client layer providing transaction interface
Master
Ratekeeper
Commit version control
Data distribution
Monitor system overhead and determine the number of transactions can be processed later
Centralized version controller
Control data distribution dynamically according to current tLog/storage subsystems status
Cluster controller
This is a light weight job. Duties are worker registration and failure monitoring. This may be assigned to an arbitrary process and we cannot specify a process to take the job.
Transaction processing subsystem
Proxy
Proxy
Proxy
Proxy
Proxy
Resolver
Check commit conflicts
Process transaction requests. A client talks to one or more proxies and send requests based on round-robin sequence
Transaction logging subsystem
tLog
tLog
tLog
tLog
tLog
tLog
tLog
tLog
Update data from tLog system
Remove data from the queue
Take data from other server and write them into local disk
Write transaction logs
Storage subsystem
In memory
Storage server
Updater peek
Writer pop
Fetch keys
Versioned data
Storage engine
Storage server
Storage server

Proxies

The proxies are responsible for providing read versions, committing transactions, and tracking the storage servers responsible for each range of keys.

To provide a read version, a proxy will ask all other proxies to see the largest committed version at this point in time, while simultaneously checking that the transaction logs have not been stopped. Ratekeeper will artificially slow down the rate at which the proxy provides read versions.

Commits are accomplished by:

- Get a commit version from the master.
- Use the resolvers to determine if the transaction conflicts with previously committed transactions.
- Make the transaction durable on the transaction logs.

The key space starting with the `\xff` byte is reserved for system metadata. All mutations committed into this key space are distributed to all of the proxies through the resolvers. This metadata includes a mapping between key ranges and the storage servers which have the data for that range of keys. The proxies provides this information to clients on-demand. The clients cache this mapping; if they ask a storage server for a key it does not have, they will clear their cache and get a more up-to-date list of servers from the proxies.

Transaction Logs

The transaction logs make mutations durable to disk for fast commit latencies. The logs receive commits from the proxy in version order, and only respond to the proxy once the data has been written and fsync'ed to an append only mutation log on disk. Before the data is even written to disk we forward it to the storage servers responsible for that mutation. Once the storage servers have made the mutation durable, they pop it from the log. This generally happens roughly 6 seconds after the mutation was originally committed to the log. We only read from the log's disk when the process has been rebooted. If a storage server has failed, mutations bound for that storage server will build up on the logs. Once data distribution makes a different storage server responsible for all of the missing storage server's data we will discard the log data bound for the failed server.

Resolvers

The resolvers are responsible determining conflicts between transactions. A transaction conflicts if it reads a key that has been written between the transaction's read version and commit version. The resolver does this by holding the last 5 seconds of committed writes in memory, and comparing a new transaction's reads against this set of commits.

Storage Servers

The vast majority of processes in a cluster are storage servers. Storage servers are assigned ranges of key, and are responsible to storing all of the data for that range. They keep 5 seconds of mutations in memory, and an on disk copy of the data as of 5 second ago. Clients must read at a version within the last 5 seconds, or they will get a `transaction_too_old` error. The SSD storage engine stores the data in a B-tree based on SQLite. The memory storage engine store the data in memory with an append only log that is only read from disk if the process is rebooted. In the upcoming FoundationDB 7.0 release, the B-tree storage engine will be replaced with a brand new Redwood engine.

Data Distributor

Data distributor manages the lifetime of storage servers, decides which storage server is responsible for which data range, and ensures data is evenly distributed across all storage servers (SS). Data distributor as a singleton in the cluster is recruited and monitored by Cluster Controller. See internal documentation.

Ratekeeper

Ratekeeper monitors system load and slows down client transaction rate when the cluster is close to saturation by lowering the rate at which the proxy provides read versions. Ratekeeper as a singleton in the cluster is recruited and monitored by Cluster Controller.

Clients

A client links with specific language bindings (i.e., client libraries) in order to communicate with a FoundationDB cluster. The language bindings support loading multiple versions of C libraries, allowing the client communicates with older version of the FoundationDB clusters. Currently, C, Go, Python, Java, Ruby bindings are officially supported.

Transaction Processing

A database transaction in FoundationDB starts by a client contacting one of the Proxies to obtain a read version, which is guaranteed to be larger than any of commit version that client may know about (even through side channels outside the FoundationDB cluster). This is needed so that a client will see the result of previous commits that have happened.

Then the client may issue multiple reads to storage servers and obtain values at that specific read version. Client writes are kept in local memory without contacting the cluster. By default, reading a key that was written in the same transaction will return the newly written value.

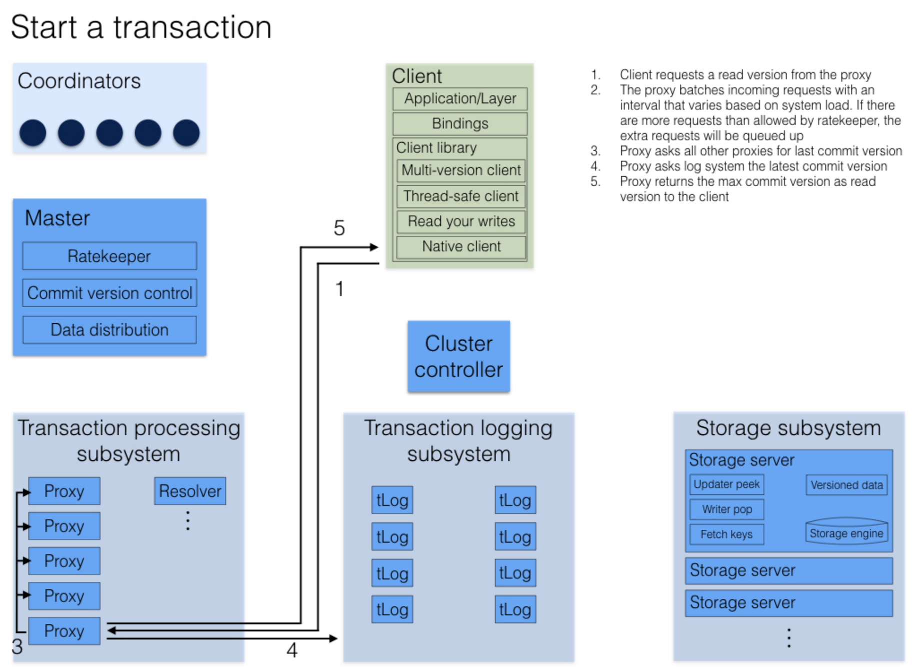
At commit time, the client sends the transaction data (all reads and writes) to one of the Proxies and waits for commit or abort response from the proxy.

If the transaction conflicts with another one and cannot commit, the client may choose to retry the transaction from the beginning again. If the transaction commits, the proxy also returns the commit version back to the client. Note this commit version is larger than the read version and is chosen by the master.

The FoundationDB architecture separates the scaling of client reads and writes (i.e., transaction commits). Because clients directly issue reads to sharded storage servers, reads scale linearly to the number of storage servers. Similarly, writes are scaled by adding more processes to Proxies, Resolvers, and Log Servers in the transaction system.

Determine Read Version

When a client requests a read version from a proxy, the proxy asks all other proxies for their last commit versions, and checks a set of transaction logs satisfying replication policy are live. Then the proxy returns the maximum commit version as the read version to the client.

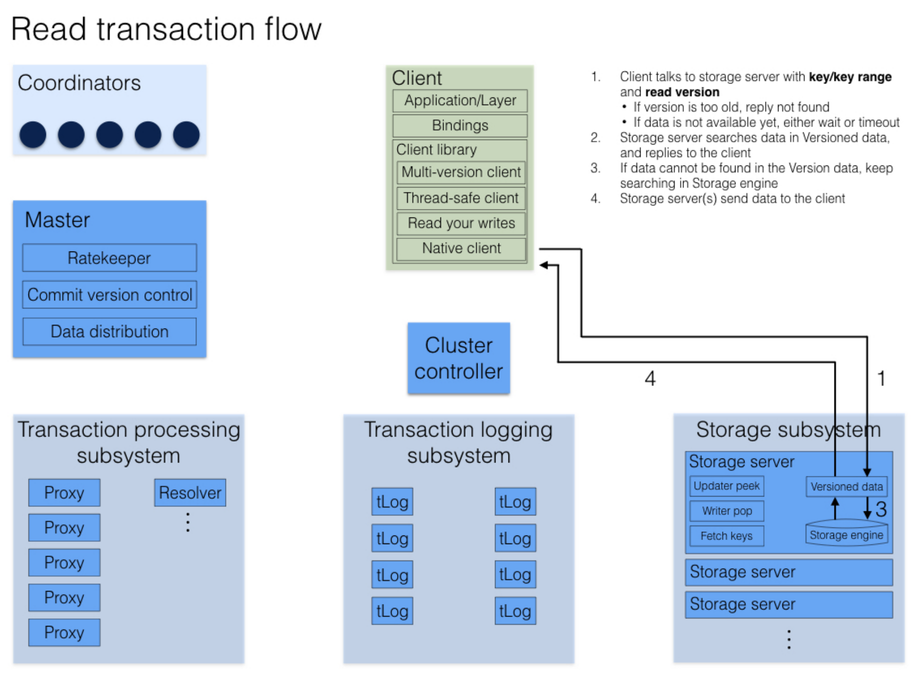


1. Client requests a read version from the proxy
2. The proxy batches incoming requests with an interval that varies based on system load. If there are more requests than allowed by ratekeeper, the extra requests will be queued up
3. Proxy asks all other proxies for last commit version
4. Proxy asks log system the latest commit version
5. Proxy returns the max commit version as read version to the client

The reason for the proxy to contact all other proxies for commit versions is to ensure the read version is larger than any previously committed version. Consider that if proxy `A` commits a transaction, and then the client asks proxy `B` for a read version. The read version from proxy `B` must be larger than the version committed by proxy `A`. The only way to get this information is by asking proxy `A` for its largest committed version.

The reason for checking a set of transaction logs satisfying replication policy are live is to ensure the proxy is not replaced with newer generation of proxies. This is because proxy is a stateless role recruited in each generation. If a recovery has happened and the old proxy is still live, this old proxy could still give out read versions. As a result, a read-only transaction may see stale results (a read-write transaction will be aborted). By checking a set of transaction logs satisfying replication policy are live, the proxy makes sure no recovery has happened, thus the read-only transaction sees the latest data.

Note that the client cannot simply ask the master for read versions. The master gives out versions to proxies to be committed, but the master does not know when the versions it gives out are durable on the transaction logs. Therefore it is not safe to do reads at the largest version the master has provided because that version might be rolled back in the event of a failure, so the client could end up reading data that was never committed. In order for the client to use versions from the master, the client needs to wait until all in-flight transaction-batches (a write version is used for a batch of transactions) to commit. This can take a long time and thus is inefficient. Another drawback of this approach is putting more work towards the master, because the master role can't be scaled. Even though giving out read-versions isn't very expensive, it still requires the master to get a transaction budget from the Ratekeeper, batches requests, and potentially maintains thousands of network connections from clients.


Read transaction flow

1. Client talks to storage server with **key/key range** and **read version**
   - If version is too old, reply not found
   - If data is not available yet, either wait or timeout
2. Storage server searches data in Versioned data, and replies to the client
3. If data cannot be found in the Version data, keep searching in Storage engine
4. Storage server(s) send data to the client

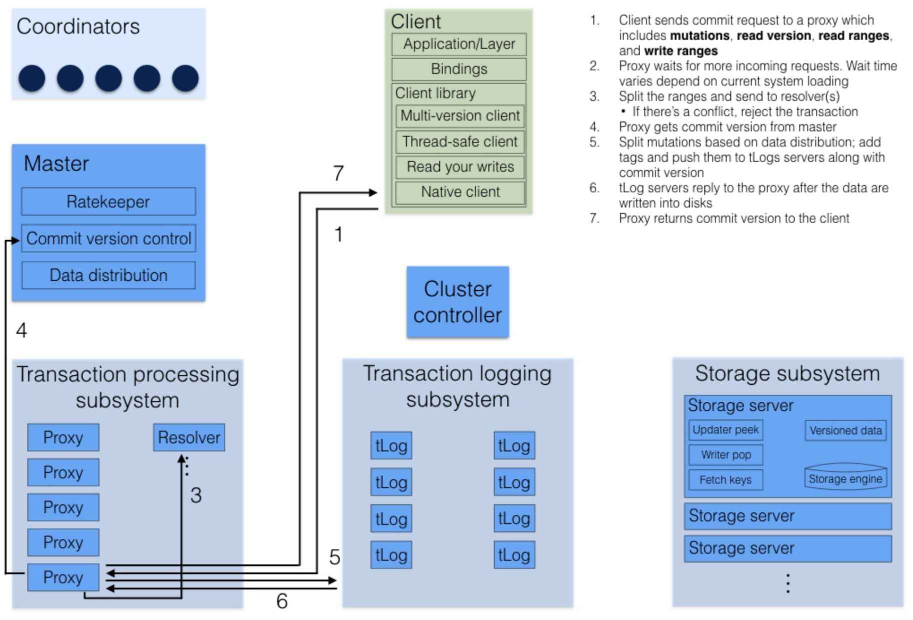
Transaction Commit

A client transaction commits in the following steps:

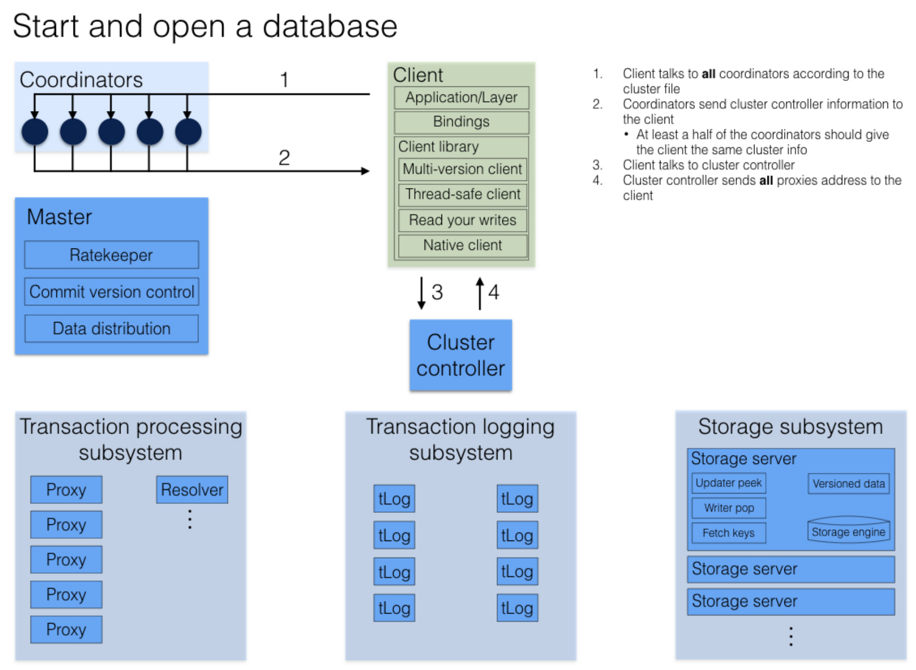
1. A client sends a transaction to a proxy.
2. The proxy asks the master for a commit version.
3. The master sends back a commit version that is higher than any commit version seen before.
4. The proxy sends the read and write conflict ranges to the resolver(s) with the commit version included.
5. The resolver responds back with whether the transaction has any conflicts with previous transactions by sorting transactions according to their commit versions and computing if such a serial execution order is conflict-free.
   - If there are conflicts, the proxy responds back to the client with a not_committed error.
   - If there are no conflicts, the proxy sends the mutations and commit version of this transaction to the transaction logs.
6. Once the mutations are durable on the logs, the proxy responds back success to the user.

Note the proxy sends each resolver their respective key ranges, if any one of the resolvers detects a conflict then the transaction is not committed. This has the flaw that if only one of the resolvers detects a conflict, the other resolver will still think the transaction has succeeded and may fail future transactions with overlapping write conflict ranges, even though these future transaction can commit. In practice, a well designed workload will only have a very small percentage of conflicts, so this amplification will not affect performance. Additionally, each transaction has a five seconds window. After five seconds, resolvers will remove the conflict ranges of old transactions, which also limits the chance of this type of false conflict.

Write transaction flow



1. Client sends commit request to a proxy which includes **mutations**, **read version**, **read ranges**, and **write ranges**
2. Proxy waits for more incoming requests. Wait time varies depend on current system loading
3. Split the ranges and send to resolver(s)
   - If there's a conflict, reject the transaction
4. Proxy gets commit version from master
5. Split mutations based on data distribution; add tags and push them to tLogs servers along with commit version
6. tLog servers reply to the proxy after the data are written into disks
7. Proxy returns commit version to the client

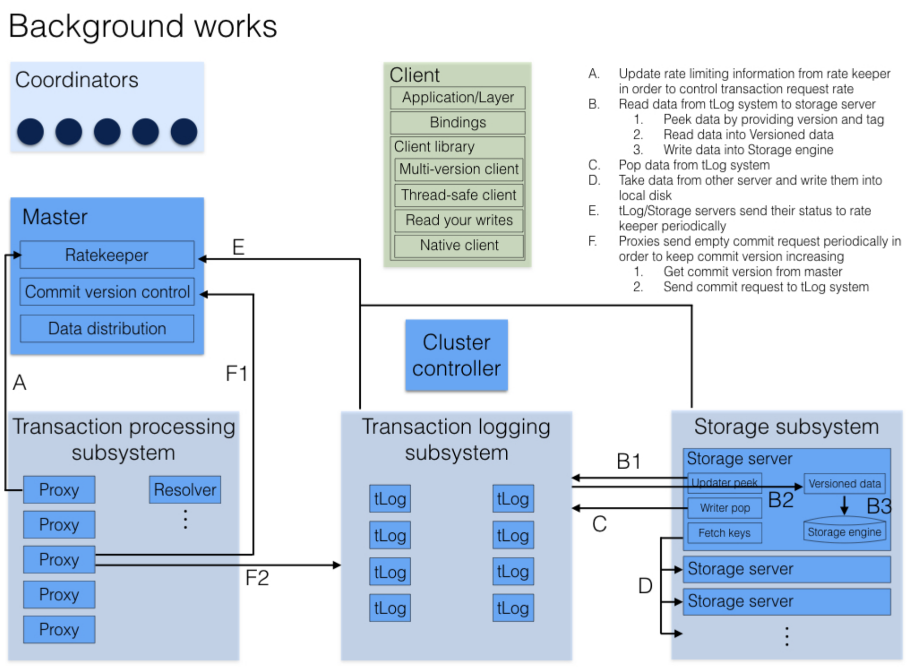

Start and open a database

1. Client talks to **all** coordinators according to the cluster file
2. Coordinators send cluster controller information to the client
    - At least a half of the coordinators should give the client the same cluster info
3. Client talks to cluster controller
4. Cluster controller sends **all** proxies address to the client

Background Work

There are a number of background work happening besides the transaction processing:

- **Ratekeeper** collects statistic information from proxies, transaction logs, and storage servers and compute the target transaction rate for the cluster.
- **Data distribution** monitors all storage servers and perform load balancing operations to evenly distribute data among all storage servers.
- **Storage servers** pull mutations from transaction logs, write them into storage engine to persist on disks.
- **Proxies** periodically send empty commits to transaction logs to keep commit versions increasing, in case there is no client generated transactions.



A. Update rate limiting information from rate keeper in order to control transaction request rate
B. Read data from tLog system to storage server
    1. Peek data by providing version and tag
    2. Read data into Versioned data
    3. Write data into Storage engine
C. Pop data from tLog system
D. Take data from other server and write them into local disk
E. tLog/Storage servers send their status to rate keeper periodically
F. Proxies send empty commit request periodically in order to keep commit version increasing
    1. Get commit version from master
    2. Send commit request to tLog system

Transaction System Recovery

The transaction system implements the write pipeline of the FoundationDB cluster and its performance is critical to the transaction commit latency. A typical recovery takes about a few hundred milliseconds, but longer recovery time (usually a few seconds) can happen. Whenever there is a failure in the transaction system, a recovery process is performed to restore the transaction system to a new configuration, i.e., a clean state. Specifically, the Master process monitors the health of Proxies, Resolvers, and Transaction Logs. If any one of the monitored process failed, the Master process terminates. The Cluster Controller will detect this event, and then recruits a new Master, which coordinates the recovery and recruits a new transaction system instance. In this way, the transaction processing is divided into a number of epochs, where each epoch represents a generation of the transaction system with its unique Master process.

For each epoch, the Master initiates recovery in several steps. First, the Master reads the previous transaction system states from Coordinators and lock the coordinated states to prevent another Master process from recovering at the same time. Then the Master recovers previous transaction system states, including all Log Servers' Information, stops these Log Servers from accepting transactions, and recruits a new set of Proxies, Resolvers, and Transaction Logs. After previous Log Servers are stopped and new transaction system is recruited, the Master writes the coordinated states with current transaction system information. Finally, the Master accepts new transaction commits. See details in this documentation.

Because Proxies and Resolvers are stateless, their recoveries have no extra work. In contrast, Transaction Logs save the logs of committed transactions, and we need to ensure all previously committed transactions are durable and retrievable by storage servers. That is, for any transactions that the Proxies may have sent back commit response, their logs are persisted in multiple Log Servers (e.g., three servers if replication degree is 3).

Finally, a recovery will fast forward time by 90 seconds, which would abort any in-progress client transactions with `transaction_too_old` error. During retry, these client transactions will find the new generation of transaction system and commit.

`commit_result_unknown` error: If a recovery happened while a transaction is committing (i.e., a proxy has sent mutations to transaction logs). A client would have received `commit_result_unknown`, and then retried the transaction. It's completely permissible for FDB to commit both the first attempt, and the second retry, as `commit_result_unknown` means the transaction may or may not have committed. This is why it's strongly recommended that transactions should be idempotent, so that they handle `commit_result_unknown` correctly.

Resources

Forum Post

Existing Architecture Documentation

Summit Presentation

Data Distribution Documentation

Recovery Documentation