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. # FoundationDB Commit Process

Who can commit?

Anyone can make pull requests. Currently the FoundationDB team at Apple is controlling the list of contributors who can merge PRs. However, there are a few rules for these people:

Rules for People with Merge-Rights

- Only merge PRs for which you are the assignee. The only exception to that is if the assignee does not have merge-rights and asks you to click the merge button.
- Only merge a PR if CI successfully finished. The only exception to that are changes that are expected to break CI (for example changes to docker containers). "This change is trivial" is not a valid reason to merge before CI is done!

- Never merge a draft PR. You can reach out to an author and ask them to make the PR a non-draft.
- You may merge your own PR as long as it has the required number of reviews (GitHub will enforce this). New changes to a PR will dismiss all existing reviews, so an approved PR can be safely merged under the assumption the reviewer(s) have seen all changes.
- A PR needs at least one approved review.

Rules for PRs

- If a PR is not done done (code complete, test complete, and performance tested) it should be a draft PR. Only if the authors believe that the PR can be merged as is, it should be a non-draft PR.
- If the author wants reviewers to look at the code, the label RFC should be applied. If you're assigned as a reviewer you should ignore draft PRs unless they have the RFC label attached to them
- If the author wants to talk about the change in our weekly meeting, the label needs-discussion should be applied.
- A PR that cherry-picks a change from a newer branch should include a title and description that describes the change being made. For example, instead of the title "Cherry-pick #XXXX", you could include the original PR title "Change feature X (release-X.Y)" and copy the original description. The PR description should include the original PR number so that readers know which PR is cherry-picked. Please refer to PR#4677 as an example.
- A PR should include links in the description to any issues that it is addressing or any PRs on other branches that it was cherry-picked from.

Code Review

- Never be offended if a PR is rejected by a reviewer!
- Doing a good job in a code review can save us (and our build cops!) a lot of trouble down the road. Please take this job seriously. If you're unsure whether a code-change has a high enough quality in order to be merged, it probably hasn't. You can use the points below as a check-list:

Code Style

- **Variable and Function Names.** Do variable and function names make sense for what they mean / how they are used? Often during development the role of a variable or function can drift, a reviewer's fresh perspective might notice confusing names more easily than the original developer.
- **Consider running git clang-format to format your changes.** In some cases, this can result in funny looking code as not all files are formatted

correctly.

Performance

- **Are all CPU-hot paths well optimized?** Try to look out for hot paths. Usually we don't care as much about code that runs very rarely if it is not optimal. However, if you find anything new in a hot code path (for example the commitBatch function in the proxy), be extra careful and look for things that can be optimized.
- **STL containers.** Using STL containers is generally fine. However, for many containers we have specialized implementations and often they will perform better in FDB. For example: whenever you see a std::vector, check whether a VectorRef would make more sense in that context. Check for new slow tasks. In simulation or in performance runs, check whether there are new trace lines of type SlowTask - these can be a problem and should be fixed as early as possible.

Testing

- Is the code tested in Simulation? For all new introduced code there should be at least one simulation test that verifies that this code works as expected. In many cases existing tests will cover this already.
- Are new parameters and knobs tested properly. For knobs, please use BUGGIFY to test different values. For configuration parameters, make sure that the simulator tests different combinations of them.
- Were simulation tests run on this PR? Not all contributors have easy access to, or automatically run, a larger-than-one-ctest amount of correctness. In these cases, you can run it on either Apple or Snowflake architecture. Please don't rely on our nightlies to do this work for you. If the author doesn't have access to Joshua (our internal testing infrastructure that allows us to run simulation tests at scale), someone from Apple or Snowflake should run this for them. Currently our CI only executes a smoke test, so unless your code is trivial (like fixing a typo), please run enough testing yourself or ask someone to do it for you.
- Are ASSERT, ASSERT_WE_THINK, and TEST appropriately and correctly used? Assertions about invariants are always nice to have. They can also be made simulation-only. Code paths that are required to have meaningful and good testing coverage should have a TEST() macro to ensure that they are actually tested.
- Does it make sense to have unit-tests? We don't do too much unit-testing in fdb. But for some components it might still be worth adding unit tests (like new data structures)
- For bug fixes: is there a test that catches regressions? For bugs that we do not find in simulation, we should first try to write a test that reproduces the problem before fixing it.

Development Process

- Every PR needs a description (title is NOT a description). The description has to describe what the change is doing and how it was tested (such as the correctness test's result summary, perf test result).
- Instead of merging up, we're cherry-picking down. So if you want a change in `release-6.3` you first need to make this change in `master` and then cherry-pick the change (if applicable).
- If a PR is made against a release-branch there must be a justification in the description (or a link to a GitHub issue).
- Every function/actor/class that has been touched needs to be documented. Currently there are no guidelines about how much documentation is needed so it's mostly at the discretion of the author and the reviewers. But the minimum is one sentence per function/actor/class has to exist. Our code-documentation is quite bad and this is an attempt of gradually improving code quality.

Additionally the following two changes will be made in the near future: * All PRs will need two reviewers (at least on release branches, for master there hasn't been a decision yet). * Each release branch will have an owner. The owner will keep track of changes made to a branch. Not all responsibilities have been defined yet, but this person will be able to revert changes if they cause problems or do not follow the process or simply are not properly documented.