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. # Overview

Before being made durable, FDB transactions must flow through several processes in an FDB cluster. This document gives a brief, high-level overview of each step in the commit pipeline. It focuses on the FDB7 implementation. For a more detailed overview of the commit pipeline, see this video.

Step 1: Client

Before a transaction is committed, all mutations are cached locally. The NativeAPI exposes a `Transaction` class which stores a vector of cached mutations. Bindings use a higher-level `ReadYourWritesTransaction` class, which also stores cached mutations in a tree that can be used to serve future read requests from the same transaction. In addition to storing vector of mutations, read- and write-conflict sets are also generated by each mutation.

Once the client calls `Transaction::commit`, a `CommitTransactionRequest` is sent to a commit proxy.

Step 2: CommitProxy

`CommitTransactionRequest`s are batched on the `CommitProxy`. Once a commit batch is ready, it is forwarded to the `commitBatch` actor, which coordinates the rest of the commit pipeline. The `commitBatch` actor first sends a `GetCommitVersionRequest` to the master server.

Step 3: MasterServer

The master server is in charge of distributing read versions and commit versions. Versions monotonically increase at a rate of approximately 1 million versions per second. The master server sends back a commit version via a `GetCommitVersionReply`.

Step 4: Back to CommitProxy

Once the commit proxy has received a commit version, it has all of the information it needs to send the commit batch to the resolvers for conflict resolution. In particular, the commit version and the read versions and read- and write-conflict sets of all transactions in a batch are sent to the resolvers in a `ResolveTransactionBatchRequest`.

Step 5: Resolver

The resolver maintains an in-memory skip-list storing the versioned write-conflict sets of all transactions committed in the last 5 seconds. This skip list is used to detect transactions in the batch that must be aborted with `not_committed` errors due to serializability violations. Furthermore, transactions with read versions more than 5 seconds old must also be rejected with `transaction_too_old` errors.

In addition to resolving conflicts, resolvers are also responsible for saving a log of mutations to the transaction state store. This log is used by proxies to maintain a consistent view of the transaction state store.

Resolvers send a `ResolveTransactionBatchReply` reply back to the commit proxy, containing both information about which transactions must be committed or aborted and information about which transaction state store mutations from other proxies must be applied locally.

Step 6: Back to CommitProxy again

When the commit proxy receives a `ResolveTransactionBatchReply`, it must apply the transaction state store mutations to its own local copy of the transaction state store. Using the transaction state store, it must then determine how all mutations in the set of committed transactions should be tagged. These mutations are then sent to the transaction log system through the `ILogSystem` interface.

Step 7: Transaction Log System

Full details of the transaction log system are outside of the scope of this page. When running with replication factor `n`, each mutation will typically be sent to `n` transaction logs, and it will only be acknowledged as successful if all `n` transaction logs report being able to make the mutation durable. Asynchronously (outside of the critical commit path), storage servers pull these mutations from transaction logs.

Step 8: Back to CommitProxy again

When the commit proxy has heard back from the transaction log system, it reports the commit version to the master server with a `ReportRawCommittedVersionRequest`.

Step 9: Back to MasterServer

The master then updates its local register of the latest committed version, and all future read versions given out must be at least this large. This is part of the way FoundationDB provides read-after-commit consistency. The master acknowledges to the proxy that it has learned about the reported committed version.

Step 10: Back to CommitProxy again

The proxy is then ready to send replies back to the committing clients. Replies either consist of a commit version for successful commits or an error for unsuccessful (or maybe successful) commits.

Step 11: Back to the Client

For successfully committed transactions, clients have access to the commit version. For unsuccessful (or maybe-successful) commits, clients can retry transactions depending on the type of error received.