A Deep Dive into NixOS: From Configuration To Boot

CS5250: Advanced Operating Systems

Chen Jingwen

A0111764L

National University of Singapore

Abstract

Mature operating systems (e.g. Windows, Fedora) are inherently stateful and imperative, adding layers of complexity by installing or upgrading software. This causes side-effects such as breaking existing software while upgrading shared libraries without maintaining backwards compatibility. NixOS is a Linux distribution designed to be purely functional, where building everything from the kernel to the web browser has no side-effects. System configuration files are written in the *Nix language*, a lazy functional domain specific language with a declarative syntax, and software packages are managed by the *Nix package manager*. A distinct feature of NixOS is the ability to declare the configuration of an entire system in one file, which is then used to build a bootable system deterministically. This report gives an overview and the motivations of NixOS, and a deep dive into how the configuration of an operating system can be derived from a single file.

Contents

1 Introduction **4**

2 Motivation **5**
2.1 Multiple versions . . . 5
2.2 Destructive updates . . . 5
2.3 Rollback difficulties . . . 6
2.4 Non-atomic upgrades . . . 6
2.5 Inability to reproduce builds . . . 6

3 NixOS Architecture **7**
3.1 Package specifications and the Nix expression language . . . 7
3.1.1 Nix expression language . . . 8
3.1.2 Derivations . . . 9
3.2 Nix store . . . 9
3.2.1 Cryptographic hash . . . 9
3.2.2 Source to binary deployment . . . 10
3.2.3 Nix database . . . 10
3.3 Nix package manager . . . 11
3.3.1 Installation . . . 11
3.3.2 Immutability . . . 11
3.3.3 Dependency management . . . 11
3.4 Putting it all together: NixOS . . . 13
3.4.1 configuration.nix . . . 14
3.4.2 System profiles . . . 14
3.4.3 Garbage collection . . . 15
3.5 Advantages . . . 15
3.5.1 Version management . . . 15
3.5.2 Atomicity . . . 16
3.5.3 Easy rollbacks . . . 16
3.5.4 Reproducibility . . . 16

4 Deep Dive: Switching Configurations **18**
4.1 System configuration . . . 18
4.2 nixos-rebuild.sh . . . 18
4.3 nix-instantiate . . . 18
4.4 nix-env . . . 18
4.5 Configuration switch . . . 19
4.6 Observations . . . 19

5 Experience feedback **20**

6 Summary **20**

7 Appendix **21**

A NixOS configuration file **21**

1 Introduction

NixOS is an operating system built in a purely functional manner on top of the Linux kernel. At its heart is **Nix**, a package management system like APT[1] on Ubuntu and RPM[2] on Red Hat.

Nix differentiates itself from other package managers by attempting to solve issues plaguing popular modern operating systems such as the inability to manage multiple versions of a package easily, rollback updates quickly, have a deterministic reproduction of system installations, and solving shared dependency issues.

The core approach of Nix package management is the use of pure functions to describe each package, otherwise known as **Nix expressions**. In each package's Nix expression, there is a **derivation** that describes the steps to take to build the package from source along with its dependencies.

Evaluating the derivations builds the package and stores the resulting contents in the **Nix store**, an immutable content-addressable folder where the entry name contains a hash of all input arguments of the derivation. Nix uses such metadata of the expressions to build a closure of the dependency graph, which can be evaluated lazily and deterministically, since every node in the graph has no side effects. The expressions are written in `.nix` files using the **Nix expression language**, a purely functional lazy programming language.

NixOS is the result of taking the Nix package manager one step further by using it to build an entire operating system from scratch. It aims to extend Nix's features to manage and centralize the configuration of the system, which includes components such as the kernel, network and filesystem drivers, bootloader, and graphical environments.

Eelco Dolstra designed and implemented Nix for his PhD thesis.[3] Armijn Hemel designed the first prototype of NixOS for his Masters thesis[4] and was further developed and documented by Dolstra, Loh and Pierron in a comprehensive paper in 2010.[5] It is under development as an open source project on GitHub,[6] and free to be downloaded and used.[7] It is currently used by commercial deployments worldwide.

We will first discuss the issues facing the current state of operating systems. Then, we will give an overview to the parts of NixOS and how their designs overcome these issues. Finally, we will do a deep dive into the core of the NixOS system, the configuration file `/etc/nixos/configuration.nix`, and follow the steps NixOS takes to turn that file into a bootable system.

[1] *Apt - Debian Wiki*. Mar. 2017. URL: `https://wiki.debian.org/Apt`.

[2] *rpm.org - Home*. Mar. 2017. URL: `http://rpm.org`.

[3] Eelco Dolstra. "The Purely Functional Software Deployment Model". In: *Utrecht University* 56.12 (2006), p. 281. ISSN: 14968975. DOI: `10.1007/s12630-009-9179-6`. URL: `http://www.st.ewi.tudelft.nl/%7B~%7Ddolstra/pubs/phd-thesis.pdf`.

[4] Armijn Hemel. "NixOS: the Nix based operating system". In: (2006).

[5] Eelco Dolstra, Andres Löh, and Nicolas Pierron. "NixOS: A purely functional Linux distribution". In: *Journal of Functional Programming* 20.5-6 (2010), pp. 577–615. ISSN: 0956-7968. DOI: `10.1017/S0956796810000195`.

[6] *Official Nix/Nixpkgs/NixOS*. Apr. 2017. URL: `https://github.com/nixos`.

[7] *NixOS Linux*. Mar. 2017. URL: `http://nixos.org`.

2 Motivation

The fundamental architecture of modern and conventional operating systems is an imperative model: every action that the user or system takes to install, update or remove software is a stateful action that modifies the global state.

New versions of packages overwrite older ones. Some packages are shared and used by other packages in the form of both static and dynamic dependencies. Packages are scattered across the filesystem hierarchy of the operating system. For example in the case of Unix systems, packages are distributed over directories like `/etc, /usr, /bin, /var, /lib`.

The more complex a system grows, the more difficult it is to keep track of where everything is. It is difficult to determine whether a file or directory is required by the system or user, or is an unneeded residual file left behind from a system update some time ago.

The design of the imperative model stems from the early days of computing, and inertia has grown so strongly with an accumulation of efforts over decades, resulting in difficulty in reengineering the operating systems to address the following issues.

2.1 Multiple versions

In a typical Unix system, installing a package `foobar` writes the compiled binary directly to `/usr/bin/foobar`, or some directory in the user's `$PATH` where binaries are located.

Assuming that `foobar-v1` is installed, updating it to some later version, e.g. `foobar-v2`, will overwrite the `v1` binary at `/usr/bin/foobar`. This works well if only one version of the package is needed at any time, and most operating systems assume this by storing only one version of each software package in the system. However, if more than one version of the package is required by the user or system, there are no straightforward methods to get around this assumption.

A common workaround is to include the version of the package in the name directly. This is seen in the `python` package, where the version numbers are part of the package name, i.e. `python27` and `python34`. This approach works as intended, but does not scale well as the number of versions grow, or even in maintaining minor version bumps.

2.2 Destructive updates

The side-effects of a package update in an imperative model might be cascading, non-obvious, and hard to detect.

To build an intuition, let's treat the operating system like a program written in a general purpose programming language. Popular shared libraries such as `glibc` are like the program's global variables with shared mutable state. Different parts of the program are able to access and update these global variables via side-effects, and if care is not taken, it might result in unpredictable states of the system.

Users of dynamically linked libraries (DLL) will otherwise know this problem as the DLL hell, where a shared dependency that their application relies on is unknowingly modified by a third party, leaving their application in a non-working state should there be an incompatible API change.

This issue extends to the metadata of the software packages, such as configuration files and registry values. It is up to the software developer of each package to determine how to safely transition the package's

metadata or on whether to maintain backwards compatibility.

2.3 Rollback difficulties

As a consequence of destructive updates and the lack of multiple versions coexisting in a system, rolling the system back to a previous state is not a simple task. This requires manually reverting the steps taken by the update process, which may not be possible if there were crucial data lost along the way.

Modern operating systems (e.g. Windows[8]), keeps system restore snapshots of the system periodically, but these snapshots take up a large amount of space, and is usually not temporally granular enough to minimize the amount of lost state.

Maintaining the integrity of the system is then left to the responsibility of the user in the form of backups and version control systems, which are not simple software for the average user.

2.4 Non-atomic upgrades

Upgrading an operating system is a perilous and complex process. During a system upgrade, users are told to ensure that their machines have enough battery power to sustain the entire operation, or to be plugged in to a electrical socket. The main reason for this is to prevent the system upgrade from halting abruptly, leaving the core system partially upgraded in an inconsistent state, and with a non-zero chance of bricking the system.

Coupling this with the immense difficulty of restoring a bricked system makes non-atomic system upgrades a painful issue for users.

2.5 Inability to reproduce builds

The transformation actions that the system had taken to reach its current state were never recorded, and as a consequence of destructive updates and mutation of global states, it is difficult to reproduce a system's state deterministically in the same system or on another machine.

[8] *Back up and restore your PC - Windows Help.* Apr. 2017. URL: `https://support.microsoft.com/en-us/help/17127/windows-back-up-restore`.

3 NixOS Architecture

The objective of Nix and NixOS is to address the issues stateful and imperative operating systems face. The architecture of NixOS breaks down into the following components:

- **Nix expression language**: a purely functional and lazy domain specific language to specify package information and declare build rules via expressions in `.nix` files.
- **Derivations**: Description of a package's metadata and build steps. This is the minimal set of data and its dependencies that, when evaluated by Nix, produces an output to be stored in the Nix store.
- **Nix store**: the immutable data store of *all* build outputs, usually in `/nix/store`.
- **Nixpkgs**: The main repository[9] of all public packages contributed by the community. At the time of writing, it has over 6500 packages.
- **Nix package manager**: The package management system that parses Nix expressions, builds the dependency graph, evaluates the build actions, and manages the Nix store.
- **NixOS**: The operating system that builds on top of the Linux kernel, Nix and Nixpkgs.

Each component is designed specifically and purposefully to maintain a purely functional approach to package management (in Nix) and configuration management (in NixOS). In the later sections, we will discuss why a purely functional approach is the key in solving issues with statefulness.

3.1 Package specifications and the Nix expression language

This is a snapshot of the GNU Hello package specification retrieved from the Nixpkgs repository on GitHub.[10] The specification is written in the Nix expression language. Here, the specification is in the form of an expression that defines everything that is needed to build the binary from source.

```
{ stdenv, fetchurl }:

stdenv.mkDerivation rec {
  name = "hello-2.10";

  src = fetchurl {
    url = "mirror://gnu/hello/${name}.tar.gz";
    sha256 = "0ssi1wpaf7plaswqqjwigppsg5fyh99vdlb9kzl7c9lng89ndq1i";
  };

  doCheck = true;
```

[9] *NixOS/nixpkgs: Nix Packages collection.* Apr. 2017. URL: `https://github.com/NixOS/nixpkgs`.

[10] *nixpkgs/default.nix at master — NixOS/nixpkgs.* Apr. 2017. URL: `https://github.com/NixOS/nixpkgs/blob/master/pkgs/applications/misc/hello/default.nix`.

```
  meta = {
    description = "A program that produces a familiar, friendly greeting";
    longDescription = ''
      GNU Hello is a program that prints "Hello, world!" when you run it.
      It is fully customizable.
    '';
    homepage = http://www.gnu.org/software/hello/manual/;
    license = stdenv.lib.licenses.gpl3Plus;
    maintainers = [ stdenv.lib.maintainers.eelco ];
    platforms = stdenv.lib.platforms.all;
  };
}
```

In this particular example, the syntax

```
{ stdenv, fetchurl } : expression
```

defines a function that takes in an argument in the form of an attribute set with two attributes, `stdenv` and `fetchurl`, which are provided by the caller of this function.

An example of an attribute set is the `meta` attribute, which contains a series of `name = expression` pairs surrounded by curly braces.

`stdenv.mkDerivation` is a helper function provided by the NixOS standard environment to build derivations easily via wrapping common build steps like `autotools` on Linux, where we can build most binaries with the `configure`, `make`, and `make install` steps. The argument to the function is an attribute set containing attributes `name, src, doCheck` and `meta`.

The `rec` keyword supports recursion in the function body. This allows expressions in the attribute set to use other named attributes declared in the same set, as seen in the use of `name` in the `src.url` attribute.

3.1.1 Nix expression language

The Nix language is a pure, functional and lazy language with its syntax and semantics designed specially for describing packages and build steps. For example, URIs can be specified as-is without quotes for convenience, as seen in the example's `meta.homepage` attribute.

It is *pure* because functions do not have any side effects when called. There are no mutable variables in the language; all values are immutable once created.

It is *functional* because functions are first class citizens in the Nix language, hence they can be passed as arguments to other functions. This is useful when it comes to creating abstractions of build patterns: it is easy for users to implement a new builder that abstracts over a different build process (e.g. Maven v.s. C++ build steps), and still be able to pass that to a package derivation that expects a builder function but has no dependency on how it is implemented.

It is *lazy* because expressions are not evaluated unless the outer scope needs the result of the expression. For example, this allows computationally-heavy expressions to hide behind a conditional. Known as *call-by-need*, the coupling of the purity and laziness properties of the language has led to performance optimization

methods by memoizing build action results. This technique, created by Dolstra, is known as maximal laziness.[11]

3.1.2 Derivations

The result of evaluating the function `stdenv.mkDerivation` is a resulting file with an extension `.drv`. The derivation contains the bare minimum amount of information that the Nix package manager requires to build the package, which usually include the name and version of the package, paths of the compilers, and any flags and options to be used during the compilation process.

We wish to note that the generation of the derivation file does not actually build the package; it is only built when the derivation is evaluated via some explicit caller. This caller is usually some other derivation that depends on this derivation, or the system itself.

The user does not specify the output path or entry to store the output of the derivation – it is algorithmically generated to be an unique location in the Nix store.

3.2 Nix store

The Nix store is the cornerstone of the Nix and NixOS. It is the target directory where the evaluation of derivations stores their build outputs, which includes binaries, configuration files, directories, and the derivations themselves. By default, it is located at `/nix/store`.

```
[nixos:/]$ ls /nix/store
total 167000
-r--r--r--   1 root root       2319 Jan  1  1970
    00fxmvpccq4xh68anri4x2dr3y663gi4-pango-1.40.3.tar.xz.drv
-r--r--r--   1 root root       2276 Jan  1  1970
    00mka9hvj7hibangxw9jhm87439fgi9h-dhcpcd-6.11.5.tar.xz.drv
dr-xr-xr-x   3 root root       4096 Jan  1  1970 i
    00n5n3g1jlffq11d4mq7hy1d6yr3x91p-unit-script
{ ... }
```

3.2.1 Cryptographic hash

As we've seen in the previous section, the description of the package is a collection of the parameters defined in the derivation itself along with the arguments of the expression. These parameters are the core ingredients that go into creating a unique entry name, usually in the following pattern:

```
/nix/store/$(cryptographic-hash)-$(package-name)-$(version)/$(package-contents)
```

The `cryptographic-hash` is a 160-bit MD5 checksum of the inputs that went into the derivation computation. The entry name generation algorithm will recursively parse and collect the inputs to the derivation, so the checksum computation consists of everything from `stdenv`, environment variables to the version of the `gcc` compiler used in the compilation.

[11] Eelco Dolstra. "Maximal Laziness. An Efficient Interpretation Technique for Purely Functional DSLs". In: *Electronic Notes in Theoretical Computer Science* 238.5 (2009), pp. 81–99. ISSN: 15710661. DOI: 10.1016/j.entcs.2009.09.042.

This is Nix's method for enforcing determinism: a different version of `gcc` might produce a totally different output binary, even if the source is identical. Computing the derivation with a different `gcc` version will ensure that the derivation output will be a brand new entry in the Nix store. Environment variables are used as part of the checksum, but Nix tries to minimize non-determinism by clearing some variables out, like `$PATH`, before computing the derivation.

If we evaluate the GNU Hello package derivation in the previous section, we should expect to see the fetched source tarball, the `.drv` derivation file, and the compiled package output.

```
[nixos:/nix/var/nix/db]$ ls /nix/store/*hello*
/nix/store/g4dl2djh719933klf04bmmrjdswfpzip-hello-2.10.tar.gz.drv
/nix/store/qpskxiffl94sy3awnwprra2id9r911zr-hello-2.10.drv

/nix/store/dd617z4bscvvv6i0d9d1x2ml96pi04nk-hello-2.10:
bin  share
```

3.2.2 Source to binary deployment

With the deterministic and immutability constraint placed into the contents of the Nix store, a transparent source to binary deployment can be implemented, where we can cache and compress pre-compiled binaries and built directories remotely, and cache them into the Nix store when the metadata in the derivations matches. This is an advantage in terms of saving computation power and time.

3.2.3 Nix database

The Nix store maintains a **sqlite3** database at `/nix/var/nix/db/db.sqlite`. This database contains metadata about the paths in the Nix store itself, such as the tracking of dependencies between store paths. This allows the user to query information about a package (gcc) as such:

```
$ nix-store -q --references /nix/store/mpi06h1i531wdjrmp6dnq4hwyrm52hcy-gcc-5.4.0-lib
/nix/store/izxnyg94352qxa4a4783dzgnpy5cwazj-glibc-2.25
/nix/store/mpi06h1i531wdjrmp6dnq4hwyrm52hcy-gcc-5.4.0-lib

$ nix-store -q --referrers /nix/store/mpi06h1i531wdjrmp6dnq4hwyrm52hcy-gcc-5.4.0-lib
/nix/store/mpi06h1i531wdjrmp6dnq4hwyrm52hcy-gcc-5.4.0-lib
/nix/store/42a55ri5xm7mgf6wb2zxlyfiilr5rcb7-groff-1.22.3
/nix/store/yhr0dk4njcmzbgxs0rnmvzc73p2v27ry-boehm-gc-7.6.0
/nix/store/fa02k3adaccc1qd2nf8qc5293i3zlgwy-nix-1.11.8
/nix/store/i0m49mp4pbang9klybr57wzrmd1mham2-nix-repl-1.11.8-2016-02-28
```

This ability to identify each package's references and referrers is crucial to building a dependency closure for reproducibility.

3.3 Nix package manager

The Nix package manager is the orchestrator and gatekeeper of the Nix store: it is the only interface the user has to manipulate the Nix store. It provides command line interfaces to install, update, uninstall and garbage collect Nix store entries.

3.3.1 Installation

A typical use case of installing a new package by using `nix-env -i firefox` invokes the package manager by searching for the Nix expression in the system, and evaluating the derivation. This usually continues with a fetch of the source/pre-built binary over the internet from Nixpkgs,[12] evaluating any derivations that the `firefox` package depends on, and stores the final output accordingly in `/nix/store`.

3.3.2 Immutability

To respect the immutability of the Nix store, locations where the binaries and configurations used to be (e.g. `/usr, /etc, /lib`) are now populated by symlinks to the actual binaries in the Nix store. Nobody other than the Nix package manager should modify `/nix/store` directly.

Once packages are built and stored, they are never rebuilt again since there is absolutely no need to rebuild a package unless some input has been changed. On the other hand, entire trees of packages will need to be rebuilt if their dependency is upgraded or modified. For example, upgrading a core system library like `glibc` will entail the rebuild of almost every package in the system. This is an intended outcome and favourable tradeoff in terms of time and space to solve the shared dependency problem.

3.3.3 Dependency management

Dependencies between packages are managed by creating symbolic links atomically. Since the Nix store contains everything in the system, the package manager will be able to resolve dependencies by creating a symlink in the dependent's Nix store entry to the dependency's entry.

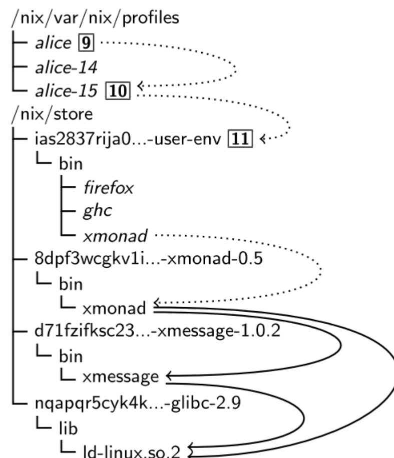


Fig. 5. The Nix store, containing xmonad and some of its dependencies, and profiles

[12] *NixOS/nixpkgs: Nix Packages collection.*

To manage individual user profiles, each user has a symlink in their home directory named `.nix-profile`. This is a symlink to a directory in `/nix/var/nix/profiles`, which is another symlink to the Nix store, where the store entry is a directory representing an user environment. In this figure retrieved from Dolstra's NixOS paper,[13] we can visually see the relationships between Nix store entries. Each dotted line is a symlink, while a solid line is a hardcoded path reference into the binary at pre-compile time.

These user-specific environments, usually implemented as `/bin/bash` on typical Linux systems, are stored as a collection of symlinks to the store.

These user environments are **generational**. Everytime the user installs or updates a package, a new user environment containing the new symlinks are created in the `/nix/store` (as seen in the diagram as `alice-14` and `alice-15`), and the user will be migrated to the new generation, with the previous generations intact and stored in the Nix store. Every generation of the user profile is numbered, so switching between generations of user environments is easily done via `nix-env --rollback` and `nix-env -G $(generation-number)`.

Here, we can see the symlinks of two generations of the user profile symlinked into the Nix store.

```
[/nix/var/nix/profiles/per-user/jin]$ l profile*
lrwxrwxrwx 1 jin users 14 Apr 13 21:58 profile -> profile-2-link
lrwxrwxrwx 1 jin users 60 Apr 13 21:56 profile-1-link
  -> /nix/store/jjafihrvz71nqjnyqvx528a10i0wwc8v-user-environment
lrwxrwxrwx 1 jin users 60 Apr 13 21:58 profile-2-link
  -> /nix/store/7apbs9f4fxycc5w0diqydps9jfy71xkf-user-environment
```

If we look into the user profile directory, there is the `bin/` folder containing the packages that this particular user installed (`hello` and `nix-repl`). Using this symlink model, together with the immutability of the store, the package manager allows users to install and modify packages without interfering other users in the system, unlike typical package managers where package installation is restricted to privileged users.

```
[nixos:/nix/var/nix/profiles/per-user/jin/profile]$ l *
lrwxrwxrwx 1 root root   60 Jan  1  1970 manifest.nix
    -> /nix/store/gwc7gfidpg0hqhdsm6lnzf8q4nspwfnq-env-manifest.nix
lrwxrwxrwx 1 root root   60 Jan  1  1970 share
    -> /nix/store/dd617z4bscvvv6i0d9d1x2ml96pi04nk-hello-2.10/share

bin:
lrwxrwxrwx 1 root root   64 Jan  1  1970 hello
    -> /nix/store/dd617z4bscvvv6i0d9d1x2ml96pi04nk-hello-2.10/bin/hello
lrwxrwxrwx 1 root root   83 Jan  1  1970 nix-repl
    -> /nix/store/i0m49mp4pbang9klybr57wzrmd1mham2-nix-repl-1.11.8-2016-02-28/bin/nix-repl
```

[13] Dolstra, Löh, and Pierron, "NixOS: A purely functional Linux distribution".

3.4 Putting it all together: NixOS

NixOS makes use the Nix philosophy and ecosystem to build an entire operating system from ground up. Instead of using Nix for just managing software packages, NixOS uses it to define, evaluate and build every operating system component from a single configuration file.

Operating system components, such as the Linux kernel, networking stacks, filesystem drivers and graphical systems, often come with stateful configuration files. These files are usually located in `/etc` or `/usr`, and users can directly modify these configuration files. However, in Nix, these configuration files are generated from Nix expressions and tracked deterministically in the Nix store. The only way to modify these configuration files is by specifying options in the main configuration file, and then rebuilding the system with the new configuration.

The implications of extending Nix to system components are immense: multiple versions of systems services can coexist in the same system without their configuration files interfering with each other.

Unlike the Nix packages derivations, NixOS organizes each operating system component into a logical unit called **modules**. Each module declare a set of `options` (e.g. a boolean flag on whether to enable the `sshd` service in the `sshd` module) and the `config` that it contributes to the global system configuration (e.g. if the `sshd` service is enabled, the system should open TCP port 22 for `ssh` connections).

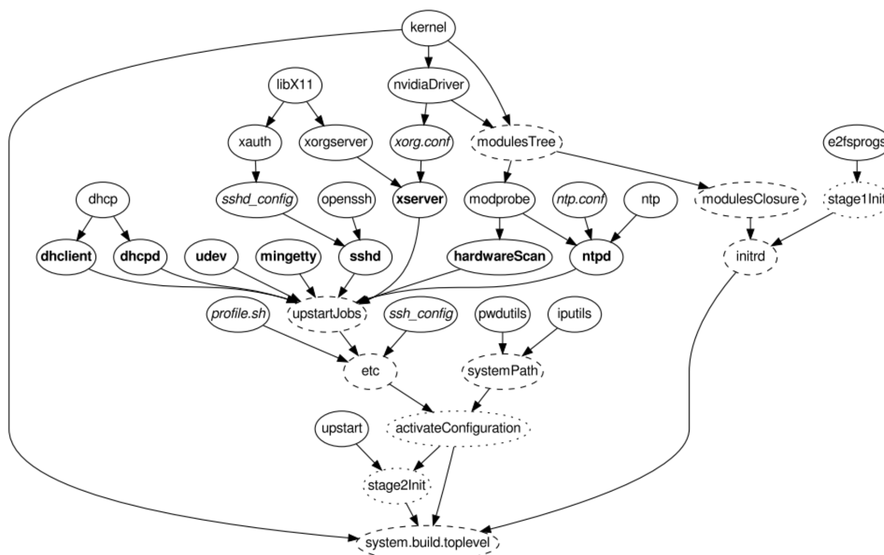


Fig. 10. A small part of the dependency graph of the derivations that build NixOS

In this figure retrieved from Dolstra's 2010 NixOS paper,[14] we can see a dependency closure of a basic Linux system from the perspective of Nix. The source node of each edge represents a dependency of the destination node, where nodes are entries in the Nix store. **Bolded** nodes represents NixOS modules such as `dhcpd`, `udev` and `sshd`. To reproduce a system on another machine, a user simply needs to copy the dependency closure to another machine.

In order to build a dependency closure of the system, there needs to be a single source of truth for the entire system's configuration.

[14] Dolstra, Löh, and Pierron, "NixOS: A purely functional Linux distribution".

3.4.1 configuration.nix

NixOS provides the user with a file at `/etc/nixos/configuration.nix` where the user can specify everything about the system. The following Nix expression is an excerpt from our NixOS machine's configuration file (full file in Appendix A):

```
{ config, pkgs, ... }:
{
  imports = [ ./hardware-configuration.nix ] ;# Include the results of the hardware scan.

  boot.loader.grub.enable = true;
  boot.loader.grub.version = 2;
  boot.loader.grub.device = "/dev/sda"; # or "nodev" for efi only

  services.openssh.enable = true;

  system.stateVersion = "17.03";
}
```

This configuration declares settings for the GRUB bootloader, OpenSSH daemon and environment constants like the selected NixOS system version. It is possible to include other stateful system services, such as creating users and specifying startup bash scripts. There is a separate auto-generated `hardware-configuration.nix` file that contains information about filesystem mount points, partition IDs and other hardware specific configuration.

`config` is a object that contains the global system configuration. This is a significant departure from traditional imperative systems that scatter system configuration throughout different directories. `pkgs` contains information about the installed and available packages that the user can use, and together with the `config` object, the user will be able to declare the entire system configuration in one Nix expression.

From this configuration, Nix can infer that the dependencies include the `grub` and `sshd` Nix modules. Then, Nix constructs a dependency closure of the entire system, including the kernel, by recursively evaluating the dependencies. The result of evaluating the nodes (build actions) in the dependency closure results in a new state of the system, which the package manager stores in a system profile.

3.4.2 System profiles

Similar to the user profiles described in section 3.3.3, a system profile stores the entire state of the system and versioned in terms of generations. The directory `/nix/var/nix/profiles` stores the `system` profiles along with the user profiles.

```
[nixos:/nix/var/nix/profiles]$ ls -al
{ .. }
lrwxrwxrwx  1 root root   13 Apr 13 19:52 system -> system-6-link
lrwxrwxrwx  1 root root   81 Mar 16 15:18 system-1-link
    -> /nix/store/517glkdiqg9yc55ys9ps4gcg6rn2rx7k-nixos-system-nixos-16.09.1829.c88e67d
```

```
lrwxrwxrwx  1 root root   81 Mar 16 15:32 system-2-link
    -> /nix/store/7i3085vy3b6qjj491jlxdk1jw0qxw25z-nixos-system-nixos-16.09.1829.c88e67d
lrwxrwxrwx  1 root root   81 Apr  5 23:45 system-3-link
    -> /nix/store/7ig7sn47gza7yslxpnnyanlii5a53g9g-nixos-system-nixos-16.09.1829.c88e67d
{ .. }
```

Each system profile is a symlink to a directory in the Nix store. Whenever a user makes a change in `configuration.nix` and calls `nixos-rebuild` to evaluate and build the new configuration, Nix creates a new generation of the system profile, and symlinks the main `system` directory to the latest generation.

```
[nixos:/nix/var/nix/profiles]$ ls system
activate                extra-dependencies      initrd          sw
append-initrd-secrets   fine-tune               kernel          system
bin                     firmware                kernel-modules  systemd
configuration-name      init                    kernel-params
etc                     init-interface-version  nixos-version
```

Each `system` directory in the Nix store contains **symlinks** to the crucial components of the system: the `kernel` bzImage, `kernel-modules` including drivers, `systemd` for process orchestration, and `initrd`, the initial ramdisk for the boot process.

The `activate` script is one of the only pieces in NixOS that is stateful, and it is responsible for applying the stateful transformations specified in the configuration file, such as the creation of user accounts.

3.4.3 Garbage collection

Looking into `/run/`, we can see that there are two subfolders, `/run/booted-system` and `/run/current-system`.

```
[nixos:/]$ l /run/*system
lrwxrwxrwx 1 root root 83 Apr 13 20:13 /run/booted-system
    -> /nix/store/33m4qhcdw4wvk9nwpp92ya1pgw1hg37x-nixos-system-nixos-17.03.896.56e5561fbd
lrwxrwxrwx 1 root root 83 Apr 13 20:13 /run/current-system
    -> /nix/store/33m4qhcdw4wvk9nwpp92ya1pgw1hg37x-nixos-system-nixos-17.03.896.56e5561fbd
```

These are two generations of system profiles - `booted-system` representing the last known bootable system configuration, and `current-system` representing the current configuration that the system is running on. NixOS uses the Nix store symlinks in these two folders, along with the other directories in `/nix/var/nix/gcroots`, as the roots of the garbage collection algorithm to remove unused files in `/nix/store`. Files in the Nix store that are not referenced by any of these files are garbage collected. This allows users to reclaim unused space without the risk of deleting files that are in use.

3.5 Advantages

3.5.1 Version management

Given the immutability enforcement of the Nix store, once a package is built, it is never removed from the Nix store unless there are no other entries that depend on it. Installing new versions of packages and

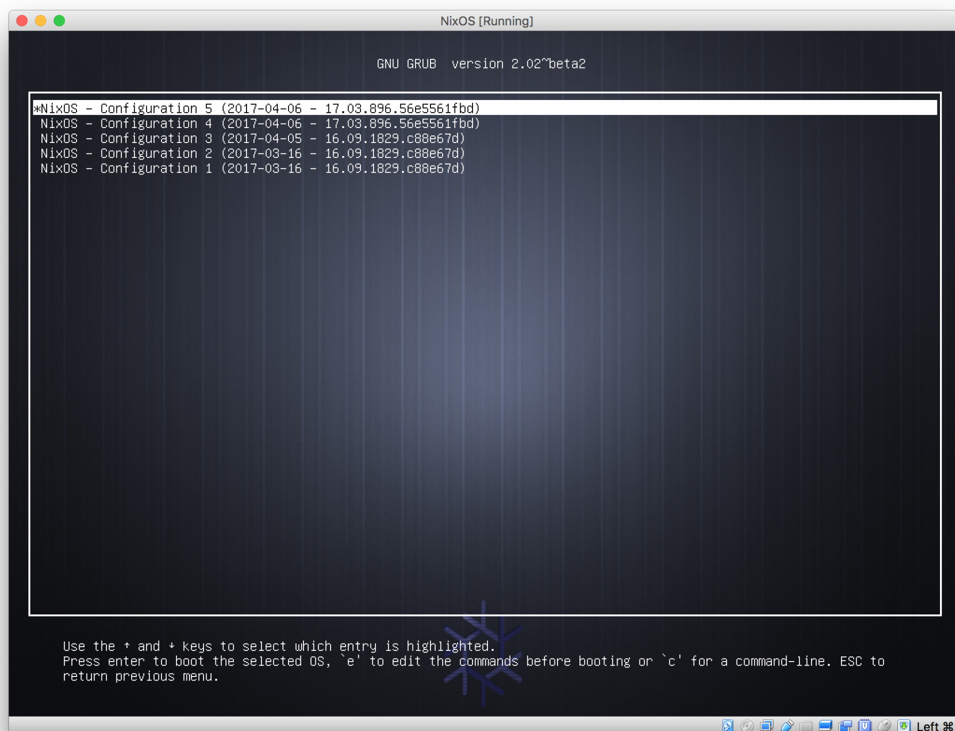
configuration do not overwrite the older ones; new entries with unique hashes and filenames are created instead, and symlinks are updated accordingly by the Nix package manager. This solves the issue of DLL hell, where shared dependencies are destructively updated.

3.5.2 Atomicity

Since neither user or system packages can be overwritten in the Nix store and creation of symlinks is an atomic process, upgrades, rollbacks and other modifications of individual packages or the system configuration as a whole are atomic as well. The system will not be stuck in a limbo state should an upgrade process exit abruptly.

3.5.3 Easy rollbacks

Since contents in the Nix store are never modified after creation, and the user and system profiles are stored generationally in the Nix store, a rollback will entail a simple symlink switch to the previous generation.



As shown in the screenshot, the rollback ability is extended to the GRUB bootloader selection menu, where the user can select an older generation of system configuration if the latest generation is not bootable.

3.5.4 Reproducibility

The purely functional approach to package and configuration management lets us think of system builds as a pure function with the inputs being the configuration in `configuration.nix`, and the output being

the build itself. Build actions do not interfere with each other as there is no concept of a global state, hence maintaining purity. NixOS forbids non-deterministic inputs such as randomness, timestamps or user inputs. This allows us to create a dependency closure of the entire system state deterministically from a configuration file, entailing the ability to **reproduce** the same system on another machine. This closure is portable across systems.

4 Deep Dive: Switching Configurations

We are interested in what goes on behind the scenes when the user presents NixOS with a new system configuration and the steps it will take to switch the system to use the new configuration. Understanding this process will give an insight to the core design principles of NixOS.

To achieve this, we will explore the implementation of NixOS in the Nixpkgs GitHub repository.[15]

4.1 System configuration

The starting point is the configuration file located at `/etc/nixos/configuration.nix`. It is the only file that the user needs to modify in order to change the state of the system. Appendix A shows an example of a system configuration file.

To activate the new configuration, the user will have to use the command `nixos-rebuild --switch`. Defined in `nixpkgs/nixos/modules/installer/tools/tools.nix`,[16] this command calls a bash script with the same name, `nixos-rebuild.sh`.

4.2 nixos-rebuild.sh

In this script,[17] Nix will first try to rebuild Nix itself, as the latest NixOS version might require an updated Nix version.

After updating Nix, the script builds the new system configuration by using `nix-instantiate`, the tool for evaluating Nix expressions.

4.3 nix-instantiate

`nix-instantiate` is a C++ program implemented in a separate repository.[18] It is the entry point to access the Nix derivation compiler that parses Nix expressions and transforms them into derivations, a computer-readable intermediate format that stores the minimum amount of information required to describe dependencies and build actions. At this point, there are no evaluation of build actions yet.

The compiler catches any syntactical and semantical errors at this step. If there are no errors, then the output of `nix-instantiate` will be a derivation file of the configuration.

Next, `nix-env`, the tool for managing Nix environments, will evaluate this derivation and build the derivation tree of dependencies.

4.4 nix-env

`nix-env`[19] is the tool to modify and query user environments. In `nixos-rebuild.sh`, `nix-env` is called with the `--set` flag with the argument as the derivation from the previous step. According to the manual, this

[15] *NixOS/nixpkgs: Nix Packages collection.*

[16] *nixpkgs/tools.nix.* Apr. 2017. URL: `https://github.com/NixOS/nixpkgs/blob/master/nixos/modules/installer/tools/tools.nix`.

[17] *nixpkgs/nixos-rebuild.sh.* Apr. 2017. URL: `https://github.com/NixOS/nixpkgs/blob/master/nixos/modules/installer/tools/nixos-rebuild.sh`.

[18] *nix/nix-instantiate.cc.* Apr. 2017. URL: `https://github.com/NixOS/nix/blob/master/src/nix-instantiate/nix-instantiate.cc`.

[19] *nix/nix-env.cc.* Apr. 2017. URL: `https://github.com/NixOS/nix/blob/master/src/nix-env/nix-env.cc`.

flag modifies the current generation of the system profile to only contain the new configuration derivation, and nothing else. It makes sense to isolate the Nix environment to only contain the derivation that we want to evaluate.

Upon evaluation of the derivation, `nix-env` builds and traverses the dependency tree, and works together with `nix-store` to build a graph of build actions. The `nix-store` C++ program[20] is responsible to manipulating the contents of the Nix store and database and allows users to query the state of the store using the `opQuery` function.

After executing all of the build actions, a new generation of the system profile will be created in the Nix store, containing the build outputs of the configuration's derivation and the necessary symlinks between dependencies.

The derivation evaluation generates the `activate` script along the way, containing stateful commands (user creation, system services start/stop). This script is stored in Nix store entry of the new system profile generation.

The next step is to perform the stateful operations in system to use the newly built configuration.

4.5 Configuration switch

In the final lines of `nixos-rebuild.sh`, a script named `switch-to-configuration` is called:

```
if [ "$action" = switch -o { ... } ]; then
    if ! targetHostCmd $pathToConfig/bin/switch-to-configuration "$action"; then
```

This script[21] is defined at `nixpkgs/nixos/modules/system/activation/switch-to-configuration.pl`.

First, the script will ensure that the bootloader is installed and updated properly. Then, it prepares the switch by figuring out what are the services that needs to be stopped/started/restarted, along with the mounting and unmounting of filesystems and swap by comparing the old and the newly generated `fstab` in the latest `system` profile generation.

Using `systemctl`, the script then begins the configuration switch process by stopping necessary processes, and calling the `activate` script mentioned in the previous section.

When the activation script completes, `systemd` is restarted and `systemctl` reloads/starts/restarts the required services.

If the services are restarted/started without problems with their respective new configurations in the Nix store, then the configuration switch is completed and the system is now running on the new configuration.

4.6 Observations

During the deep dive in the source code, we've observed that a seemingly simple task of a configuration switch involves decoupled parts of NixOS. This suggests that the author of NixOS is strongly influenced by the Unix philosophy ("Do one thing well"). By decoupling the tools, it helped us in easily understanding the separation of concerns between each tool and how they complement each other.

[20] *nix/nix-store.cc*. Apr. 2017. URL: `https://github.com/NixOS/nix/blob/master/src/nix-store/nix-store.cc`.

[21] *nixpkgs/switch-to-configuration.pl*. Apr. 2017. URL: `https://github.com/NixOS/nixpkgs/blob/master/nixos/modules/system/activation/switch-to-configuration.pl`.

5 Experience feedback

To experiment with NixOS, we installed it in a virtual machine using VirtualBox. As the operating system is still under heavy development, there was no graphical user interface or automated prompts to partition and mount filesystems. This is not a big issue as NixOS is well documented with a comprehensive manual with step-by-step instructions to set up a new NixOS system.

The command line interface is usually the first impression a user gets with a tool, but the Nix command line is not the easiest to learn. Running a command-line autocomplete on NixOS 17.03, we get a large list of related commands.

```
[nixos:/etc/nixos]$ nix
nix-build               nix-install-package    nixos-option
nix-channel             nix-instantiate        nixos-rebuild
nix-collect-garbage     nix-log2xml            nixos-version
nix-copy-closure        nixos-build-vms        nix-prefetch-url
nix-daemon              nixos-container        nix-pull
nix-env                 nixos-generate-config  nix-push
nix-generate-patches    nixos-help             nix-shell
nix-hash                nixos-install          nix-store
```

This is not a great user experience for beginners to learn the intricacies of Nix. We've noticed that this issue is under development on the issue tracker,[22] where the solution is to use a single command, `nix`, for all operations.

6 Summary

We have given an overview of the issues plaguing modern and conventional operating systems, and identified the main cause of these issues to be the stateful and imperative model that governs the architecture. Designed to be purely functional from ground up, Nix and NixOS provides certain guarantees such as the lack of side effects in software package dependency and configuration management. This lead to crucial benefits such as the ability to reproduce a system deterministically and solving the shared dependency problem once and for all.

We have taken a deep dive into understanding how NixOS transforms a simple and readable configuration file into a bootable Linux operating system while maintaining the aforementioned advantages of the stateless model.

NixOS brings important advantages to both users and system administrators by solving entire classes of problems with well-executed features, and we hope that the lessons from purely functional model can be eventually extended and implemented in the popular operating systems in use today.

[22] *Redesign of the nix command line.* Apr. 2017. URL: `https://github.com/NixOS/nix/issues/779`.

7 Appendix

A NixOS configuration file

```
{ config, pkgs, ... }:
{
  imports =
    [ # Include the results of the hardware scan.
      ./hardware-configuration.nix
    ];

  # Use the GRUB 2 boot loader.
  boot.loader.grub.enable = true;
  boot.loader.grub.version = 2;
  boot.loader.grub.device = "/dev/sda"; # or "nodev" for efi only

  time.timeZone = "Asia/Singapore";

  # Enable the OpenSSH daemon.
  services.openssh.enable = true;

  # Define a user account. Don’t forget to set a password with passwd.
  users.extraUsers.jin = {
    isNormalUser = true;
    uid = 1000;
    extraGroups = [ "wheel" ];
  };

  # The NixOS release to be compatible with for stateful data such as databases.
  system.stateVersion = "17.03";

  virtualisation.virtualbox.guest.enable = true;
}
```