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. (This document is based off discussions that can be found in this forum post)

Building indexes from existing data (especially if you add the additional constraint that you want to do it from a live dataset while concurrently permitting reads and writes) can be somewhat tricky, but it also ends up being a necessary primitive that many sufficiently-advanced layers will end up needing to support schema evolution. This document would go through a sketch here of how that might be done.

The naive approach could be run a transaction which looks like "for each existing record, insert the index" and then commit that transaction. But there are several problems with this solution:

- Transaction size and time limit. Sufficiently large data sets may take longer than five seconds to read, and additionally, a transaction can do at most `10 MB` of writes (actually, probably best to keep them under `1 MB`, if possible)
- Conflicts. This transaction will necessarily have a read conflict range that spans the entire data set, so any concurrent write will cause that index-building transaction to fail.(In theory, one could get around this

by writing a key in FDB to "lock" the data, i.e., you would enforce client-side that any write first check to see if the lock key is set and then fail the write in your application rather than submitting it to be committed.)

So, it becomes essentially a necessity for any layer that implements some kind of index maintenance scheme to also implement a multi-transaction index builder. How to do that? The simplest way would be (1) start a transaction, (2) read some number of keys starting from the beginning of the data range, (3) write the appropriate index keys for the keys read, (4) commit the result, (5) go back to (1), but if (4) succeeds read from the last key written rather than the beggining of the range. This will usually work, with a few caveats:

1. There must be some mechanism to avoid using the unbuilt index to answer queries while simultaneously updating the index as data are inserted and deleted from the main data set. This usually necessitates being able to make indexes enter a "write-only" mode, and this state needs to be persisted somewhere, and then when the build completes, it can mark the index as readable by mutating that state.
2. This will only do the right thing if index updates are idempotent. For simple indexes, that assumption holds, so the only issue is that index updates might end up duplicating work. One example where that won't work can be the this: consider an index that counts how many times we see a value–a histogram index. (You might do this efficiently using our atomic mutation API.)
3. If whatever is building the index dies midway through, the entire index build must be restarted.

To solve (2) and (3), the solution can be designed to be slightly more sophisticated by atomically writing a key with the the index updates that states how far through the index build has gotten. That way, if the index building job dies part way through, it can be resumed from where it left off rather than the beginning. And then once that information is available, it can also be used to solve (2) by updating a non-idempotent write-only index if and only if the key being added is within the built range. If it's outside the built range, the index builder will pick it up. If it's within the range, then the initial write will index it. Either way, it is indexed exactly once.

So, the final question is then how to implement parallelism. To do that, the aforementioned technique can be extended to write multiple keys instead of just one saying which ranges of the main dataset have been built. Then multiple index builders can work on different ranges at once and use those keys to avoid stepping on each other. The details are a little complicated, but that's the general idea. If the distribution of the data is roughly known, it would be straightforward to pick boundaries to shard upon. FDB also offers an API that can be used to find the boundary keys of the underlying shards. Note that the boundaries are not required for correctness, just performance, so once boundaries are chosen, they can be used for the entirely of the build (or even change them at will depending on how the progress of each ranged is being tracked), and things will just “work”.