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. Here are some helpful debugging tips when you encounter with simulation failures.

1. Why I cannot reproduce failures reported by Joshua on my local machine?
   - Simulation tests are only reproducible on the same OS type, and with the same build environment. A binary from a CMake build in docker may not execute the same as a CMake build in the centos7 VM. Libstdc++ vs libc++ makes determinism change.
   - Make sure your test is deterministic by checking the unseed numbers are same across multiple runnings. (The seed initializes the random number generator for the test, and then the unseed is the last random number generated.)
   - Verify the `trace*.xml` is well-formatted. If you know your test is successful on your local machine, check the last `trace*.xml` file using command like `mono ./correctness/bin/TestHarness.exe summarize trace.0.0.0.0.0.1595012319.MJbTZt.0.1.xml summary.xml "" "" false`. See whether summary.xml has something the same as your Joshua report. (Maybe open the XML file in the browser

directly also help.)

2. How to use `valgrind` to help diagnose memory problems?

   - Edit `CMakeCache.txt`. Set `USE_VALGRIND=ON` then recompile fdbserver. Then `valgrind --log-file=valg.log ./bin/fdbserver -r simulation -s 366751840 -f ../tests/fast/SwizzledRollbackSideband.toml` can work.
   - If illegal instruction error appears when you run `valgrind` (says `vex amd64->IR: unhandled instruction bytes...`), set `USE_AVX=OFF`.
   - You can also submit `packages/valgrind-7.0.0.tar.gz` to Joshua. It takes more time to finish one assemble.

3. How to choose a subset of tests to run on Joshua?

   - Edit `TEST_INCLUDE:STRING, TEST_EXCLUDE:STRING`. Notice that the regex doesn't have `.toml` suffix, and the path can be related to `${FDB_REPO_ROOT}/tests` (ex. `fast/.*`), or not (ex. `Swizzled.*` means all test files under subfolders started with Swizzled).
   - Only test spec files under `${FDB_REPO_ROOT}/tests/{rare,restarting,slow,fast}` will be choosen. The files under ./tests/ are some tests that used to work or be useful but not anymore.

4. How to get useful information in `trace*.xml`, considering there're too many lines?

   - Start grep Trace events from `fdbserver/tester.actor.cpp` and locate you failed in which test phase.
   - Function `waitForQuietDatabase` in `QuietDatabase.actor.app` also contains many useful trace message in simulation test.

5. Get you to know which line causes the problem.

   - use coredump files. (ex. `gdb ./bin/fdbserver core.17426`)
   - use addr2line. The problematic trace stack is logged in trace files and stdout. (ex. `addr2line -e ./bin/fdbserver -p -C -f -i 0x7fae93006630 0x26b5f98 0x26eff9b 0x26dc132 0x26df565 0x26dfaea 0x25c8798 0x25c4f45 0x25c20d8 0x25c1e0e 0x25c0118 0x25b816c 0x25bf888 0x25bf4ba 0x25bc968 0x25bc27a 0x1eb6148 0xf34538 0xf34316 0x287a431 0x287a1f1 0xfc6c78 0x28edc37 0x28eda96 0x28ed731 0x28edae2 0x28ee16c 0xfc6c78 0x29d1016 0x29c5bfe 0x28df844 0x13f3c18 0x7fae92745555`)

6. A restart test failed. How do I reproduce it locally?

   - See https://github.com/apple/foundationdb/wiki/How-to-reproduce-a-restart-test-failure.

7. Try to set `TestConfig.simpleConfig=true` to make the fdb cluster as a small cluster to make debug easier.

8. `MutationTracking.h` and `MutationTracking.cpp` file contains useful functions to track all mutation activities of up to 2 keys in simulation test.

Try to set the key you want to track, and `grep MutationTracking`.