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

This document is meant as:

1. An internal documentation for Snowflake. It is meant to contain what we learned and learn while running FoundationDB clusters of variable sizes on different cloud providers.
2. These things are potentially useful to other FoundationDB users and we try to either keep Snowflake specific information out of this document or point the requirements out clearly.

We are currently running FoundationDB deployments in several regions on Amazon Web Services (AWS), Microsoft Azure (Azure), and Google Cloud Platform (GCP). Most of the documentation here applies generally to all of them, but for examples and specifics it covers mostly AWS. If something is drastically different for other cloud providers, we'll try to point them out.

Choosing the right Disks

When running in the cloud a user typically has the option between several disk offerings. For AWS these are:

1. Ephemeral SSDs
2. Several flavors of Elastic Block Storage (EBS):
    1. EBS Cold HDD
    2. EBS Throughput Optimized HDD
    3. EBS General Purpose SSD (gp2)
    4. EBS Provisioned IOPS SSD (io1)

Generally speaking: ephemeral ssd are probably the best solution for most use cases as they provide the highest number of IOPS for the lowest price. This being said, EBS has some benefits. We'll explore the options below.

Ephemeral SSDs

For most deployments, this is probably what you want. It has the following benefits over EBS:

1. Very high number of IOPS compared to EBS.
2. Much cheaper.

The list of drawbacks is a bit longer, although as mentioned above, FoundationDB provides features to mitigate most of them:

1. **Less reliable**: disks can fail and if an ephemeral disk fails, it will lose all its data. FoundationDB can automatically recreate a lost copy. In theory however, you would lose all your data if a whole region would go down due to a power failure.
2. **Missing Features**:
    1. No Snapshots: if you want to do snapshots this will increase the operational work as you would need a filesystem that supports snapshots and you will need operational tools to copy these snapshots off the machine. EBS can do all of this for you.
    2. No machine resize: if you start/stop your machine, you will lose your data on that disk. This will make some operational tasks harder.

It is a good idea to use something like Fearless or DR in addition to backups if you run on SSDs. With EBS it is much less likely to lose three disks within a cluster. Backups are great to safe you against accidental corruption by a user while satelite tlogs will save you from disk and machine failures.

EBS

In general EBS can provide valuable benefits over ephemeral disks. However, they solve problems that FoundationDB already solves for you - so depending on your sensitivity to risk of data loss and generally your topology and additional safety mechanism EBS can either give you additional piece of mind and some valuable features or it could be a waste of money.

A typical FoundationDB cluster with EBS can be more than twice as expensive than a FoundationDB cluster with ephemeral disks - and you can expect slightly higher latency and lower throughput. This being said, there are some benefits:

1. **Much lower probability for data loss**: If Amazon loses machine due to a hardware failure, you will be able to just mount the data volume to another machine. If you use ephemeral disks, losing a machine means that you lose your data. This also means that you could safely run in double or even single replication mode. However, in practice you probably won't do that (it will still lower your availability and the high cost of EBS might offset the cost savings due to a lower replication factor).
2. **Fewer Disk Failures***: You generally can expect EBS to be more robust. Disk failures are usually transparent to the user (Amazon can often repair your disk without you noticing). Think of it as a managed RAID. This can save you some operational work. This being said: EBS volumes can break. However, if they break you usually will only lose availability and your data will be back after restarting the machine.
3. **Additional Features**:
    1. You can resize a machine and even change the machine type (for example from `m5` to `c5`) without data loss. This is not possible with ephemeral disk as it requires a stop/start of the machine (which will result in your ephemeral data to be lost). This will speed up some operational work significantly. On ephemeral disks you would need to use `include` and `exclude` to "wiggle" your data over to a new disk which will take significantly longer.
    2. **Snapshots**: FoundationDB 6.2 introduced a new snapshot feature which will can use file system snapshots to consistently snapshot a whole cluster. However, this feature requires the filesystem to support snapshoting. For Linux there are not many great solutions: ZFS is not part of the Kernel (and doesn't support KAIO), btrfs is not great for B-Trees (and FoundationDB has many B-Trees on disk), and LVM snapshots slow down disk operations. EBS supports snapshots and can even write them automatically to S3. This makes snapshots a compelling alternative to traditional backups.
    3. **Sizing**: If you get an ephemeral SSD, the size of the disk will typically be determined by the size of your VM (this is a bit more complex/flexible on GCP). With EBS you can get almost any size you want. Additionally you can even add more space on the fly (which

for GP2 will increase IOPS). However, a resize also needs to be supported by the file system.

EBS Cold HDD

This is not a good option for any processes that run FDB. However, it could be used to store backups - although S3 seems to be the better option for this. We generally don't use that.

EBS Throughput Optimized HDD

We didn't explore this offering yet. In general we believe that this would be a bad choice to run storage nodes on (we only run FoundationDB on SSDs). This being said: this could be an interesting for TLog as HDDs are cheaper than SSDs and writing a log on a HDD is often sufficiently fast. However, this has not yet been tested. One problem could be disk spilling: if TLogs spill, they will index the data in a b-tree, which could cause a significant slow-down.

EBS General Purpose SSD (gp2)

This is probably the most useful EBS offering for FoundationDB. In our experience it is as fast and as stable as io1 but much cheaper. The only real drawback is that the maximum per disk IOPS limit is lower for gp2 than for io1 and io1 is more elastic.

With gp2 you get a certain amount of IOPS per GB disk space and you get billed only depending on the size of the disk. The limits are usually high enough for a single storage process (though a storage process can saturate any EBS volume).

This is the EBS type that Snowflake uses.

EBS Provisioned IOPS SSD (io1)

io1 is similar to gp2 but mostly with different pricing. Instead of only paying for disk size, you'll also pay for provisioned IOPS. This makes this offering much more expensive. Price is the biggest drawback and for most users the benefits will probably not justify the much higher price. The benefits are:

1. **Higher IOPS limits**. io1 can give you up to 64,000 IOPS - gp2 can only scale up to 16,000 IOPS (at the time of writing - these numbers might change). However, there are two main arguments why this is not super useful for FoundationDB:
    1. There's also a machine-limit for IOPS that is determined by the network adapter that this machine provides. These limits can be found here. Only large instances will be able to actually utilize more than the max IOPS that gp2 provides.

2. Instead of having disks with many IOPS it is much cheaper to have several storage processes where each process gets a disk. So instead of having 4 storage processes running on a io1 disk with 64K IOPS, it will be cheaper to run for processes each having its own gp2 disk with 16K IOPS.

We also found that io1 is so expensive that it is usually cheaper to get machines with a lot of memory and set the cache-parameter for storage servers to a high value. This way, most read operations will be served out of memory and therefore won't consume any IOPS (and this is also faster). So only writes have to be calculated against this budget. This should probably work for most real-world workloads. Only if you have a lot of data and your workload is very uniform, this might not be a great option (most workloads have some significant skew though).

Machine Types

One great thing about running in the cloud is that it provides great flexibility to chose type and size of the machines you want to run. In general we think there are two valid strategies how to chose the right machine sizes: many small machines or few large machines.

Large machines have the benefit that you might not have noisy neighbors. It will also help to understand physical machine failures: if you have two i3.metal machines, you know that a machine failure will only bring down one machine. However, if you run 72 i3.large instances, a machine failure could cause anything between 1-32 virtual machines to fail.

Warning: Above argument about machine failures and big instances doesn't seem apply to GCP (at least at the time of writing). Google currently doesn't provide any guarantees about machine placement. So if a physical machine fails, it might look to you like a whole data center goes down.

This being said, this shouldn't be too much of an issue in practice for several reasons:

1. You should distribute your data across several AZs. Two VMs in two different AZs are guaranteed to run on two different machines.
2. You probably want to size your cluster in a way that you can survive the failure of a full AZ.
3. At least AWS provides an anti-placement feature you can use to get yourself some additional security.

In general small VMs have several benefits over large VMs:

1. If a VM crashes (or a disk fails) the amount of data FoundationDB has to re-replicate is small.

2. It increases elasticity: the smaller the machines, the smaller the steps for scaling are.
3. You can scale process types independently of each other.

A sample configuration would be (we assume the region you run on has at least 3 AZs):

- 36 storage nodes, one process per machine, i3en.large as machine type, with ephemeral SSD. You can use ~8GB of the memory for the page cache (`--cache_memory` argument to `fdbserver`)
- 6 Tlogs, one per machine on i3en.large
- 6 Stateless processes, one per machine, on m5.xlarge

For the rest of this documentation, we assume that this is the cluster we want to deploy - but making changes here should be simple.

Anti-Placement

AWS support Spread Placement Groups to ensure that multiple VMs run on different physical machines. One important limitation to spread placement is that only up to 7 EC2 instances can be in one spread placement group.

Currently FoundationDB can not make use of anti-placement which is a serious limitation. The main problem with that is that one might lose data if an AZ and a machine (in a different AZ) go down at the same time.

Currently solving this issue manually is possible by using a combination of these strategies:

1. As long as the cluster consists of fewer than 21 machines of each type, anti-placement can be used (7 machines per AZ).
2. One can use `.metal` machines (or the biggest VM available which will usually run on dedicated hardware). An important thing to look out for when doing this is that these machines often are NUMA-machines (`numactl` can be used to place `fdbserver` processes into NUMA regions).

This is obviously not optimal and we'll hopefully find a better fix for this. #2126 tracks this.

FoundationDB Configuration

Here comes the more interesting part: how to configure FoundationDB for maximum reliability. The main things to keep in mind is the following:

1. Whenever the FoundationDB documentation talks about data centers, it assumes that these data centers are geographically far apart. In AWS terminology this would correspond to a region.
2. FoundationDB doesn't (yet) have a concept of availability zones.

3. Zones in FoundationDB refer to a fault domain (this could be used for example to group all machines within a rack together). In a cloud environment we currently don't know of any users who make use of this.

Replication Mode

FoundationDB supports several replication modes and all of them are documented here.

The recommended replication mode for most users running in the cloud is `three_data_hall`. With this mode a cluster can survive the failure of one machine and one AZ. The policy works as follows:

1. For storages: each AZ holds exactly one copy of data. If an AZ fails, the cluster will be available and run with only two copies of data (until a third AZ comes back).
2. For TLogs: There can be at most two copies of data in one AZ and there are four copies in total. So if an AZ and a machine fail, there is one copy left.

It is important that there are enough processes of each class so that the cluster can recovery if one AZ and one machine fail. This means specifically that one should have at least three machines per AZ and each machine should run at least one tlog process.

Setting Localities in the Configuration

Locality is configured through the `foundationdb.conf` file:

```
[fdbserver]
command = /usr/sbin/fdbserver
public_address = auto:$ID
listen_address = public
datadir = /var/lib/foundationdb/data/$ID
logdir = /var/log/foundationdb
# logsize = 10MiB
# maxlogssize = 100MiB
# class =
# memory = 8GiB
# storage_memory = 1GiB
# cache_memory = 2GiB
# locality_machineid =
# locality_zoneid =
# locality_data_hall =
```

The only locality which is strictly needed is `locality_data_hall` (the `machineid` and `zoneid` is set to a random value by default and will be set on a per machine basis). All machines within the same AZ have to have the same data hall locality.