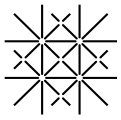

University of Basel

Gachix:
A binary cache for Nix over Git

Bachelor Thesis

University of Basel - Faculty of 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
ephraim.siegfried@proton.me

December 23, 2025

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 is evaluated through performance benchmarks measuring memory usage and package retrieval speed, alongside functional tests of the peer-to-peer replication and Nix interface.

Acknowledgments

I would like to thank Prof. Dr. Christian Tschudin and Dr. Erick Lavoie for giving me the opportunity to work on this thesis. I especially thank Dr. Erick Lavoie for his guidance, patience and giving me a broader perspective. Many ideas presented in this thesis originate from the many fruitful discussions we had.

Table of Contents

List of Figures

1
Introduction

This project is motivated by the hypothesis that core features of the Nix package manager can be implemented using Git's internal object model. This concept is explored through the development of Gachix, a binary cache for Nix and the primary subject of this thesis.[1]

Gachix uses Git as a backend storage system for Nix packages. By mapping the Nix store's structure directly onto Git objects (blobs, trees, and commits) this project aims to bridge the gap between these two technologies. Functionally, Gachix acts as a binary cache; it serves pre-built binaries and other artifacts to clients that have the Nix package manager installed. It is designed as a complementary tool, rather than a replacement for Nix or the local Nix store.

Adopting Git as a storage layer allows Gachix to inherit several advantageous features. Most notably, it leverages Git's native deduplication and compression capabilities, which significantly reduces the size of the package database compared to a traditional Nix store. Furthermore, Gachix operates independently of a local Nix installation, allowing it to run on any machine. Finally, by using Git's synchronization protocol, Gachix enables efficient, peer-to-peer package replication and exchange across multiple nodes.

[1] https://github.com/EphraimSiegfried/gachix

2
Background

2.1. Nix

Nix is a declarative and purely functional package manager founded by Eelco Dolstra. [1] 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. To solve this issue, Nix provides a functional language with which the developer can declaratively define all software components a package needs. Nix then ensures that these dependency specifications are complete and that Nix expressions are deterministic, i.e. building a Nix expressions twice yields the same result. [2]

2.1.1. Nix Store

The Nix store is a read-only directory (usually located at `/nix/store`) where Nix stores objects. Store objects are source files, build artifacts (e.g. binaries) and derivations among other things. These objects can refer to other objects (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>`. [3]

There are two primary methods for computing the hash. The standard approach is to hash information contained within a object's derivation, known as input-addressing. A derivation is a file specifying the build recipe and the complete graph of dependencies. [4] There is

also an experimental feature called content-addressing, where the hash is computed directly from the final contents of the built object itself. [5]

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. In the sandbox, only explicitly 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. [1] The result of the build process is an object in the Nix store. Subsequently, we denote the output objects as artifacts.

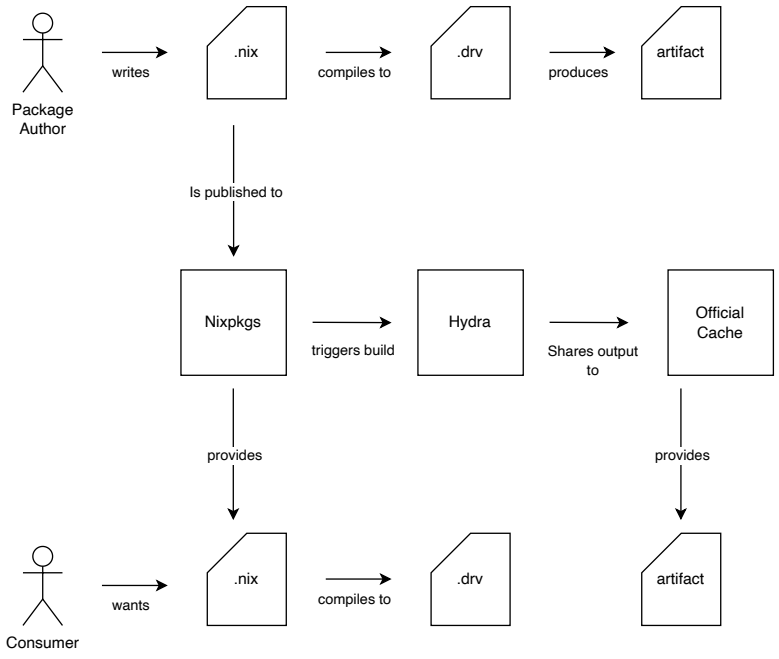

Figure 1: Nix Deployment Pipeline

If an author wants to share a package with others, the author needs to put the Nix expression which produces the artifact in a public registry. The official Nix registry called *Nixpkgs* is the place where most packages get published. It is maintained as a Git repository stored

on GitHub. 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 Nix expression 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 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.[2] This cache has most artifacts which are in the official Nixpkgs registry. These official artifacts are built 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 Nix expressions on a private registry and publish artifacts on a custom binary cache such as Cachix, which is a platform which both hosts and manages binary caches.[3] The benefit of publishing on the Nixpkgs registry is that artifacts will be made available at the official cache, which is set as trusted and available in every Nix installation.

2.1.3. Nix Archive (NAR) Format

Nix has a custom format for deserializing files and directories which is called Nix Archive (NAR). It is used to send packages over the network. It does not compress the contents of files. The specification of the Nix Archive is displayed in Listing 1. The specification closely follows the Extended Backus-Naur form, except for the *str* function, which writes the size of the bytes to be written, the byte sequence specified and a padding of 0s to a multiple of 8 bytes. [6]

[2] https://cache.nixos.org/

[3] https://www.cachix.org

```
nar = str("nix-archive-1"), nar-obj;

nar-obj = str("("), nar-obj-inner, str(")");

nar-obj-inner
  = str("type"), str("regular") regular
  | str("type"), str("symlink") symlink
  | str("type"), str("directory") directory
  ;

regular = [ str("executable"), str("") ], str("contents"),
str(contents);

symlink = str("target"), str(target);

(* side condition: directory entries must be ordered by
their names *)
directory = str("type"), str("directory") { directory-
entry };

directory-entry = str("entry"), str("("), str("name"),
str(name), str("node"), nar-obj, str(")");
```

Listing 1: Specification of the Nix Archive

2.1.4. Narinfo

A Narinfo is a plaintext metadata file used in Nix binary caches to describe a store object and its associated NAR file. It contains key-value pairs separated by newlines, providing information for retrieving and verifying binary package data. [7] It contains the following keys:

- **StorePath**: The full store path.
- **URL**: The URL of the NAR fetching endpoint, relative to the binary cache.
- **Compression**: The compression format of the served NAR.
- **FileHash**: The hash of the compressed NAR.
- **FileSize**: The size of the compressed NAR.
- **NarHash**: The hash of the NAR.
- **NarSize**: The size of the NAR.

- **Deriver**: The derivation which specifies the store object.
- **System**: The platform type of this binary, if known.
- **References**: A set of store paths which are direct runtime dependencies, separated by whitespace.
- **Sig**: A signature over the StorePath, NarHash, NarSize, and references fields using ED25519 public-key signature system.

2.1.5. Binary Cache Interface

The Nix binary cache interface exposes a set of HTTP endpoints that allow clients to retrieve package metadata and contents. [7]

The core endpoints of the API are:

- `GET /<store-hash>.narinfo` Retrieves the Narinfo (metadata) for a specific store path. The `<store-hash>` corresponds to the unique hash substring found in a package's `/nix/store` path.
- `HEAD /<store-hash>.narinfo` Efficiently checks if a specific package exists in the cache without downloading the full metadata body.
- `GET /<url>` Downloads the compressed NAR (Nix Archive). The exact path for this endpoint is defined dynamically within the URL field of the previously fetched Narinfo file. While the path is configurable, it often follows the pattern `GET /nar/<nar-hash>.nar.<compression>`.

2.1.6. Daemon Protocol

The Nix daemon is a service which runs Nix specific operations on behalf of non-root users. Most of the operations it can execute, can also be run via the Nix command line interface (CLI).

The main purpose of the Nix daemon is for multi-user Nix installations. In this mode the Nix store is owned by some privileged user to prevent other users to manipulate the Nix store in a malicious way (e.g. install a Trojan horse). When untrusted users use the Nix CLI, control is forwarded to the Nix daemon which is run under the owner of the Nix store. In this scenario, commands to the Nix daemon are dispatched through interprocess communication via a socket located at `/nix/var/nix/daemon-socket/socket`. [8]

There is also another usage of the Nix daemon. Nix is able to connect to remote Nix machines and perform operations on them. The initializer can for example order builds (`nix build --builders <remote-url>`) on the remote machine and fetch packages from the remote store (`nix copy --from <remote-url> <nix-path>`). Inter-

nally, Nix does this by connecting to the remote machine via SSH and starting the Nix daemon with `nix daemon --stdio`. This command starts a Nix daemon on the remote machine and makes the remote daemon listen on standard I/O. Subsequently, both nodes communicate via the Nix daemon protocol. [9]

The Nix protocol starts with a handshake, where both parties agree on the protocol version they will use. They also exchange some configuration options. Subsequently there are 47 operations the parties can issue by sending the corresponding Opcode. [10]

2.2. Git

The official Git documentation advertises Git as a distributed version control system. More abstractly, it is a tool for manipulating a directed acyclic graph (DAG) of content-addressable objects and replicating these objects across repositories. With the internal tools provided by Git (so called plumbing commands), it is possible to use Git as a backend for applications which incorporate a similar DAG-based data structure.

2.2.1. Objects

Git objects are immutable, i.e. they can be added or deleted from the object database but never updated. 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. The metadata of the blob, 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**: A tree is a collection of pointers to trees or blobs. It associates a name and other metadata with each pointer.
- **Commit**: A commit is a record which points to exactly one tree. It also contains the author of the commit, the time it was constructed and it can point to other commits which are called parents.
- **Reference**: A reference is a pointer to a Git object. **Direct References** can point to blobs, trees and commits; they supersede what are commonly known as references and tags. **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`. [11]

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 downloaded to the local repository and how the received references should be named. This operation also downloads all objects which are reachable from the references specified. The *refspec* is a string which is structured as `<remote_references>:<local_references>`. For example, the command `git fetch 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`. [11]

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.

Much 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 objects in Git are content-addressed.

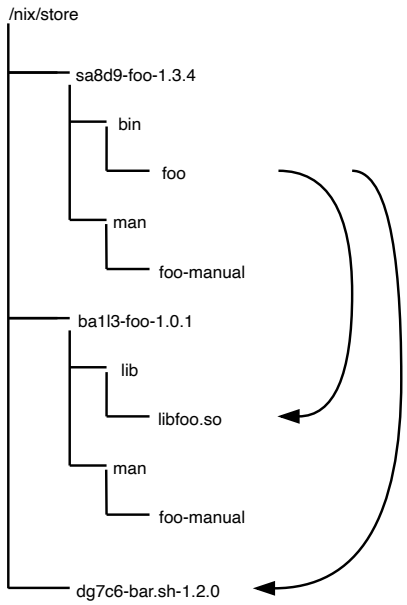

Figure 2: Nix Store. The arrows indicate dependencies between packages.

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.

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 the file 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.

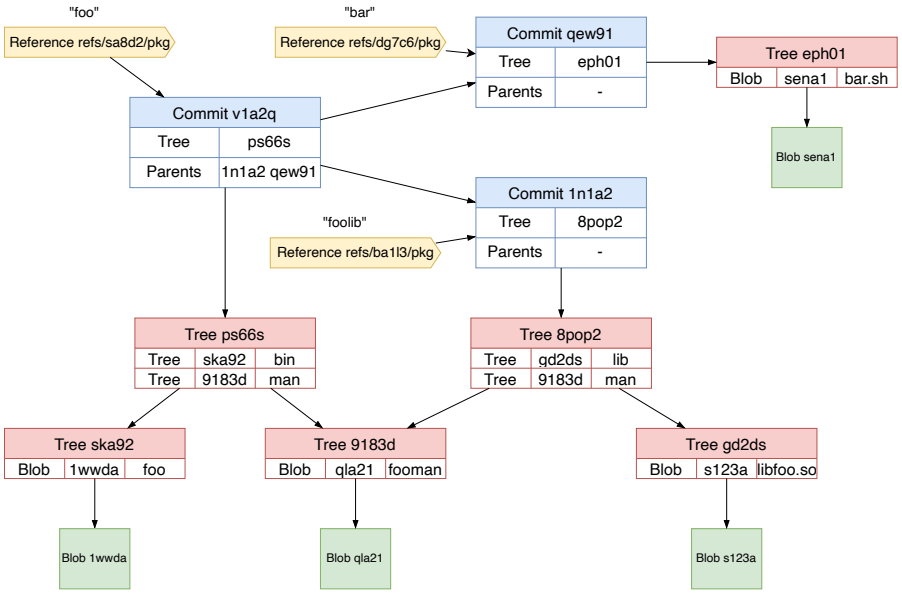

Figure 3: Gachix Git object model after transforming the Nix store shown in Figure 2

3.1.2. Dependency Management

In order to track which packages have which runtime dependencies, Nix manages a Sqlite database. This database records which Nix paths have which references to other paths in the Nix store. [12] This information helps Nix to copy package closures to other stores. A package closure is a set of store paths that are directly or transitively reachable from that store path. [13] Keeping track of references also 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 globally associated with exactly one commit hash regardless of when and where it was created (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 store package hash, because the Nix binary cache interface expects that packages are requested by the store hash. To maintain a mapping between store hashes and commit hashes we have

considered 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. The garbage collector will remove all objects which are not reachable from any reference. 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 2.1.5.). With this interface, a Nix user can add the URL of a Gachix instance to the set of substitutors (Nix's term for binary caches). 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 (See Section 2.1.4.). While many cache implementations compute Narinfo files on demand, Gachix computes the Narinfo only once. A blob containing the Narinfo is created immediately when a package is added.

Additionally, to associate the blob with a package, a reference is added under *refs/<package-nix-hash>/narinfo* pointing directly to that blob. Whenever a Narinfo is requested for a specific Nix store hash (via `GET /<package-store-hash>.narinfo`), Gachix resolves this reference to serve the blob's contents.

With the request `HEAD /<package-store-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

A Narinfo file provides a URL for downloading a package's NAR, typically using the NAR hash as the identifier. But because this endpoint can be chosen arbitrarily, we 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, it is enough and faster to include the tree hash in the URL. The Nix user can 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. This approach maintains integrity by allowing the user to verify the download: they can simply unpack the NAR, convert the resulting directory back into a Git tree, and ensure the calculated tree hash matches the one provided 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 on 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. It returns the commit ID associated with that package if it is able to fetch the closure of the package. If it is not able to do so, it returns *NIL*. It tries to recursively add package contents to the Git repository. At its core, it is similar to a depth first search algorihm (See lines 24-34 in Algorithm 1). The algorithm iterates through a package's direct dependencies (line 26). For each dependency d, it makes a recursive call to *add_closure(d)* (line 28). 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 35).

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 in the set P if they have already replicated the package closure
6:      for p ∈ P do
7:          if has_package(p, nix_hash) then
8:              return fetch_closure(nix_hash)
9:          end
10:     end
11:     ▷ Ask Nix daemons in the set D if they can provide package contents
12:     package ← NIL
13:     for d ∈ D do
14:         if has_package(d, nix_hash) then
15:             package ← fetch_package(nix_hash)
16:             break
17:         end
18:     end
19:     ▷ Return None if there doesn't exist a daemon which can build the package
20:     if package = NIL then
21:         return NIL
22:     end
23:     ▷ Get the nix hash of all dependent packages
24:     dependencies ← package.dependencies()
25:     parents ← {}
26:     for d ∈ dependencies do
27:         ▷ Recursively fetch all commit oids
28:         dep_commit_oid ← add_closure(d)
29:         ▷ A dependency could not be built/fetched, so we have to return None
30:         if dep_commit_oid = NIL then
31:             return NIL
32:         end
33:         parents ∪ {dep_commit_oid}
```

```
34:   end
35:   commit_oid ← commit(package.tree, parents)
36:   add_reference(path, commit_oid)
37:   return commit_oid
38: 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).

The algorithm can be extended to request packages from trusted binary caches beyond the Gachix ecosystem. Additionally, performance can be optimized by implementing a heuristic that orders peer contact based on proximity, thereby minimizing latency.

3.3.2. Fetching Packages from Replicas

This section explains what the *fetch_closure* function does in Algorithm 1 on line 8. Its main task is to fetch commits and references associated with a package. We 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-store-hash>/*:refs/<nix-store-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. and Section 3.2.1.). When fetching with this refspec, Git will fetch the blob from the *narinfo* reference and it will fetch all commits reachable from the *pkg* reference. 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 [14]. 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 explicitly. 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.

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. In the function *fetch_package* in Algorithm 1 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. After receiving the NARs, Gachix unpacks them to Git trees.

4
Implementation

This section explains how a binary cache was implemented using the ideas presented in Section 3. The objective was to create a store for Nix packages using Git and provide them using the common Nix cache interface. The name of this cache is Gachix and the source code is available on Github.[4]

The project is written in the Rust programming language. This compiled language is ideal for resource-intensive tasks such as parsing a large number of NAR files. The language is also effective for concurrent tasks, which is used in Gachix for serving multiple connections at once. Rust guarantees memory and thread safety. It eliminates many classes of bugs (e.g. use after free) at compile time.

4.1. Architecture

A high-level overview of the implementation can be seen in Figure 4. The top-level boxes represent the most relevant modules in the code base and the nested boxes their most important functions.

The Command Line Interface (CLI) module gives a friendly interface for interacting with the cache. It also manages the state of the configuration for the binary cache. Gachix can be configured via environment variables or a YAML configuration file. The CLI module merges the configuration options coming from these different sources. With the CLI, the user can start the web server or add a package (and all its dependencies) to the cache.

[4] https://github.com/EphraimSiegfried/gachix

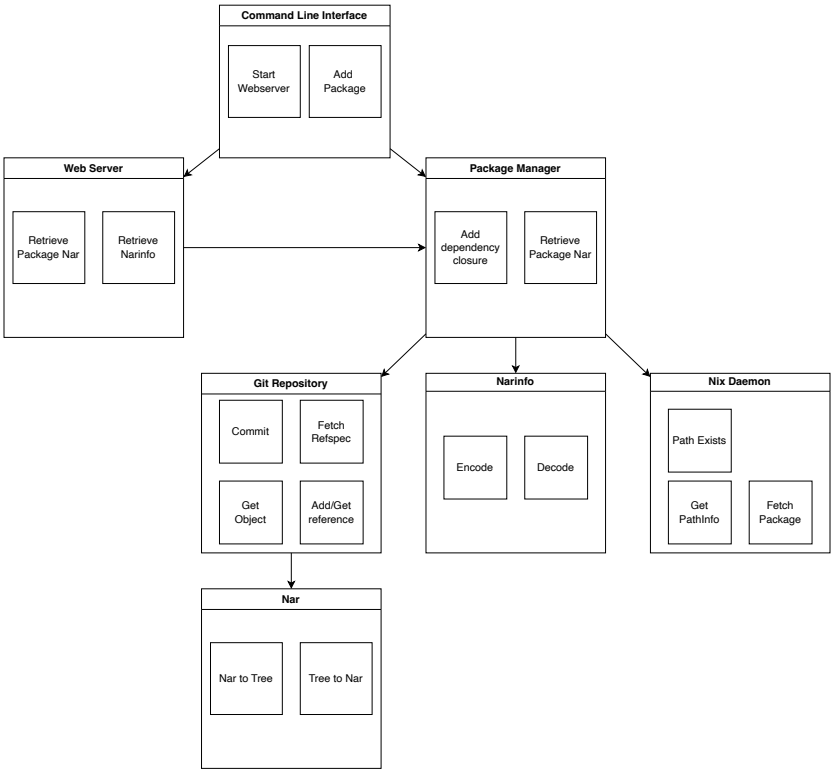

Figure 4: Gachix Architecture Overview

The web server implements the Nix binary interface (see Section 2.1.5.) and serves clients which connect to it via HTTP. It makes requests to the internal package manager module and forwards responses from it to clients.

The package manager module is the core module. It is responsible for doing the Nix to Git mapping discussed in Section 3.1. It also includes the *add_closure* algorithm discussed in Section 3.3.1.

The Repository module mainly performs Git operations using the Git2 library.[5] Git2 is a Rust wrapper of the C++ libgit implementation, which provides low-level Git operations.[6] The Repository module gives abstractions over common Git operations used for the binary cache, e.g. creating commit objects with constant author, message and date values.

The Nix Daemon module is responsible for communicating to either the local Nix daemon (i.e. the daemon which runs on the same machine as Gachix) or to remote Nix Daemons via the SSH protocol. It contains code for setting up SSH connections, retrieving metadata about store paths and retrieving store objects in the NAR archive format. It depends on a

[5]https://github.com/rust-lang/git2-rs

[6]https://libgit2.org/

custom fork of a library which implements the Nix daemon protocol.[7] The fork includes low-level code for retrieving the NAR which did not exist in the original library.

The Narinfo module constructs a Narinfo data structure from the Nix object metadata retrieved from the Nix daemon. It is also able to encode this metadata as a string, which is then stored as a blob in the Git database. Additionally, it signs the fingerprint of the Narinfo and appends it to the Narinfo (See Section 2.1.5.).

The NAR module transforms trees to NARs and vice versa. It is used to transform NARs retrieved from Nix daemons to equivalent Git trees. It encodes trees as NARs when Nix cache clients request packages. It does not have to load the whole Git tree onto memory because it is able to stream the NAR, i.e. decode the tree in chunks and serve these chunks continously.

4.2. Concurrency

To increase the performance of the binary cache, it is crucial to handle requests concurrently. Concurrency can lead to inconsistent state or crashes if handled incorrectly. Inconsistent state happens most often when multiple threads modify the same objects. In Gachix this threat is eliminated by ensuring that threads never modify objects.

Let us consider the two most prevalent operations: Retrieving and adding packages. When a package is retrieved, Gachix looks up the corresponding reference, transfers the referred package tree into a NAR and streams it to the user. The only operations involved in this process are read operations. It does not cause conflict when multiple threads read the same object at the same time.

Adding a package can happen by contacting other Gachix peers and fetching the needed Git objects from them. In this case, only new objects will be added to the Git database. Two threads cannot store two different objects with the same name because trees, blobs and commits are content addressed and references are named after unique Nix hashes. The only scenario that can happen is that two threads try to add the same object but this does not cause a conflict. When fetching NARs from Nix daemons they are transformed to equivalent Git trees. In this process, also only new objects are added.

Since all operations either read or add objects to the database, there is no conflict between multiple threads. Therefore no locking mechanisms have to be used when serving the packages concurrently.

[7] https://codeberg.org/siegii/gorgon/src/branch/main/nix-daemon

4.3. Nix Archive

When serving clients, Gachix encodes packages which are stored as Git trees into Nix Archives (NARs). The specification of the Nix Archive Format is shown in Listing 1 (Section 2.1.3.).

4.3.1. Simple Encoder

A simple approach to encode a given tree into a NAR is to explore the tree in a depth first search manner, retrieve the metadata about the objects and write this in a byte buffer according to the NAR format. Trees are mapped to directories, blobs to symlinks or regulars depending on their type. This approach is fine for shallow trees but is inefficient and can even lead to errors for large trees. This is because the buffer size can exceed the RAM limit for large packages.

4.3.2. Stream Encoder

A more efficient approach is to stream data chunks to the client, allowing the NAR to be constructed incrementally. To enable this, Gachix implements the Stream trait from the futures crate (A Rust interface) for the encoder module.[8] Having implemented this trait allows the HTTP server library called Actix Web to asynchronously stream the NAR to the client.[9] Implementing this trait requires implementing the function `poll_next` which returns the next chunk of bytes to be sent. Recursing the Git tree as in the first approach is not possible anymore because control has to be given back to the caller of `poll_next`.

To address the inability to use standard recursion within the asynchronous `poll_next` method, the implementation replaces the call stack with an explicit state stack (`Vec<TraversalState>`). This transforms the traversal into an iterative state machine. The TraversalState enum captures the specific phase of visiting a node, i.e. whether the encoder is initializing a node (StartNode), iterating through a directory's children (ProcessTreeEntries), or closing a syntactic scope (FinishNode). When the encoder encounters a directory, instead of making a blocking recursive call, it simply pushes the corresponding states onto the stack. This effectively schedules the processing of child nodes and the finalize markers for future execution cycles, allowing the function to yield control back to the async runtime after every chunk.

This architecture decouples the logic of tree traversal from the actual transmission of data. As the state machine advances, it serializes specific components of the NAR format such

[8] https://docs.rs/futures/latest/futures/prelude/trait.Stream.html

[9] https://docs.rs/actix-web/latest/actix_web/struct.HttpResponseBuilder.html#method.streaming

as padded headers, names, or file contents and queues them into a `pending_chunks` buffer. The `poll_next` function prioritizes draining this buffer to the consumer before processing further states from the stack. This ensures that even massive Git trees can be encoded and streamed incrementally without exceeding the server's RAM limits.

4.4. Nix Daemon

In Gachix the Nix daemon protocol is used to interract with the Nix API, i.e. to fetch packages and retrieve metadata information about them. One benefit of using the Nix daemon rather than utilizing the Nix command line interface is that it allows interracting with the Nix daemon as a non-root user. Another benefit of using it is that it prevents calling system commands such as `nix store dump-path` and `nix path-info`. Spawning a new process for each request to Nix is considerably slower than utilizing inter-process communication to interact with the API through the persistent daemon. Lastly, using the Nix daemon protocol makes it straightforward to interact with remote Nix daemons (see Section 2.1.6.).

4.4.1. Libraries

There are few publicly available libraries that implement the Nix daemon protocol interface, and finding them is difficult. The existing implementations identified are typically found as submodules or components within larger projects, rather than as standalone, dedicated libraries. Consequently, these modules are often highly tailored to the specific requirements of their parent projects, often implementing only a limited subset of the full Nix daemon operations necessary for their particular use case. Libraries that were found are:

- **Snix**: Snix reimplements the Nix package manager (more in Section 6.1.). It has a module for interracting with existing Nix protocols. In this module there is a submodule which implements the daemon handshake protocol and a limited set of operations.[10]
- **Harmonia**: Harmonia implements the Nix binary cache interface. It also implements the Nix daemon protocol for the client.[11]
- **Gorgon**: Gorgon is a continous integration framework. It contains a submodule which implements both the client and server daemon protocol.[12]

[10] https://git.snix.dev/snix/snix/src/branch/canon/snix/nix-compat/src/nix_daemon

[11] https://github.com/nix-community/harmonia/tree/main/harmonia-daemon

[12] https://codeberg.org/gorgon/gorgon/src/branch/main/nix-daemon

Gachix used the Gorgon module because it is the most generic implementation and it has most operations implemented which are needed in Gachix. It only lacks the operation to fetch NARs from Nix stores. This missing feature was implemented in a custom fork.

4.5. Content and Input Addressing Schemes

Nix uses two methods to define the identity of packages. The prevalent way is to use the hash of the input dependency graph, which is stored in the derivation of a package. This type of identity is called input-addressed. The other method is to hash the NAR of a package, which is called content-addressing.

Gachix uses Git to store packages, which also identifies packages by their content. However, the two systems differ in their hashing algorithms, object formats, and encoding methods. While Nix typically employs SHA-256 to hash Nix Archives (NARs) [5], Git relies on SHA-1 to hash its internal object format [11]. Furthermore, their representation of these hashes diverges: Git represents digests using hexadecimal encoding, whereas Nix traditionally uses a specialized base-32 encoding.

There was some effort made to adopt the same hashing scheme as Nix within Gachix. However, this was not finalized because it proved simpler to maintain a mapping between Nix hashes and Git hashes directly within Gachix. This approach offers the benefit of allowing input-addressed Nix hashes to be used for identifying packages in Gachix. Furthermore, we believe that the performance and storage overhead of using references, rather than the Git object hashes directly, is negligible.

4.6. Limitations

In the current implementation on Gachix a few features are missing that other Nix binary cache services provide. The most notable ones are:

- Normally packages are sent as compressed NARs (most often compressed with `xz`). In the current implementation Gachix only sends plain NARs.
- There is no command implemented which removes packages. However, this is rather easy to implement. It involves removing the reference of the package and calling the Git garbage collector.
- The `.ls` Nix binary cache endpoint is not implemented. This endpoint normally lists the entries of a package.

Furthermore, Algorithm 1 (Section 3.3.1.) can be enhanced by implementing a peer-selection heuristic designed to minimize the latency of package addition. The current algorithm is inefficient because it contacts peers in an arbitrary fixed order.

5
Evaluation

This section shows independently verifiable features of Gachix. The following claims about Gachix are made:

1. Gachix achieves the lowest median latency but shows slower average performance. (Section 5.3.)
2. Gachix is more storage efficient than other cache services. (Section 5.4.)
3. Gachix can be deployed on any Unix machine, including on systems without Nix installed. (Section 5.5.)
4. Gachix is transparent to Nix users as it can be used to fetch Nix packages using the Nix substituers interface. (Section 5.6.)

Because we compare Gachix with similar projects, we will present these projects in Section 5.1. The specifications for the benchmarking environment are detailed in Section 5.2.

5.1. Functional Comparison to other Cache implementations

There are a few projects which implement the Nix binary cache interface. The most notable ones are:

- **nix-serve**: This is the first cache implementation developped by Eelco Dolstra (i.e. the founder of Nix).[13] It is written in Perl.
- **nix-serve-ng**: This is the successor of nix-serve. It is written in Haskell.[14]
- **harmonia**: This is a modern implementation of the binary cache interface with many features.[15] It is written in Rust.

[13] https://github.com/edolstra/nix-serve

[14] https://github.com/aristanetworks/nix-serve-ng

[15] https://github.com/nix-community/harmonia

A notable difference between Gachix and the caches presented above is the other caches directly use the Nix store for storing packages. In contrast, if a Nix user wants to serve her packages with Gachix, she has to copy the packages from the Nix store to Gachix. With the other implementations, this is not necessary.

On the other hand, the benefit of using Gachix is that it does not rely on any Nix infrastructure (such as the Nix store) and it can be deployed on a Unix machine without Nix installed. All other implementations expect that Nix is installed on the host machine.

5.2. Test Machine Specification

The experiments were conducted on a desktop workstation with the following hardware configuration:

- **CPU**: Intel Core i7-14700K (8 P-cores, 12 E-cores, 28 Threads) @ 5.60 GHz (Max Turbo)
- **GPU**: AMD Radeon RX 6600 (8 GB GDDR6)
- **Memory**: 32 GiB DDR5 @ 6000 MT/s
- **Storage**: Kingston's NV2 PCIe 4.0 NVMe SSD (Partition size: 239.25 GiB)
- **Swap**: Disabled

The software environment includes:

- **Operating System**: NixOS 25.11 (Xantusia)
- **Kernel**: Linux 6.12.60
- **Nix Version**: 2.31.2
- **Nixpkgs Commit Hash**: 28bb483c11a1214a73f9fd2d9928a6e2ea86ec71

5.3. Package Retrieval Latency

To test whether the retrieval speed of packages is acceptable, Gachix was compared against the cache services presented in Section 5.1.

5.3.1. Methodology

In this benchmark, the process began by retrieving the full collection of packages from the *nixos-24.11* and *nixos-25.11* branches of the Nixpkgs registry, which represent two distinct stable releases. These packages were compiled into two separate lists, from which 325 items were randomly selected. The resulting 650 packages, along with every required dependency, were built and stored locally in the Nix Store before being added to the Gachix cache. This resulted in 5123 packages that were added both to the Nix store and Gachix.

The selection of packages is representative because it is a subset of packages the official binary cache stores. Having packages from two distinct releases ensures coverage across different software versions and ecosystem states.

To evaluate performance, each cache service was initialized on the same machine as the benchmarking tool to minimize network interference. The benchmark then iteratively requested both the Narinfo and the Nix Archive for every package. During this process, the system recorded the end-to-end latency (request send to full respons received) for each request.

The list of packages and all measurements can be found in the benchmarking tool repository.[16]

5.3.2. Result

The median, 95th percentile, 99th percentile, the maximum, mean and standard derivation were computed over all 5123 packages for each cache service. These computations are presented in Table 1 for the Narinfo latency and in Table 2 for the NAR latency.

Cache Service	Median	p95	p99	Max	Mean	Std
gachix	0.849	1.592	2.812	8.844	0.947	0.417
harmonia	2.960	4.469	4.881	31.970	2.673	1.333
nix-serve	4.026	11.455	13.151	19.473	5.178	2.710
nix-serve-ng	1.006	1.985	3.027	23.982	1.127	0.581

Table 1: Summary Statistics for Narinfo Latency (ms)

Cache Service	Median	p95	p99	Max	Mean	Std
gachix	4.812	142.129	840.199	9931.217	49.347	327.198
harmonia	8.530	119.514	604.240	3316.529	41.912	146.475
nix-serve	42.063	101.205	447.337	2749.474	57.757	107.114
nix-serve-ng	7.689	105.879	616.550	4832.431	37.989	182.101

Table 2: Summary Statistics for NAR Latency (ms)

The data is visualized as a boxplot in Figure 5. Notice that the outliers are not visualized.

[16] https://github.com/EphraimSiegfried/thesis-metrics

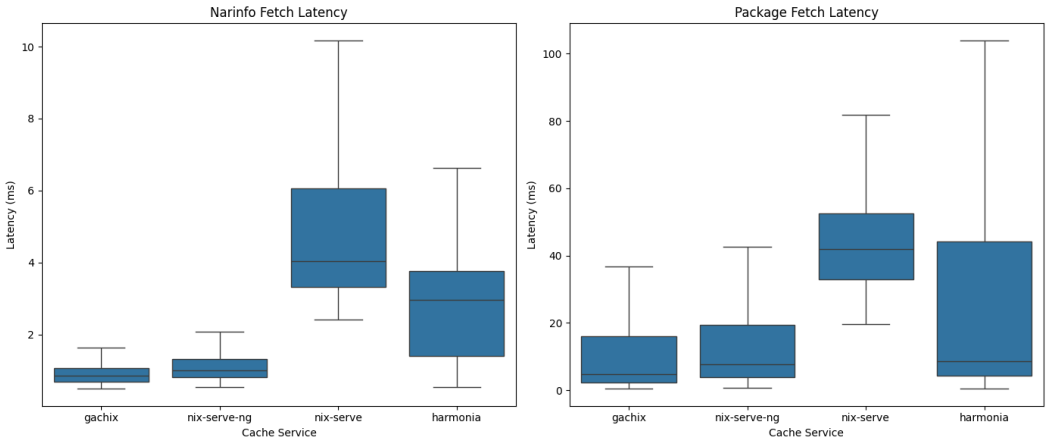

Figure 5: Narinfo Latency Distribution (No outliers)

The scatter plot in Figure 6 illustrates the relationship between package size and latency, with each individual measurement represented as a single point.

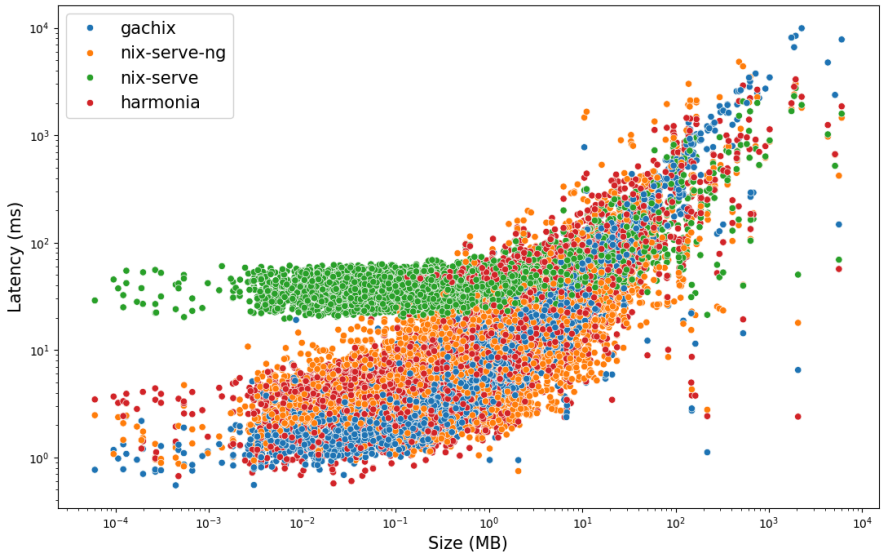

Figure 6: Package size vs Latency

5.3.3. Discussion

From the data we can see that Gachix has the lowest median and lowest mean latency followed by nix-serve-ng when serving Narinfos.

When serving NARs, Gachix has the lowest median latency of (4.8 ms) but has the second highest mean latency. The reason for this observation is that Gachix usually performs well but has a few extreme outliers in which it performs badly. These outliers can be seen in Figure 6, where measurements where the latency is larger than one second is mostly from

Gachix. All of these outliers have in common, that the packages are larger than 136.5 MB. This suggests that Gachix is slower when serving large files but indicates good performance for small to medium sized files.

The reason why *nix-serve* has a much slower latency than the other services is probably because Perl (the language that nix-serve was written in) is an interpreted language and all other languages are compiled.

From the results we can conclude that *gachix* is reasonably fast and can compete with other products in this area.

5.4. Package Storage

This benchmark compares the disk storage usage of Gachix to the cache services presented in Section 5.1.

5.4.1. Methodology

In this experiment, 5123 randomly selected packages from Nixpkgs were added to both the Nix store and Gachix. It's the same packages as specified in Section 5.3.1.

To assess storage consumption, the total storage used by Gachix was measured by the size of its `.git` directory. This was compared against the sum of the size of all 5123 packages in the Nix store.

Note on Comparison: The sum of the package sizes in the Nix store serves as a lower-bound estimate for the storage required by other cache services. This estimate is conservative because it does not account for potential operational overhead or internal metadata that other caching mechanisms might introduce.

5.4.2. Result

The sum of the package sizes in the Nix store is 77.84 GB. The size of the `.git` repository is 13.45 GB. This is a size reduction of 82.72%.

5.4.3. Discussion

We believe there are two primary reasons why we observe this size reduction.

Firstly, Gachix compresses its objects using zlib. [11] The Nix store does not contain any compressed packages.

Secondly, since the Git object database is a Merkle tree and every object is identified by its hash, identical files in the Nix store are only stored once in the Git database. We have computed the amount of objects that have an indegree greater than one, i.e. the objects

which are pointed by more than one tree.[17] We found that out of the 706,848 unique objects reached, 154,371 (21.84%) had an indegree larger than one. This deduplication of files is also a reason why the size is smaller in Gachix.

5.5. Deployment on Systems without Nix

This section shows that Gachix can be deployed on Unix machines without Nix installed.

5.5.1. Methodology

In this experiment we will run Gachix inside a Docker container without Nix installed. We will then show that Gachix can populate its cache by fetching packages from remote Nix daemons. Gachix will add the package *hello*, a standard lightweight example package frequently used for testing. As the remote Nix daemon we will choose the daemon located at the host machine. Therefore this experiment only works on machines with Nix installed.

For this experiment a Dockerfile and Docker Compose file was written which can be found on the Gachix repository. In the Dockerfile, Gachix is built in an environment with Rust installed. After the Gachix binary is built, it is placed in a separate Debian container, where the binary will be run.[18] The docker compose file places the necessary files inside the container and sets configuration values for Gachix.[19] The experiment proceeded as follows:

- Generate a ssh key pair with: `ssh-keygen -t e25519 -N "" -f ~/.ssh/id_ed25519`
- Add the following to the `configuration.nix` file :

  ```
  nix.sshServe.enable = true;
  nix.sshServe.keys = [ "ssh-dss AAAAB3NzaC1k..." ];
  ```

  Ensure that the contents of the generated public key is inside the `keys` list.[20]
- Clone the Gachix source repository with `git clone https://github.com/EphraimSiegfried/gachix.git`
- Launch the container with `docker compose up`
- Add a package to Gachix with `docker exec gachix_service /usr/local/bin/gachix add $(nix build nixpkgs#hello --print-out-paths --no-link)`. The subcommand `nix build nixpkgs#hello --no-link --print-out-paths` will add the package *hello* to the Nix store and print the path of the package to *stdout*.

[17] https://github.com/EphraimSiegfried/gitics/tree/master
[18] https://github.com/EphraimSiegfried/gachix/blob/master/Dockerfile
[19] https://github.com/EphraimSiegfried/gachix/blob/master/docker-compose.yml
[20] https://nix.dev/manual/nix/2.18/package-management/ssh-substituter

5.5.2. Result

The last command should print out something similar to:

```
INFO gachix::git_store::repository: Using an existing Git
repository at ./cache
INFO gachix::git_store::store: Repository contains 0
packages
INFO gachix::git_store::store: Succesfully connected to Nix
daemon at host.docker.internal
INFO gachix::git_store::store: Adding closure for
hello-2.12.2
INFO gachix::git_store::store: Added 5 packages
```

5.5.3. Discussion

The output confirms that the container successfully connected to the host machine. Gachix communicated with the remote daemon to fetch the *hello* package and its closure, resulting in the addition of 5 packages to the database. This demonstrates that Gachix operates effectively on systems without Nix installed and is capable of replicating packages using the Nix daemon protocol.

5.6. Nix Transparency

This experiment verifies whether Gachix correctly implements the Nix binary interface. We confirm this by demonstrating that a user can successfully substitute (fetch) a package using the standard Nix command line tools backed by Gachix.

5.6.1. Methodology

In this test we add the package *hello* to Gachix. We then use the `nix build` command which will try to substitute the package by using binary caches. The experiment proceeds through the following steps:

1. We create a key pair which is used for signing Narinfos with `nix-store --generate-binary-cache-key my-gachix-cache key.private key.public`
2. We add a package to Gachix with `GACHIX__STORE__SIGN_PRIVATE_KEY_PATH=key.private gachix add $(nix build nixpkgs#hello --no-link --print-out-paths)`. The environment variable `GACHIX__STORE__SIGN_PRIVATE_KEY_FILE` tells

Gachix where the private key is located (this could have also been configured using a YAML file).

3. We remove the package from the local Nix store to prevent a local cache hit and ensure the package must be fetched remotely.: `nix store delete nixpkgs#hello`
4. We can now start the Gachix HTTP binary cache server with `gachix serve`. By default this listens on: `http://localhost:8080`.
5. Finally, we fetch the hello package again, explicitly designating the Gachix server as the substituter: `nix build nixpkgs#hello --substituters http://localhost:8080 --trusted-public-keys $(cat key.public) -vv --no-link`. Substituters and trusted public keys are normally specified in a Nix configuration file but can be overridden in the command line.

5.6.2. Result

Upon running the final command, the output logs should display the following:

```
copying path '/nix/store/2bcv91i8fahqghn8dmyr791iaycbsjdd-hello-2.12.2' from 'http://localhost:8080'...
downloading 'http://localhost:8080/nar/cd533301f886090bb173bee7a3aaa67a2b140a8d.nar'...
```

5.6.3. Discussion

This output confirms that Nix successfully fetched the binary from Gachix. Without the `--substituters` and `--trusted-public-keys` flags, Nix would have ignored the server, built the package locally, and logged the compilation steps instead.

Notice that setting `trusted-public-keys` only works if the user executing the Nix command is marked as trusted.

6
Related Work

6.1. Snix

Snix is a re-implementation of Nix in Rust. Although differing internally, Snix implements most Nix interfaces such as the Nix binary interface and the interface to interact with registries such as Nixpkgs. It has individual modules with clear purposes and is more decoupled than the offficial Nix implementation. [15]

It has a custom store implementation where all objects are content addressed. To store objects it uses a Merkle tree with objects similar to objects in Git. [16] It differs from Git by hashing objects with `BLAKE3` algorithm instead of `SHA1`. It also uses the canonical Protobuf serialization format for serializing trees instead tree encoding Git uses. Furthermore, for identifying blobs Git hashes the content of files with a Git specific prefix. The prefix makes the hash of the file very Git specific and not portable to other content-addressed systems. Snix avoids this by only hashing the file content. [17]

Gachix and Snix are quite similar due to Gachix's use of Git for package storage. Snix gains the benefit of fixing Git's specific shortcomings but pays the cost of having to recreate a significant amount of "battle-tested" functionality that Git already provides.

6.2. Extending Cloud Build Systems to Eliminate Transitive Trust

In the paper *Extending Cloud Build Systems to Eliminate Transitive Trust*, the authors address the issue of trust within the Nix package supply chain. [18]

Under the current trust model, each package possesses a fingerprint that consists of its input hashes and an output hash. This fingerprint is signed using the private key of a binary cache (or a Nix builder). A Nix user maintains a set of public keys for trusted caches, which allows them to verify that a package originates from a valid source. Once the signature is

verified, the user implicitly trusts the mapping between the input hashes and the resulting output hash.

However, the authors highlight a critical limitation: the Nix user cannot currently verify whether a package was actually compiled by a trusted builder. To resolve this, the authors propose a solution that extends the signature mechanism, enabling the user to cryptographically verify that the package originated specifically from a trusted builder.

The new fingerprinting mechanism presented in the paper could be incorporated into future versions of Gachix. By doing so, a Nix user would no longer need to place trust in the Gachix instance itself, but could instead rely solely on the builder's signature.

7

Conclusion

In this thesis, we demonstrated that Git's object model can be effectively used to implement a binary cache, which we named Gachix. Gachix operates as a peer-to-peer system that serves build artifacts to Nix clients without requiring a local Nix installation; instead, it populates its cache via other Gachix peers and Nix daemons.

For storing packages we mapped Nix store objects such as directories and files to equivalent trees and blobs. This approach enabled content-addressable storage, resulting in an 82% reduction in storage size compared to the traditional Nix store.

In order to track the dependencies between packages, we have used commit objects. We associate every package with exactly one commit, where the commit tree contains the contents of the package and the parents of the commit represent the packages's runtime dependencies. This has the benefit of efficiently retrieving the dependency closure of a package and also simplifies the synchronization between peers.

Furthermore, we demonstrated that Git trees and blobs can be efficiently transformed into Nix Archives (NARs). This facilitates rapid package delivery to clients; in our benchmarks, Gachix frequently outperformed existing binary cache implementations in terms of serving speed.

Future work should focus on achieving full feature parity with standard Nix binary caches. This includes implementing NAR compression and adding support for missing endpoints, such as `.ls`. Additionally, the replication protocol could be optimized by designing heuristics that prioritize peer contact based on latency. Finally, the trust model could be strengthened by implementing package verification within the distributed system.

Bibliography

[1] E. Dolstra, “The Purely Functional Software Deployment Model,” PhD Thesis, Utrecht University, 2006.

[2] The NixOs Foundation, “How Nix Works.” Accessed: Sept. 19, 2025. [Online]. Available: https://nixos.org/guides/how-nix-works

[3] Nix Dev, “Store Object.” Accessed: Dec. 16, 2025. [Online]. Available: https://nix.dev/manual/nix/2.32/store/store-object.html

[4] Nix Dev, “Input-addressing derivation outputs.” Accessed: Dec. 16, 2025. [Online]. Available: https://nix.dev/manual/nix/2.32/store/derivation/outputs/input-address.html#input-addressing-derivation-outputs

[5] Nix Dev, “Content-Addressing Store Objects.” Accessed: Dec. 15, 2025. [Online]. Available: https://nix.dev/manual/nix/2.29/store/store-object/content-address

[6] Nix Dev, “Nix Archive (NAR) format.” Accessed: Dec. 04, 2025. [Online]. Available: https://nix.dev/manual/nix/2.22/protocols/nix-archive

[7] F. Zakaria, “Nix Binary Cache.” Accessed: Dec. 04, 2025. [Online]. Available: https://fzakaria.github.io/nix-http-binary-cache-api-spec/#/

[8] Nix Devhttps://github.com/NixOS/nix/blob/5f69fd3e8d99889c571f558113786bd05f0e38ee/src/libstore/ssh-store.cc#L201-L220, “Multi-User Mode.” Accessed: Dec. 16, 2025. [Online]. Available: https://nix.dev/manual/nix/2.28/installation/multi-user

[9] NixOS, “nix.” Accessed: Dec. 16, 2025. [Online]. Available: https://github.com/NixOS/nix/blob/5f69fd3e8d99889c571f558113786bd05f0e38ee/src/libstore/ssh-store.cc#L201-L220

[10] Snix, “Operations.” Accessed: Dec. 16, 2025. [Online]. Available: https://snix.dev/docs/reference/nix-daemon-protocol/operations/

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

[12] NixOS, “nix.” Accessed: Dec. 17, 2025. [Online]. Available: https://github.com/NixOS/nix/blob/f435634a29551754d5f7303b0a60cd8fe2df2079/src/libstore/schema.sql

[13] Nix Dev, “Glossary.” Accessed: Dec. 17, 2025. [Online]. Available: https://nix.dev/manual/nix/2.28/glossary

[14] 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

[15] Snix, “Snix.” Accessed: Dec. 16, 2025. [Online]. Available: https://snix.dev/

[16] Snix, “Data Model.” Accessed: Dec. 16, 2025. [Online]. Available: https://snix.dev/docs/components/castore/data-model/

[17] Snix, “Why not Git?.” Accessed: Dec. 16, 2025. [Online]. Available: https://snix.dev/docs/components/castore/why-not-git/

[18] M. Schwaighofer, M. Roland, and R. Mayrhofer, *Extending Cloud Build Systems to Eliminate Transitive Trust*. Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3689944.3696169