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. This document describes how cluster coordination works. This should answer the following questions:

- What happens when a `fdbserver` process joins a cluster?
- How are node failures and network partitions detected?
- How can a process find processes of a specific role?

Important Roles

All server nodes within a cluster participate in the cluster coordination process. Servers join the cluster by participating in the leader election process (or rather by submitting a candidacy) and then stay in the cluster by sending an occasional heartbeat to the leader.

The three roles (explained in more detail below) that participate in this process are:

1. The **coordinators** are typically chosen by the administrator and a cluster typically runs between one and 5 of those.

2. The **cluster controller** is the leader and is elected by the coordinators. Any node in the cluster can become leader, but an administrator can make the election of certain processes much more likely by setting a role.
3. **Workers** are a fundamental building block for a FoundationDB cluster. Every process runs this role. It's main functionality is to allow to start other roles (like storage, tlog, master, resolver, etc).

The Coordinators

The coordinators run in the processes that are listed in the `fdb.cluster` file. Coordinators are the first entry point and, apart from other things, they elect the leader (called cluster controller).

When a `fdbserver` process starts, it will send a candidacy to become leader to all coordinators. This means that all nodes in the cluster will try to become leader. Each coordinator will reply with a `LeaderInfo` object - containing the interface of what it chose to become leader. If a majority of all coordinators agree on the leader, this node will start the cluster controller role and all other nodes will connect to it.

The relevant actor here is `tryBecomeLeaderInternal` which can be found in `fdb-server/LeaderElection.actor.cpp`.

If everything goes well, the server will afterwards have knowledge of the leader.

The Cluster Controller

The cluster controller is the leader which got elected by the coordinators. The definition of its RPC interface can be found in `fdbserver/ClusterRecruitmentInterface.h` (part of it is reachable from the client side - this can be found in `ClusterIn-terface` in `fdbclient/ClusterInterface.h`).

The cluster controller has many responsibilities (and not all of them will be explained in this document):

1. It keeps a list of all server nodes that are currently running in the cluster.
2. It broadcasts the `ServerDBInfo` to all nodes in the cluster (see below for more explanations).
3. It monitors all servers and decides on failures - it is the main authority to resolve network partitions.
4. It kicks off recovery by electing a master (only very briefly explained later).
5. It recruits workers for certain roles (not explained in this document).
6. It serves as the endpoint to clients for opening the database (not ex-plained in this document).

7. It can generate the `status` json document for `fdbcli` and clients (not explained in this document).

The Workers

Every `fdbserver` process that joins the cluster runs a worker role. It's interface can be found in `fdbserver/WorkerInterface.actor.h` (`WorkerInterface` only visible to other servers) and `ClientWorkerInterface.h` (`ClientWorkerInterface` also visible to clients).

On startup the worker tries to become a leader and it will get a cluster controller interface as a result (this could be an interface to itself if it won the election). It will then let the cluster controller know of its existence by calling `registerWorker` on the CC interface. This allows the CC to curate a list of life workers.

Furthermore the worker will check whether there are existing files of interest in the data directory on startup. This means specifically that it will check for tlog, storage, and coordinator files when it starts and if one or more of these files exist, it will automatically reboot that role.

The workers provide the following functionality:

1. They provide RPC interfaces to initialize/start new roles (like tlog, storage, master, resolver, etc). This is used by the CC and master during recovery.
2. A `waitFailure` interface (explained below). This is used to detect worker failures.
3. An interface to execute disk snapshots (not explained in this document).
4. Some useful debugging facilities.
5. An RPC interface to ask for certain logged events (`eventLogRequest`). This is mainly used for `status json`: the CC can use that to ask for metrics that are then forwarded to the client.

Detecting Failures

As in any distributed system, detecting failures is generally an unsolvable problem. Therefore we have to rely on timeouts of sorts to make a best effort. Generally, if a node doesn't reply to a request within a time window, any of the following could've happened:

1. The node failed (either the process or the machine died).
2. The actor that is listening to the corresponding endpoint died.
3. The node is very slow (for example because it is experiencing very high load and the actor that should reply to the request is starving).

4. There is a network partition - this is especially tricky as it could mean that other nodes in the cluster can talk to that node just fine.
5. The network is very slow or experience a high number of packet losses (FDB is using TCP - so these two scenarios will have the same effect).

FoundationDB uses heuristics to detect partitions and node failures. In many cases there are important trade-offs to consider when doing this. For example if the cluster controller believes that the master server died (due to a timeout), it will elect a new master which will cause a complete recovery. But recoveries have a large impact on the system (all in-flight transactions are aborted and no new transactions can be started for a short amount of time). Therefore, false positives are expensive. To keep the probability of such a false positive low, the cluster controller can wait longer for the master server to respond to heart beat requests. However, this means that whenever the master actually fails, it will take longer for the cluster controller to initiate a recovery which will in turn mean that downtimes are longer whenever the master fails.

This section describes several concepts implemented within FoundationDB that help a system to detect node failures and network partitions:

1. FailureMonitor is used to detect node failures. A node is marked as failed globally if it can't communicate with the cluster controller.
2. WaitFailure client/server is a service that is ran by some nodes. It allows to detect machine failures and actor failures. WaitFailure uses FailureMonitor to achieve this.
3. Ping messages within the transport layer. This is an orthogonal concept and is used to detect partitions. It is not used to mark nodes as failed. Instead it is used to implement at most once and at least once semantic for message delivery.

FailureMonitor

FailureMonitor is the mechanism used to detect failed nodes. A node is globally marked as failed if it doesn't talk to the cluster controller for more than 4 seconds (this is a bit simplified - consult the implementation `failureDetectionServer` to get a more precise definition). This means that the cluster controller is the authority that marks nodes as down.

FailureMonitor is implemented as a client and server pair. The cluster controller runs the server (`failureDetectionServer` in `fdbserver/ClusterController.actor.cpp`) and every worker runs the client (`failureMonitorClient` in `fdbclient/FailureMonitorClient.actor.cpp`). Clients also used run a version of the failure monitor but with less functionality - in more recent versions clients don't rely on this mechanism anymore (client functionality is not discussed in this document).

When a worker starts up, it starts the failure monitor client, which will send a request to the failure detection server (running in CC). It will receive a list of addresses as a response and all addresses are considered to be up. It will continue to send this request every 100ms and will get a diff of addresses that the CC considers to be up.

The main feature one gets from this is that each worker will maintain a list of addresses that it considers being up. Therefore, even if a worker can not communicate with another worker, it won't mark it as failed unless the failure monitor marks it as failed.

WaitFailure

Sidetrack: How `broken_promise` works across the Network

A `Future` will be set to `broken_promise` if all promises associated with it get destroyed. So if you `wait` on such a future this error will be thrown.

It is important to understand that this also works for futures if the corresponding `ReplyPromise` was sent to another process. This is implemented in `networkSender` (can be found in `fdbrpc/networksender.actor.h`). `networkSender` is an uncancellable actor and it is started whenever we deserialize a `ReplyPromise<T>`. This actor will wait on the Future (for the reply) and send back the result. If it receives a `broken_promise` (or any other error) it will forward this to the remote that holds the corresponding future.

WaitFailure Server and Client

The failure monitor can be used to detect failed processes (with a possibility to run into false positives). However, failure monitor won't detect whether a remote worker is still executing a certain role. This can be problematic in certain cases: consider the case where the actor that executes the master role dies but the process doesn't. The CC needs to detect this case (so that it can chose a new master and kick off recovery). This is where wait failure comes in.

WaitFailure is implemented as a client/server process. Each server can serve many clients. The implementation can be found in `fdbserver/WaitFailure.actor.cpp`.

WaitFailure Server

The server part is very simple: `waitFailure` is of type `FutureStream<ReplyPromise<Void>>`. The server will receive `ReplyPromise<Void>` and put that request into a queue. Usually it will never send back an answer (this is not quite correct, as there is an upper limit how large this queue is allowed to grow - but this is only an optimization).

So the only important function of the wait failure server is to hold on to these promises without destroying them. If the caller of `waitFailureServer` dies (and therefore this actor gets cancelled), all clients will receive an error of type `broken_promise`.

WaitFailure Client

The client combines the concept of promises being able to live on a different machine with the failure monitor.

To detect failing actors it would be sufficient to just send a reply promise to the wait failure server and wait on it. If the corresponding role fails we will receive a `broken_promise`.

However, this will only detect failed roles if the process that runs the role doesn't die as well. But we want to detect both. So instead of just waiting on that request, we also wait on the failure monitor. So for wait failure client to return one of two conditions must be true:

1. The client sent back a `broken_promise` or
2. `FailureMonitor` marked the remote as being down.

The problem with 2 is that it is timeout based and we could therefore see false positives. There are two counter measures:

1. We need to make sure, that whenever we use wait failure, that it also works if there is a false positive. As a consequence, a false positive will cause the cluster to do unnecessary work, but it mustn't introduce any bugs.
2. If we do something that is very expensive as a result of a failure, we might want to wait a bit longer before we start.

The first is the responsibility of the developers. For the second we can pass a reaction time and a reaction slope. The implementation of `waitFailureClient` calls into `RequestStream<T>::getReplyUnlessFailedFor` in order to achieve that. This function will wait on the future and the failure monitor.

Connection Monitor

We now have the ability to detect global failures and to detect actor failures of a non-failed process. However, we could still run into the issue that two processes can't talk to each other because of a local network partition.

For that purpose we start a connection monitor (`connectionMonitor` in `fdbrpc/FlowTransport.actor.cpp`) for each open connection. This connection monitor will ping the remote every second and close the connection after a timeout (marking the connection as failed - not the host).