Nix's Substituter List Is Not a Routing Table

Published May 24, 2026 | 14 min read | Source software programming nix

▼ Table of Contents

Nix's substituter model is one of those designs that is *almost* right, but isn't *exactly* there. It is simple in itself: you list a few binary caches in `nix.conf`, the daemon walks them in order, and if the path you want is anywhere on the internet a build doesn't have to happen on your laptop. The binary cache is often listed as a *strength* of NixOS, but it's actually a strength of Nix and a bare minimum for something like NixOS to work for users. Which is to say that it's actually perfectly fine until you have a large enough multi-cache setup for your configuration's dependencies, or in rarer cases, your projects dependencies. By which I mean multiple binary cache instances because you decided to fetch massive C++ and Rust projects from the internet.

In such a case the first cache in your list is almost always `https://cache.nixos.org`. It is fast, it is global, and it does not have your overlay's packages. The second tends to be something like the `nix-community` cache, because you usually pull something really useful [1] from the nix-community organization. In less common cases you also add a third party project's Cachix and occasionally, if you're technical enough, your homelab's private cache. In such a setup every narinfo lookup walks that list. Every `nix-shell -p hello` becomes a serialized scan across four hosts on three continents because Nix has no concept of *which substituter is most likely to answer for this path*. It just asks them all, in the order you wrote them down, one after the other.

1
Cough nh c
and it's ann
building fro
because Ru

The Shape of the Problem

To explain why a proxy is the right answer, you have to be honest about what Nix's substituter logic *is*:

- A loop over `substituters`, in order.
- For each: `HEAD /<hash>.narinfo`. If 200, fetch and use it. If 404, continue.
- No concurrency. No latency tracking. No memory of which cache won *last* time you asked for this hash.

The substituter list is a *preference*, not a *routing table*. There is a `priority` field, but it is a static integer chosen at config time. It does not know that `cache.nixos.org` is 40 ms away on your home connection and 800 ms away

from the office VPN. It does not know that your private cache is the only one that has the path. It does not know anything, because there is nothing to know. The daemon is stateless on every request. For a tool that prides itself on being a model of declarative purity, the network layer underneath is still static and request-local.

ncro, Briefly

ncro—Nix Cache Route Optimizer, pronounced Necro—is a small HTTP proxy that sits between `nix-daemon` and your substituters. It is about three thousand lines of clean, performant Rust code. It does three things:

1. On a narinfo lookup, it *races* all configured upstreams in parallel with `HEAD` and remembers which one won.
2. On a NAR fetch, it streams the body straight through to the client. No disk. No buffer. No NAR ever lives on the proxy.
3. It keeps a small, bounded SQLite table of route decisions so a restart doesn't force it to relearn the entire world.

That is the whole product. It is deliberately *not* another `ncps`, which mirrors caches to disk and gives you all the cache-invalidation grief that comes with mirroring caches to disk. ncro does not retain payload data once it is streamed through.

What's Actually In It

The interesting parts are the ones the architecture diagram (which you might or might not have paid attention to) doesn't show:

1. **The race.** ncro's router groups candidates by `priority`, then for each priority tier spawns a `FuturesUnordered` of `HEAD` requests and breaks on the first success. The tier loop is what lets you say "prefer my private cache, but only if it answers—otherwise fall through to cache.nixos.org" without writing any of that logic into Nix itself. There's a deadline pinned to the select loop so a single hung upstream can't stall the entire lookup. Failures are classified as *not found*, *network error*, and *timeout* because "every upstream returned 404" and "every upstream's TCP handshake died" deserve different answers to the client.

2. **The cache, in two layers.** A moka `Cache` sits in front of SQLite, with a 1024-entry capacity and TTL bound to the route TTL. SQLite underneath, with `narinfo_bytes` stored alongside the route so a hot path doesn't even need a second upstream fetch. Eviction is throttled to fire every hundred writes by abusing `AtomicU64::fetch_add`'s pre-increment semantics. This is a detail that bit me in review because `count % 100 == 0` fires on the *first* write, when the counter is zero. The fix was one character, but the impact was real: latency metrics were skewed until this edge case was corrected.

3. **Health, with EMA.** $L_t = \alpha R_t + (1-\alpha) L_{t-1}$, $\alpha = 0.3$. The first sample bypasses the smoothing. Otherwise, the first probe permanently anchors to whatever junk was in `ema_latency` at startup. Consecutive failures bin the upstream into Active / Degraded / Down with multiplicative backoff (x1, x4, x10) so a dead cache stops costing you probe traffic. Sorting falls back to `priority` only when two upstreams are within 10% of each other, which is the only honest interpretation of "ties" when you're dealing with network latency that jitters by milliseconds.

4. **Inflight dedup.** If two clients ask for the same narinfo at the same time, only one race happens; the other waits on the mutex and reads the LRU on the way out. The cleanup path uses `remove_if` with `Arc::ptr_eq` because a *naïve* `remove` will happily delete *someone else's* goddamned `Arc` that landed in the slot between the time your guard read it and the time it dropped. This avoids a TOCTOU class bug in the dedup map.

5. **Signatures.** ed25519 via `ed25519-dalek`, public keys parsed from Nix's `name:base64(key)` format, the fingerprint reconstructed in the exact `1;StorePath;NarHash;NarSize;refs` shape that `nix store sign` writes. If you configure a key for an upstream, narinfos that don't verify are rejected before they hit the cache. This is the part you cannot skip: a proxy that strips signature checks is a proxy that turns one compromised upstream into *every* machine pulling through you.

What I Did Not Build

There is no NAR cache. There is no mirror. There is no "warm-up" job. ncro will happily serve the same 200 MB closure ten times in a row by streaming it from upstream ten times, and that is the *correct* behaviour for a router. The optimization is in **which upstream you ask**, not in **avoiding the upstream entirely**. The moment you start storing NARs you are signing up for free-disk-space alerts, content-addressed-but-not-really cache poisoning, and a maintenance surface that has nothing to do with what you actually wanted, which was for `nix build` to stop spending 400 ms on DNS for a cache that never has the path. There is also no mesh by default. There *is* an optional gossip layer with signed UDP packets for the trusted-peer case, but it is off, and it should be off unless you have a clear trust and threat model for peer exchange. I plan to expand upon this in the future.

Was It Worth Writing?

Yes.

The proxy is small enough that I can keep the whole thing in my head, which I cannot say about most software I *actually depend on*. It solves one problem, the problem it set out to solve, and it does not try to grow into a content cache or a CI artifact store or a peer-to-peer Nix mesh—even though all three are tempting and each would complicate the design substantially. Nix's substituter logic could support this kind of behavior, but in practice it has remained a static ordered loop for years. A dedicated proxy is a practical way to add dynamic routing without patching the daemon itself.

Perhaps it interests you too. Give it a try!

← Back to all posts