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. ### 4/29/2020

- Work continuing on Persistent ART
  - Another ART implementation in HOPE if it's better/faster/more tested
  - One could maintain an ART of N second windows to re-use the existing Arena-based one.
  - One could maintain an arena for every N seconds, and then concurrently walk through and remove references into older arenas.
  - If one maintains version history in values rather than keys, then a single-version structure would work. Atomic op-heavy workloads would be the worst case for doing this, though.
  - Entirely different strategy: throw more memory at MVCC buffer?
- Rusty's tree changes likely to go in soon
- Rusty comparing memcpy implementations, to hopefully get an overall speedup
  - It might possibly be worth dusting off the "Align arena memory" PR
- Trevor has another PR for reducing copies with Futures/Promises
  - And has more changes planned for reducing copies in the proxy

4/22/2020

- Allow moving out of futures and actor compiler changes to reduce copies
  - Avoiding unnecessary copies from movable futures
  - R-value references are complicated
- Daniel ran Rusty's ART benchmark
  - Benchmark used random strings, which biased towards making ART work well
  - Result was 7x faster, which looks promising
  - ART code itself relied on area for memory safety
  - Struggled to get refcounting on children working right
- Daniel re-did benchmarks, and found out his target bandwidth is lower than expected.
  - Will look into RocksDB PR after spending a bit of time on ART
  - Better storage server write throughput is needed, something closer to Memory than Redwood
  - We should check with **Markus**, who mentioned someone on RocksDB might have gone peeking into this
  - There's still feelings of uncertainty about RocksDB correctness, and FDB simulation testing wouldn't help find them, as RocksDB would be running on its own threads outside of FDB's control

4/15/2020

- Avoid unnecessary copies in PromiseStream #2915 landed, and mostly resulted in write bandwidth improvements
- Investigating doing more similar changes
  - Modifying actor compiler to get perfect forwarding in more places #2928
  - Potentially creating a MovableFuture class, so that one can move out of futures.
  - A longer discussion occurred on design choices around futures, actor compiler changes, and resumable functions
  - Eliding copies from reply generation could have similar improvements.
  - Using a better memcpy implementation resulted in a 5%-10% perf increase.
    - ⋆ **Rusty** to look into this after current work is done. Ask **Kao** for brain dump.
  - make-linux-fast-again.com to disable kernel security patches when comparing perf across versions

4/8/2020

- Storage server profiling and optimizations

  - Daniel and Rusty have both not found low hanging fruit to further optimize
  - **Daniel** to run and profile a test with Rusty's three way comparison merged
  - 5% of CPU time spent allocating std::vector, but unsure if there's a good way to reduce that
- Daniel seeing stalls during performance tests
  - Are there recoveries? OOMs? Seems not.
  - Likely Ratekeeper?
  - **Alex** to connect with Segment folk about getting FoundationDB metrics into Datadog. (Sent!)
- P-Tree improvements/rewrites
  - Daniel chatting with Steve about if there's a better datastructure to use than the existing P-Tree
  - Neelam's previous investigation suggested that there's no low-hanging optimizations to do on the P-Tree itself
  - Persistent Adaptive Radix Tree might be a good candidate
  - (Persistent in the versioned sense, and not persistent as in disk.)
  - HOPE would maybe be useful to reduce in-memory size of data?
    - ⋆ **Rusty** to bug Pavlo about releasing the code
    - ⋆ Update: HOPE source was released
- New Slack channel of #write-throughput-discuss made for this project
- Proxy optimizations
  - 550micros spent in batching
  - Via tweaking knobs, this was reduced to 200micros
  - In the process of this work, found Avoid unnecessary copies in PromiseStream, which might help for more loaded cases.

4/1/2020

- Storage server profiles show 25% CPU time in operator<
  - Rusty's approach: reduce the number of calls to operator< made by lower_bound and upper_bound
  - Daniel's approach: Pass StringRef by value in comparison operators
  - Rusty has a larger change in the works to move from operator< to operator<=> for more gains
  - Taking an `ssd` profile again would be good?
  - Try tweaking knobs to lower the number of versions a storage server keeps in memory to see what effect that has
- Proxy CPU
  - Had used debug transaction to build up stats on where time is going in commits
  - Looked like more time is spent batching transactions on proxy than in TLogs
  - Exploring ways to cheat the commit path, and drop strict serializ-

    - ability if desired
    - Policy engine optimizations might have a large impact on proxy cpu time
- Will continue adding more debug transactions to get better pipeline visualization

3/25/2020

- Rusty's updates
    - Initial results from network loop optimization resulted in a 1%-2% speedup
    - Next focus will be on storage server optimizations
- Daniel's benchmarking run results
    - Tracing of DebugCommit doesn't seem to show large pipelining waits
    - AWS seems to show ~0.1ms for within AZ, and ~0.3ms and ~0.7ms for across different AZs
    - Adding more clients now shows storage server saturation
    - Will continue to run more benchmarks, take and forward profiles to others to examine more closely

3/18/2020

- Pipelining
    - With Chrome Tracing, we now have some tools we can use to investigate commit pipelining.
    - How do we identify issues? What do we think could be issues? What tests should we be running to find issues?
    - Setting `\xff\x02/timeKeeper/disable` will disable timekeeper, which is the every 10s debug transaction.
    - Trevor is doing work in this area also. Will invite him to future meetings.
        - ⋆ Investigating instances where more time being spent in proxy than waiting for tlog fsync.
    - Resolvers cannot just be turned off for conflict-free workloads due to their involvement in `\xff`
- Other improvements
    - Rusty's event loop 2.0?

To do for next week: * The suspicion is there there should be one proxy that is slower than the others. * Confirm from the trace that resolvers are waiting on pipelining, and preferably that one proxy is indeed slower * Look into or add proxy batching to traces * If there is indeed one slow proxy, then work on single proxy CPU usage and profiling * Sync with Trevor on his current state of proxy profiling