University of Basel

DRAFT VERSION

01.01.1980

Gachix: A binary cache for Nix over Git

Bachelor Thesis

Department of Computer Science
Department of Mathematics and Computer Science
Computer Networks Group
cn.dmi.unibas.ch

Examiner: Prof. Dr. Christian Tschudin
Supervisor: Dr. Erick Lavoie

Ephraim Siegfried
e.siegfried@unibas.ch

January 01, 1980

Abstract

This project develops a binary cache for Nix packages using Git's content-addressable filesystem. This approach improves upon traditional input-addressable packages by reducing memory usage and enhancing trust. Leveraging Git provides a key advantage: simple peer-to-peer replication of the binary cache across multiple nodes. The core work involves modeling package dependency graphs and user profiles within Git and creating an interface for Nix to interact with this system. The project will be evaluated through performance benchmarks measuring memory usage and package retrieval speed, along-side functional tests of the peer-to-peer replication and Nix interface.

Table of Contents

1 Introduction

2 Background

2.1. Nix

Nix is a declarative and purely functional package manager. The main purpose of Nix is to solve reproducibility shortcomings of other package managers. When packaging an application with RPM for example, the developer is supposed to declare all the dependencies, but there is no guarantee that the declaration is complete. There might be a shared library provided by the local environment which is a dependency the developer does not know of. The issue is that the app will build and work correctly on the machine of the developer but might fail on the end user's machine. Nix provides a functional language with which the developer can declaratively define all software components a package needs. Nix ensures that these dependency specifications are complete and that Nix expressions are deterministic, i.e. building a Nix expressions twice yields the same result. [1]

2.1.1. Nix Store

The Nix store is a read-only directory (usually located at /nix/store) where Nix store objects are stored. Store objects are source files, build artifacts (e.g. binaries) and derivations among other things. These objects can refer to other store objects in the Nix store (e.g. dependencies in binaries). To prevent ambiguity, every object has a unique identifier, which is a hash. This hash is reflected in the path of the object, which is constructed as /nix/store/<nix-hash>-<object-name>-<object-version>.

There are two major ways of computing the hash. The prevalent way is to hash the package build dependency graph. There is also an experimental feature where it is computed by hashing the store object contents. In the first case the addressing scheme is called input-addressing and in the latter content-addressing.

2.1.2. Deployment Pipeline

To produce a package in Nix, a derivation has to be produced. A derivation is a build plan that specifies how to create one or more output objects in the Nix store (it has the .drv extension). It also pins down the run and build time dependencies and specifies what the path of the output will be. It is an intermediate artifact generated when a Nix package expression is evaluated, analogous to an object file (*.o) in a C compilation process.

To build a derivation, Nix first ensures all dependent derivations are built. It then runs the builder in an isolated sandbox. It ensures that in the build process of a component only exclusively declared build and runtime dependencies can be accessed (e.g. /bin gets pruned from the environment variable PATH) and network access is limited. This makes component builders pure; when the deployer fails to specify a dependency explicitly, the component will fail deterministically. The result of the build process is an artifact which is also in the Nix store. [2]

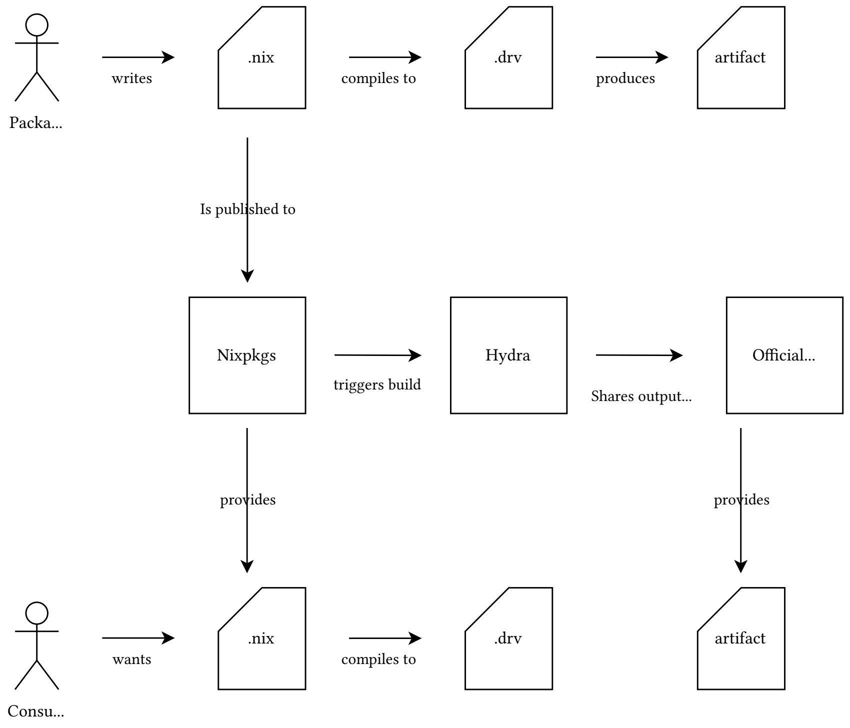

Figure 1: Nix Deployment Pipeline

A package author might wish to share her package with others. To do this, the author needs to share the package in a registry. The official Nix registry is located at the official Nixpkgs repository. To add a package there, the author needs to make a pull request with the new package expression to be added. The expression gets reviewed by a trusted set of community members. Once accepted, the new package will be added to the registry.

Users can build packages by specifying the registry and the name of the package. Nix will download the expression from the registry and produce a derivation. The derivation specifies the path of the output artifact in the Nix store. To avoid building the artifacts locally, which can take a long time, users can benefit from binary caches, which are called substituers in Nix. With the default installation of Nix, there is only one substituer which is the official binary cache. This cache has most artifacts which are in the official Nixpkgs registry. These official artifacts are build on Hydra, which is a continous build system. Using the package identifier retrieved from the derivation, Nix will fetch the package from the binary cache.

Package authors can also publish packages on a private registry and publish artifacts on a custom binary cache such as Cachix. The benefit of using the official registry is that they will be made available at the official cache, which is set as trusted and available in every Nix installation.

2.1.3. Binary Cache Interface

2.1.4. Daemon Protocol

2.2. Git

Git is advertised as a distributed version control system. More generally, it is a tool for manipulating a directed acyclic graph of objects and replicating these objects across repositories.

2.2.1. Objects

Git stores objects in a append-only manner. Objects can only be added to the object database and never be modified. There are four types of objects in Git:

- **Blob (Binary Large Object)**: A blob is a sequence of bytes. It is used to store file data. Its semantics, e.g. whether the file is executable or a symlink, is stored in the object that points to the blob, i.e. a tree.
- **Tree**: Is a collection of pointers to trees or blobs. It associates a name and other metadata with each pointer.
- **Commit**: Is a data structure which points to exactly one tree. It also records the author of the commit, the time it was constructed and it can point to other commits which are called parents.

- **Reference**: Is a pointer to a Git object. **Direct References** can point to blobs, trees and commits. **Symbolic References** point to direct references.

Blobs, trees and commits are compressed and stored in the .git/objects directory. All objects in this directory are content addressed, i.e. they are identified by the hash of their contents. References are identified by their chosen name and are stored in .git/refs. [3]

2.2.2. Replication

Git can replicate objects across multiple repositories. One of the replication protocols is the *fetch* operation. With this operation, a repository can request and download objects from another repository, which are called remotes. The connection to remotes happens via HTTP or SSH.

To specify which objects should be downloaded, a *refspec* can be used. The *refspec* specifies which remote references should be downloded to the local repository and how the received references should be named. This operation also downloads all objects which are reachable from the references speciefied. The *refspec* is a string which is structured as <remote_references>:<local_references>. For example, the refspec refs/foo:/refs/bar copies the remote reference refs/foo and names it refs/bar locally and downloads all objects reachable from refs/foo. It is also possible to specify multiple references by using globs, e.g. the reference refs/heads/* specifies all references which are in the namespace refs/heads.

3 Design

3.1. Mapping Nix to Git

This chapter introduces how Nix concepts are mapped to the Git object model. Figure 2 displays a simple Nix store with a package called *foo* which depends on packages *libfoo* and *bar*. This section shows which transformations are taken to have an equivalent model in Git which is represented in Figure 3.

Just like the Git object database, the Nix Store is an immutable collection of data. Apart from packages the Nix store also stores source files, derivations, links and other types of objects. Since a binary cache only serves binary packages, we can focus on only those type of objects.

Every top-level entry in the Nix store is uniquely addressable by some hash. There exist also files which are not uniquely addressable, which are inside those directories (e.g. a directory called ‘bin’). When mapping Nix entries to Git objects we have to make sure that the corresponding objects in Git are also uniquely addressable. Fortunately, this is already the case in Git, because Git uses the hash of the data content to address it.

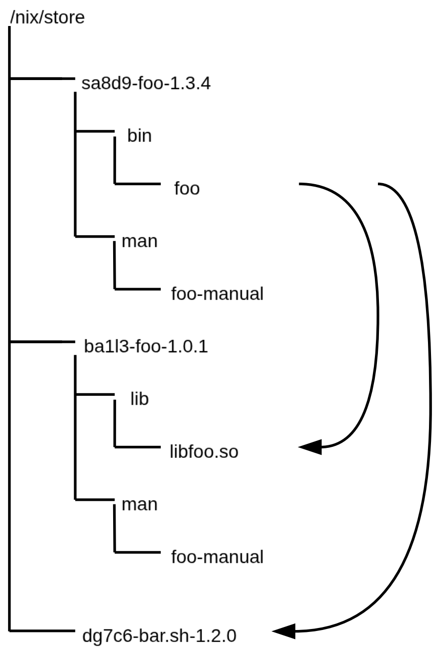

Figure 2: Nix Store

3.1.1. Packages

A Nix package is either a single file executable or a directory containing an executable. In Git, files can be mapped to blobs. This mapping will lose file metadata information: When constructing a blob from a file, and then reconstructing the file, it cannot be known whether the file is executable. There are many ways to solve this issue. One way to solve this, is to always assume that top-level files are executable, because the binary cache only serves executables. When sending the Nar of a top-level file to a client, we can always mark it as executable. Another way to solve this, is to construct a Git tree with the blob of the file as the single entry. In this tree, we can mark the blob as executable. In order to distinguish this tree from other trees, the blob in this special tree can be named with a unique magic value which marks the tree and the blob as a single file executable. The latter approach is favourable, because as presented in Section 3.1.2. we will create commits for each package, and commits can only point to trees and not blobs.

Package directories can be mapped to Git trees. These directories can contain symbolic links, files and other directories. Directories can be mapped to trees, symbolic links and files can be mapped to blobs, while metainfo such as the name of the objects can be stored in the trees.

Notice that after mapping Nix store entries to Git objects, the size of the database will have decreased. One reason is that Git compresses its objects. Another reason is that all objects in Git are content addressed. If two package directories in the Nix store contain a file with the exact same content, Git will store this file as a blob only once. The package directories will be mapped to trees which point to the same blob.

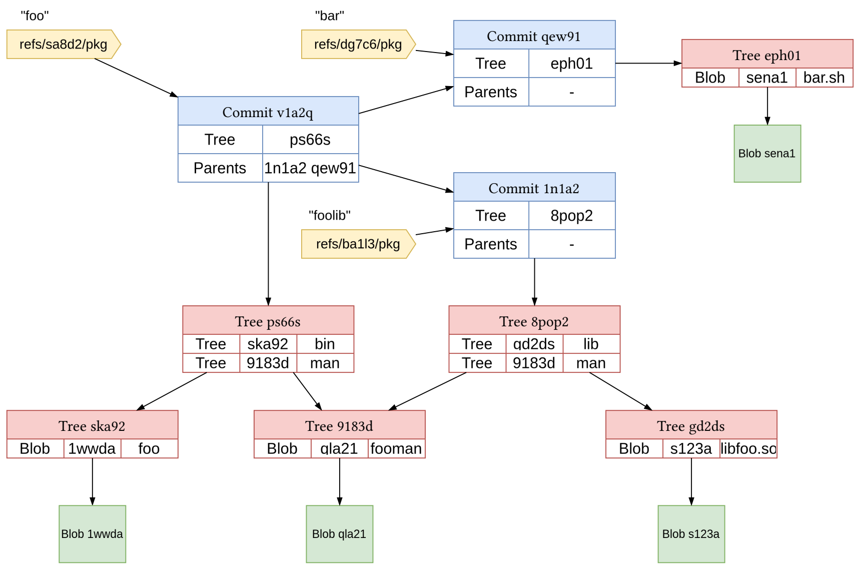

Figure 3: Gachix Git object model after transformaing Nix store in Figure 2

3.1.2. Dependency Management

In order to track which packages have which runtime dependencies, Nix manages a Sqlite database. This database tracks which Nix paths have references to other paths in the Nix store. This information helps Nix to copy package closures to other stores and it prevents deleting packages which have references to them. In Gachix this dependency management is achieved using commit objects. For each package, a commit object is created where the commit tree is the tree containing the package contents as described above. The parents of the commit are the dependencies of the package which are also represented as commits. To find out what the dependency closure of a package called 'foo' is, we can recursively follow the parent pointers which is equivalent to running git log <foo-commit-hash>. To enable easy replication, we'll need to ensure that every package is associated with exactly one commit hash (more in Section 3.3.). To ensure this, for every commit on each replica the commit message, the time stamp and the author field of the commit are set to constant values.

As a binary cache, Gachix needs to locate the corresponding commit hash given a Nix package hash, because the Nix binary cache interface requires that Nix packages are requested by the Nix hash. To maintain a mapping between Nix hashes and commit hashes I propose two solutions. One solution is to create a Git tree which serves as an index.

This special tree has all package commits as entries where the name of each entry is the respective Nix hash. With this special tree we can quickly lookup a package. A downside of this approach is that for each new entry the tree has to be copied(and the new entry be appended), because we cannot alter objects in the Git database. Another solution is to create a Git reference for each package which points to the corresponding commit object. The name of each reference contains the Nix hash. This approach also allows fast lookups. It is faster and requires less space since no objects have to be copied after package deletion or insertion. If we want to delete a package, we only have to remove the reference and call the garbage collector and Git will remove all objects associated with that package since these objects won't be reachable from a reference anymore. Because of these positive properties, Gachix uses the second approach.

3.1.3. User Profiles

Although this is not used in Gachix, an equivalent of Nix profiles and user environments can be achieved with the use of Git references. We can create a reference namespace (let's say for example *refs/myprofile*) containing symbolic references to direct references (e.g. *refs/myprofile/foo* pointing to *refs/<foo-nix-hash>/pkg*) which point to package commits. The symbolic references in this namespace point to the packages the user wishes to have in the environment. We can then create a worktree by merging all commits reachable from this reference namespace. This worktree is identical to the contents of */nix/store* except that only active packages are contained in it.

3.2. Cache Interface

To integrate Gachix into the existing Nix ecosystem as a binary cache, it is easiest to implement the same API that the existing binary caches provide (see Section 2.1.3.). With this interface, a Nix user can add the URL of the Gachix service to the set of *substitutors*. Everytime the user adds a package, Nix will ask Gachix for package availability and fetch it if it is available.

3.2.1. Narinfo endpoint

The binary cache needs to serve metadata about the packages which is called Narinfo in Nix. Usually in other cache implementations the narinfo is computed on demand. In Gachix everytime a package is added to Gachix, the Narinfo is computed once and it is stored as a blob. Additionally, to associate this blob with a package, a reference is added under *refs/<package-nix-hash>/narinfo* which points to this blob directly. Everytime a Narinfo is requested (with GET /<package-nix-hash>.narinfo) for a given Nix hash, Gachix serves the contents of this blob which it gets through this reference.

With the request HEAD /<package-nix-hash>.nar a client can ask the cache if it has the corresponding package. This request is handled by checking whether a reference exists containing the requested hash.

3.2.2. Nar endpoint

The Narinfo contains an URL, under which the nar of a package can be downloaded. This link usually contains the hash of the Nar of the package. But because this endpoint can be chosen arbitrarily, I decided to put the hash of the package tree instead. It would also be possible to put the hash of the package commit, which would enable sending the whole package closure to the client. But since Nix only expects the contents of the requested package and not its dependencies, it is enough and faster to include the tree hash in the URL. The Nix user can then request a package with GET refs/nar/\<tree-hash\>.nar. Gachix then directly accesses the tree, converts it to a nar and sends the archive to the client. The user can verify whether the hash corresponds to the package by unpacking the received nar, converting it to a Git tree and comparing the tree hash with the hash in the URL.

3.3. Replication Protocol

This section explains how packages are added to the cache and replicated across peers. Gachix itself does not build packages and relies on external services. Gachix can communicate to the local Nix daemon, to remote Nix daemons and to remote Gachix peers (i.e. Git repositories managed by Gachix). With the Nix daemon protocol, Gachix can request metadata of store paths (e.g. runtime dependencies) and retrieve a package from the Nix store in the nar format. With Gachix peers, Gachix uses the Git protocol to replicate commits and to fetch whole package dependency closures.

There is no policy yet on when Gachix adds packages to its repository. In the current version, it has to be manually run in the command line. A possible policy would be to always try to add a package when a Nix user requests one via the HTTP interface and it does not exist on the local repository. Gachix should then fetch the package from trusted replicas or may build it using the daemon protocol.

3.3.1. Constructing Package Closures

The current algorithm of adding a package closure is displayed in Algorithm 1. The algorithm receives the Nix store hash as an argument and returns the commmit id associated with that package. It tries to recursively add package contents to the Git repository. At it's core, it is similar to a depth first search algorihm (See lines 19-28 in Algorithm 1). The algorithm iterates through a package's direct dependencies (line 21). For each dependency d, it makes a recursive call to *add_closure(d)* (line 23). This means it fully processes one dependency, including all its dependencies, before moving to the next direct dependency in the list. Once it has collected all commit hashes of the dependencies, it constructs a commit using the hash collection as parent commits (line 26).

Algorithm 1: Add package closure

```
 1: procedure ADD_CLOSURE(path)
 2:     if package_exists(nix_hash) then
 3:         return commit_oid(nix_hash)
 4:     end
 5:     ▷ Ask gachix peers if they have already replicated the package closure
 6:     for $p \in P$ do
 7:         if has_package(p, nix_hash) then
 8:             return fetch_closure(nix_hash)
 9:         end
10:     end
11:     ▷ Ask Nix daemons if they can provide package contents
12:     for $d \in D$ do
13:         if has_package(d, nix_hash) then
14:             package ← fetch_package(nix_hash)
15:             BREAK
16:         end
17:     end
18:     ▷ Get the nix hash of all dependent packages
19:     dependencies ← package.dependencies()
20:     parents ← {}
21:     for $d \in$ dependencies do
22:         ▷ Recursively fetch all commit oids
23:         dep_commit_oid ← add_closure(d)
24:         parents ∪ {dep_commit_oid}
25:     end
26:     commit_oid ← commit(package.tree, parents)
27:     add_reference(path, commit_oid)
28:     return commit_oid
29: end
```

There are three base cases of the recursive algorithm. The algorithm does not recurse if it finds a leaf in the package dependency tree, i.e. a package without dependencies. It does also not recurse if a package already exists in the local repository (lines 2-4). Another base case is when the package can be retrieved from peer replicas (lines 6-10).

3.3.2. Fetching Packages from Replicas

This section explains what the *fetch_package* function does in Algorithm 1 on line 8. Its main task is to fetch commits and references associated with a package. I will present two algorithms which perform this task. The first one is short and fast, but does not work because of an upstream Git bug. The second will be the one which is used in the current implementation.

We can fetch packages by using refspecs (see Section 2.2.2.). With the refspec _refs/<nix-hash>/*:refs/<nix-hash>/*_ we specify that we want all references under the remote Nix hash namespace. Each such namespace has a *pkg* and a *narinfo* reference (see Section 3.1.2.). Git will fetch the blob from the *narinfo* reference and it will fetch all commits reachable from *pkg*. With this we will have all package contents from the whole package closure. What we will not have are the references to the dependencies of the package. For example, if a package *foo* has the dependency *bar* and we fetch with git fetch peer refs/<nix-foo-hash>/*:refs/<foo-nix-hash>/* we will receive the package contents of *foo* and *bar* but not the reference namespace *refs/<bar-nix-hash>*. To also get references from dependencies, we could add the namespace *refs/<nix-hash>/deps* containing symbolic references to all dependent packages. In the example above, when a peer constructs the package *foo*, it should also add the symbolic references *refs/<foo-nix-hash>/deps/<bar-nix-hash>/pkg* and *refs/<foo-nix-hash>/deps/<bar-nix-hash>/narinfo*. With this approach we can recursively reach all references by symbolic references. The expected behavior is that when fetching all references from _refs/<nix-hash>/*_ that all reachable references will also be fetched. Unfortunately, this is not the case because of a Git inconsistency which has been reported. When fetching symbolic references, Git resolves the symbolic reference to direct references [4]. This makes this approach of fetching package unusable, as the references of the dependencies won't be fetched. But once this upstream bug is fixed, the explained method is a viable approach.

In the second approach there is no *deps* namespace and Gachix fetches each reference explicitely. The initial fetch downloads all Git objects associated with the package closure. The subsequent fetches only download the references. The fetching of the references is done in a breadth-first-search manner.

TODO: should I add the algorithm?

3.3.3. Nix Daemons

If packages don't exist locally and no other replica has the package, it has to be fetched from a Nix store. The Nix daemon protocol can be used to achieve this. With this protocol, we can talk to the local Nix daemon (if one exists) via a socket or to remote Nix daemons via the SSH protocol. Gachix uses the Nix daemon protocol to fetch package metainfo information called “Path Info” in Nix and to download package contents in the nar format. Gachix parses the nar files and unpacks them as Git trees or blobs.

4 Implementation

5
Evaluation

6 Related Work

7 Conclusion

Bibliography

[1] The NixOs Foundation, "How Nix Works." Accessed: Sep. 19, 2025. [Online]. Available: https://nixos.org/guides/how-nix-works

[2] E. Dolstra, "The Purely Functional Software Deployment Model," 2006.

[3] S. Chacon and Straub Ben, *Pro Git*. Apress, 2025. [Online]. Available: https://git-scm.com/book/en/v2

[4] GitLab and GitLab Organization, "'git remote -t' should list tags for specific remotes." Accessed: Dec. 02, 2025. [Online]. Available: https://gitlab.com/gitlab-org/git/-/issues/175