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

FoundationDB is a key-value storage database with transactional support. When a client submits a transaction, the corresponding data will be sent to a proxy. The proxy will go to the master for the version of the transaction. After versioning information is retrieved, potential read and write conflicts will be sent to the resolver. The resolver will determine if the conflicts can be resolved or reject the commit. In the case the resolution is found, the proxy will forward the key-value mutations to tLog servers, where the journals are stored and consumed by storage servers. In one transaction, multiple key-value pairs might be updated. In the current schema, all the pairs will be sent to one single proxy. If the volume of the key-value pairs is large, proxy processing, which is both CPU and network consuming, will be blocking and requests from other clients will be thus delayed. A straightforward solution of this would be, split the large transaction into multiple parts, and deliver the parts to different proxies. The details will be discussed in the following sections.

Design

Client

When a transaction comes in, the total size of the values will be estimated by summing up the size of the values of each key-value pair. If the total size is above a certain criteria, the client will split the transaction into parts. The number of parts is determined by the number of proxies the client connects to. If the client connects to N proxies, then the transaction will be split into N parts, and each proxy will receive one of them, respectively. Since the parts need to be merged back later, they will share a same unique ID, denoted by split ID. The content of the transaction will be distributed over the parts. For read_conflicts and write_conflicts, since the elements are unordered, round-robin arrangement is sufficient; while for mutations, items are ordered as they are not commutative. To ensure the ordering, each part will handle a continuous subversion of the mutations. For example, if there are ten mutations and three proxies, then one possible distribution will be: proxy 1 receives mutation 1, 2 and 3; proxy 2 receives mutation 4, 5 and 6 whereas proxy 3 receives all the remaining mutations.

Proxy

Each proxy will receive one part of split transaction. Even the current implementation will batch the commits it have received for the purpose of performance, a part of a split transaction will **not** be batched, i.e. it will always be treated as a single transaction. There are certain reasons for this design:

1. If a part of split transaction is batched with some other non-split transactions, while the split transaction is rejected by the resolver, then the rejection will include unnecessary victims.
2. If a part of split transaction is batched with some other parts of split transactions, this will cause the split transactions share the same version.

Both of the issues are not unsolvable, yet unnecessary complexity would be introduced; thus it is decided that parts of the split transation will always be committed alone.

Master Server

The master server provides version for each transaction. Without split transaction, the responding version is strictly monotonically increasing. However, for a split transaction, all parts should use the same prevVersion and version, to ensure they are processed at the same time. To achieve this, when the master server receives a version request from a split transaction for the first time, the split ID and version will be stored in a cache. The rest of the parts, identified by the split ID, will re-use the version in the cache. Each split ID/version pair in the cache will have a timestamp. The timestamp is used to determine if

the pair is expiring or not. The same approach is used to ensure prevVersion is consistent for all parts of a split transaction.

Resolver

The resolvers need to retrieve all read_conflicts and write_conflicts for a split transaction to carry out its task. Currently, this is done by waiting for all proxies sending the conflicts and merging them later. An alternative approach is to let one part, denoted by **P**, carry all the conflicts, and cache the result of the resolution per split ID. Parts arrived before **P** will need to wait; while parts arrived after **P** can immediately receive the result from the cache. The benefit is that the resolvers do not need to wait for all parts; while the bandwidth might be the bottleneck since only one proxy retrieves and delivers the conflict range.

Transaction Log

Similar to the resolvers, the transaction log servers has to retrieve all assigned mutations before it commits the transaction. The mutations are sorted by subversion, as TLogs depends on the subversion of the mutations during the persistent process.