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. # Background

- https://forums.foundationdb.org/t/how-do-bindings-get-an-appropriate-copy-of-fdb-c/311
- https://forums.foundationdb.org/t/new-api-to-get-the-version-of-a-running-cluster-without-needing-a-matching-libfdb-c-library/437

New fdb c API to ask a cluster its protocol version

Proposed API:

```
// Get the server's protocol version, optionally passing in an expected protocol version.
// If you only want to know what the current server protocol is, then pass NULL for currentProtocol.
// Otherwise, the returned future becomes ready when the protocol version different from *currentProtocol.
FDBFuture* fdb_get_server_protocol(const char* clusterFilePath, const uint64_t* currentProtocol);
fdb_error_t fdb_future_get_uint64( FDBFuture* f, uint64_t* out );
```

Basically you can get a future where if it becomes ready, then a majority of coordinators are/were running fdbservers with this protocol version. This

will work even with older versions of fdbserver before we added this feature, since they already share their protocol version.

This can be combined with the following existing (but currently undocumented) feature to get the bare minimum answer to a popular question - "Is my client protocol-compatible with a majority of the coordinators it's trying to connect to?"

```
DLLEXPORT const char* fdb_get_client_version();
```

Which returns version information formatted like `$MAJOR.$MINOR.$PATCH,$GIT_HASH,$HEX_PROTOCOL_VERSION`.

We could conceivably also have `fdb_get_server_protocol` return all the same version information that `fdb_get_client_version` does, but this can't be done in a way that will work with old fdbservers since they just don't share that information.

Implementation

- https://github.com/apple/foundationdb/pull/3858
- https://github.com/apple/foundationdb/pull/4028

The road to getting `npm install --save foundationdb` (and similar) to just work

The current proposal is that we start offering "fat bindings" (similar to the fat jar) which bundle a copy of libfdb_c for every supported platform. Such a package would then be set up to query the server for its protocol version, and if necessary download an appropriate libfdb_c based on that. This downloaded libfdb_c could then be loaded as an external client by the multiversion client. Loading an external client after setting up the network isn't currently supported, so we'd have to fix that.

Retrieving `libfdb_c.so`

We currently have two ideas for retrieving the libfdb_c.so file.

1. Downloading directly from coordinators
2. Downloading from blob store

Both ideas have some considerations that go along with them.

Downloading from coordinators would require thought into where to write the file and how to get write permissions into that directory. We would also have to make sure not to overload coordinators with download requests.

Downloading from blob store would require clients to go to an external source (namely the blob store) to download the libfdb_c file, which some clients may not accept.

Other ideas

- Download libfdb_c directly from coordinators
- Where to write the file? How to get permissions?
- Distribute libfdb_c to all clients before upgrade
- Avoid performance penalty for going through two network threads
- Rules for preferred client when there are multiple protocol-compatible clients
- Don't ping coordinators with all clients
- Push patch updates to clients through server - something like `fdbcli --exec 'distribute client /path/to/libfdb_c.so'`
- Don't overload coordinators
- Measure client downtime caused by recovery, and don't make it worse
- Implement fdb_get_server_protocol natively in clients?