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. ## Overview

Unlike most roles in the cluster, coordinators must be configured statically, so selecting them requires more careful consideration. Coordinators carry state, and this state is not automatically re-replicated if a coordinator is lost. This means that you must select enough coordinators to fulfill your desired fault tolerance. Coordinators are also quorum-based, which means that you need additional coordinators to ensure that a quorum is available. Losing a majority of your coordinators will take the database completely unavailable. On the other hand, having more coordinators than you need can be a problem for performance and operations, so you want to avoid recruiting additional coordinators if it does not provide significant marginal benefit.

Upper Bounds on Coordinators

Coordinators are responsible for three things and all of these would become significantly slower if you have a large number of coordinators:

1. They elect the cluster controller (cluster controller is the leader of a cluster). This election process is done in an iterative way: election takes a

short amount of time but if it fails, this election time is increased. Having more coordinators means that the probability of the election failing increases. Therefore you would need more iterations. I would imagine that with 100 coordinators election would be very very slow.
2. They store a global state. During recovery this state is read, locked, and rewritten to all coordinators. If the system has more than one master running, one of them will eventually fail. But as this protocol involves several round-trips to all coordinators, it would slow down the process. Therefore, your recovery times will probably go up significantly if you have many coordinators.
3. Clients connect through coordinators. This path is optimized to prevent clients from keeping unnecessary connections, but in the worst case a client will always need to talk to all coordinators. Therefore having many coordinators will make connection establishment slower.

Recommended Coordinator Counts

For a single-DC config with replication factor `R`, you want `2*R-1` coordinators. The expected fault tolerance with this configuration is `R-1`. Having `2*R-1` coordinators means that after losing `R-1` you will still have a majority of the original coordinators available, which is required for the database to be available.

For multi-DC configs, you will likely want to be resilient to at least 1 DC and 1 other machine going down simultaneously. Losing a DC is likely to be a failure mode that takes longer to recover from, and you will want some additional fault tolerance while you are in this failure mode. For these kind of configurations, we recommend 9 coordinators, spread evenly across at least 3 DCs. This would ensure that no DC has more than 3 coordinators. In the event that you lose 1 DC and 1 machine, you would lose at most 4 of your 9 coordinators, leaving you with a majority available.

If you are only storing data in two DCs, we recommend that you provision 3 processes in a third data center to serve as coordinators.

In three data hall mode, we also recommend 9 coordinators, by the same logic.

Replication Mode	Coordinator Count
single	1
double	3
triple	5
three_data_hall	9
three_datacenter	9
multi-region	9

Determining Which Processes to Use

There are two things that you should guarantee when determining which processes will serve as coordinators:

- Every coordinator should be on a different fault domain, and thus have a different zoneid.
- Coordinators should be spread as evenly as possible across data centers and data halls.

These two rules should ensure that your cluster maintains its fault tolerance goals.