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. # GitHub Issue

https://github.com/apple/foundationdb/issues/2302

Forum Discussion

https://forums.foundationdb.org/t/rpc-layer-requirements-and-design/1817

Feature Definition

Feature Summary

Our client/server exchange relies on a custom protocol and the presence of sophisticated logic on the client side of that exchange. The proposed RPC layer would provide as an additional option a server that exposes a more standard RPC interface (e.g. using gRPC), with all of the complicated logic located on the RPC server. Client applications would be then be able to talk directly to these RPC servers without having to wrap our native client library or deal with configuring the multi-version client behavior.

Also as part of this feature, language binding implementations for RPC clients will be provided in at least Java and Python.

Terminology

- Client - An application, script, etc. that connects to and runs transactions against the cluster.
- FoundationDB client library - The native C library that currently provides the only way for clients to communicate with a FoundationDB cluster. The version of the library must match the version of the cluster, and any higher level language bindings are required to utilize this library (e.g. by using JNI in Java).
- Multi-version client - An API that provides the ability for a client to load multiple different FoundationDB client libraries at different versions. By configuring the multi-version client appropriately, clients are able to continue operating during an upgrade of the cluster. However, the details of this configuration are somewhat complicated and unintuitive. See the documentation here (https://apple.github.io/foundationdb/api-general.html#multi-version-client-api).
- Read your writes - The FoundationDB client library provides the ability to read values previously read or written in the same transaction from cache without interacting with the cluster. Because writes are not sent to the cluster until commit time, the ability to read mutations from this cache is required in order for a transaction to be able to read the writes it has made. The cache also provides a significant performance benefit when reading these keys but comes with the cost of significant implementation complexity in the client library.
- Cluster file - A file containing a connection string that is required in order to connect to the cluster. This consists of two identifier strings and a list of the coordinators in the cluster. See the documentation here (https://apple.github.io/foundationdb/administration.html#cluster-files).

Background

As mentioned above, the existing architecture designed around a native client library with a matching version to the cluster has a variety of shortcomings:

1. Clients cannot use pure implementations in their language and instead must wrap native libraries.
2. Configuring multi-version support is not intuitive and prone to mistakes. It is also unlike most other systems without a compelling reason to be

so (at least from the client perspective).

3. Language bindings are typically not shipped with the native library because of the common need to mix and match versions, further increasing the complexity of getting started. In many cases, this is also quite unusual for a user (e.g. in Node.js among others, one would typically expect that installing the foundationdb package would be sufficient and not require a separate installation).
4. The observability of the client library from a client application is not great. For example, there is no good way to integrate logs of the client library with those of the application, and the resource usage and behavior of the client library can lead to problems that are difficult for a user to diagnose.
5. The nature of the client library imposes some constraints on the API that otherwise might not be desired. For example, in Java it is required that Transactions be explicitly closed in order to free native resources, whereas it might be more natural to rely on garbage collection.
6. Details from the client library's trace logs are often useful to a cluster/service operator when debugging issues, but there is no built-in mechanism for these to be available to that operator. Instead, it is necessary for clients to make logs available externally if that is desired.

There are a few reasons for the current approach. First, the logic contained in the client library is fairly complex (for example, see read your writes) and would be difficult to implement and maintain if it were implemented in each language that wanted to use the cluster. Further, testing of these implementations would be much harder as they could no longer rely on our simulated testing.

Second, the protocol used to communicate between the client library and the server processes is a custom and unstable protocol with no cross-version compatibility support. Much of the reason for this is again related to testing, as the simulator does not support testing across versions. It's unstableness is largely a function of the complexity of interaction between the client library and the cluster, and in truth even if it were stable it would probably be undesirable for a client to have to implement all of it.

As extra reading, there are a variety of discussions on the forums related to the pain points discussed in this section:

1. https://forums.foundationdb.org/t/how-difficult-would-it-be-to-implement-the-wire-protocol-in-other-languages/69
2. https://forums.foundationdb.org/t/new-api-to-get-the-version-of-a-running-cluster-without-needing-a-matching-libfdb-c-library/437
3. https://forums.foundationdb.org/t/grpc-binding-gateway/1599
4. https://forums.foundationdb.org/t/using-foundationdb-without-installing-client-libraries/1667
5. https://forums.foundationdb.org/t/how-do-bindings-get-an-appropriate-copy-of-fdb-c/311 (https://forums.foundationdb.org/t/how-do-bindings-

get-an-appropriate-copy-of-fdb-c/311/7)

Detailed Feature Description

The RPC layer is intended to address the described shortcomings by exposing a simple and stable protocol with a straightforward versioning story. This will be done by implementing a server process that wraps the client library (or some internal component of it) and exposes a new RPC interface, which will shift the boundary between client and service in a way that eliminates complexity for clients and improves observability for the service operator.

There are other potential benefits that can be exposed by an RPC layer:

1. Assuming that the RPC endpoints are at known locations and an external discovery mechanism exists, it may not be necessary for clients to have the cluster file in order to connect to the cluster.
2. Rather than having every client application connecting to various processes in the cluster (proxies, storage, coordinators), these processes will instead connect to the RPC instances. This can help us to increase the number of clients supported by a cluster, depending on how many clients each RPC instance can support.
3. Having a well defined RPC boundary could make it easier to create a simple mock implementation of FoundationDB that can be used for testing, etc.

Testing

If we choose to depend on external libraries for RPC and serialization, it's likely that we won't be able to fully test the new functionality in simulation. We could potentially mock up an RPC request generator that would be usable in simulation, though.

The binding tester should be able to evaluate whether the results of a test client sequence are equivalent when run using the native client library and the RPC layer.

Our workload-based performance testing mechanism will need to be extended to support testers that interact with the cluster using RPC-based transactions in order for us to do adequate performance testing of the new features.

End-User Documentation

The following items should be documented with respect to using this feature as a client:

- Differences and trade-offs between using FoundationDB client library and RPC layer

- How to setup clients to discover and use RPC layer
- How upgrades are managed

Scale

It should be possible that any cluster is capable of running at least 2x more RPC server processes than would be needed to saturate that cluster, up to the current and projected legal cluster sizes.

We roughly estimate this would mean running at most as many RPC server processes as storage server processes in a cluster. Hopefully with more realistic workloads this would only require closer to one third the number of RPC server processes.

Because the number of RPC processes could need to scale up with the number of storage server processes, with each RPC instance potentially talking to all storage servers, there is an N^2 communication concern that may ultimately limit our scalability. This issue likely doesn't need to be addressed in the initial implementation, but may need to be considered in the design as a point for future improvement.

The RPC layer will provide an opportunity to increase the number of clients supported by the cluster. Because it serves as an intermediary between clients and the cluster, it can eliminate the current possibility that some processes in the cluster need to maintain connections with every client. We aren't setting a specific goal for the total number of clients supported via the RPC layer, though we should make an effort to measure how many we can have in practice.

Contract Requirements

The RPC layer will have a contract with at least the following requirements and guarantees:

- A transaction must take place entirely against a single RPC server process, which is responsible for maintaining transaction state.
- A transaction against a failed RPC process must be restarted. If a commit was in flight, it should be regarded as having an unknown result.
- Ordering of operations within a transaction as performed on the client must be preserved if the operations aren't commutative.
- A client is allowed to have multiple in-flight transactions at once.

Performance

There will be an additional performance cost associated with having to perform an extra hop between the client and the cluster. We expect the latency cost for this to be small (estimated 0.5ms upper bound) for each operation.

Operations that would be performed locally in the FoundationDB client library need to avoid introducing an extra serial RPC call when possible. There are exceptions to this, such as with reads to the read-your-writes cache, which will not be locally available on clients. Instead, reading values that have already be read or written will now require an RPC call. Other operations, such as reading the current transaction size, may also require an extra round trip in at least some cases.

Throughput of a single RPC server process should be comparable to that of a single client process using the FoundationDB client library directly. Client processes may be able to drive more load than before, as they will not have to worry about saturating the network thread.

Availability

When used, the RPC layer will be involved in all interaction between the clients and the cluster. As a result, its availability directly impacts the availability of the cluster itself from the perspective of the client, and it should therefore be possible to configure this layer such that it provides the same high availability guarantees of FoundationDB.

During an upgrade of the cluster, this layer should be able to maintain availability for clients without any action taken by them.

Security and Privacy Requirements

Data Storage

The RPC layer will not persist any data to disk. Its storage requirements should consist of a cluster file that does not need to be protected and a set of trace logs that may contain all or part of some keys, etc. It will be the responsibility of the platform running the layer to protect these logs, if required.

Privacy

Like any client of FoundationDB, the RPC layer instances will have access to the entire database. There are currently no planned intentions to build key-based access control or permissions into the layer, so users of the RPC layer will have full access to the cluster. Like with the current client, any such capabilities would need to be built on top of the RPC layer.

Because a single RPC instance may be handling transactions from multiple clients, it is possible that one of the RPC instances could be tricked into revealing data from one transaction to an unrelated client. Because access to the RPC layer instances grants access to the entire cluster, this may not be a significant issue.

Access Control

In order to communicate with the rest of the cluster, the RPC instances will be required to configure and use the FDB TLS plugin if the cluster requires it.

The RPC instances must also provide a mechanism for secure access from allowed clients, preferably including through use of mutual TLS. It may be desirable that this mechanism is configurable separately from that of the rest of the cluster, which would allow an operator to have independent restrictions to the RPC and non-RPC cluster interfaces.

Operational and Maintainability Requirements

Configuration

The RPC layer will need to provide the ability to configure the following:

- It should be possible to easily control the number of RPC server instances running against the cluster.
- Each RPC instance will need to be able to control the address it is listening on for RPC connections.

Optionally, there may be value in providing some additional configuration options:

- A mechanism to block/disable an RPC instance in the event that something is wrong with it.
- A limit to the number of clients that connect to each RPC server.
- A limit to the number of in-flight transactions or operations on each RPC server instance.

Operator Documentation

Additional documentation will be required to describe how to configure and run RPC instances. We will also need to document the new metrics produced in status, trace logs, etc. that are important for operating these instances.

KPI's and Health

New metrics/monitoring required:

- Number of RPC server processes
- Locality-aware fault tolerance of RPC server processes

- Cluster awareness of the dependence on RPC server processes (i.e. it should be knowable at some level whether the client is using the RPC server processes and would therefore require them to be present for availability).
- Counters for the various operations performed on each RPC server process (e.g. number of transactions, reads, mutations, commits, etc.)
- Latencies of different operation types as observed from the RPC processes. This could be used as a more accurate replacement for the cluster-side measurements used to measure against latency SLOs. In particular, it would include more parts of the request chain without being subjected to client-side effects.
- Statistics about the clients connected to a cluster via the RPC server processes
- The ability of RPC processes to connect to the cluster
- Maybe: version compatibility of each instance with the cluster

Additionally, any information about a process that could be useful for the purposes of indicating that clients should prefer other RPC instances (such as version compatibility mentioned above) should be published.

Roll-out

The intended procedure for rolling this feature out onto a cluster that is already operational would be as follows:

1. Add new RPC instances without modifying existing clients
2. Start new replacement clients connected to RPC instances
3. Migrate traffic from old clients to new clients
4. Shut down old clients

It may be the case that migrating traffic from old to new clients instead involves shutting down a client and restarting it with an updated version of the client code. Additionally, steps 2 and 3 could be done all at once or could be done in once for each client (i.e. you could restart clients one by one).

Other Considerations

Dependencies

The implementation of this feature may involve the use of 3rd party RPC and serialization libraries, such as gRPC and protobuf.

Our choice of the libraries to use will consider the performance, ease of integration with flow, and ease of use by clients (including language support).