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. The global configuration framework is an eventually consistent configuration mechanism to efficiently make runtime changes to all clients and servers. It works by broadcasting updates made to the global configuration key space, relying on individual machines to store existing configuration in-memory.

The global configuration framework provides a key-value interface to all processes in a FoundationDB cluster. The first part of this document describes how to use the global configuration framework to read and write data. The second part goes into implementation details.

Using the global configuration framework

Reading data

The global configuration framework is exposed through the `GlobalConfig::globalConfig()` static function. The API is defined in `fdbclient/GlobalConfig.actor.h`. There are separate ways to read a value, depending on if it is an object or a primitive.

Reading a primitive The recommended way to read a primitive from the global configuration is to use the templated get function (note that this requires you to know the type of the value). A default value must be supplied in case the global configuration framework cannot find the specified key.

```
auto& config = GlobalConfig::globalConfig();
double value = config.get<double>("path/to/key", 1.0);
```

Instead of hardcoding a string literal for the key, it is recommended to store your global configuration keys in the global configuration implementation file (at the top).

Reading an object Reading an object returns both the object as well as a reference to the arena where the objects memory lives. As long as the client keeps the arena around, the object's memory is guaranteed to stick around.

```
auto& config = GlobalConfig::globalConfig();
auto configValue = config.get("path/to/key");
```

The type returned is `ConfigValue`, defined as

```
struct ConfigValue {
    Arena arena;
    std::any value;
};
```

If the global configuration framework does not have a value for the given key, `configValue.value.has_value()` will return false. Otherwise, the value can be cast to the appropriate type with `std::any_cast`.

```
ASSERT(configValue.value.has_value());
auto str = std::any_cast<StringRef>(configValue.value);
```

Range read Range reads of global configuration data are also available. See `get(KeyRangeRef range)` in the global configuration header file.

Writing data

Writing data is done using transactions written to the special key space range `\xff\xff/global_config/ - \xff\xff/global_config/0`. Data must always be encoded as tuples. See `fdbclient/Tuple.h` for the encode/decode API.

You are responsible for avoiding conflicts with other global configuration keys. For most uses, it is recommended to create a new directory. For example, an application that wants to write configuration data should use the `global_config/config/` namespace, instead of storing keys in the top level. Global configuration keys should be stored as constants in `flow/GlobalConfig.actor.cpp`, and you can check this file to avoid naming conflicts.

Here is an example of writing to the global configuration.

```
// In GlobalConfig.actor.h
extern const KeyRef myGlobalConfigKey;
// In GlobalConfig.actor.cpp
const KeyRef myGlobalConfigKey = LiteralStringRef("config/key");

// When you want to set the value..
Tuple value = Tuple().appendDouble(1.5);

FDBTransaction* tr = ...;
tr->setOption(FDBTransactionOptions::SPECIAL_KEY_SPACE_ENABLE_WRITES);
tr->set(GlobalConfig::prefixedKey(myGlobalConfigKey), value.pack());
// commit transaction
```

Clearing data Clears and clear ranges are supported and behave in the same way as normal transaction clears.

Implementation details

The global configuration framework stores key-value pairs using FDB itself, with some abstraction between storage and retrieval. Durable configuration data lives in the `\xff/globalConfig/` key space and is maintained in the same way as all other key-value pairs in FDB.

Storage Internals

The `\xff/globalConfig/` key space is broken up into three ranges.

- `\xff/globalConfig/k/ - \xff/globalConfig/k0`: storage of the key-value pairs that make up the global configuration
- `\xff/globalConfig/h/ - \xff/globalConfig/h0`: a truncated list of the last three updates made to the global configuration. Keys in this range represent the commit version the update was made at, and the value is a serialized string of `MutationRef`s representing the update
- `\xff/globalConfig/v`: serialized versionstamp of the last update made to the global configuration, followed by the protocol version

Updates

The cluster controller creates a watch on `\xff/globalConfig/v` and updates its `AsyncVar<ClientDBInfo>` with the most recent mutations when the version changes. `ClientDBInfo` is broadcast to all nodes in the system on any change, and eventually each individual node will receive `ClientDBInfo` containing the three most recent updates made to the global configuration. If the node does not know about the oldest mutation in the mutation list, it must re-read the

entire global configuration from the `\xff/globalConfig/k/` range. Otherwise, it can use the history in `ClientDBInfo` to update its global configuration. This is the mechanism through which all nodes eventually converge on the most recent configuration.

Special Key Space

The special key space performs extra work when writing to the global configuration. The following actions are performed when a transaction making at least one write to `\xff\xff/global_config/` is committed.

1. Reads the `\xff/globalConfig/h/` range and deletes the oldest key-value pairs (according to version) to make room for the new update.
2. Transforms writes made to `\xff\xff/global_config/` into writes made to `\xff/globalConfig/k/` for durable storage.
3. Writes the newest set of mutations into a new `\xff/globalConfig/h/<version>` key, where `<version>` is the versionstamp of the commit.
4. Updates `\xff/globalConfig/v` with the versionstamp and protocol version used for the transaction.

The special key space also provides functionality for reading global configuration values. A transaction may read `\xff\xff/global_config/<your-key>` to read values set in the global configuration namespace. This is for information and debugging purposes only, and should not be used by client applications (see Reading Data above). The special key space read functionality will return all values as strings.