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

This documentation gives a short introduction into how watches are implemented in FoundationDB and what kind of guarantees a client is getting. Furthermore, this document should give some insight into back practices.

An equivalent Functionality implemented in the Client

Understanding how watches are implemented is valuable to understand its performance implications. But in order to use watches correctly, one needs to understand the guarantees that watches provide. Understanding this part is easier if you think of watches as an optimization for a client feature.

Let's assume you want to be informed whenever the value for a certain key changes. You could do this without a watch by polling:

```
@fdb.transactional
def key_changed(tr, key, value):
    v = tr[key]
    if v ≠ value:
        return True
```

```
    return False

def watch_key(db, key, value, watch_sleep_time):
    while not key_changed(db, key, value):
        time.sleep(watch_sleep_time)
```

So this will just query the value in a loop and return as soon as the value changed (and keeps changed long enough for it to be observable). If you use this method you will quickly find that finding a good value for `watch_sleep_time` is very hard: if the value is very small each watch will put significant load onto the storages within the cluster. If the value is too large, it will take much longer until a change is observed.

It is important to understand that the only thing that database watches solve is that one can basically set this hypothetic `watch_sleep_time` parameter to `0` without paying for the polling. Or phrased differently: think of watches as an optimization rather than a feature.

Implementation Overview

In general there are two parts of this implementation: one part is implemented in the client and one is implemented on the server side. These play together to achieve the right functionality.

Watches are registered through a transaction and are registered with a storage server. The storage will keep all watches in a map internally. For each mutation that is executed on the storage, it will then look up corresponding watches from that map and, if there are any, will send the version at which the change was made back to the client.

Since the initial implementation of watches there have been a few optimizations that were done. The first is on the client side, where watches are tracked for a given key and a request is only sent to the storage server if one has not been sent for a previous watch with the same key/value pair. The second optimization is on the server, where the duplication of key/value pairs is reduced by collapsing the actors for same key/value watches.

Guarantees

A watch might get notified on changes. However, you can miss changes for two reasons: * The value changed more than once while the client or a storage server was in some faulty state and the watch had to be reregistered. In this case the watch might only fire once. * Watches are subjected to the ABA-problem: if a value changes twice and after the second update it has the same value as before, the client might not be notified of that update.

Failure Handling

Client Failures

If a client fails, the storage will keep that watch until it will be triggered the next time as the storage server will not be made aware of the fact that the client died. This is problematic, as a client could watch a key that never changes. To handle this case the storage server will eventually time out any watch that didn't see any writes for a long time (usually after ~15 minutes). If the client didn't fail, it will receive a timeout from the server and can reregister the watch (it will do so automatically).

Server Failures

Watches on storage servers are ephemeral. So if a storage server fails, it will lose all current watches. While clients are waiting for a watch reply, they will ping the server (by default once a second) - if this times out they will automatically reregister the watch on a different server.

Usage

The client side code for registering a watch and using the watch is often in this pattern:

```
    // First transaction waits for a change to a key
  state Reference<ReadYourWritesTransaction> tr(new ReadYourWritesTransaction(cx));
    state Future<Void> watchFuture;
    loop {
        try {
            ... // Optional<Value> value = wait(tr->get(someKey));

            watch = tr->watch(watchKey);
            wait(tr->commit());
            wait(watchFuture);
            break;
        } catch (Error& e) {
            wait(tr->onError(e));
        }
    }

    // Second transaction reads the key
    tr->reset();
    loop {
        try {
            // Optional<Value> value = wait(tr->get(watchKey));
```

```
            ...
            wait(tr->commit());
            break;
        } catch (Error& e) {
            wait(tr->onError(e));
        }
    }
```

Note the first transaction `wait` on the transaction commit and then `wait` on the `watchFuture`. Specifically, the `tr->commit()` in the first transaction sets up the `watchFuture` to receive changes from the committed version of the first transaction. As a result, if another transaction `Tx` modifies the `watchKey` after the first transactions, the above pattern guarantees that when `wait(watchFuture)` returns, `Tx` has already committed and the second transaction runs after it (thus will see the effect of `Tx`).