Introduction

As we all know, certain kind of bugs within FoundationDB can have catastrophic consequences if they are triggered in a production environment. We currently rely mostly on testing to find those bugs. However, while we believe our testing is very thorough, it is clear that it can only prove the existence of bugs - never the absence of bugs.

Additionally we should make changes to FoundationDB to make it less likely that a bug can cause catastrophic damage. Optimally, bugs should only be able to cause crashes or stalls, but not data corruption or data loss.

We already have some mechanisms in place that try to do this. One example is checksumming in storages. Another example is the concept of file identifiers in flatbuffers (so if we try to read a message of a wrong type there's a high chance we crash in flatbuffers - while the old streaming serializer would just read garbage).

This wiki page is meant as a place to brainstorm ideas on how to reduce the probably that bugs can cause catastrophic failures. Ideas shouldn't be limited to new features but can also include new testing methodologies. However, in this document we are not interested in ideas on how to reduce the number of bugs, only how to make FoundationDB more robust against bugs.

Idea 1: Add invariant checking

Each component, e.g, proxy and tLogs, may have invariants for each of its functionalities. For example, a tLog should not receive mutations whose tags are not for the tLog.

It has two benefits: a) Crashing the system early when invariant is violated can help uncover potential data corruption situation.
b) The invariant failure can help uncover the root cause of a simulation failure and production failure. This saves time both in development and in production error diagnosis.

This requires code change and expertise in each component. It can be hard to find these invariants. Linux kernel development uses invariants to crash the system early instead of letting a system wobble in undetermined state.

Idea 2: Add verification Engine

This is a similar to idea 1 but might go a bit farther.

The proxies currently keep the txnStateStore in memory and every transaction (whether system- or user-transaction) has to go through a proxy. Additionally, every message written to a TLog is written by a proxy.

This means that we could use (and maybe additionally introduce) some data redundancy in the txnStateStore that would allow us to do further verification. After we resolved a batch of transactions we could then run every batch through this verification engine. Afterwards we could also run all TLog message through this engine before we actually write to the tlog. This would allow us to catch certain type of bugs early and simply crash the proxy before we write anything to a disk.

Examples for things we could verify:

- We could verify that we don't duplicate messages to a tag.
- We could verify that shard assignment messages and private mutations go to tags that make sense (for example when removing a shard from a tag, we can double-check there that this corresponds to the shard-mapping we currently have).
- For replication factor `N`, we could verify that every non-private message goes to at least `N` number of tags and `N` number of tlogs and that we don't violate the replication policy.

Idea 3: Buggify++ (Bug Injection in Simulation)

Similar to buggify, we could introduce byzantine failures in several places of the code (for example, in the serialization code we could randomly deserialize wrong data). As FDB is by definition not resilient against these kind of failures, test runs will need to run differently. This is what I would imagine could work well:

1. We would run a simulation without bug injection enabled and run `SaveAndKill` at the end.
2. We would then run a restart test with bug injection enabled. In this run, we would ignore all Sev 40s (as they are excpected) and we won't expect any progress. Basically the hope is that this will crash at most but it won't change the on disk state in any harmful way.
3. Last there would be another restart test. This would then verify that no data corruption has happened in the previous run.