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. # Mandatory Fields

Type - The name of the trace event.

LogGroup - The value of the `--loggroup` parameter passed to `fdbserver`. This is useful when trace logs from multiple databases are ingested into the same system.

Severity: * 10 - SevInfo - General information events * 20 - SevWarn - Indicates a generally rare event, but not indicate a actual problem * 30 - SevWarnAlways - Indicates an event that could be causing performance problems in a cluster * 40 - SevError - Indicates one of the assumptions of the database has been violated. Generally caused by hardware failures.

ID - The ID of the role which produced the event. For instance, messages logged by the master will contain the master's ID.

Machine - The IP address and port of the process which produced the event. For client trace logs the pid of the process is logged instead of the port.

Time - The time at which the event happened.

Roles - A comma separated list of two letter codes which correspond to all of the roles running on the process at the time the message was logged.

- StorageServer - SS
- TLog - TL
- MasterProxyServer - MP
- MasterServer - MS
- Resolver - RV
- ClusterController - CC
- Tester - TS
- LogRouter - LR
- DataDistributor - DD
- Ratekeeper - RK
- Coordinator - CD

Common Fields

Error - Events which are logged because of an error include the associated error code.

Backtrace - Some high severity events include a backtrace which provides the call stack of the system at the time the event was generated.

TrackLatestType - The trace files generated by FoundationDB will automatically roll after a period of time. Some events are re-logged every time the trace file is rolled, so we will always have a history of the last time the message was called. If the TrackLatestType is `Original` that means the event just happened. If the TrackLatestType is `Rolled` it means the event happened sometime in the past. For `Rolled` trace events, the `OriginalTime` field is the time at which the original event happened.

SuppressedEventCount - Many events have a maximum cap on the number of times they can be logged in a given time interval. Events that have this time of suppression include the number of events that were not logged in the next message which does get written.

Counters - Fields which have values that are three space separated numeric values are called counters. The first number is the the amount the value has increases per second since the field was last logged. The second number is a roughness, where larger numbers represent that the increases to this field are very bursty, and smaller values represent that the value has increase at a very regular interval. The third number is the total amount this value has increased since the start of the role logging the event.

Periodic Messages

TLogMetrics

The difference between `BytesInput` and `BytesDurable` is the amount of memory the TLog is using. Multiple TLogs can be recruited on the same process, but only one of them can be active. `SharedBytesInput` and `SharedBytesDurable` represent the combined memory used by all TLogs on the same process. Once this difference reaches 1.1GB batch priority transactions will be throttled. Once it reaches 1.5GB the TLog will start take writing mutations from memory to disk. Future reads for these mutations will need to read them from disk. Once this difference reaches 2.0GB Ratekeeper will start slowing down client traffic, with the goal of ensuring TLog memory does not exceed 2.4GB. `SharedOverheadBytesInput` and `SharedOverheadBytesDurable` show the amount of memory being used on overhead, and should generally be much smaller than the total amount of memory in use.

- Version
- QueueCommittedVersion
- PersistentDataVersion
- PersistentDataDurableVersion
- KnownCommittedVersion
- MinPoppedTagVersion
- MinPoppedTagLocality
- MinPoppedTagId
- KvstoreBytesUsed
- KvstoreBytesFree
- KvstoreBytesAvailable
- KvstoreBytesTotal
- QueueDiskBytesUsed
- QueueDiskBytesFree
- QueueDiskBytesAvailable
- QueueDiskBytesTotal
- PeekMemoryReserved
- PeekMemoryRequestsStalled

ProxyMetrics

- TxnStartIn
- TxnStartOut
- TxnStartBatch
- TxnSystemPriorityStartIn
- TxnSystemPriorityStartOut
- TxnBatchPriorityStartIn
- TxnBatchPriorityStartOut
- TxnDefaultPriorityStartIn
- TxnDefaultPriorityStartOut

- TxnCommitIn
- TxnCommitVersionAssigned
- TxnCommitResolving
- TxnCommitResolved
- TxnCommitOut
- TxnCommitOutSuccess
- TxnConflicts
- CommitBatchIn
- CommitBatchOut
- MutationBytes
- Mutations
- ConflictRanges
- KeyServerLocationRequests
- LastAssignedCommitVersion
- Version
- CommittedVersion
- CommitBatchesMemBytesCount

StorageMetrics

- GetKeyQueries
- GetValueQueries
- GetRangeQueries
- QueryQueue
- FinishedQueries
- RowsQueried
- BytesQueried
- WatchQueries
- EmptyQueries
- BytesInput
- BytesDurable
- BytesFetched
- MutationBytes
- SampledBytesCleared
- Mutations
- SetMutations
- ClearRangeMutations
- AtomicMutations
- UpdateBatches
- UpdateVersions
- Loops
- FetchWaitingMS
- FetchWaitingCount
- FetchExecutingMS
- FetchExecutingCount
- ReadsRejected
- LastTLogVersion

- Version
- StorageVersion
- DurableVersion
- DesiredOldestVersion
- VersionLag
- LocalRate
- FetchKeysFetchActive
- FetchKeysWaiting
- QueryQueueMax
- BytesStored
- ActiveWatches
- WatchBytes
- KvstoreBytesUsed
- KvstoreBytesFree
- KvstoreBytesAvailable
- KvstoreBytesTotal

ProcessMetrics

MachineMetrics

NetworkMetrics

MemoryMetrics

LogRouterMetrics

ClusterControllerMetrics

MovingData

TotalDataInFlight

DDTrackerStats

RkUpdate

RkUpdateBatch

DatacenterVersionDifference

SpringCleaningMetrics

Edge Triggered Messages

Generic Failure Info

SlowTask

Net2SlowTaskTrace

FailureDetectionStatus

ConnectingTo

ConnectionClosed

ConnectionTimedOut

ConnectionTimeout

ProgramStart

Role

TLogDegraded

LargeTransaction

ProxyCommitBatchMemoryThresholdExceeded

TooManyNotifications

TooManyStatusRequests

LargePacketSent

LargePacketReceived

HugeArenaSample

GetMagazineSample

SlowKAIOTruncate

SlowKAIOLaunch

KVCommit10sSample

StorageServerUpdateLag

TLogUpdateLag

TraceEventThrottle_*

TLS*

ServerTag

ServerTagRemove

Master Recovery

MasterRecoveryState

MasterRecovering

MasterRecoveredConfig

MasterRegistrationReceived

MasterRecoveryDuration

MasterTerminated

BetterMasterExists

CCWDB

TLog Recovery

TLogRestorePersistentState

TLogRestorePersistentStateDone

TLogRecover

TLogReady

TLogPoppedTag

TLogLockStarted

TLogLocked

TLogStop

GetDurableResult

GetDurableResultWaiting

Storage Server Recovery

Rollback

ReadingDurableState

RestoringDurableState

SSTimeRestoreDurableState

StorageServerStartingCore

Data Distribution

StatusMapChange

TeamHealthChanged

BuildTeamsBegin

ServerTeamRemover

MachineTeamRemover

DDInitTakingMoveKeysLock

DDInitTookMoveKeysLock

Coordination

NominatingLeader

ReplacedAsLeader

LeaderTrueHeartbeat

LeaderFalseHeartbeat

LeaderNoHeartbeat

ReleasingLeadership

Client Messages

ClientStart

TransactionMetrics