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. # Introduction

As we all know, certain kind of bugs within FoundationDB can have catastrophic consequences if they are triggered in a production environment. We currently rely mostly on testing to find those bugs. However, while we believe our testing is very thorough, it is clear that it can only prove the existence of bugs - never the absence of bugs.

Additionally we should make changes to FoundationDB to make it less likely that a bug can cause catastrophic damage. Optimally, bugs should only be able to cause crashes or stalls, but not data corruption or data loss.

We already have some mechanisms in place that try to do this. One example is checksumming in storages. Another example is the concept of file identifiers in flatbuffers (so if we try to read a message of a wrong type there's a high chance we crash in flatbuffers - while the old streaming serializer would just read garbage).

This wiki page is meant as a place to brainstorm ideas on how to reduce the probably that bugs can cause catastrophic failures. Ideas shouldn't be limited to new features but can also include new testing methodologies. How-

ever, in this document we are not interested in ideas on how to reduce the number of bugs, only how to make FoundationDB more robust against bugs.

Idea 1: Add invariant checking

Each component, e.g, proxy and tLogs, may have invariants for each of its functionalities. For example, a tLog should not receive mutations whose tags are not for the tLog.

It has two benefits: a) Crashing the system early when invariant is violated can help uncover potential data corruption situation.
b) The invariant failure can help uncover the root cause of a simulation failure and production failure. This saves time both in development and in production error diagnosis.

This requires code change and expertise in each component. It can be hard to find these invariants. Linux kernel development uses invariants to crash the system early instead of letting a system wobble in undetermined state.

Idea 2: Add verification Engine

This is a similar to idea 1 but might go a bit farther.

The proxies currently keep the txnStateStore in memory and every transaction (whether system- or user-transaction) has to go through a proxy. Additionally, every message written to a TLog is written by a proxy.

This means that we could use (and maybe additionally introduce) some data redundancy in the txnStateStore that would allow us to do further verification. After we resolved a batch of transactions we could then run every batch through this verification engine. Afterwards we could also run all TLog message through this engine before we actually write to the tlog. This would allow us to catch certain type of bugs early and simply crash the proxy before we write anything to a disk.

Examples for things we could verify:

- We could verify that we don't duplicate messages to a tag.
- We could verify that shard assignment messages and private mutations go to tags that make sense (for example when removing a shard from a tag, we can double-check there that this corresponds to the shard-mapping we currently have).
- For replication factor `N`, we could verify that every non-private message goes to at least `N` number of tags and `N` number of tlogs and that we don't violate the replication policy.

Idea 3: Buggify++ (Bug Injection in Simulation)

Similar to buggify, we could introduce byzantine failures in several places of the code (for example, in the serialization code we could randomly deserialize wrong data). As FDB is by definition not resilient against these kind of failures, test runs will need to run differently. This is what I would imagine could work well:

1. We would run a simulation without bug injection enabled and run `SaveAndKill` at the end.
2. We would then run a restart test with bug injection enabled. In this run, we would ignore all Sev 40s (as they are excpected) and we won't expect any progress. Basically the hope is that this will crash at most but it won't change the on disk state in any harmful way.
3. Last there would be another restart test. This would then verify that no data corruption has happened in the previous run.