## 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. # Status

This proposal is currently on hold as it seems most users want the transaction to fail if the mutation doesn't have an effect.

## Semantics

The `compare_and_set` mutation would consist of a key, an expected former value, and a value to set the key to if the former value matches the expected former value. The expected former value can either be absent (meaning the key is not set) or a value to be compared with the former value. Likewise, the value to set if the comparison succeeds can be absent (indicating that the key should be cleared), or present (indicating the value should be updated). This way it can be used to set a key to a value only if it was previously absent.

### Caveats

Like other mutations, the transaction succeeds or fails solely based on the read conflict range of that transaction. The transaction committing success-

fully does not tell you any information about what the former value of the key was. Similarly, each of these mutations in a transaction would behave independently. For example, a scheme that attempts to swap the values of two keys using two of these mutations would not be sound, since it could be that only one of these mutations has an effect.

### Examples

`Key: "foo", ValueToCompare: "old", ValueToSet: "new"`

If the key "foo" is currently set to "old", then after applying this mutation the value would be "new". Otherwise this mutation would have no effect.

`Key: "foo", ValueToCompare: <absent>, ValueToSet: "new"`

If the key "foo" is currently not set, then after applying this mutation the value would be "new". Otherwise this mutation would have no effect.

`Key: "foo", ValueToCompare: "old", ValueToSet: <absent>`

If the key "foo" is currently set to "old", then after applying this mutation the key "foo" would not be present. Otherwise this mutation would have no effect.

## Proposed c api

```
DLLEXPORT void fdb_transaction_compare_and_set(FDBTransaction* tr,
                                               uint8_t const* key_name,
                                               int key_name_length,
                                               uint8_t const* former_value,
                                               int former_value_length,
                                          fdb_bool_t former_value_present,
                                      uint8_t const* potential_new_value,
                                          int potential_new_value_length,
                                 fdb_bool_t potential_new_value_present);
```

## Changes to MutationRef serialization

MutationRef is currently a typecode and two arbitrary bytestring payloads. We can keep this scheme, and match other atomic ops by using the first string payload as the key. The second bytestring payload can be serialized pair of optional values. The format can be an implementation detail. I think we can use either BinaryWriter or ObjectSerializer. To be more "future proof" we can consider using the ObjectSerializer, but I don't anticipate us evolving the schema.

## Protocol version compatibility

We may want to introduce this in a new protocol version so that we can be sure that the server understands it if the client sends these mutations.

## Downgrade considerations

In order to support downgrade, we would need to teach the previous protocol version how to read and interpret this mutation, but we should not allow clients at the previous protocol version to write this mutation. This is to avoid an old server on that protocol version not understanding a mutation from a new client on that protocol version. This allows us to add the ability to downgrade to an already-existing protocol version.

## Backup compatibility implications

Presumably backups taken on the new version with this feature would not be possible to apply to with older versions.

## Implementation details

WIP

### RYW cache

### Storage server

### Storage cache

### Other places that need to know how to interpret mutations?

## Test plan

1. Test combining this feature with backup and dr to make sure that applying a backup still works.
2. Add this to the `FuzzApiCorrectness` workload.
3. Add to binding tester.
4. Test downgrades with a workload that uses this mutation.
5. Add this to the `AtomicOpsApiCorrectness` workload.