## 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. ## Overview

FoundationDB supports safe, automatic failover in the event of a data center failure through multi-region replication, which is also known as Fearless DR. In a Fearless DR configuration, all commits are synchronously committed to transaction logs in multiple data centers before being confirmed to the client, and are asynchronously replicated to storage servers in multiple data centers. This approach allows you to make different choices about where you place the transaction logs and the storage servers to balance the requirements for low-latency commits with the requirements for geographically-distributed reads.

All of FoundationDB's multi-region configurations require at least three data centers in order to survive the loss of one data center.

## Concepts

In a multi-region configuration, you define two regions, and two or more data centers in each region. The data centers within a region should have a low enough latency for it to be acceptable to have that latency for every commit.

At any time, one region will be the primary region, meaning that commits are made synchronously to that region. The primary region will be where the master and cluster controller processes are recruited.

Regions can be assigned priorities, which tell the database which region you would prefer to use as the primary. Regions with higher priorities are preferred over regions with lower priorities. You can set a region's priority to be `-1` to indicate that the database should never recruit that region as the primary region.

Each region has a list of two or more data centers, which are identified by an `id`. This `id` should match the value you pass to the `locality_dcid` parameter for processes in that data center. You can configure whether a data center is a satellite or the main DC by setting the `satellite` flag to `1` or `0`. The satellites can also be assigned a priority, to indicate which satellite we should prefer to use when recruiting satellite logs.

You can also control how many satellite logs the database will recruit in each region through the `satellite_logs` parameter. This has a similar role to the `logs` field in the top-level database configuration, but it is specified on the region object instead.

The `satellite_redundancy_mode` is configured per region, and specifies how many copies of each mutation should be replicated to the satellite data centers.

`one_satellite_single`: Keep one copy of the mutation log in the satellite datacenter with the highest priority. If the highest priority satellite is unavailable it will put the transaction log in the satellite datacenter with the next highest priority.

`one_satellite_double`: Keep two copies of the mutation log in the satellite datacenter with the highest priority.

`one_satellite_triple`: Keep three copies of the mutation log in the satellite datacenter with the highest priority.

`two_satellite_safe`: Keep two copies of the mutation log in each of the two satellite data centers with the highest priorities, for a total of four copies of each mutation. This mode will protect against the simultaneous loss of both the primary and one of the satellite data centers. If only one satellite is available, it will fall back to only storing two copies of the mutation log in the remaining datacenter.

`two_satellite_fast`: Keep two copies of the mutation log in each of the two satellite data centers with the highest priorities, for a total of four copies of each mutation. FoundationDB will only synchronously wait for one of the two satellite data centers to make the mutations durable before considering a commit successful. This will reduce tail latencies caused by network issues

between data centers. If only one satellite is available, it will fall back to only storing two copies of the mutation log in the remaining datacenter.

At this time, FoundationDB only supports configuring one or two regions.

## Example: Three Data Centers

In this example, we have three datacenters: `dc1`, `dc2`, and `dc3`. Let's say we have the following latencies between our data centers.

| From DC | To DC | Latency |
|---|---|---|
| dc1 | dc2 | 10ms |
| dc1 | dc3 | 15ms |
| dc2 | dc3 | 12ms |

Let's also say that we want the primary to be in `dc1`, and our secondary to be in `dc3`.

To make this configuration work, we would define two regions: one where `dc1` is the main DC, and one where `dc3` is the main DC. In the first region, we would have `dc2` and `dc3` as possible satellites. In the second region, we would have `dc1` and `dc2` as possible satellites.

In the happy case, we commit synchronously to main logs in `dc1` and satellite logs in `dc2`. If the client processes are in `dc1`, that would mean we're adding about 10 ms to the commit latency in order to get the data to `dc2`. After a brief delay, the new commits would be replicated onto storage processes in `dc1` and `dc3`, and the commit data would be removed from the transaction logs.

If `dc2` goes down, the database will recover into a new configuration where we are putting the satellite logs in `dc3`. This would add a small amount to the commit latency. The database would become fully healthy after the few seconds it takes to recover.

If `dc3` goes down, the database will still be able to accept commits, but would not be able to replicate data into `dc3`. This would lead to a large volume commits being stored on the transaction log processes in `dc1` and `dc2`. This would not have an immediate impact on clients, but it would become a problem eventually. Setting the `usable_regions` configuration to `1` rather than `2` would disable replication into `dc3`, and prevent these commits from being batched up.

If `dc1` goes down, the database will recover into a configuration where commits are made synchronously to main logs in `dc3` and satellite logs in `dc2`. Any commit data that had not been replicated into `dc3` would be recovered from the copies stored in `dc2`. This would have similar risks as the scenario where `dc3` went down.

In all of these scenarios, if the failed data center comes back, we will automatically go back to the happy case configuration. If the `usable_regions` setting is changed, however, it will require manual reconfiguration to get back to the original configuration.

The region configuration for this would be:

```
[
    {
        "priority":1,
        "satellite_redundancy_mode": "one_satellite_double",
        "satellite_logs": 3,
        "datacenters": [
            {"id":"dc1", "satellite":0 },
            {"id":"dc2", "satellite":1, "priority": 2},
            {"id":"dc3", "satellite":1, "priority": 1}
        ]
    },
    {
        "priority":0,
        "satellite_redundancy_mode": "one_satellite_double",
        "satellite_logs": 3,
        "datacenters": [
            {"id":"dc3", "satellite":0 },
            {"id":"dc2", "satellite":1, "priority": 2},
            {"id":"dc1", "satellite":1, "priority": 1}
        ]
    }
]
```

## Example: Four Data Centers

In this example, we have four datacenters: `dc1`, `dc2`, `dc3`, and `dc4`. Let's say we have the following latencies between our data centers.

| From DC | To DC | Latency |
|---|---|---|
| dc1 | dc2 | 5ms |
| dc1 | dc3 | 60ms |
| dc1 | dc4 | 60ms |
| dc2 | dc3 | 60ms |
| dc2 | dc4 | 60ms |
| dc3 | dc4 | 5ms |

Let's also say that we want the primary to be in `dc1`, and our secondary to be in `dc3`.

In this case the latency between the first two DCs and the second two DCs is too high for us to accept in our commit pipline, so we will want to have a narrower set of options for satellites.

To make this configuration work, we would define two regions: one where `dc1` is the main DC, and one where `dc3` is the main DC. In the first region, we would have `dc2` as the only satellite. In the second region, we would have `dc4` as the only satellite.

In the happy case, we commit synchronously to main logs in `dc1` and satellite logs in `dc2`. If the client processes are in `dc1`, that would mean we're adding about 5 ms to the commit latency in order to get the data to `dc2`. After a brief delay, the new commits would be replicated onto storage processes in `dc1` and `dc3`, and the commit data would be removed from the transaction logs.

If `dc2` goes down, the database will fail over to use `dc3` as the primary DC, and `dc4` as the satellite DC. This will lead to higher latencies from clients in `dc1`, so the clients may need to do failover at their layer as well. Any data that has not been replicated into `dc3` will be recovered from the replicas in `dc1`. Once the database fails over, we will be able to asynchronously ship data to `dc1`, and the failed state of `dc2` will not cause any problems for the cluster.

If `dc3` goes down, the database will still be able to accept commits, but would not be able to replicate data into `dc3`. This would lead to a large volume commits being stored on the transaction log processes in `dc1` and `dc2`. This would not have an immediate impact on clients, but it would become a problem eventually. Setting the `usable_regions` configuration to `1` rather than `2` would disable replication into `dc3`, and prevent these commits from being batched up.

If `dc4` goes down, there will be no immediate impact on the database, but we would lose the capability to fail over to `dc3`.

If `dc1` goes down, the database will recover into a configuration where commits are made synchronously to main logs in `dc3` and satellite logs in `dc4`. Any commit data that had not been replicated into `dc3` would be recovered from the copies stored in `dc2`. This would have similar risks as the scenario where `dc3` went down. Unlike the scenario where `dc2` went down, in this scenario we will be unable to replicate commits onto the remote storage servers.

In all of these scenarios, if the failed data center comes back, we will automatically go back to the happy case configuration. If the `usable_regions` setting is changed, however, it will require manual reconfiguration to get back to the original configuration.

The region configuration for this would be:

```
[
    {
        "priority":1,
        "satellite_redundancy_mode": "one_satellite_double",
```

```
        "satellite_logs": 3,
        "datacenters": [
            {"id":"dc1", "satellite":0 },
            {"id":"dc2", "satellite":1, "priority": 1}
        ]
    },
    {
        "priority":0,
        "satellite_redundancy_mode": "one_satellite_double",
        "satellite_logs": 3,
        "datacenters": [
            {"id":"dc3", "satellite":0 },
            {"id":"dc4", "satellite":1, "priority": 1}
        ]
    }
]
```

## Process Count Recommendations

In locations that are only serving as satellites, you will only need to run log processes. These log processes will have to be spread across 3 failure domains, just like they would in the main DC. If you're running multiple FDB processes on each machine, then you may want to use smaller machines in the satellites to account for the fact that you'll only be running one per machine in the satellites.

It's generally best to run and recruit the same number of logs in the satellites that you have in the main DC. This will help to avoid performance and disk space issues when you're running in a degraded state where one of the main DCs is unavailable. In the happy case, you should be fine recruiting about 1/3 as many satellite logs as main logs.

If you are running a 3-DC config, you should be aware that there will be some situations where a data center needs to serve as a main DC and a satellite DC at the same time. This will require provisioning enough logs to be able to handle the number you are going to be recruiting for each of these rules. In this scenario you can either recruit the same number of main logs and satellite logs, which means you would need to provision twice as many, or recruit fewer satellite logs than main logs, which means that you could run into problems if you ran with only 2 DCs up for an extended period of time.

## Changing Region Configuration

When you change region configuration, the database applies additional checks to make sure that the change is safe and well-defined.

1. You cannot change the `usable_regions` setting and the `regions` setting in the same configuration command.
2. You cannot change the `usable_regions` setting when you have more than one region with priority >= 0. Another way of putting this is that you cannot change the `usable_regions` setting when automatic failover is enabled.

As a consequence of these checks, changing region configurations can be a multi-step process that must be carefully managed. After making a configuration change, you should wait for the database to become healthy to confirm that the replication changes and data movement have finished.

#### Example: Going from 1 region to 2 regions

1. Add second region with `priority: -1`.
2. Set `usable_regions=2`
3. Change priority on second region to `0`.

#### Example: Going from 2 regions to 1 region

1. Set priority on second region to `-1`.
2. Set `usable_regions=1`.
3. Remove second region

#### Example: Moving secondary region to a different data center

Because FoundationDB only supports a maximum of 2 regions, if you want to change the data center for one of the regions, you must go down to a single region before adding the new region. To change the secondary region to a different data center, you would follow the steps for "Going from 2 regions to 1 region", then follow the steps for "Going from 1 region to 2 regions".