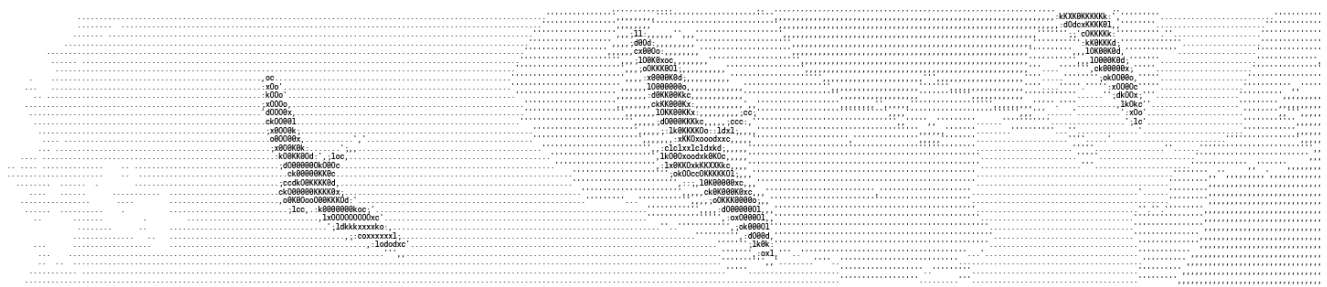


Testing sync at Dropbox

 dropbox.tech/infrastructure/-testing-our-new-sync-engine



1. Testability
2. Randomized Testing
3. CanopyCheck
4. Trinity
5. Conclusion

...and how we rewrote the heart of sync with confidence.

Executing a full rewrite of the Dropbox sync engine was pretty daunting. (Read more about our goals and how we made the decision [in our previous post here](#).) Doing so meant taking the engine that powers Dropbox on hundreds of millions of user's machines and swapping it out mid-flight. To pull this off, we knew we would need a serious investment in automated testing. Our testing strategy gave us confidence that we were on the right track throughout the rewrite, and today it allows us to continue building and shipping new features on a quick release cycle.

First, we'll discuss the types of testability considerations that went into the design of Nucleus, our new sync engine, and then we'll get into some of the randomized testing systems that we built on top of our test-friendly architecture.

Testability

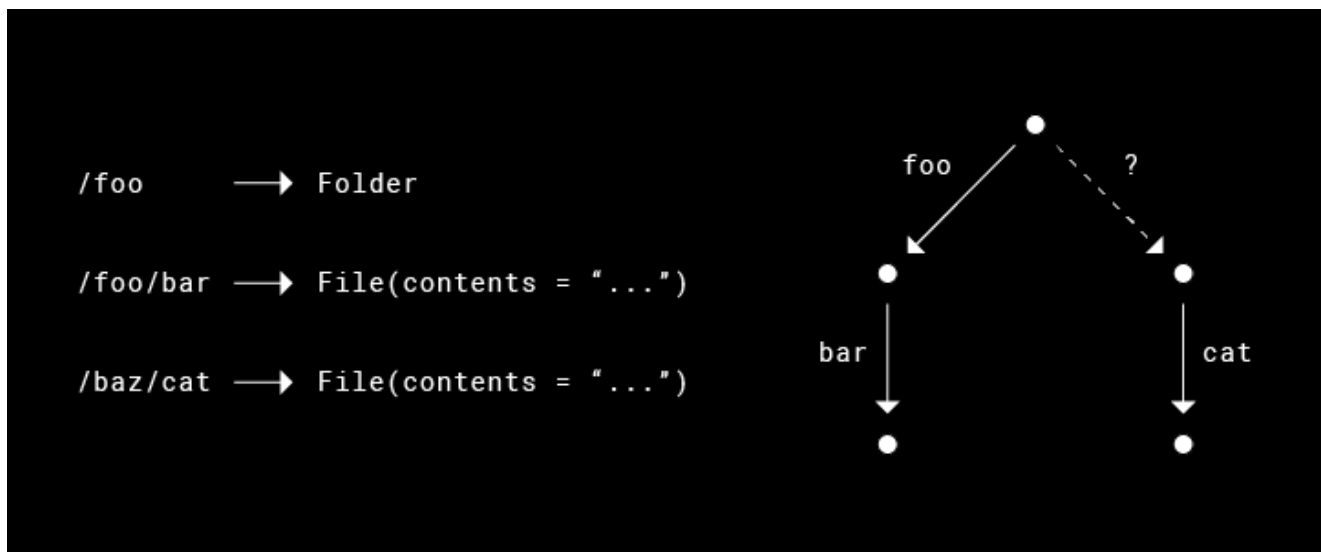
When we embarked on the rewrite, one thing was clear: to have a robust testing strategy, the new system would have to be testable! Emphasizing testability early, even before implementing the associated testing frameworks, was critical to ensuring that our architecture was informed appropriately. But what does testability even mean?

We can look to [Sync Engine Classic](#), the legacy system, for insights. Why wasn't the old system testable? What made it so hard to avoid regressions and maintain correctness in that system? And what did we learn that informed the architecture of our new system?

Protocol and data model

For one, the server-client protocol and data model of Sync Engine Classic were designed for a simpler time and a simpler product, before Dropbox had sharing, before Dropbox had comments and annotations, and before Dropbox was used by thousand-person enterprise teams. Dropbox has evolved quite a bit in the 12+ years since we designed the original system, and the requirements have changed greatly.

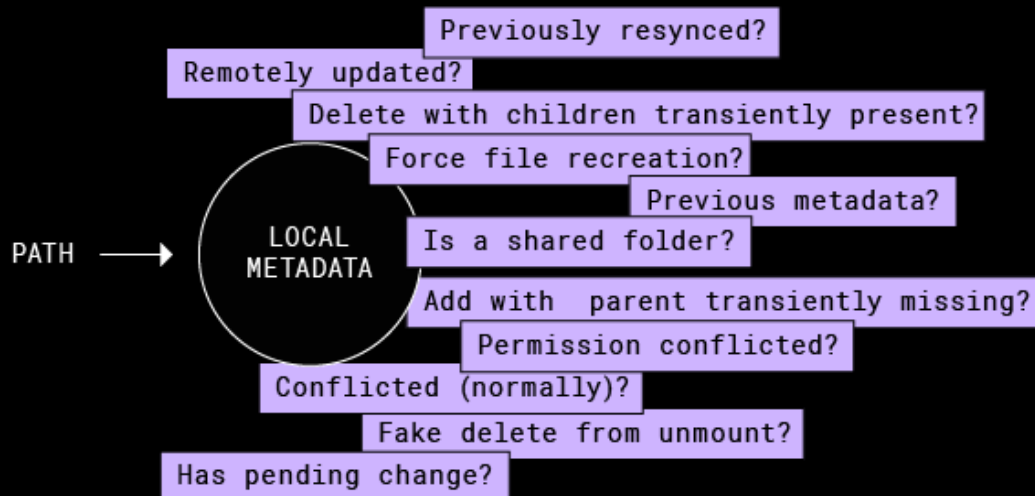
Sync Engine Classic's client-server protocol often resulted in a set of possible sync states far too permissive for us to be able to test effectively. For example, a client could receive metadata from the server about a file at `/baz/cat` before receiving its parent directory at `/baz`. Correspondingly, the client's local database (SQLite) needed to represent this orphaned state, and any component that processed filesystem metadata needed to support it. This in turn made it impossible to distinguish many types of serious inconsistencies (for example, orphaned files) from a client merely being in some acceptable transient state.



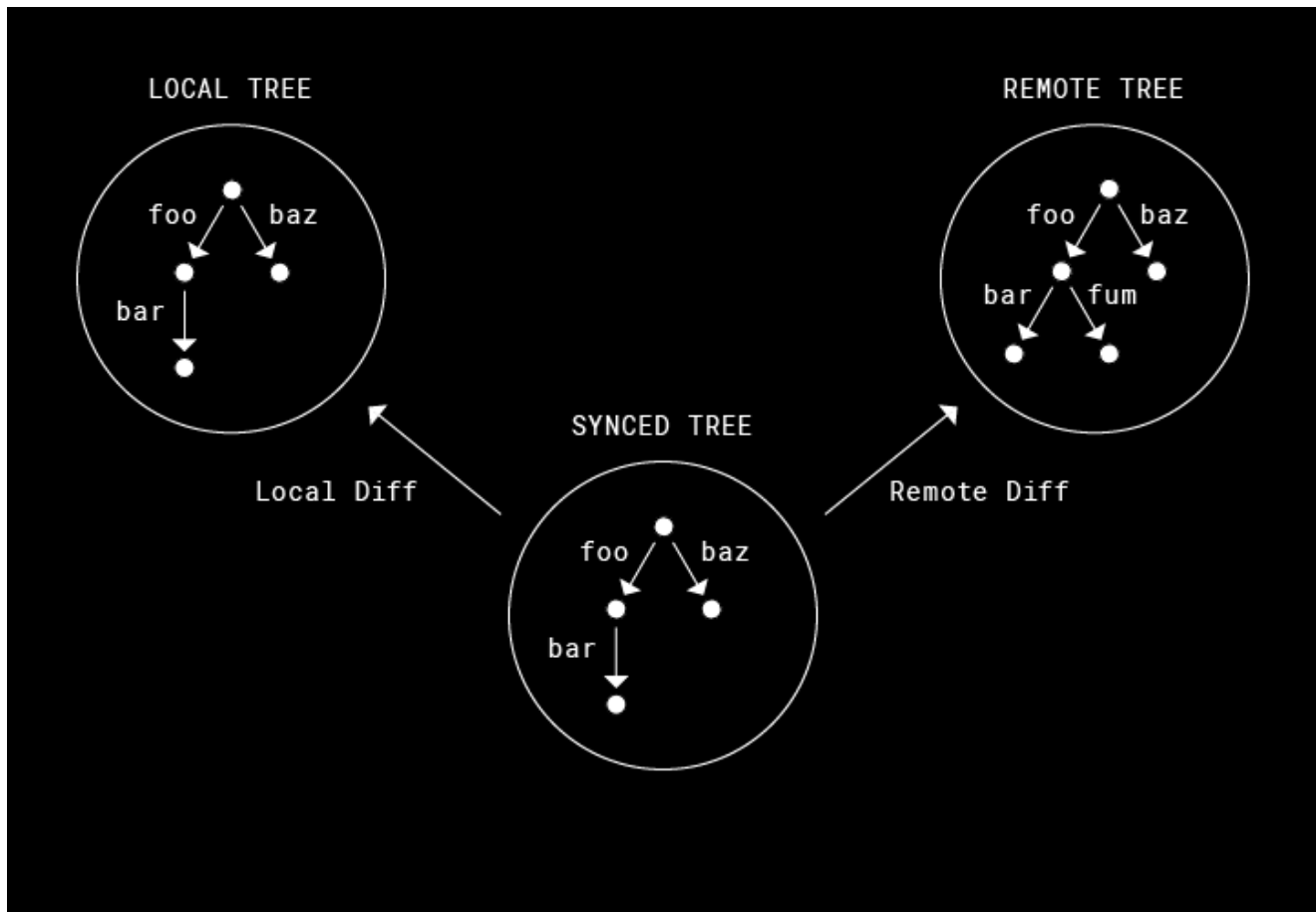
In Nucleus, the protocol prevents us from getting into this state to begin with! In fact, one of our core architectural principles is “Design away invalid system states.” (In a future post, we’ll discuss how we also leverage Rust’s type system to support this principle.) In the case of the stray cat, we report a critical error at the protocol level before the client can enter such a state. The persisted data model and higher-level components need not consider such a possibility. The additional strictness affords a new, testable invariant: in the database no file or folder can exist (even transiently) without a parent directory.

Sync Engine Classic and Nucleus have fundamentally distinct data models. The legacy system persists the outstanding work required to sync each file or folder to disk. For example, it stores whether a given file needs to be created locally or if it needs to be uploaded to the server. By comparison, Nucleus persists *observations*. Instead of representing the outstanding sync activity directly, it maintains just three trees, each of which represents an individually-consistent filesystem state, from which the right sync behavior can be *derived*:

- The *Remote Tree* is the latest state of the user's Dropbox in the cloud.
- The *Local Tree* is the last observed state of the user's Dropbox on disk.
- The *Synced Tree* expresses the last known "fully synced" state between the Remote Tree and Local Tree.



Sync Engine Classic data model



Nucleus data model

The Synced Tree is the key innovation that lets us unambiguously derive the correct sync result. If you're familiar with version control, you can think of each node in the Synced Tree as a merge base. A merge base allows us to derive the *direction* of a change, answering the question: "did the user edit the file locally or was it edited on dropbox.com?" In the graphic above, we can derive that the file at path `/foo/fum` was added remotely because the Local Tree (on disk) matches the merge base expressed by the Synced Tree. Without the Synced Tree, we wouldn't be able to distinguish this scenario from if the user had actually deleted `/foo/fum` locally.

We arrived at this data model because it is extremely testable! With this data model, it is easy to express a key goal of the system: to converge all three trees to the same state. When the user's local disk looks the same as dropbox.com (i.e., Local Tree matches Remote Tree), sync is complete! It allows us to enforce strict invariants—for example, no matter how the three trees are configured at the beginning of a test, all three trees must still converge.

In the Nucleus data model, nodes are represented by a unique identifier. This is in stark contrast to Sync Engine Classic, which keyed nodes by their full path. As a result, in the legacy system, a rename of a file was represented in the database as a delete at the source path and an add at the destination. For folders, this meant exploding a single move into $O(n)$

deletes and adds for all descendants! This would then result in a series of successive deletes and adds on disk. Until all of a directory's descendants had been deleted and added (one at a time), the user would see two inconsistent subtrees.

In Nucleus, the system is stricter. Because nodes are represented with a unique identifier, a move is just an update to the moved node's attributes in the database. This update is then replicated to the filesystem with a single atomic move, providing us with an additional invariant that is enforced in testing—any moved folder is visible in exactly one location.

Concurrency model

The concurrency model of Sync Engine Classic made testing extremely challenging and was another area we were particularly determined to get right the second time around.

In Sync Engine Classic, components were free to fork threads internally. This meant that when it came to execution order, we were completely at the mercy of the underlying OS. Coordination between components was accomplished through a series of global locks. Timeouts and backoffs were hard-coded. As you can imagine, this frequently resulted in flaky tests and frustrating debugging. To address these testability shortcomings, tests often resorted either to sleeping an arbitrary amount of time or to manually serializing execution via invasive patching and mocking.

In Nucleus, we sought to make writing tests as ergonomic and predictable as possible. Nearly all of our code runs on a single “control” thread. Operations benefiting from concurrency (e.g. network I/O, filesystem I/O, and CPU-intensive work like hashing) are offloaded to dedicated threads or thread pools. When it comes to testing, we can serialize the entire system. Asynchronous requests can be serialized and run on the main thread instead of in the background. Say goodbye to flaky tests! This single-threaded design is the key to the determinism and reproducibility of our randomized testing systems.

Randomized Testing

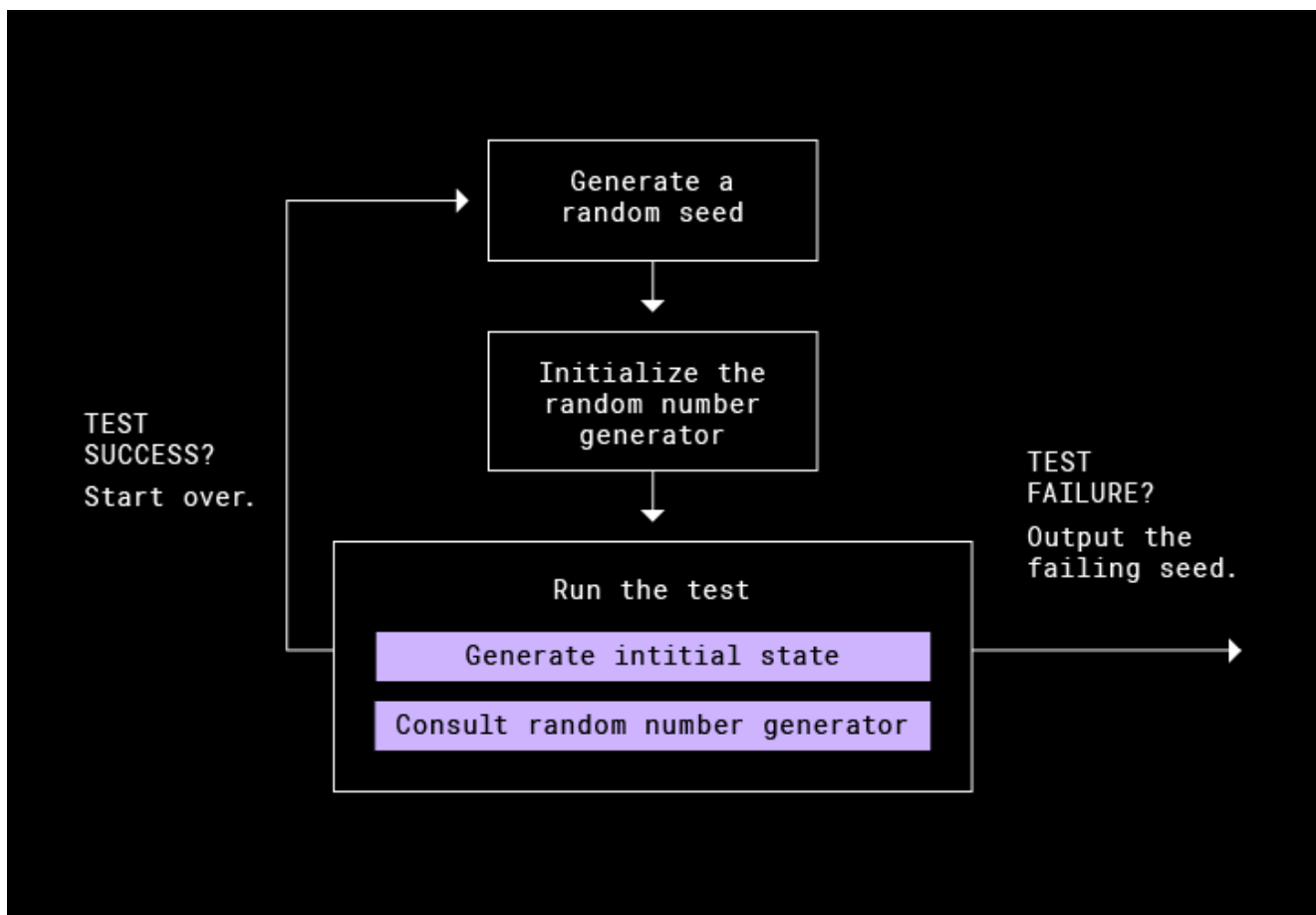
Randomized testing is the most essential part of our testing strategy. As with any complex system, just when you think you've covered all the bases, you discover another edge case. And another. And another. Dropbox runs on hundreds of millions of users' machines, each a wildly different environment. We often tell new engineers on the team: even the most obscure corner cases *will* show up in the wild. Regular unit and integration tests and manual testing simply won't cut it, since humans just aren't that great at proactively anticipating edge cases. Randomized testing is what gives us the confidence that our system is truly robust.

Depending on your past experiences, “randomized testing” might not be your favorite phrase. Many randomized testing systems are sources of extreme frustration—failing only intermittently, in ways that are impossible to reproduce. And when a failure isn't easily reproducible, you have no hope of adding more logging or breakpoints in order to diagnose

it. In fact, Sync Engine Classic had more than a few of these systems! We spent many hours poring over randomized testing logs only to find that the logs we *really* needed simply weren't present, which blocked any further investigation. So with Nucleus, we were determined to build randomized testing systems that could not only give us the coverage we desired but do so in an ergonomic way. The developer experience was paramount. To this end, we set a challenging requirement for ourselves: **All randomized testing frameworks must be fully deterministic and easily reproducible.**

In order to provide the desired determinism guarantees, all our randomized testing systems share the following structure:

1. At the beginning of a random test run, generate a random seed.
2. Instantiate a pseudorandom number generator (PRNG) with that seed. (Personally, given its name, I like [this one](#).)
3. Run the test using that PRNG for all random decisions, e.g. generating initial filesystem state, task scheduling, or network failure injection.
4. If the test fails, output the seed.



Every night we run tens of millions of randomized test runs. In general, they are 100% green on the latest master. When a regression sneaks in, CI automatically creates a tracking task for each failing seed, including also the hash of the latest commit at the time. If an engineer

needs more logging to understand what happened in the test run, they can simply add it inline and re-run the test locally! It's guaranteed to fail again.

In order to uphold this guarantee, we take great care to make Nucleus itself fully deterministic, provided a fixed PRNG input. For example, Rust's default HashMap uses a *randomized* hashing algorithm under the hood to resist denial of service attacks that can force hash collisions. However, we don't need collision-resistance in Nucleus, since an adversarial user could only degrade their *own* performance with such an attack. So, we override this behavior with a deterministic hasher to make sure all behavior is reproducible. Note also the importance of the commit hash, as another type of "test input" alongside the seed: if the code changes, the course of execution may change too!

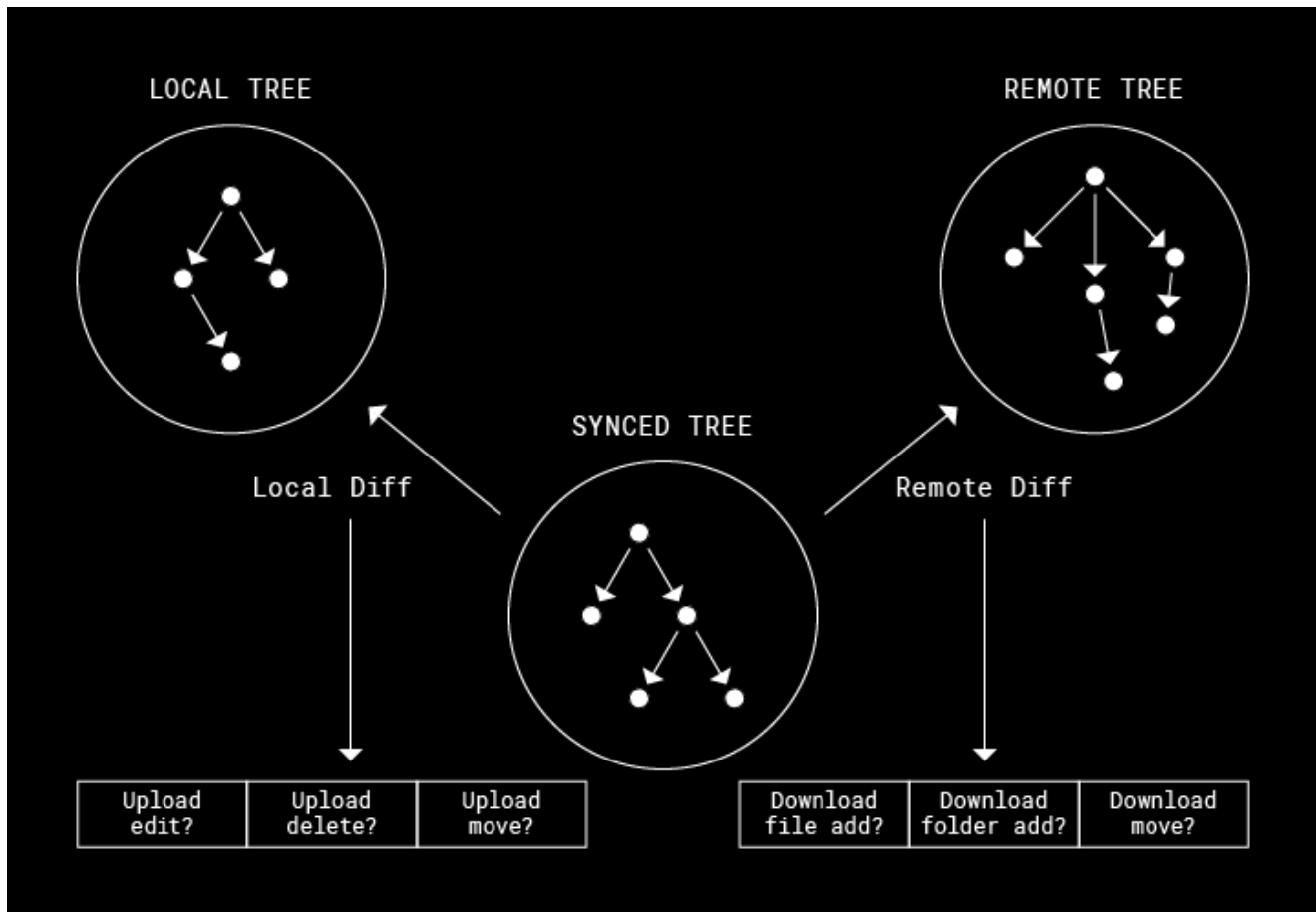
In this post we'll discuss two of the randomized testing systems protecting Nucleus: *CanopyCheck*, which tests our ability to bring the three trees into sync, and *Trinity*, which tests the concurrency of the engine at large.

CanopyCheck

Not all randomized testing must be end-to-end. Building narrowly-tailored randomized testing systems to exercise particular components can offer increased coverage and also allow for asserting stronger invariants. CanopyCheck is specifically designed to identify bugs in the *planner*.

Planner

The planner is the core algorithm behind sync at Dropbox. As input, it takes the three trees—the core of Nucleus's data model—which together we refer to as "Canopy." Recall that the three trees express the current state of the user's Dropbox on the server, the current state of the user's Dropbox on disk, and a merge-base expressing the sync progress made so far. The planner's job is to output a series of operations to incrementally converge the trees. For example, "create this folder on disk," "commit an edit to this file to the server," etc. It batches these operations into groups that are safe to be executed concurrently; for example, it knows that a file cannot be created until after its parent directory has been created, but that it is safe to concurrently edit or delete two sibling files.



It's essential that the planning algorithm be correct. Here's an example of a basic handwritten unit test we have for the planner. It also shows off how heavily we use Rust's macro system for testing ergonomics: under the hood, the `planner_test!` macro initializes the trees, generates operations according to the planner, successively updates the trees to reflect each operation's result, and asserts the final equality property shown, as well as some internal consistency properties.

```

#[test]
fn test_remote_add() {
    planner_test! {
        initial synced, local: {
            /foo: 1 = Directory,
            /foo/bar: 2 = File contents: hello,
            /baz: 4 = Directory,
        }
        initial remote: {
            /foo: 1 = Directory;
            /foo/bar: 2 = File contents: hello,
            /foo/fum: 3 = File contents: world,
            /baz: 4 = Directory,
        }
        final remote, synced, local: {
            /foo: 1 = Directory,
            /foo/bar: 2 = File contents: hello,
            /foo/fum: 3 = File contents: world,
            /baz: 4 = Directory,
        }
    }
}

```

Note: The above test verifies that the planner emits an appropriate plan to download a remotely added node.

As you might expect, we have hundreds of tests like this that put the three trees into various configurations and assert that the final state determined by the planner is acceptable. But the number of possible non-trivially distinct tree configurations is astronomical, even if we limit ourselves to small trees. How can we be confident that the planner handles all possible inputs appropriately? Enter CanopyCheck.

Initialization

CanopyCheck is a testing framework that generates test cases like the example above. It randomly generates the initial trees and programmatically infers at least some subset of what the final trees should look like.

Generating random inputs isn't trivial: if we generated three random trees independently, we would fail to exercise interesting scenarios. In other words, if the three trees had completely disjoint sets of files at non-overlapping paths, the planner wouldn't exercise any of its logic for deletes, edits, moves, etc. Instead, we first randomly generate one tree, and then we randomly perturb it to arrive at the other two. This better explores the space of all possible sync cases at a high-level while still getting good random coverage of the specifics.

Once we have three trees randomly generated, we can start planning!

Runtime

The run loop of a given random CanopyCheck test looks like this:

1. Ask the planner for a batch of concurrent operations.
2. Randomly shuffle the set of operations (to verify that order doesn't matter).
3. For each operation, *pretend* the operation succeeded by updating the trees accordingly.
4. Repeat, until the planner returns no further operations.

In step 3, CanopyCheck drives the sync process forward, but without actually performing the requested operations—removing any need to mock components or worry about concurrency. If everything goes according to plan, after some finite number of iterations of this loop, all three trees will have converged and we will be synced!

Invariants

The framework looks simple, given that it skips out on a lot of the juicy parts of Nucleus like I/O and concurrency, but it's quite powerful nevertheless. With CanopyCheck, we can verify all sorts of invariants.

Termination

Sync is an incremental process. First we do one batch of operations, then another, then another, until we're synced. But how do we know that sync will ever terminate? What prevents us from accidentally looping and infinitely syncing? CanopyCheck verifies that, no matter how crazy or contrived the input, the planner produces no infinite loops (approximately speaking, of course, using a heuristic cutoff of 200 planning iterations).

No panics

We liberally assert! in the planner (and throughout Nucleus in general) to be as defensive as possible. CanopyCheck provides early, comprehensive coverage of these asserts, assuring us they won't result in runtime panics in the wild.

Early on in Nucleus's development, this alone caught an enormous number of bugs, exposing flawed assumptions in our design that drove us back to the drawing board. In fact, it was CanopyCheck that exposed the Archives/Drafts/January directory cycle bug we described in [our previous blog post](#). CanopyCheck was able to find this condition where applying a local move and a remote move together created a cycle. The seed then triggered an assertion error within our tree data structure and failed the test.

Sync correctness

We enforce that all three trees are equal at the end of a test run. This is the definition of sync! But if this is all we enforced, we'd be vulnerable to really extreme bugs: suppose the planner always went out of its way to blow away everything on dropbox.com and on the user's disk. No matter the input, we'd always end up with three identical *empty* trees, yet the test would still pass! Thus, in addition to requiring that all three trees converge, we also

enforce numerous additional correctness invariants. Because these are randomized tests, it's important that the invariants be simple enough to apply in all cases, yet aggressive enough to be consequential. Typically some (ideally simple) property of the three trees is derived at initialization, and then a related property is enforced at the end of the test.

For example, one invariant we enforce is that if a given file exists only in the Remote Tree, but not in the other two trees (i.e., the file is only available on the server), then it must exist in all three trees at the end of a test run. Intuitively, this captures the notion that whenever there's unsynced data on the server, it must be synced down to the user's disk. Deleting it would be an error! We enforce the symmetric local invariant, for uploads, as well. Another example invariant pertains to Smart Sync: we verify that any locally added file remains downloaded as long as it's not moved into an "online only" folder. This ensures that the planner doesn't evict local contents from disk prematurely.

Minimization

The name "CanopyCheck" is a reference to [QuickCheck](#), a Haskell testing library. Like QuickCheck, when CanopyCheck finds a failing test case, it next attempts to find a minimally-complex input that reproduces the failure.

The test's simple input format (the three trees) allows a very natural approach to minimization. Suppose a test case failed some of the correctness invariants described above, or perhaps it ran for too many iterations, suggesting an infinite loop. Upon finding the test case, CanopyCheck searches for the minimal input by iteratively removing nodes, thereby shrinking the trees' initial state, and checking if the failure persists.

Because the initial randomly generated input is often quite convoluted, this process is extremely valuable to us developers. Tracking dozens of nodes across three trees is simply not easy for a human to do, as the real bug is often hidden alongside many red herrings. This minimization removes such distractions, often making it much easier to diagnose at a glance that, for example, "oh, we're not handling the case where the user adds a node under a parent directory that was moved remotely."

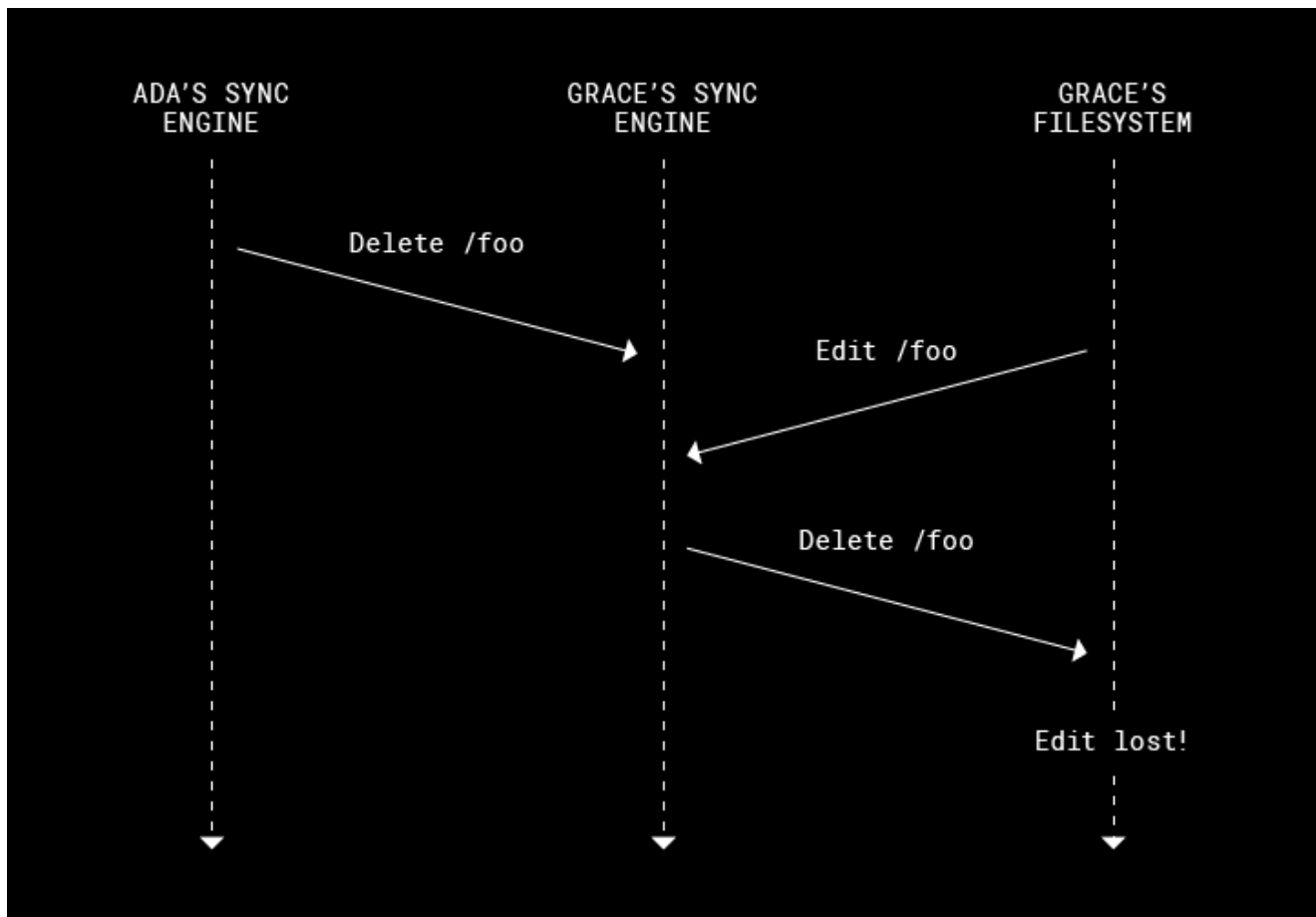
Trinity

CanopyCheck is very effective at testing our planning algorithm, but there is much more to Nucleus that still needs testing coverage. This is where Trinity comes in.

The trickiest bugs in sync are often caused by subtle race conditions, appearing only in rare circumstances when operations align with particularly problematic timing or ordering. Here's an example:

- In a shared folder, Ada deletes foo on her machine.
- Grace's sync engine learns that foo should be deleted.

- Simultaneously, Grace writes some new data into foo on her computer.
- Grace's sync engine erroneously deletes foo, clobbering her recent change.



Losing data the user intended to sync to Dropbox must be avoided at all costs. Trinity helps us discover races of this sort (and, of course, more complicated ones) *before* deploying our product to users.

Initialization

At the beginning of a run, Trinity initializes the backend state (i.e., the user's Dropbox on dropbox.com) and filesystem state (i.e., the Dropbox folder on disk). Note that here we're not initializing Nucleus's data model directly, but rather the external state of the system that Nucleus observes and syncs. It then instantiates Nucleus, akin to a user linking the Dropbox desktop client and selecting a previously existing Dropbox folder on their disk.

Execution

Trinity then alternates between scheduling Nucleus and scheduling itself on the main thread. Until Nucleus reports that it has reached a synced state, Trinity aggressively agitates the system by modifying the local and remote filesystems, intercepting Nucleus's asynchronous requests and reordering responses, injecting filesystem errors and network failures, and simulating crashes.

Verification

Once Nucleus reports it's synced, Trinity asserts that the system is in a consistent state. Additionally, it verifies its own determinism by re-running the same test (with the same seed) and asserting that the same final state is reproduced.

Mocking

Nucleus is parameterized over several compile-time dependencies, allowing Trinity to pass in wrapped versions of the filesystem, network, and timer during initialization. Trinity uses these wrapped implementations to intercept asynchronous requests and serialize responses. If Trinity chooses to satisfy a request, it proxies it through to an underlying concrete implementation.

Filesystem

Trinity replaces the native platform filesystem with an in-memory mock. This mock allows Trinity to inject failures in any filesystem operation, to reorder requests and resolve them in whatever order it wants, and even to simulate system crashes by snapshotting and restoring old filesystem states. Most importantly, though, using an in-memory filesystem provides a huge performance boost, allowing for about 10x more test runs and hence more random exploration.

Network

The entire server backend—metadata database, file content storage, notification services, etc.—is swapped out for a Rust mock, allowing Trinity to arbitrarily reorder, delay, and fail any RPC to the server. The mock emulates all server-side services that Nucleus depends upon, and mimics production behavior as closely as possible.

Time

The Nucleus codebase uses a generic, mockable timer object. For example, if Nucleus requests a 5-minute timeout for downloading an online-only placeholder, Trinity can intercept that request, fast-forward time arbitrarily, and fire the timeout whenever it wants.

Simulating concurrency

Thanks to all this mocking, Trinity is privy to all asynchronous activity in the system. Trinity drives Nucleus on the main thread, buffering numerous intercepted requests to each of the mocked components. After some time, Nucleus passes control back to Trinity, at which point Trinity randomly chooses whether to satisfy or fail those requests or to take its own actions to perturb the system's external state (as described under "Execution" above).

Now, you might be thinking, "how does Trinity achieve scheduling both itself and Nucleus on the main thread?" The answer is that Nucleus itself is a Rust Future, and Trinity is essentially a custom executor for that future that allows us to interleave the future's execution with its

own additional custom logic. If you're not already familiar with futures in Rust, it may be helpful to check out [this excellent overview](#) and/or the [latest official documentation](#) before we dive deeper in this section.

Futures are composed of other futures, so it makes sense for Nucleus itself to be one giant future. Internally, it's composed of numerous other worker futures, which are in turn composed of more futures, creating a tree-like structure. For example, the "upload worker" is a future responsible for transferring file contents to the Dropbox servers. Internally it manages an [unordered set of futures](#) representing the concurrent network requests it drives.

Rust's [Future trait](#) expresses execution via the function `poll()`, which causes a future to make some progress and then return whether it's completed or not. If a future returns `Poll::Ready(result)`, it has completed with output result. If it returns `Poll::Pending`, it is blocked on one of its child futures, and an external party is responsible for calling `poll()` again when any of those child futures have made progress. The top-level Nucleus future has the type `impl Future<Output = !>`. The `!` is Rust for "the uninhabited type," meaning it can only ever return `Poll::Pending`. However, each `poll()` invocation on a future still allows its subsystems to make progress.

During testing, Trinity is the external party in charge of polling that top-level sync engine future. In its main run loop, Trinity alternates between running its own code, calling `poll()` on Nucleus, and calling `poll()` on all mocked filesystem and network requests that it intercepted. When Nucleus becomes blocked on one or more outstanding requests, Trinity chooses some to either succeed or fail (driven, as always, by the PRNG). This not only simulates the concurrency Nucleus would normally experience in production but also amplifies the probability of less likely execution orderings!

Limitations

While Trinity is great at finding unexpected interactions between different components internal to Nucleus, mocking out so many external sources of nondeterminism comes at a cost.

Native filesystem interactions

Our native platform layers are themselves fairly complex, including OS-specific logic to manage permissions, modify extended file attributes, hydrate Smart Sync placeholders, and so on. When running Trinity against our in-memory filesystem mock, we lose all coverage of this area. To cover this layer of our codebase, we also run Trinity in a "native" mode, targeting the platform's actual filesystem. However, running against the native filesystem incurs a huge performance penalty (roughly 10x), which in turn means Trinity Native can't test as many different seeds.

In order to deliver on its reproducibility promises, Trinity serializes all calls into the native platform APIs, to avoid any nondeterminism that could arise from the OS interleaving system calls. Of course, real users can make system calls whenever they want, so such interleavings could very well be a source of race conditions in the wild.

Lastly, some platform behavior is still out of scope: Trinity cannot actually reboot the machine mid-test, so it can't validate that we use fsync in all the right places to ensure crash durability on each platform.

Network protocol

Mocking out all of Dropbox's server architecture might hide any number of bugs in the real client-server communication protocol. Given the complexity of our production environment, there is high risk that our mock's behavior may drift from reality. On the other hand, mocking out the network allows Trinity to run quickly and independent of connectivity issues.

To test the protocol, we have a separate test suite called Heirloom. Heirloom operates on the same deterministic random seed principle for controlling the client's execution, but because it necessarily talks to a real Dropbox server over the network, it must trade off some determinism guarantees. And due to the overhead of this approach (communicating across multiple language boundaries and through our full backend stack), Heirloom runs about 100x slower than Trinity. How exactly Heirloom works is a subject for a future blog post.

Minimization

Another tradeoff Trinity makes is that, because it mocks out less of the system than CanopyCheck does, it cannot minimize failing test cases as easily. The more of Nucleus's complex, emergent behavior we try to validate, the less we can perturb any given test's input without making its behavior totally diverge. Even a small change to the three trees' initial state might change the order of network request scheduling down the line, and thereby invalidate a failing random seed—hard-won at the cost of many CPU-days! We're currently thinking about how to solve this problem (e.g., by decoupling the global PRNG into several independent ones), but for now, the developer must analyze the scenario themselves, inserting new logging and fine-tuning their grep filter to focus on what's most relevant in the trace.

Ultimately, any verification effort faces some limitations to how much coverage it can offer. We maintain several other test suites in our CI that run at higher levels of abstraction, which trade off Trinity's ease of use and performance in order to provide some (admittedly weaker) end-to-end coverage for whatever lies outside Trinity's scope. But when it comes to testing Nucleus itself, Trinity allows us to deploy our production builds with a level of confidence that we never would have dreamed of with Sync Engine Classic.

Conclusion

Focusing on testability from the get-go with Nucleus allowed us to build fully deterministic, randomized testing systems like CanopyCheck and Trinity. The target of our rewrite had over a decade to stabilize, meaning we needed to avoid regressing 10+ years of bug fixes. But thanks to these test frameworks we were able to safely get it done. And now, the heart powering Dropbox sync is stronger than ever. ❤️

Special thanks to Ben Blum and Sujay Jayakar for contributions and to Geoff Song, John Lai, Liam Whelan, Gautam Gupta, Iulia Tamas, and the whole Sync team for comments and review. And thanks to all current and past members of the team who've contributed to building Nucleus!

// Copy link

Link copied