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. In this article we will implement a data structure on top of a FoundationDB database. We will focus on how to test our implementation, using pytest. This page assumes familiarity with the python bindings.

Hello World

We'll need an instance of a `db` object which obeys the contract of the FoundationBD client. Assuming we have that, we can write a simple test like this

```python
def test_simple(db):
    @fdb.transactional
    def simple(tr):
        tr[b'x'] = b'y'
        assert tr[b'x'] == b'z'
    simple(db)
```

We'll use a pytest fixture to provide a `db` implementation.

The simplest implementation would be this:

```python
@pytest.fixture(scope='function')
def db():
    yield fdb.open()
```

Putting it all together, here's our "hello world" test:

```
$ ls
test_simple.py

$ cat test_simple.py
import pytest
import fdb
fdb.api_version(620)

@pytest.fixture(scope='function')
def db():
    yield fdb.open()

def test_simple(db):
    @fdb.transactional
    def simple(tr):
        tr[b'x'] = b'y'
        assert tr[b'x'] == b'z'
    simple(db)

$ pytest
...
tr = <fdb.impl.Transaction object at 0x7fe207915ba8>

    @fdb.transactional
    def simple(tr):
        tr[b'x'] = b'y'
>       assert tr[b'x'] == b'z'
E       AssertionError: assert b'y' == b'z'
E         At index 0 diff: b'y' != b'z'
E         Use -v to get the full diff

test_simple.py:13: AssertionError
================================================== 1 failed in 0.07s ======================================
```

Indeed 'y' != 'z', and the test is working.

Cleaning up the keyspace

If you want to start each test with an empty keyspace, you might implement your fixture like this instead:

```python
@pytest.fixture(scope='function')
def db():
    @fdb.transactional
    def database_empty(tr):
        return len(list(tr.get_range(b'', b'\xff', limit=1))) == 0
    db = fdb.open()
    assert database_empty(db)
    yield db
    del db[b'':b'\xff']
```

This double checks that the database is empty at the beginning of the test (make sure you're using a test-only database), and clears the keyspace after the test.

Testing a "counted set" abstraction

A common design pattern is to record the number of key-pairs in a subspace in a separate key, and update it transactionally. This way you can get the count without doing a full range scan.

Here's an example implementation (counted_set.py)

```python
import fdb
fdb.api_version(620)
import struct


@fdb.transactional
def add_item(tr, set_name, item):
    keySubspace = fdb.directory.create_or_open(tr, (b'countedSet', set_name))
    if tr[keySubspace.pack((item, ))] == None:
        tr[keySubspace.pack((item, ))] = b''
        countSubspace = fdb.directory.create_or_open(
            tr, (b'countedSet', set_name, b'count'))
        tr.add(countSubspace.pack((b'count', )), struct.pack('<q', 1))


@fdb.transactional
def remove_item(tr, set_name, item):
    keySubspace = fdb.directory.create_or_open(tr, (b'countedSet', set_name))
    if tr[keySubspace.pack((item, ))] != None:
        del tr[keySubspace.pack((item, ))]
        countSubspace = fdb.directory.create_or_open(
            tr, (b'countedSet', set_name, b'count'))
        tr.add(countSubspace.pack((b'count', )), struct.pack('<q', -1))
```

```
@fdb.transactional
def cardinality(tr, set_name):
    countSubspace = fdb.directory.create_or_open(
        tr, (b'countedSet', set_name, b'count'))
    if tr[countSubspace.pack((b'count', ))] == None:
        return 0
    return struct.unpack('<q', bytes(tr[countSubspace.pack((b'count', ))]))[0]
```

The main invariant we want for this abstraction is that the cardinality is the same as the number of keys in the subspace.

We can test this invariant like so:

```
def test_cardinality(db):
    @fdb.transactional
    def debug_cardinality(tr, set_name):
        s = fdb.directory.create_or_open(tr, (b'countedSet', set_name))
        assert cardinality(tr, set_name) == sum(1 for _ in tr[s.range()])
    for i in range(10):
        add_item(db, b'mySet', i)
    debug_cardinality(db, b'mySet')
    for i in range(10):
        remove_item(db, b'mySet', i)
    debug_cardinality(db, b'mySet')
```

Exercising the retry loop

A possible bug in the implementation of `add_item` is to increment the count even if the key is already present, but currently our test will not catch this bug.

We can use client buggify to increase our test coverage with a simple change to our test fixture.

```
@pytest.fixture(scope='function')
def db():
    @fdb.transactional
    def database_empty(tr):
        return len(list(tr.get_range(b'', b'\xff', limit=1))) == 0
    db = fdb.open()
    fdb.options.set_client_buggify_enable() # add this line
    assert database_empty(db)
    yield db
    fdb.options.set_client_buggify_disable() # add this line
    del db[b'':b'\xff']
```

Now our test will catch the bug (you may need to run the test a few times)

This will cause your transaction retry attempts to backoff as if the cluster were under high load, so to speed up your tests you may want to consider lowering the max retry delay in your db fixture (again make sure you're using a test-only database, where a lower retry delay is appropriate).