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. Storage queue in FoundationDB is an in-memory, queryable structure that keeps 5s worth of versioned data on a storage server. It's commonly referred to as PTree in the FDB code base.

Data structure

Storage queue has been implemented using the 'Treap' data structure. It's technically a balanced binary search tree, but O(Log n) height is not guaranteed. It uses randomization and binary max heap property to maintain balance with high probability. Expected time complexity of search, insert and delete in a Treap is O(Log n).

Every node of Treap maintains two values: * Key: Follows standard BST ordering (left is smaller and right is greater) * Priority: Randomly assigned value that follows Max-Heap property.

Partial Persistence

Partial persistence in a data structure is the ability to read at any given version, but the ability to modify only the latest version. Storage queue in FDB offers partial persistence for 5s worth of versioned data. There are three common techniques to achieve partial persistence in a data structure: * Copy-on-write * Fat Nodes * Path copy * https://en.wikipedia.org/wiki/Persistent_data_structure

PTree has implemented partial persistence using a combination of Fat node and path copy technique. In this hybrid technique, one additional updated pointer is stored in the PTree nodes. And an update at a new version uses the updated pointer, if available. Else, it creates a new node and follows the path copy technique.

Node structure

PTree nodes are fast allocated (using the FDB fast allocator) and reference counted. Each node is separately fast allocated. i.e. there is no guarantee what arenas they will end using. Each node is fixed in size at 96 bytes. Nodes contains references to keys and values in the mutation log and an integer priority (and hence fixed size). Each node contains 3 pointers: left child, right child, updated child if any.

Whenever the 3rd pointer (i.e. updated child) in the node is occupied, that means the node contains an update at a newer version. It's important to note that the update actually happened to one of the child nodes (left or right) and not actually to the node. And the node will have the following special fields set: * Updated flag: Indicates that the node was updated (imp to look at this in addition to the updated child pointer. Because the child pointer might be present because of an uncleared node as well) * replacedPointer: Indicates which child (left or right) was actually updated * lastUpdateVersion: The version at which the update happened

Versioned Map

Versioned Map is the structure that's exposed to the storage servers. It's a std::deque of pairs of versions and corresponding PTree root nodes. Since the versions are monotonically increasing, the data is sorted intrinsically. So we don't need to store this information in an in-order data structure (such as std::map which is a red-black tree underneath).

Versioned map provides a logical abstraction of a separate PTree at any given version. Physically it's a single PTree which contains data across the 5s worth of versions.

It helps the storage server process find the root of the tree at a given version during various read operations. Whenever a new version is created, root node corresponding to the latest version is copied into the new entry and may get modified as operations are performed on the PTree.

Storage Queue Mutations and Operations

Storage queue provides many read-write operations to it's clients, such as the storage servers. Such as inserting new <key, value> pairs, clearing certain key-ranges and range reads.

Clear Range Mutation

This operation erases a [startKey, endKey) range at a given version. Note that the endKey of range is not-inclusive. Clear range is a two step process: * Step 1: Erase the <key, value> pairs in the given range from the PTree as of the given version * Step 2: Insert a special marker <startKey, endKey> (aka tombstone) indicating the presence of clearRange Tombstone insertion is necessary to have for readRange at the storage server level to work correctly. This is because, at the storage server level, the result of the readRange operation is the merge of read results from the storage queue and the storage engine. The presence of the tombstone ensures that we skip over that range while reading from the storage engine as well.

Set Mutation

This operation inserts a <key, value> pair at the latest version. The steps involved in inserting a <key, value> pair in the storage queue are as follows: * Locate the root of the PTree at the latest version * Traverse the PTree to find the correct position - typical BST way * Create the node with given <key, value> and attempt to insert at the desired position * Generate a random priority value and check if the max heap property for priority has been violated * Causes a rotation if violated

Presence of clearRange mutations needs special handling during Set mutation When inserting a key that happens to fall in the middle of a clearRange, we need to split that clearRange. e.g. Let's assume that there already exists a clearRange [A, E) in the PTree, now inserting C would cause the clearRange to split into: [A, C) and [D, E).

Read Range Operation

As mentioned above, read range operation has been implemented at the storage server level. As the name suggests, it returns values with-in the specified

key-range. At storage server level, it's the merge of read range results from in-memory PTree and the storage engine.

Recall that PTree is a binary search tree under the covers and hence there is no super-efficient way to do a range scan (like a B+Tree where the leaf nodes are connected can do a blazing fast range scan).

Also note that the FDB PTree implementation doesn't store parent pointers, which might help going back to the ancestor. With these limitations, and with the goal of achieving efficient range scans, this operation uses an iterator with a finger vector: like a stack of nodes visited. This helps remember the ancestor in the absence of a parent pointer. This works well while moving forward and saves a bunch of work. But note that there is more wasted work while moving backwards.

Range read finds the low bound based on key range beginning and then can move forward or backward (depending on the direction of range read) using the finger vector of the iterator.

Interesting properties

Presence of clearRange mutations leads to some complexity in the read-Range path. readRange gets split into multiple readRange requests if there are clearRange mutations present in the desired key-range. End bound of current readRange iteration is at the start of next clearRange. Then data is read from the storage engine for that (sub)range and the results are merged with the PTree read results.

PTree is traversed once to establish these end bounds and then again while reading the data between start bound and end bound in the merge step. This double traversal is important for efficiency and correctness in the presence of clearRange mutations. Because we don't want to read the key-range of a clearRange from storage engine. But it can be wasteful in the absence of a clearRange. We can potentially cache the results of PTree traversal while establishing the end bound and use that during the merge process.

Potential known issues with Storage Queue

Storage queue growth can be a concern in certain scenarios. If the storage queue gets backed up, it can have a negative impact on the system's overall performance.

Scenarios that can lead to storage queue backing up: * When the underlying disks are not able to keep up, and the data versions start to accumulate in the storage queue. * When there are relatively few data mutations, but large number of versions. This makes the storage queue large * In the current imple-

mentation, the storage queue keeps everything since last persisted version to very latest. Even if that corresponds to more than 5s worth of data versions * For a write heavy workload (any workload can have periods of high write rate, e.g. during initial or incremental loads), if the number of storage server processes is not configured properly (if it's less than the optimal value per disk), that can also lead to the storage queue being backed up. Because there won't be enough disk write requests and they won't reach their peak IOPS.