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. Storage servers are the primary source of data for client requests. Each storage server is responsible for certain key-ranges (aka shards). As more writes come in and the storage server starts getting full or certain shards becomes write hot, the data distribution layer starts splitting those shards and move them around to different storage servers. Data distribution indicates so to the storage servers by writing special private mutations which when processed on the storage server take special actions to change the shard boundaries. More details on the data distribution internals can be found here: https://github.com/apple/foundationdb/blob/master/design/data-distributor-internals.md

update(): Main actor that keeps running on a storage server. It pulls mutations from the transaction log system (TLog) and applies those mutations to the storage server.

Private Mutations: Private mutations exist for certain in-frequent events. As the name suggests, they are private to a specific storage server. Private mutations are off of the end of the database and are injected by **ApplyMetadataMutation()** to tell the storage server to do something special. e.g. the most common private mutation is to change the shard boundaries.

update() calls **applyPrivateData()** to apply these private mutations. Which takes special action based on the type of private mutation. Here, we will focus on the private mutations that change the shard boundaries.

It could be the addition of a new shard, or the deletion of an existing shard. Please note that, we don't allow overlapping or conflicting shard boundaries at the storage server level. The data distribution layer makes sure that the storage servers only receive non-overlapping key-ranges.

There is a set of two private mutations to indicate a change in the shard boundaries. First mutation in the set will contain begin key of the shard range and second will contain end key of the shard range. After establishing this range, it calls **changeServerKeys()** which drives the addition or removal of a shard from the storage server. It is responsible for splitting or coalescing intersecting shard ranges as well. This only happens when there is an addition of a new shard ranges that intersects with an in-progress delete range. Because the deletion takes some time and new ranges overlapping ranges might get added during that duration. changeServerKeys() then calls **fetchKeys()** to fetch the data for the newly added key-range, from other storage servers. It does so by starting a database transaction and issuing a normal range read.

fetchKeys() fetches data at an established **fetchVersion** which is the current version of the storage server when it processes the mutation which caused the fetch to be initiated. After fetching, it applies the fetched data directly to the storage engine.

While fetchKeys() is in progress, the storage server might receive new updates to the keys that fall in the range that's being currently added in fetchKeys(). fetchKeys() keeps track of them separately, outside the storageServer while it's in progress. After fetchKeys() is done fetching and applying the new key-range to the storage engine, it waits for durablization of that data.

And then, it chooses the **transferredVersion** that ensures that * The transferredVersion can be mutated in versionedData * The transferredVersion isn't yet committed to storage (so we can write the availability status change) * The transferredVersion is <= the version of any of the updates in batch, and if there is an equal version, its mutations haven't been processed yet

Then it transfers that list of separately tracked new updates to update() actor to be applied to the storage queue at transferredVersion. There is a complicated song and dance that takes place between the update() and fetchKeys() actors.

The final step in fetchKeys() is to wait until the transferredVersion gets durablized. This essentially means waiting until the new shard data gets committed and durable.

Just a side note that the applyMutation() code is re-used, to both apply the mutation to the storage server, as well as track the newly coming updates while fetchKeys() is in progress. This happens based on the shard state. If the shard is in “readWrite” state, the mutation is applied to the storage server, else if the shard is in “Adding” state, the mutation is added to tracked shard state.