# NixOS: the Nix based operating system
# INF/SCR-05-91

Armijn Hemel

Aug 24, 2006

**Abstract**

The subject of this thesis is how the Nix package management system can be applied to manage a whole Linux distribution. Many conventional package management systems have drawbacks that Nix fixes. But, Nix has never been used to deploy and manage a whole system.

In this thesis a proof of concept Linux distribution called NixOS is described. NixOS uses the Nix package management system to manage all software that is installed on the system, including the Linux kernel, all software and system services.

Using Nix to manage all software on a system, as is done on NixOS, has several advantages. Developers don't need to be worried that unwanted dependencies are picked up during the build of a software package, since these are completely eliminated. System administrators get the possibility to deploy services using Nix and how they can immediately use all benefits from Nix, including atomic upgrades and rollbacks, without going through a painful cycle of rolling back a service, with all its, possibly also updated, dependencies.

This thesis describes the implementation NixOS, including pitfalls that were encountered and choices that were made. Also shown are some concrete results of running NixOS and how NixOS can be bettered.

# Contents

## Chapter 1

# Introduction

Software management is important on modern computer systems. The way software is installed and maintained affects the ways a system can be effectively used by its users (endusers and developers) and its system adminstrators.

If software on a system is not managed properly (or not managed at all) a system soon turn into chaos. New versions of software are installed on top of old versions, software might be installed or deinstalled incompletely, dependencies of software packages might be missing, parts of two (or more) different versions of a package might get mixed up and it would be hard to see what is actually installed on a system. It would be a maintenance nightmare and the system would become unusable over time.

For these reasons operating systems use package managers. Package managers take care of all the details of installing, removing and configuring software on a system. Every mainstream operating system has some form of package management. Some package managers are quite sophisticated (RPM on Linux), others are quite limited in what they can do (various installation tools on Microsoft Windows).

Package managers help a lot in maintaining software on an operating system, but it seems that many package manager are not using the full potential of what a package manager can do. Newer systems, such as Nix[12] can do a lot more, but have not been used to manage a whole system.

In this thesis it will be shown that using Nix to manage a whole system is very well possible and how it solves a few issues that happen on systems using conventional package management systems for free.

## 1.1 Issues with package managers

Package managers make life for users, developers and administrators a lot easier, by taking care of all the details of installing, removing and configuring software on a system. Package managers are not perfect and many suffer from a number of issues.

In current mainstream operating systems software is installed in fixed locations in the filesystem. On Unix(-like) systems this is often `/bin`, `/usr/bin` or `/usr/local/bin`. Users who have these locations in their default search path (through the environment variable `PATH`) can make use of these programs. If a new program is installed in any of the fixed locations users can immediately make use of the program, without having to change anything to their setup. While this mechanism makes sense, it also has a few implications.

One of the implications is that nearly always only one version of a software package can be installed on the system by the package manager. Installing multiple versions of a software package on a system is possible, but the packager has to do extra effort to make this work, for example by making the other version of the package look like a different package (for example by changing the names of all programs by appending version numbers), or by installing it into a location that differs from that of the original package.

Installing each package in a different location or under a different name avoids packages conflicting with each other, but a user is exposed to this mechanism: if a user wants to use a particular version, the exact location or the name of a certain version of the program has to be known in advance, or the search path for programs has to be adapted to make sure that that particular location is searched as well, and before any other locations other versions might be installed in. So, the user has to have explicit knowledge of where the software is installed on the system.

An upgrade or update of a package is often done by removing the old version of the package from the system and installing the new version in its place. This process is not an atomic operation, because during removal of the old version and installation of the new version there is a time window where a user is still able to start the program, even though it is not even completely installed, or removed, yet. For a short period of time the two programs are mixed.

Rollbacks to an older installed version of a package are difficult. Extra state has to be kept, such as the package itself, and all its configuration information. If a package has dependencies, these have to be be rolled back to the appropriate state as well. Only some package managers keep state, but only if they are explictely configured to do so.

Packagers have to be very careful when building packages to prevent that unwanted and undeclared dependencies are used during the build of a package. The configuration process of many packages often contains hardcoded links to tools and other packages, or it searches for tools and other packages on the system, which are not declared as a dependency by the build scripts. If these packages are used by the program, but not deployed on a target machine, the package might fail to run. In such situations the dependencies of the package are not complete. If the target system has packages installed which satisfy the dependencies of the package, for example because the dependency system is based on names, but which actually do not implement the same functionality, (for example, there are several packages that have the same name, such as the `aterm` library and the `aterm` terminal) the package will install, but might not run. In this situation the dependency is not correct.

### 1.1.1 Requirements for a better system

**Multiple versions** It should be possible to install and use multiple versions or variants of a particular software package next to each other, without one version of the package interfering with another version.

**Atomic updates** Updates of software packages should be done in an atomic fashion. That is, the update process should never interfere with the execution of a program in any way.

**Atomic rollbacks** If a program does not work correctly it should be possible to roll back to an older version easily and atomically, complete with its configuration information.

**Correctness and completeness of dependencies** Every dependency of a package also has to be installed when that package is installed. Also, the right version of the dependency should be installed. All the dependencies for a package should be complete and correct.

Most current software deployment systems don't offer any or just a few of these features.

## 1.2 Existing Unix package management systems

In the Unix family of operating systems there is a huge amount of package management systems available. Every Unix system from every vendor comes with its own package management system and often there are third party replacement package management systems available.

Linux has grown to be one of the most popular operating systems in the Unix and Unix-like operating systems family.

Unlike other Unix(-like) operating systems Linux is not a whole operating system from one single vendor, but merely a kernel developed by a lot of companies and individuals from all over the world. What is commonly referred to "Linux" is in fact a combination of various programs, which are maintained by different independent developer groups. For example, the Linux kernel is maintained by the Linux kernel team, while the C library, compiler and many utilities needed to create a functioning system are maintained by the GNU project. The combination of all these different software packages, along with an installer, maintenance tools and documentation, is often referred to as a "distribution". Popular distributions are Red Hat Enterprise Linux, Fedora Core Linux, SUSE Linux, Debian GNU/Linux, Ubuntu, Slackware and Gentoo. In total there are nearly 500 Linux distributions [1] and this number has been growing steadily over the years.

Because Linux is just a kernel and there is no central organization defining how a Linux-based operating system should be created, there is also no standard way to install software on a Linux system.

On the majority of Linux systems software is installed either using the RPM package manager (SUSE Linux, Red Hat Linux and Fedora Core are prime examples), dpkg (Debian GNU/Linux, Ubuntu and others), or a home-grown solution (like Gentoo Linux and many others).

### 1.2.1 RPM (Linux)

The most popular package management system on Linux is RPM[2]. The RPM convention is that packages install in standard, fixed, locations such as `/usr/bin`.

#### Installing multiple versions

Installing multiple versions of software packages is possible with RPM, as long as files inside the packages don't try to install in the same location. If two files share the same location there is a conflict.

#### Atomic updates

In a default RPM setup updating a package is not done in an atomic way. The update process hides nothing from the user. During an update action the old software is overwritten by the new software and it is therefore possible that when a users starts the program during the update, files from both the old version and new version are used.

### Rollbacks

RPM installs and updates can be transactional (but they do not have the ACID properties from databases). When one or several packages are installed with RPM it is put in a transaction. The RPM database keeps a record of which transaction a particular package was installed in. If a package is upgraded or deleted the old version is first packed into a new RPM file and kept in a directory. Rollbacks need to be explicitely enabled by administrators and are based on times/dates, not on versions[3][4].

### Dependencies

Dependencies in RPM (both at run time and at build time) can be declared in various ways, including other RPM packages, libraries, paths and capabilities, all of which could be satisfied by different packages.

If a dependency is a RPM package then any RPM package that has the same name as the RPM package that is needed is enough to satisfy the dependency. The dependencies can be very generic (just a name), or more specific (name and version number).

Two different packages could both provide a library called `libfoo.so.1`, both of which could implement completely different functionality. If a package needs `libfoo.so.1` the dependency will be satisfied by either of the two packages.

The same is true for paths. Two different packages could provide a program that would be installed in the same path, for example `/foo/bar/bin/baz` with completely different behaviour. If a package would depend on this path either two packages would satisfy the dependency.

Capabilities are a bit fuzzier than the previously mentioned dependency, because they describe a functionality, such as “webserver” or “mailserver”. These functionalities often describe well known functionality, where different implementations can be used interchangebly, because they work according to well defined interfaces.

RPM is a system for binary deployment, but it also offers an environment to build packages. Builds of RPM packages are done using a so called “RPM spec file”, which is a recipe for building the package. Inside this file configuration information is recorded, such as which files have to be installed, where files have to be installed, which configuration options have to be used and what dependencies a package has, and so on.

RPM does not enforce that all dependencies are met during a build, or during an install. It is not guaranteed that a package built on one system will actually install on another system.

A drawback of RPM is that during builds of RPM packages on a system it matters a lot what software is already installed on the system. One of the things that can happen is that the build process automatically detects software that is installed on the system and subsequently uses it, without it being declared as a dependency in RPM. A package can build on a developer's machine and install perfectly on a deployment machine, but fail to run, because the non-declared dependencies are not met on the deployment machine.

The whole environment, including the search path for executables, `PATH`, is left intact during a build with RPM. If a developer happens to have certain paths in `PATH` containing software not maintained by RPM, the build process can pick up this software during the build and introduce unwanted dependencies that are not detected by RPM. As these dependencies are not recorded they can not be fulfilled during install time, or go unnoticed until run time, when a package suddenly breaks.

### 1.2.2 FreeBSD

Another package management philosophy can be seen on the BSD operating systems (and, for that matter, all other commercial Unix operating systems). The BSD systems setup diffr from the Linux one. First of all, there is a distinction between a "base system" and software that is installed additionally. The base system consists of the kernel, C library and a set of tools that give a minimal, yet complete working Unix environment. The base system is not managed by any package manager.

Additional software is installed in different part of the filesystem hierarchy, apart from packages in the base system. Typically additional packages are installed under `/usr/local`. The most popular BSD operating system is FreeBSD.

In FreeBSD additional software packages are kept in the "ports collection"[5]. The ports collection contains descriptions how to build certain packages, including patches. This distinction between the base system and ports means that there are two different ways to install software on the system.

#### Installing multiple versions

A software package can be installed as a part of the base system, or as a port. Inside the base system only one version of a package can be installed. If multiple versions are installed, one of these is always from the ports, since the base system can only contain one version, or all versions are installed from the ports collection. As with other package managers, it is necessary to know the exact name and path of the version you need.

### Atomic updates

When a new base system is installed, policy says that the system should be rebooted to single user mode by the root user when installing the new base system[8]. This prevents users from accidentily having logged into the system when the update is done. This is just policy and the install mechanisms don't enforce this. The install process itself is not done in an atomic way.

Ports are first built in a separate build directory, before installation. The default install process will not remove the old port first, but simply overwrite the old package and record the new package in the database. If the old package is subsequently removed files from the old package will be removed, even if these are also part of the new package.

To avoid this the `portupgrade` is frequently used. This tool first deinstalls and backs up a previously installed port before installing a new port and reinstalls the old version in case the new install fails.

### Rollbacks

Packages in the base system are supposed to be installed as one set of packages. A rollback of a package in the base system typically means rolling back the whole base system.

Tools like `portupgrade` and `portdowngrade` [6] help with keeping management of software installed from the ports a bit sane but doing a rollback is often a chore, because the tools don't keep backups themselves[7].

### Dependencies

The base system has no concept of dependencies as it is built and deployed as if it were one package. It can be configured to include or exclude certain features.

Packages inside the ports collection only have dependencies on other packages in the ports collection. The base system is always assumed to just "be there". Like with RPM systems build time dependencies and run time dependencies can accidentily be picked up during compile time.

## 1.3 Nix package management system

The problems described above are not impossible to solve and are in fact already solved in the the Nix deployment system[12]. Nix is a modern and versatile software deployment system. Its main merit is safe and complete installation of software.

```
{stdenv, fetchurl, ncurses}: [1]

stdenv.mkDerivation {
 name = "vim-7.0";
 builder = ./builder.sh; [2]
 src = fetchurl {
  url = ftp://ftp.vim.org/pub/vim/unix/vim-7.0.tar.bz2; [3]
  md5 = "4ca69757678272f718b1041c810d82d8";
 };
 buildInputs = [ncurses];
}
```

Figure 1.1: A Nix expression for the Vim editor

With Nix it is possible to always differentiate between different versions of software, even if the difference is not in the software itself, but in the configuration options that were used to build the software, or when something upstream in the build process differs, for example in the toolchain (compiler, linker, assembler and so on) or other dependencies that were used to build the software.

It can tell versions apart by taking all "inputs" of the build process of a package, like other packages the package depends on, configuration options, compile flags, environment variables, sources and so on, to compute a unique cryptograhic hash. This hash is embedded into the path the software is installed into. Nix keeps all packages it builds in the so called "Nix store".

The build process is described by a Nix expression. An expression can take parameters, just like a function in a programming language. An expression for the Vim editor can be found in figure 1.1. The parameters [1], the builder script [2] and the sources [3] for the Vim editor itself are inputs. The result of evaluating this expression is that Vim will be compiled and installed into the Nix store in a path containing the hash that Nix computed using all inputs.

If one of the inputs is changed and the package is rebuilt, a different hash is computed and the package will be installed into a different path in the Nix store. This way an upgrade of a software package will not overwrite a previously installed version. That is, there is no "destructive upgrading" in Nix, as there is in other software deployment systems.

When a package is added to the Nix store it is marked as "read-only" to prevent that its contents are changed later on. Because packages cannot be changed after installation packages the installation process becomes deterministic: a build of a package with exactly the same parameters always yields the same result.

When a software package is installed with Nix it is guaranteed that all depen-

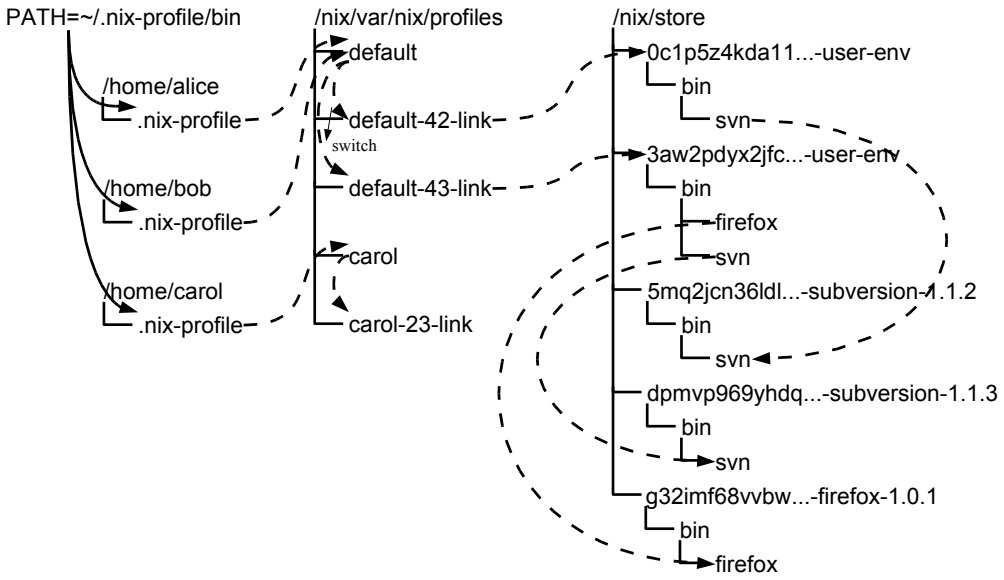


Figure 1.2: Nix profiles (from [12], figure 2.11)

dencies are installed as well. This is because Nix only relies on what is installed inside the Nix store. If a dependency for a package is missing in the Nix store, it will be installed by Nix.

If a user installs software using Nix, he is not confronted with the hashes that Nix uses. Nix creates a user environment instead, based on which packages a user has chosen to install. A user environment is nothing but a set of symbolic links into specific versions of the package the user wants to have installed. When a package is added to a user environment it is first installed inside the Nix store, with a unique path containing the Nix hash. After that a new user environment is created, containing all symbolic links from the old user environment and symbolic links to the new package. As soon as the user installs or deinstalls a package the user environment is updated in an atomic operation.

A user can use any of the various generations of a user environment. By default the user uses the latest generation of his user environment, but it is easy to switch. All generations of a user environment, including the current link to the user environment that is used, is called a profile (see figure 1.2).

Switching between two different versions of a profile only involves making a symbolic link to the right profile, which can be done in one atomic operation. Because old versions from packages are not directly deleted from the harddisk, old instances that were already running can keep running without a hitch.

#### Installing multiple versions

Multiple versions of a package can be safely installed next to each other. They do not conflict because of the way Nix uses cryptographic hashes in the path the software is installed into.

#### Atomic updates

Updates in Nix are atomic. A new package will not be availabe in a Nix user environment unless it has been completely installed inside the Nix store under a certain path with a cryptographic hash. The user never directly interacts with the paths inside the Nix store, but only with the user environment. Switching between an old user environment, without the new package and the new environment is an atomic operation in Nix.

#### Rollbacks

Because packages can be installed next to each other with Nix and packages are not overwritten during the update of a package, it is easy to do a rollback to a previous version. The rollback of a package is the same action as an update or install of a package and can be done with an atomic operation.

#### Dependencies

Dependencies are guaranteed to be correct and complete in Nix. During the build process of a software package these dependencies are recorded and will be installed when the software package is installed.

### 1.4 Goals

The Nix deployment system provides all the needed basic functionality, but so far it has not been used to manage a complete system. Nix is being used on many systems, but always next to another package manager, such as RPM. That is, the base operating system is not managed by Nix.

The following questions that arose during this project were:

1. Can Nix be used as a package manager to manage a complete system?
2. Can Nix be used to manage and store system configurations? If not, why not?

To answer these questions a Linux distribution, called “NixOS” was made. Linux was chosen as the basis for NixOS for various reasons:

1. The Linux kernel is developed and distributed separately from the rest of the Linux operating system, like the C library and tools. Because all core components are distributed and developed separately Linux is more componentized than other Unix(-like) operating systems, such as Solaris and FreeBSD, where the core components are developed and released as a whole.

2. The Nix deployment system has been developed and tested primarily on Linux, therefore it currently is the most mature platform for developing NixOS.

## 1.5 Issues for NixOS

Nix has proven itself as a reliable software deployment system for many "normal" packages, but it has not been used to install and manage a whole system, including the kernel and programs that are started during system startup.

### 1.5.1 Purifying the Nix Packages collection

The Nix Packages collection, a collection of software packages that can be installed with Nix and which is maintained by the Nix developers, could not be built on a pure system. A pure system is a system where all software packages are managed by Nix and where there are no fixed locations on the system.

The scripts that manage making a Nix environment expected certain tools to already be present in fixed locations, such as `/usr/bin/gcc`, which is not true in a pure Nix environment. For NixOS these scripts had to be rewritten.

### 1.5.2 Linux kernel

The kernel is the software that ultimately drives the computer. It makes sure processes are scheduled correctly (starting, stopping, etcetera), that programs can make use of hardware devices, and a lot more. Only one kernel can be running at any time.

The Linux kernel supports loading kernel modules at runtime. Kernel modules are small pieces of software that add functionality to the kernel, for example a device driver for a certain piece of hardware. Many modules are, for various reasons, not included in the official Linux kernel. These so called external kernel modules could still be at an alpha quality level and not stable enough to be included, or have a license that is incompatible with the license of the Linux kernel, which makes it legally impossible to distribute them with the main kernel.

External kernel modules have to be rebuilt manually by the user whenever a new kernel is installed. The package manager will not take care of compiling the extra kernel modules for a new kernel. With Nix combinations of these external modules and the kernel can be built and deployed more easily.

The way each separate kernel module is installed differs per module. Some modules are installed in a separate directory, others are put it in the same directory as the modules that were installed with the kernel. This approach conflicts with the way packages are installed in the Nix store, where a package should not change after it has been installed.

If a module needs to be loaded into the kernel, the tools that load the module first have to find the right module, for which additional management needs to be done. The tools that load modules into the kernel expect all modules to be present in one directory, for example `/lib/modules/`. If the kernel and modules are kept in separate directories in the Nix store, they need to be combined for the tools to work properly.

### 1.5.3 Services

An important part of any operating system are "services". A "service" is a program or set of programs that adds a particular functionality to a system, such as the "networking service", or a "web server". There are many examples of services on Linux-systems, most of which are started at boot time. Services are started by the so called "init scripts". On most conventional Linux systems these scripts can be found in the `/etc/init.d` directory.

Many services require a certain order in which they have to be started. For example, a web server won't be able to run if the networking service is not started. Frameworks that do this exist for many Linux distributions. So far, services deployment using Nix has been done on a 'one-off' basis[10]. There is no generic framework for Nix for deploying services, which takes things like the order in which services have to be started and stopped in account.

Services are unique from other programs because, unlike "normal" programs: often one version can run at a time with one particular configuration. For example, only one instance of the remote login service (`sshd`) can be actively listening on TCP port 22 on a network interface of a machine to accept connections to that machine. More instances of the remote login service can be active using on other ports, but for the configuration that uses TCP port 22 only one service can be active.

There are run time dependencies in a service, which cannot be expressed at build time, such as "for this service to work, networking needs to be enabled", or dependencies which express a more generic service, such as "needs a mailserver",

without specifying which implementation is needed. The implementation of a dependency like "mailserver" doesn't matter, as long as there is a service that accepts and sends mail using a certain protocol. Other dependencies are optional, such as "a logging service would be nice, but the service can still run just fine if the logging service is not present".

Another issue with services is that many services keep state configuration files, which should not be kept inside the Nix store. An example of this is the system password file, which stores account information of users. If this file is kept inside the Nix store in several versions, with various programs using different versions of the password file, it is not known what will happen. Maybe some accounts will be disabled or not exist in some situations, but perhaps, and even worse, old logins are reactivated, or two existing accounts may clash. As a rule of thumb, mutable state should not be kept inside the Nix store.

These restrictions make installing and managing (starting, stopping, restarting), of a service different from managing other software.

### 1.5.4 NixOS installer

NixOS would not be complete without an installer. The main task of the installer is preparing the target drive (partitioning, formatting), initializing the Nix database, registering packages, installing packages, installing and configuring the bootloader, etcetera.

NixOS has to be completely self-contained, so the NixOS installer has to be built using Nix as well. It is highly desirable to build NixOS on a NixOS system, without having to rely on another Linux distribution to bootstrap NixOS.

### 1.5.5 NixOS contributions

The design of NixOS can be split in a few subgoals, that are relatively independent of each other:

1. Purify the Nix packages collection so the whole build process is clean.
2. Build and install a bootable Linux kernel using Nix.
3. Make a framework for managing services.
4. Make NixOS installable on real hardware, for example via a CD. The install image has to be created via Nix itself, so new versions of NixOS can be made on NixOS itself.

## 1.6 Roadmap

This thesis is structured as follows. This chapter discussed deployment systems for Unix (and Linux in particular) and their characteristics and also described our solution, the Nix deployment system, as well as a high level description of the NixOS will be given. Chapter 2 describes how the purification of the Nix Packages collection was done. Chapter 3 describes how the Linux kernel is built with Nix and how external kernel modules are built and deployed. In chapter 4 services will be described. Chapter 5 will briefly touch configuration management. In chapter 6 the design and implementation of the NixOS installer will be explained. Chapter 7 presents some results of running and using NixOS and chapter 8 gives a list of recommendations of how to improve NixOS.

Chapter 2

# Purifying the Nix Packages collection

## 2.1 The Nix `stdenv`

The Nix Packages collection contains a component, called `stdenv`, that includes a minimal set of tools and libraries that are necessary to compile C and C++ packages. This `stdenv` contains tools such as a Bourne (compatible) Unix shell and Unix tools such as `rm` and `grep`, a C/C++ compiler, linker, assembler, and so on. In the Nix Packages this standard environment is passed to definitions of packages as a dependency (see figure 2.1).

For other operating systems the standard environment is not as complete as the environment for Linux.

### 2.1.1 Creating a pure environment

There are a few ways to realise an environment that is needed by Nix. The easiest way is to use the tools that are already installed on the host system (for example `/usr/bin/gcc`). This used to be the default for the Nix Packages collection on Linux and is still the default on all other platforms than Linux.

The result is that the environment is built in an "impure" way and Nix does not always have control over the tools that are used to realize the standard environment. This is the case if the tools are installed using another package

```
vim = (import ../applications/editors/vim) {
  inherit fetchurl stdenv ncurses;
};
```

Figure 2.1: Package definition in the Nix Packages collection.

manager, such as RPM. The packages that the standard environment is built with by Nix can be changed without Nix noticing it and can lead to environments that differ, but which are still seen by Nix as the same.

An alternative to using the tools the host system offers is to use tools that are built with Nix to realize a standard environment. Using Nix ensures purity, but it leads to a bootstrapping problem: without an environment programs can't be built with Nix, without programs built with Nix an environment can't be built.

To counter this problem there are a few solutions:

1. Install tools with Nix and mimic an "impure" environment.
2. modify the Nix scripts in such a way that programs from the Nix store can be used.
3. use pre-built tools to build a pure Nix environment.

The first solution could be realized by making links from a well know location in the filesystem (such as `/usr/bin/`) to the programs in the Nix store. While you can do without having the tools that are needed in the well known location strict dependencies are still not recorded this way. Changing the location in the Nix store from one program in `/usr/bin/` to another version in the Nix store will not be noticed by Nix, making it as bad as the original situation.

The second solution replaces calls to external programs in the Nix scripts with calls to the fully qualified path of equivalent programs in the Nix store. The result is pure, since no programs outside the Nix store are used. However, for this to work there already needs to be a fully installed Nix environment, so the bootstrap problem is still present.

Another issue is the potential explosion of the number of possible standard environments. If the standard Nix environments in two Nix installations differ they will also produce different standard environments. Because they are different they cannot be shared.

The third solution uses pre-built tools and libraries to create a pure standard environment. These pre-built linked tools are treated as source inputs (not built by Nix) in the build process. These tools are partly distributed inside the Nix packages collection and partly downloaded from the Nix website. Because they are treated by Nix as an input, they are also used in the calculation of derivations. Changing the inputs will be recorded by Nix and a new

Because the same pre-built tools are used the result (the standard environment) is the same, which makes sharing easier.

The drawback of the third solution is that it involves some manual work to compile the tools. For every new platform for which support is added to Nix a

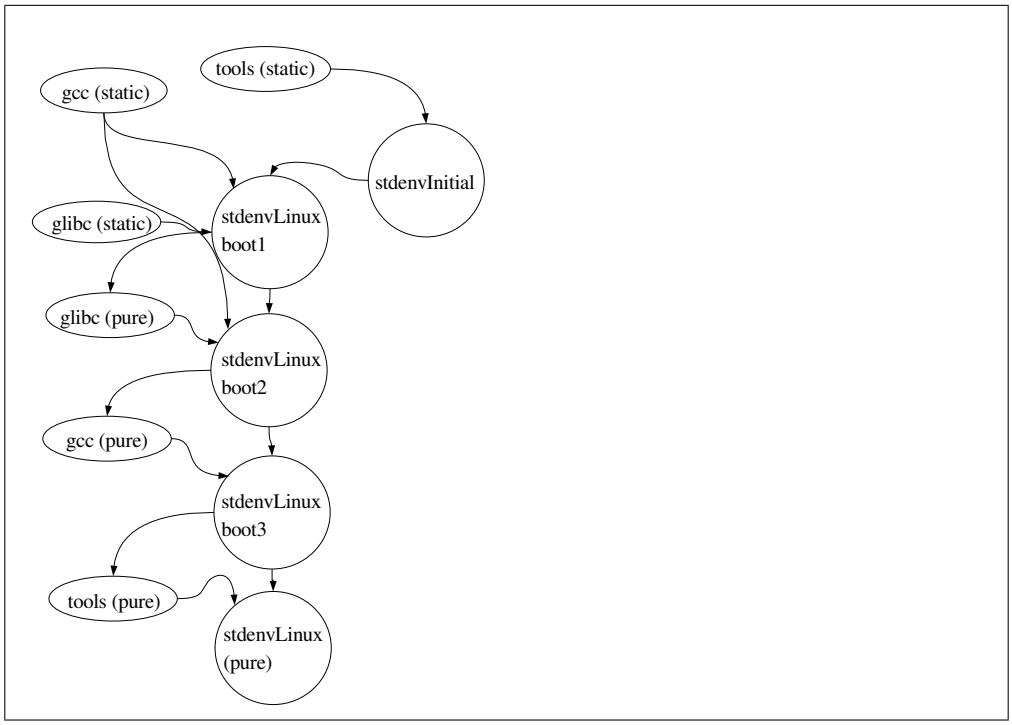


Figure 2.2: The flow of packages in build process of a completely pure environment using static tools in the Nix packages collection.

new set of pre-built tools has to be made and tested.

With this pre-built environment it is possible to completely bootstrap a new initial environment inside a pure Nix system like NixOS, eliminate all impure paths from packages and be completely self-building, whether building on a pure or impure system.

The third approach is the one taken in the Nix Packages collection. Pre-built tools are used in various phases to create a completely pure environment. In every new phase a tool, library or set of pre-built tools is replaced by a newly built, pure, equivalent, until in the end the whole environment is pure (see figure 2.2). To make the process easier and more self contained these tools are statically linked and need no external libraries to run.

The `stdenv` is built in several stages. First, the pre-built tools are used to set up an initial environment (`stdenvInitial`). Using this environment, the compiler and C library a new environment is created (`stdenvLinuxBoot1`). Using the `stdenvLinuxBoot1` environment a new C library is built. The C library in `stdenvLinuxBoot1` is replaced with this new C library to create `stdenvLinuxBoot2`. With `stdenvLinuxBoot2` a new compiler is built, which

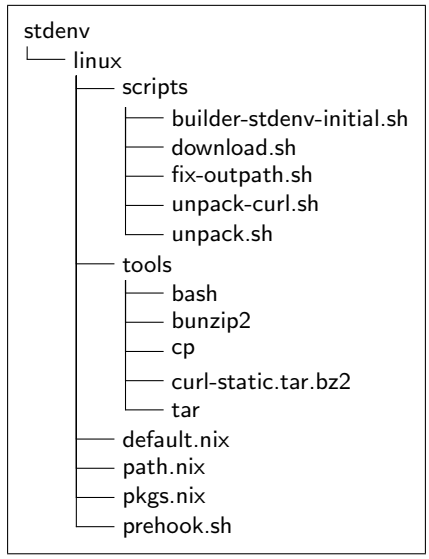


Figure 2.3: Directory structure of the buildscripts of the static Linux environment

replaces the compiler in `stdenvLinuxBoot2` to create `stdenvLinuxBoot3`. With `stdenvLinuxBoot3` all tools that haven't been rebuilt yet are rebuilt to create `stdenvLinux`. This is the final environment with which all other packages in the Nix Packages collection are built.

### 2.1.2 Implementation

The static environment as used on NixOS can be found in the subdirectory `stdenv/linux` of the Nix packages collection (see figure 2.3). The subdirectory `tools` contains four statically linked binaries: `bash`, for executing all shell scripts found in the directory `scripts`, `bunzip2` and `tar` to unpack archives with more tools and `cp` to copy files and scripts to other locations. Also present is a statically linked `curl`, which is stored in an archive and unpacked before being used.

[4] The `allPackages` function is a parameter which is passed on from the outside and which describes the set of packages for which build definitions were defined. This will be the Nix Packages collection in most situations.

[5] The first step is to unpack the tarball containing a statically linked `curl`. The `curl` program is now available to other functions to download files from the network.

[6] [7] There are two convenience functions for downloading and unpacking archives. Using the statically linked tools the downloaded archives are unpacked and de-

```
{allPackages}: 4

rec {

  curl = derivation {  5
    name = "curl";
    builder = ./tools/bash;
    tar = ./tools/tar;
    bunzip2 = ./tools/bunzip2;
    cp = ./tools/cp;
    curl = ./tools/curl-7.15.1-static.tar.bz2;
    system = "i686-linux";
    args = [ ./scripts/unpack-curl.sh ];
  };
```

```
 download = {url, md5, pkgname}: derivation {  6
   name = baseNameOf (toString url);
   system = "i686-linux";
   builder = ./tools/bash;
   inherit curl url;
   args = [ ./scripts/download.sh ];

   outputHashAlgo = "md5";
   outputHash = md5;
 };

 downloadAndUnpack =    7
 {url, md5, pkgname}:
 derivation {
   name = pkgname;
   system = "i686-linux";
   builder = ./tools/bash;
   tar = ./tools/tar;
   bunzip2 = ./tools/bunzip2;
   cp = ./tools/cp;
   args = [ ./scripts/unpack.sh ];
   tarball = download {inherit url md5 pkgname;};
 };
```

```
staticTools = downloadAndUnpack {  8
  url =
    http://nix/dist/tarballs/stdenv-linux/static-tools.tar.bz2;
  pkgname = "static-tools";
  md5 = "90578c603079313123e8c754a85e40d7";
};
```

```
binutils = downloadAndUnpack { ... }  9

gcc = (downloadAndUnpack { ... }      10
 // { langC = true; langCC = false; langF77 = false; };

glibc = downloadAndUnpack { ... }     11
```

compressed.

8 More statically linked tools are downloaded from a pre-defined location and unpacked. These tools form the minimal set of tools needed for Nix to, given a compiler, help bootstrap a new compiler. The tools in the package are `gzip`, `bzip2`, `coreutils`, `make`, `patch`, `grep`, `findutils`, `tar`, `sed`, `gawk` and `patchelf`.

9 10 11 A compiler toolchain, consisting of a statically linked compiler (`gcc`), statically linked tools (assembler, linker, archiver in `binutils`) and a C library (`glibc`) is downloaded. Together with the previously downloaded static tools this is enough to completely bootstrap a standard environment that is suitable for Nix to compile programs.

12 The initial environment just contains the downloaded tools and the statically linked tools that are already present on the system in the Nix Packages collection (such as `bash`).

13 There is a function to create a new standard environment. The main parameters for this function are `gcc`, `binutils` and `glibc`. Initially these are set to the statically linked tools and are gradually replaced with newly built, pure, instances.

14 The first real standard environment is created. This environment contains all the statically linked tools.

15 The collection of packages that can be built with the first standard environment.

16 Using the statically linked tools a new `glibc` is built. This instance of `glibc` is pure.

```
stdenvInitial = let {   12
  body = derivation {
    name = "stdenv-linux-initial";
    system = "i686-linux";
    builder = ./tools/bash;
    args = ./scripts/builder-stdenv-initial.sh;
    inherit staticTools;
  }  // {
    mkDerivation = attrs: derivation
            ((removeAttrs attrs ["meta"]) // {
      builder = ./tools/bash;
      args = ["-e" attrs.builder];
      stdenv = body;
      system = body.system;
    });
    shell = ./tools/bash;
  };
};
```

```
stdenvBootFun =    13
  {glibc, gcc, binutils, staticGlibc, extraAttrs ? {}}:

  import ../generic {
    name = "stdenv-linux-boot";
    param1 = if staticGlibc then "static" else "dynamic";
    preHook = ./prehook.sh;
    stdenv = stdenvInitial;
    shell = ./tools/bash;
    gcc = (import ../../build-support/gcc-wrapper) {
      stdenv = stdenvInitial;
      nativeTools = false;
      nativeGlibc = false;
      inherit gcc glibc binutils;
    };
    initialPath = [
      staticTools
    ];
    inherit extraAttrs;
  };
```

```
stdenvLinuxBoot1 = stdenvBootFun { 14
  inherit glibc gcc binutils;
  staticGlibc = true;
  extraAttrs = {inherit curl;};
};

stdenvLinuxBoot1Pkgs = allPackages { 15
  bootStdenv = stdenvLinuxBoot1;
};
```

```
stdenvLinuxGlibc = stdenvLinuxBoot1Pkgs.glibc; 16
```

```
stdenvLinuxBoot2 = stdenvBootFun { 17
  glibc = stdenvLinuxGlibc;
  staticGlibc = false;
  inherit gcc binutils;
  extraAttrs = {inherit curl;};
};

stdenvLinuxBoot2Pkgs = allPackages { 18
  bootStdenv = stdenvLinuxBoot2;
};
```

```
stdenvLinuxBoot3 = stdenvBootFun {  19
  glibc = stdenvLinuxGlibc;
  staticGlibc = false;
  inherit (stdenvLinuxBoot2Pkgs) gcc binutils;
  extraAttrs = {inherit curl;};
};

stdenvLinuxBoot3Pkgs = allPackages {  20
  bootStdenv = stdenvLinuxBoot3;
};
```

17 18 A new standard environment is created. The difference with the first standard environment is that `glibc` is now pure and dynamically linked.

19 20 A new standard environment is created. The difference with the third standard environment is that `gcc` and `binutils` are now pure and dynamically linked to the pure `glibc` from the second standard environment. The static tools are still used in this standard environment.

21 As a final step all static tools are rebuilt using the pure `glibc`, `binutils` and `glibc`. The result is a completely pure environment, with which the Nix Packages collection can be built in a completely pure way.

This mechanism of using statically linked tools to bootstrap a standard environment has been succesfully used in Nix for many months and has been verified to work on NixOS and now is part of the standard build procedure of Nix for Linux.

### 2.1.3 Building the pre-built tools

Most of the pre-built tools that are used are statically linked using `dietlibc`, a C library with a smaller footprint than `glibc`. For almost all packages the fact that `dietlibc` has a smaller footprint than `glibc` is the only reason that `dietlibc` is used. The only exception is `curl`. To resolve hostnames to IP addresses `glibc` dynamically loads an additional library at run time, even if an application is statically linked with `glibc`. This is a problem for `curl`, which needs to resolve hostnames to download the statically linked tools. If `dietlibc` is used instead of `glibc` this problem does not occur and `curl` can resolve hostnames without any problems. Only `bash` cannot easily be linked with `dietlibc` and is statically linked with `glibc`.

All of the statically linked tools that are used to build the pure Nix environment are built using Nix. Nix expressions for these tools are available in the standard Nix Packages collection.

```
 stdenvLinux = (import ../generic) { 21
   name = "stdenv-linux";
   preHook = ./prehook.sh;
   initialPath = [
     ((import ../common-path.nix) {pkgs = stdenvLinuxBoot3Pkgs;})
     stdenvLinuxBoot3Pkgs.patchelf
   ];

   stdenv = stdenvInitial;

   gcc = (import ../../build-support/gcc-wrapper) {
     stdenv = stdenvInitial;
     nativeTools = false;
     nativeGlibc = false;
     inherit (stdenvLinuxBoot2Pkgs) gcc binutils;
     glibc = stdenvLinuxGlibc;
     shell = stdenvLinuxBoot3Pkgs.bash ~ /bin/sh;
   };

   shell = stdenvLinuxBoot3Pkgs.bash ~ /bin/sh;

   extraAttrs = {
     curl = stdenvLinuxBoot3Pkgs.realCurl;
     inherit (stdenvLinuxBoot2Pkgs) binutils /* gcc */;
     inherit (stdenvLinuxBoot3Pkgs)
       gzip bzip2 bash coreutils diffutils findutils gawk
       gnumake gnused gnutar gnugrep patch patchelf;
   };
 };
}
```

## Chapter 3

# Kernel

In an operating system the kernel is a crucial program. It interacts with the hardware, manages processes and memory, takes care of networking, and so on. A kernel provides the necessary basis for computer programs to run on.

There are many different kernels, most of which are shipped as part of an entire operating system. Exceptions are Linux and GNU/Hurd, where the kernels[15][16] are distributed separately from the other components of the operating system, such as the C library, shell and other utilities.

Building a Linux kernel – NixOS is based on Linux, so only the Linux kernel is considered here – with Nix differs slightly from than building other packages with Nix. The most important reason for this is that the Linux kernel has support for loadable kernel modules. Kernel modules provide extra functionality, such as support for a certain type of soundcard, or support for a filesystem, that can be loaded into the kernel and removed from the kernel at runtime.

Many device drivers are installed in module form and only loaded into the kernel when the functionality they implement is needed. Kernel modules keep its memory footprint smaller. Code that resides in a module is only loaded when it is needed. If it is never needed, it is never loaded into the kernel.

Not all kernel modules are included in the main kernel source. Some are distributed separately, such as the modules that are needed for proper 3D support in NVidia and ATi graphics cards. Installing a new kernel also means that these modules need to be rebuilt and reinstalled in order to work with the new kernel. In conventional Linux distributions rebuilding and reinstalling these modules is often a manual process, which is time consuming. With Nix these steps can be automated. When a new kernel is built, the right external kernel modules will be automatically built and installed as well.

## 3.1 Using Nix for building the kernel

The Linux kernel uses version numbers. External tools to load kernel modules into the kernel at runtime use the version number to determine if the kernel module is allowed to be loaded. If the version number of the kernel and the version that is recorded in the kernel module do not match, it is not allowed to be loaded into the kernel.

Version numbers are not completely safe though. Two kernels that have a different, incompatible, configuration can still have the same version number. Incompatible in this case means that in one kernel certain capabilities were built in that are not present in the other. An example would be that the new kernel has support for SCSI, while the old kernel hasn't, with modules in the new kernel that need SCSI support to work correctly. Using the version number alone as a mechanism to see if two kernels are different will not work.

The fact that version numbers are not reliable to distinguish kernels has to be kept in mind when building, installing and managing the Linux kernel with Nix.

There are several approaches which can be used to build, install and manage the kernel.

One approach is to create a Nix profile for kernels. The profile is adapted when another version of the kernel is installed. The kernel compilation process is not altered in any way, except that the installation directory is a directory in the Nix store.

This approach does not record the Nix hash in the kernel image, or in the kernel modules. If two kernels have the same kernel version, but are compiled in a different way there is no way to tell the kernels apart, at least not for the external tools that take care of loading and unloading modules.

This is a problem, as the following scenario shows:

1. boot a system with a specific Linux kernel, for example 2.6.11.11
2. build a new kernel with a different and incompatible configuration, but with the same version number.
3. switch the kernel profile, but do not reboot.

Now there are two kernels with the same version number, but different capabilities. Loading a module into the kernel after updating the profile, but before rebooting, might fail because the modules of the newer kernel might not be correctly loaded into the running kernel. The problem here is that two different versions of the kernel and their modules are mixed, which is not guaranteed to work at all.

Another approach would be to embed the Nix hash into the kernel image. This can easily be done by adapting the value of `EXTRAVERSION` inside the kernel `Makefile`. The value of `EXTRAVERSION` will be appended to the kernel version, which can be displayed with `uname -r`. This approach is not exotic. Many vendors already adapt this attribute. For example, `uname -r` reports the following on a Fedora Core 5 distribution:

```
$ uname -r
2.6.17-1.2145_FC5smp
```

Everything after `2.6.17` is defined in the `EXTRAVERSION` variable in the kernel buildscripts.

The changes to the build process are relatively easy. The value of `EXTRAVERSION` is replaced by the old value of `EXTRAVERSION` and it is concatenated with the Nix hash. The kernel is built and installed in the Nix store.

All drawbacks from the first approach are not present in the second approach. Embedding the hash into the kernel also ensures runtime purity because modules simply won't load into a kernel if they were not built for that kernel.

### 3.1.1 Implementation

The Linux kernel has a relatively self-contained build process and isn't restricted to specific locations. With only a very small configuration tweak the kernel can be built and installed with Nix.

The changes to the build process are minimal. The value of `EXTRAVERSION` is replaced by the old value of `EXTRAVERSION` concatenated with the hash of the Nix expression for the kernel. Most of the work that is done in the builder for the Nix expression of the Linux kernel has to do with properly embedding the Nix hash in the kernel.

[22] The kernel configuration is copied to the right location where the kernel compilation process can find it.

[23] The hash part of the name of the Nix store path is extracted and saved, because it will be reused during building and installing.

[24] The `EXTRAVERSION` variable in the Makefile is replaced with something containing the Nix hash.

[25] The location where the modules are usually installed needs to be overridden, to ensure that the modules are installed in the Nix store instead of in the default location (`/lib/modules`).

```
source $stdenv/setup

buildPhase() {
 cp $config .config  [22]
 hashname=$(basename $out)
 if echo "$hashname" | grep -q '^[a-z0-9]{32}-'; then
   hashname=$(echo "$hashname" | cut -c -32) [23]
 fi

 extraname=$(grep ^EXTRAVERSION Makefile)
 perl -p -i -e "s/^EXTRAVERSION.*/$extraname-$hashname/" \
    Makefile [24]
 echo "export INSTALL_PATH=$out" >> Makefile
 export INSTALL_MOD_PATH=$out [25]
 make
 make modules_install
```

Figure 3.1: A part of the kernel buildscript

To build kernel modules for the kernel at a later time just the configuration and header files for the kernel are needed, which can be found in the directory `build` in the kernel module directory (`/lib/modules/$version/build`). Normally this directory is a symbolic link to the kernel build directory. When building with Nix this is a temporary directory in `/tmp`, for example `/tmp/nix-1234/`, which is created by Nix to compile the software in and removed after the kernel has been successfully compiled. To ensure kernel modules can be built for the kernel at a later time, the header files and kernel configuration are copied to the Nix store to the directory `$kernel/lib/modules/$version/build/`.

## 3.2 Kernel modules

Not all kernel modules are shipped with the mainline kernel sources. Some modules that are shipped separately from the mainline kernel as modules have not been accepted for inclusion in the mainline kernel for whatever reason.

Sometimes vendors distribute their own kernel modules, which have to be shipped separately from the mainline kernel for licensing reasons. The modules are shipped under a restrictive license, which prevents the distribution of these modules with the mainline kernel source. The best known examples of these are kernel modules for 3D graphics cards (Nvidia, ATi) and virtual machines (VMware).

Installation of a new kernel version often forces the administrator of a machine

to recompile and reinstall the extra kernel drivers. Managing the kernel and all extra modules with a Nix expression makes it much easier to upgrade a kernel without needing to recompile and reinstall all extra kernel modules explicitly.

### 3.2.1 Building modules

Kernel modules can be built in two ways. The first way is during compilation of the main kernel. During the build of a a default kernel also a lot of kernel modules are built. These modules are then stored in a subdirectory of the kernel install directory in the Nix store. The kernel build process in Nix already takes care of building these modules.

The other way is when kernel modules are built separately, apart from the main kernel compilation process. Add-on drivers and experimental modules, hereafter referred to as "external kernel modules" (because they are "external" to the main kernel sources), are built this way. An increasing number of modules only need a configured kernel source tree (header files and kernel configuration) to be built. These files can be found in a subdirectory of where the kernel is installed, called `lib/modules/$VERSION/build`, where `$VERSION` is the version of the kernel, as reported by the command `uname -r`. The kernel build and installation process already makes sure that all the right files are copied to this directory.

The build process differs per external kernel module. To prevent linking with the wrong header files from `glibc` the flag `NIX_GLIBC_FLAGS_SET` should be set to `1` in the Nix expression for the kernel module. The `glibc` library keeps its own copy of kernel header files and these might possibly not match those of the kernel the module is compiled for. A kernel module should also be compiled with the same `gcc` the kernel is compiled with to prevent loading errors.

### 3.2.2 Installing modules

When installing external kernel modules a few complications arise. First of all there is no standard location where external kernel modules should be installed. Some modules, such as the modules that are needed for VMware, are installed in a separate directory, like `/lib/modules/$version/misc/` (where `$version` is again the version of the kernel). Others are installed in the same directory as previously built modules that were built during the kernel build process itself. An example of this are the OV511 modules that provide support for webcams based on the OV51x chipsets. The installation script of the OV511 driver wants to install the modules with the modules of the existing kernel, which would be `/lib/modules/$version/kernel/drivers/usb/media/` on conventional Linux distributions.

With Nix these external modules should be kept in their own directories in the Nix store, instead of installing them with the kernel they were built for. Storing these modules in the same directory in the store as the kernel they were compiled for, would not be pure. Furthermore, a path in the Nix store is tagged read-only after installation, making installation impossible without first granting write permission.

The install process often expects that the modules will be installed in a subdirectory of `/lib/modules/$version`. Since this is not a path in the Nix store it is changed to `$out/lib/modules/$version`. Here `$out` is the same as the variable `$out` from the Nix installation process and represents the install location in the Nix store. The value of the `$out` variable is different from `$version` which is (again) the version of the kernel the modules were compiled for. The result is that the install path of the kernel is not littered with external kernel modules, but remains pure.

### 3.2.3 Loading modules

Loading and unloading of kernel modules into the Linux kernel is done by tools in the `module-init-tools` package. This package contains tools that can load (`insmod`) or remove (`rmmod`) a single module, or load a stack of modules (`modprobe`), where a module and all modules that particular module depends on are loaded in the right dependency order.

Most programs in the `module-init-tools` package are agnostic of where the modules are located on disk (`insmod` can even read from standard input), there are a few tools in this package that need to know where the modules are stored.

With `modprobe` a whole stack of modules can be loaded at once using the module name. The module name is looked up in configuration files. The first file where `modprobe` looks for module names file is `modules.dep`, which is a file that is generated by `depmod`, but a module name can also be defined in a number of other places, namely `modules.alias` (also generated by `depmod`) and `/etc/modprobe.conf`, where aliases are defined for kernel modules for easier loading of kernel modules. This file is often generated by the installer, or by a hardware detection program, like `kudzu`.

The file `modules.dep` is also used to determine the dependencies of a module recursively, so all the right modules can be loaded with one command invocation.

In the sources for `depmod`, `modprobe` and `modinfo` the constant `MODULE_DIR` is defined with a default value of `/lib/modules/`. The tools use this constant to find the default location where kernel modules are installed. The `MODULE_DIR` variable can be redefined at compile time. In the Nix Packages collection these

tools are patched to get this location at runtime from the environment variable `MODULE_DIR`. If this variable is not set, the value `/lib/modules/` is used instead.

The files that `modprobe` uses to read dependency information from are generated by `depmod`. The `depmod` program looks at information inside modules. Using this information it determines what other modules are needed by a module. This dependency information is written to a set of files.

The `depmod` program expects all modules to be installed in a single directory and its subdirectories and also writes its own result in this same directory. In Nix all modules are located in multiple directories in the Nix store and not stored in one directory such as `/lib/modules/$version/`.

To circumvent this problem we use a method similar to what GNU stow[19] uses to manage software installations. The fact that we used the version of the kernel in the name of the path that the modules were stored now pays off:

1. Create a directory to store kernel modules in (done automatically by Nix using the `$out` variable).
2. For every subdirectory in `$kernel/lib/modules/$version/` that contains kernel modules make a directory in `$out` with the same name.
3. For every kernel module that is present in any of the subdirectories of the installed kernel make a symlink to the kernel module in the Nix store from `$out/lib/modules/$version/`.
4. Do the same for every external kernel module.

After this all kernel modules will seem to be in one directory in the Nix store. The `depmod`, `modprobe` and `modinfo` tools can be run as if it were a normal system after setting the `MODULE_DIR` environment variable to the right location in the Nix store.

### Implementation

Using a normal Nix expression the combination of kernel and external modules can be easily managed (see figure 3.2).

The internal directory structure of the modules directories of the kernel and the external kernel modules is recreated and symlinks to the kernel modules themselves are made. This is done by the builder script for the Nix expression (see figure 3.3).

**26** The top path for the module directory is in the Nix store. The directories that are built by this script will be in a direct subdirectory of this path. The

```
{stdenv, module_init_tools, kernel, modules}:

stdenv.mkDerivation {
  builder = ./builder.sh;
  name = "kernelscripts-0.0.1";

  inherit module_init_tools kernel modules;
}
```

Figure 3.2: The Nix expression describing the combination of the Linux kernel and external modules.

```
source $stdenv/setup

export MODULE_DIR=$out/lib/modules/ [26]

kernelVersion=$(cd $kernel/lib/modules/; ls -d *)
mkdir -p $out/lib/modules/$kernelVersion

cd $kernel
find . -not -path "./lib/modules/$kernelVersion/build*" \
   -type d | xargs -n 1 -i% mkdir -p $out/% [27]
find . -not -path "./lib/modules/$kernelVersion/build*" \
     -a -not -path "./System*" -a -not -path "./vmlinuz*" \
     -type f | xargs -n 1 -i% ln -s $kernel/% $out/% [28]

for i in $modules; do
  cd $i
  find . -not -path "./lib/modules/$kernelVersion/build*" \
  -type d | xargs -n 1 -i% mkdir -p $out/% [29]
  find . -not -path "./lib/modules/$kernelVersion/build*" \
  -type f | xargs -n 1 -i% ln -s $i/% $out/% [30]
done

$module_init_tools/sbin/depmod -ae $kernelVersion [31]
```

Figure 3.3: The builder script that combines kernel and external modules.

environment variable `MODULE_DIR` is used by `depmod` to find the right directory to find the modules to inspect and write its own files to.

[27] [29] The directory structure is recreated. For every subdirectory in the directories an equivalent directory is created in the target directory.

[28] [30] The individual modules are symlinked to the right modules in the store.

[31] The `depmod` command is run to determine and record all the dependencies for every module, for easier loading by `modprobe` and other tools.

## 3.3 Upgrading to a new kernel

The Linux kernel developers tend to add new functionality with every new kernel release. Because of this old configurations can often not easily be used for building a newer kernel. The kernel configuration script will ask what to do with the new features and wait for user input when it is run with an old configuration. In Nix the kernel buildscript runs in non-interactive mode so the build process would get stuck. There is currently no solution for this. Upgrading to another kernel will require making a new configuration by hand.

## 3.4 Booting the kernel

The first code that is executed on a PC after it is powered on is located inside the BIOS chip. This BIOS code does some rudimentary hardware probing and memory checks and also starts the bootloader program. The bootloader program has the task to boot an actual kernel and operating system. The PC standard dictates that the first 512 bytes on a harddisk are reserved for the bootloader. This section is called the "Master Boot Record" or MBR. A small part of the bootloader, often called "first stage" will be installed inside the MBR, or, alternatively, in the first 512 bytes of a PC partition. Subsequent stages of the bootloader can be loaded from the harddisk and do the actual work of loading and booting the kernel and the rest of the operating system.

For Linux on the x86 platform there are two bootloaders that are used for nearly 100% of all installations. The first one is LILO[17], which has been the default bootloader for many years, but it is rapidly being replaced by GRUB[18]. GRUB has the advantage that after a configuration change it does not have to be reinstalled in the MBR of the drive, unlike LILO, but it can read its configuration from disk. GRUB has built-in support for many of the popular Linux filesystems and knows how to read this format and read its configuration file.

For NixOS we have chosen to use GRUB as a bootloader for the following reasons:

1. GRUB is rapidly becoming the standard for Linux bootloaders, especially on the x86 platform.
2. GRUB targets more systems than just Linux on PC. Even though NixOS only targets Linux on the x86 platform right now, it might target other platforms in the future as well. Having one bootloader makes installation and configuration easier.

### 3.4.1 Bootloader configuration

If GRUB is used the kernel can be booted directly from the Nix store, as long as the filesystem that contians the Nix store is supported by GRUB. If this is not the case, then the main kernel image should first be copied to a part of the disk that GRUB can read and boot from.

Regardless of which bootloader is chosen, the bootloader itself cannot reside inside the Nix store, but has to be installed on the first 512 bytes of the disk. However, the tools that install the bootloader on the disk and the data that is installed, can safely be kept in the Nix store.

### 3.4.2 Initial ramdisk

A so called "initial ramdisk" is often used at boot time in many distributions. An initial ramdisk is used to perform some system setup before the root filesystem is mounted. An example is loading kernel modules for specific hardware support, for example to be able to mount the root filesystem that is on a filesystem for which there is no direct support in the kernel image, but for which there is a loadable module or to load exotic device drivers that are needed for just one device on a particular machine.

An example of initial ramdisk usage can be found in the Fedora Core Linux distribution, where, starting with version 3, the root filesystem is on a partition which is managed by Logical Volume Management (LVM), which is a method to let multiple disk partitions appear as one. Inside the kernel there is no direct support for LVM, but there is a module which adds LVM support to the kernel. When the initial ramdisk is loaded by the kernel the scripts inside the initial ramdisk will load the module for LVM support and the root filesystem can be mounted.

An initial ramdisk is often used to keep the kernel smaller and keep machine specific hardware support in modules. This is done to prevent building a monolithic kernel with support for hardware that is not present in the target system.

Distributors can keep their kernels small, but can still support a wide range of hardware. During installation of a new kernel on a machine typically a script is run which determines what modules need to be inside the initial ramdisk and an initial ramdisk is created with the right modules. The script looks at what hardware is in the machine and what filesystems are used. An initial ramdisk specifically for that configuration is then created.

As the initial ramdisk changes on a per machine basis, it can therefore not be easily shared between machines or configurations, unless the machines have the same hardware. It is maybe not advisable to keep the initial ramdisk inside the Nix store, but the scripts that generate an initial ramdisk, given a system configuration, can easily be managed and built with Nix.

NixOS currently has no support for building and installing an initial ramdisk.

## 3.5 Tracking sources in the Linux kernel

As seen above Nix a kernel built with Nix is a lot stricter about which modules can be loaded into a running kernel because the Nix hash is embedded in the version number. The tools that load modules into a kernel use this version number to determine if the module can be loaded. An added feature is that Nix, because Nix also tracks all buildtime dependencies, can also help to better track down kernel errors.

Errors in the Linux kernel can obviously have a severe impact on system stability. Unlike programming errors in other programs, which only effect a small subset of programs, a programming error in the kernel can lead to computer crashes. Kernel errors are inherently hard to debug.

Building a working Linux kernel involves many components. Not just the various source files (C source files, C header files, platform specific assembler), but also the tools used to compile the sources into a working binary image, such as the compiler, assembler and linker, influence the build. There are numerous reported cases where these tools generate code that prevents the kernel from working correctly, especially on architectures that are not widely spread, such as the sparc64 platform. For a long time the default Linux kernel compiler has been `gcc` 2.95, which is unable to generate correct code for the sparc64 architecture. Kernels built with this compiler will simply fail to boot.

With Nix all sources and tools that are used during the build are tracked, giving a precise dependency tree. The need to be able to track which source files are used during the build of the kernel has also been acknowledged by the Linux kernel developers and they have developed a mechanism to track which C files are used. While the mechanism that is implemented is at least better than having nothing, it is not as powerful as Nix which tracks all dependencies that

are used in a build, including configuration options, compiler and linker tools, and so on.

The mechanism developed by the kernel developers takes the C input files that are needed to build a particular kernel module and computes a checksum based on these sources. This checksum is calculated with the MD4 hashing algorithm and embedded into the binary image of the module. In the module the checksum is called `srcversion` and can be retrieved easily from the module by using the `strings` command on a kernel module:

```
$ strings ext3.ko | grep srcversion
srcversion=FF0DC8177E8E88CF33E6B42
```

There are a few weaknesses in this approach. The first weakness is that the mechanism is limited to just modules and does not apply to the main kernel binary image. So while this mechanism can help with debugging modules it will not help if a bug is in the main kernel binary.

Another weakness is that this mechanism is not enforced. It is an optional feature which has to be explicitely configured at kernel build time.

Furthermore, it doesn't take into account other components that are used that influence the build process, such as the compiler that is used to build the kernel. Problems that might occur as a result of a malfunctioning toolchain and not as a result of a programming error in the kernel code itself will not be easier to catch with this mechanism.

Configuration options needed to make choices during compile time, for example symbol definitions used in `#ifdef` constructions, are also not used to compute the hash.

This is shown by doing two compiles of a stock 2.6.11.11 kernel, with slightly different configuration options. One kernel is compiled with the "JBD (ext3) debugging support" option, the other is compiled without this option. Both times the modules are built in the exact same environment (a vanilla Fedora Core 3 installation).

The resulting kernel module without debugging support is compiled has the following `srcversion`:

```
$ ls -l ext3.ko
-rw-r--r--  1 root root 1392174 Jun  8 16:37 ext3.ko
-bash-3.00$ strings ext3.ko | grep srcversion
srcversion=FF0DC8177E8E88CF33E6B42
```

If debugging support is included the result is:

```
-bash-3.00$ ls -l ext3.ko
```

```
-rw-r--r--  1 root root 1401683 Jun  8 17:19 ext3.ko
-bash-3.00$ strings ext3.ko | grep srcversion
srcversion=FF0DC8177E8E88CF33E6B42
```

As can be seen the two kernel modules have a different size and different functionality. The `srcversion` attribute is nevertheless the same in both cases, so there is no way to distinguish the two modules by using only the `srcversion` attribute.

Building the kernel with Nix has the same benefits that this mechanism offers for free, but without the defects, and will also track all other components that were used in the build.

## Chapter 4

# System services

A crucial part of any Linux system are so called services. A service is a piece of functionality, that allows the system to perform a specific task. Services can be very concrete, such as a web server or FTP server, or be a bit vaguer, such as the "networking service".

Services are implemented by a program or script or a set of programs or scripts. On a regular Linux system a lot of important services are started at boot time: usually the network is brought up and various programs are launched, such as `syslogd` for logging, a firewall for keeping unwanted network traffic out or `sshd` for allowing remote logins into the machine. When the system is running, services can be stopped or restarted, or additional services can be started on the system. Services are probably the most important pieces of software on the system from a functionality point of view.

A service can have a dependency on other services before it can do its work. For example, the remote login service (`sshd`) won't work if the networking service is not started first. These dependencies are also often very loose. Many services that need a mail server to be running actually don't even care which mail server is running, as long as it is running and is accepting mail.

Services can also have special requirements on system resources, like TCP/IP ports, or certain devices or files. These system resources typically cannot be shared. Only one service can use a certain port. For example, only one web server can use TCP port 80.

This makes managing services different from installing and managing other programs.

## 4.1 Services on Linux

There are several ways to manage services on Linux and two approaches are in common use. The first is to start everything from one big script at boot time. This method is borrowed from the BSD Unix systems. The second method is to have a script for starting and stoppping per service. The scripts are run in a certain order to ensure that service run time dependencies are met. This method comes from the System V Unix systems. Historically the BSD method was used on Linux, but these days the majority of Linux systems use the System V method, often referred to as "System V runlevels". In this thesis only the System V method is looked at.

### 4.1.1 System V runlevels

Most Linux distributions use so called "runlevels" which come from System V Unix. A runlevel is a set of programs or services which are started and stopped together when that level is entered. Most distributions give the same meaning to runlevels. For example, "runlevel 3" often means "fully networked multi-user system with NFS enabled, no graphics", while "runlevel 5" means "fully networked multi-user system with NFS and graphics system enabled. When a runlevel is entered its startscripts are run, and when a runlevel is left its stopscripts are run.

During the startup of a Linux system the first program that the kernel normally runs, is `init`[1]. The `init` program launches other programs on startup, by interpreting a file called `inittab`, normally located in the directory `/etc`, which describes what programs should be started during boot time (see figure 4.1).

Every line in the configuration file describes what action should be performed in which runlevel. The line starts with an identifier, followed by a collection of runlevels (or all), followed by the action that should be taken. The last field describes the command (including parameters) which should be run.

**32** The default runlevel is set to a runlevel that should be entered after system boot. Runlevel 5 is typical for graphical fully networked desktop machines.

**33** The runlevel scripts are run with the right runlevel parameter.

**34** Various signals can be trapped.

**35** Logins on virtual consoles are started.

A few programs are started directly, like the programs to manage virtual consoles and serial logins (`mgetty`, `agetty` or in `mingetty`), or graphical logins (`xdm`, `gdm`

[1] Even though other programs could be run as the first program as well.

```
id:5:initdefault:  [32]

si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0  [33]
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

ca::ctrlaltdel:/sbin/shutdown -t3 -r now  [34]

1:2345:respawn:/sbin/mingetty tty1  [35]
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

Figure 4.1: An example System V `inittab` configuration

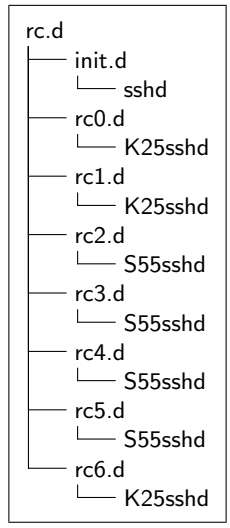


Figure 4.2: Directory structure of System V runlevels

or `kdm`). These programs are not considered services, because no scripts to start and stop these programs are used, but `init` handles these programs directly. Services on the other hand are often started by a generic script.[33]

These scripts, located in `/etc/rc.d/init.d/`, are executed in a certain order. This is done because some services need other services to be running to work correctly. For example, networking should be started before starting a web-server, otherwise the webserver simply won't run. A runlevel is constructed by making symbolic links from the directory for the runlevel (often a directory in `/etc/rc.d/`). Only for the services that are part of the runlevel symbolic links are made. For every service two symbolic links can be made: one for starting the service and one for stopping the service. The symbolic links for starting start with "S", followed by a number, the symbolic links for stopping the services start with "K", followed by a number. The scripts are executed with either the "start" or "stop" argument, and in lexical order.

Runlevels give an adminstrator a uniform interface to starting and stopping services, and the possibility to choose which services will be started together during boot time.

The scripts can also be executed later on to start additional services, or to stop or restart an already running service.

## 4.2 Dependencies in services

There are several types of dependencies in services. The first type is a buildtime dependency. This dependency is used during the buildprocess. This dependency can be often be easily expressed in Nix by using "buildInputs" in the Nix expression.

The second type is a run time dependency. An example of this is when a program executes another program during execution of the program, which is common in shell scripts. These run time dependencies can also be dealt with easily in Nix, by replacing invocations of programs in shell scripts with the fully qualified path to those programs in the Nix store.

The third type is also a run time dependency, but it is a lot looser. It occurs when programs communicate through some interface, such as a pipe, port or Unix socket, without actually knowing (or needing to know) what program is on the other side of that interface. The whole communication is based upon a protocol that both sides understand. One program needs another program to just implement a certain protocol and act accordingly. In this thesis this type of dependency will be called "remote dependency". This does not mean that the dependency has to run on another machine, but that it communicates with the dependency through an interface.

An example of a remote dependency is `syslog`, which is the system logger on Linux. It is used by many programs, but these programs don't run `syslog` directly. Instead they use the function `openlog` from the system C library to communicate with `syslog` and log messages. This function in turn uses the socket `/dev/log` for communicating with `syslog`. Basically any program that can process entries from `/dev/log` would fullfill the dependency on `syslog`. For Linux there are various implementations that offer `syslog` functionality, namely `sysklogd` and `syslog-ng`. Replacing one implementation with another does not make any difference for programs that depend on `syslog`. This dependency is a lot harder, if not impossible, to express directly in Nix.

If we would express remote dependencies in Nix the situation could occur that one service depends on one implementation of a server and another service depends on another implementation. These different implementations cannot run at the same time, because they would want to use the same resources such as TCP/IP ports, or sockets.

Apart from the various types of dependencies there is the additional complication that it is not mandatory that the dependencies themselves are running, but are optional. For example, many services want to log to the system logger, but it is not fatal for these programs if the logging service is not available. In other cases a dependency is mandatory, such as for a networked service (`webserver`)

which cannot run if the network is not enabled.

The type of dependency that is not mandatory is from now on called "soft dependency", while the dependency that is mandatory is called "hard dependency".

## 4.3 Services with Nix

For NixOS a uniform interface to control services, similar to System V runlevels on other Linux distributions was created.

The main design decisions were:

1. It should be possible to start services at boot time, but also to start or stop services when the system is already up and running.
2. It should be possible to switch from one implementation of a service to another, for example switch from the Sendmail mailserver to the Postfix mailserver.
3. Remote dependencies should be taken care of.
4. Launching a service should launch all services it depends on, if these are not already running.

### 4.3.1 Installing a service

To make managing packages easier an extra attribute called "nicename" is used. The "nicename" is a name by which the service is known to other programs. This name is more generic than the actual name of the program. It is an abstraction over the functionality it provides. For example, the `wuftdp` FTP server will have a "nicename" `ftpserver`, because it provides FTP server funcionality. Similarly the `vsftpd` FTP server will also have the nicename `ftpserver`. Instead of specifying the name of the package as the dependency, the 'nicename" attribute (for example `ftpserver`) is used to specify the dependency. A package that needs to have a FTP server running uses the dependency `ftpserver` and not `wuftpd`. This mechanism makes it easier to switch between different implementations of services.

Other examples of these nicenames are `mailserver` for a mailserver, such as `sendmail`, `postfix` and `exim`. The `sysklog` and `syslog-ng` system loggers use the "nicename" `syslog`, and so on.

Nix profiles can be used to manage services and their configuration. Using profiles has the benefit that rollbacks can be used for services. Reverting back to an old service becomes a lot easier than on conventional Linux systems, where

often services are not touched unless absolutely necessary, if they are touched at all ("if it ain't broke don't fix it").

With Nix an upgrade is a lot less troublesome, because you can do rollbacks very easily. During an upgrade the service has to be stopped, the profile is changed to a new version and the new service is started. If it doesn't work you simply do a rollback and relaunch the old version of the service.

There are various ways to keep scripts in profiles. The most straightforward way is to keep the control scripts for a service in a profile of which the name matches a so called "nicename". This way it is possible to upgrade or roll back on a per service basis but with all the separate profiles you can't revert a whole set of services at one time easily.

### 4.3.2 Starting a service

Before a script can be started, it checks whether its dependencies are already running. If not, it tries to start its dependencies recursively. If the dependency is not already installed on the system it is built and installed in a profile, for example in `/etc/rc.d/$nicename/`, where `$nicename` is the "nicename" attribute of the service (so `mailserver`).

Dependencies of a service are started by first querying the system to see if the dependency is already running. Currently this is done by asking the start/stop script of the dependency for its "nicename" and checking if this name is registered in a central directory where services keep their run time information, for example in `/var/run/nix-services/`. The Nix store would not be the right place to keep this information, since this is state information which is changing frequently during run time. The Nix store is regarded as stateless.

If the name of the dependency is present in the central directory, it is assumed that the dependency is already running. If the dependency is not yet running it is started.

If the dependency is a "soft dependency" any start up failures of the dependency will be silently ignored. If the dependency is a "hard dependency" and it fails to start, the start/stop script of the calling service will exit.

If all hard dependencies for the service are started correctly the service itself can be started using by the start script. If the service itself was also started successfully it registers itself with the system as "running" in the central directory with run time information.

### 4.3.3 Stopping a service

When a service is stopped, or removed from the system, the only thing that is needed is simply invoking the stop script of the currently running service. The service will stop and deregister itself from the system. Stopping a service will not stop the dependencies that are needed by or were started by the service. Even though it is possible to determine which services are not currently needed by services (this could be done by recursively checking if the dependencies are needed by other, running, services) this is no guarantee that the other services are not needed anymore. Other programs which are not managed by the services mechanism might still be using one of the dependencies. For example, there are a lot of networked applications (webbrowser, FTP program, newsreader) that need the "network service" to be enabled, but which are not started from a start/stop script. Stopping the "network service" because no other service that is running needs it, will break these other programs.

### 4.3.4 Anatomy of a start/stop script

The start/stop scripts can be made as fancy as you want. For NixOS a sample implementation was implemented which is minimal, but functional. All start/stop scripts are generated using Nix.

A start/stop script for a particular service can easily by generated using Nix. A script typically has the following dependencies:

- The service itself.
- A set of generic scripts that implements functionality such as registering and deregistering the service with the system and launching all dependencies recursively. Figures 4.5, 4.6, 4.7 and 4.8 give a walkthrough of the sample implementation from NixOS. The full versions can be found in the Nix Packages collection in the directory `servers/server-scripts/generic`.
- A script that implements service specific behaviour. In figures 4.9, 4.10 and 4.11 a walkthrough of the service scripts of `ssh` is given. The original scripts can be found in the Nix Packages collection in the directory `servers/ssh-script`.
- Other run time dependencies.

[36] A set of generic scripts is passed as a parameter. These scripts implement various functions that all scripts use.

[37] [38] All "soft" dependencies are specified separately from the normal hard dependencies.

```
{stdenv, ssh, bash, coreutils
, initscripts [36]
, key ? null, syslog ? null
, networking}:

stdenv.mkDerivation {
  name = "ssh-script-0.0.1";
  nicename = "sshd";
  server = "ssh";
  builder = ./builder.sh ;
  softdeps = [syslog]; [37]
  deps = [networking]; [38]
  inherit bash ssh initscripts coreutils;
  script = [./sshd];
}
```

Figure 4.3: A Nix expression to generate a start/stop script for `sshd`.

```
source $stdenv/setup

ensureDir $out

sed -e "s^@bash^$bash^g" \  [39]
    -e "s^@sshd^$ssh^g" \
    -e "s^@initscripts^$initscripts^g" \
    -e "s^@coreutils^$coreutils^g" \
    -e "s^@softdeps^$softdeps^g" \
    -e "s^@deps^$deps^g" \
    < $script > $out/control

chmod +x $out/control
```

Figure 4.4: Nix builder to generate a start/stop script for `sshd`.

```
#! @bash@/bin/sh -e

STATEDIR=/var/run/nix-services [40]
RCDIR=/etc/rc.d/ [41]
```

Figure 4.5: A generic script for services (part 1)

[39] The builder generates a script called `control`. In this script all calls to programs are replaced by the absolute paths to these programs in the Nix store.

### Generic service scripts

The control script is written in such a way it can be invoked with the usual stop/start/status arguments. It is up to the start/stop script to hide the specifics of the different programs, such as commandline options.

[40] A central directory where scripts keep state information is needed. This script is either created during install time, or created by a top level script if it doesn't exist yet. Here it is just expected to exist.

[41] All profiles are kept in one global directory. Every directory in this directory contains a profile which is managed by Nix.

[42] [46] If a profile does not exist it is created first.

[44] If a hard dependency fails the script exits.

[47] If starting a soft dependency fails it is silently ignored.

[48] [50] The service specific functions from the server specific script are called. These functions implement the functionality that cannot be shared between all the scripts.

[49] No action is needed if the service is not running and has not registered with the system.

[53] A convenience function to return the name of the service. This is not the name of the concrete instance of the service, but the same as the "nicename" of the server.

### Service specific scripts

[54] A function that implements the service specific details with regard to starting the service, such as supplying configuration files, and so on.

[55] A function that takes care of the service specific details regarding stopping the service, such as cleanups, cleaning caches (if any), and so on.

```
start_deps() {
    for i in $deps; do

        name=‘$i/control name‘

        if ! test -a "$RCDIR/$name"; then
            @nix@/bin/nix-env -p $RCDIR/$name -i $i 42
        fi

        $i/control start 43
        RETVAL=$?
        if test $RETVAL != 0; then 44
            exit $RETVAL
        fi
    done
}

start_softdeps() {
    for i in $softdeps; do

        name=‘$i/control name‘ 45

        if ! test -a "$RCDIR/$name"; then
            @nix@/bin/nix-env -p $RCDIR/$name -i $i 46
        fi

        $i/control start
        RETVAL=$?
            if test $RETVAL != 0; then 47
            continue
        fi
    done
}
```

Figure 4.6: A generic script for services (part 2)

```
start() {
    if test -a $STATEDIR/$prog; then
        exit 0
    fi

    start_deps

    RETVAL=$?

    if test $RETVAL != 0; then
        echo $prog failed
        exit $RETVAL
    fi

    start_softdeps
    startService [48]

    RETVAL=$?

    if test $RETVAL != 0; then
        echo $prog failed
    exit $RETVAL
    fi

    register
}

stop() {
    echo "stopping $prog"
    if ! test -a $STATEDIR/$prog; then [49]
        exit 0
    fi
    stopService [50]
    echo "unregistering"
    unregister
}
```

Figure 4.7: A generic script for services (part 3)

```
register() { 51
    touch $STATEDIR/$prog
}

unregister() { 52
    rm $STATEDIR/$prog
}

status() {
    if test -a $STATEDIR/$prog; then
        echo "running"
    else
        echo "stopped"
    fi
}

name() { 53
    echo $prog
}
```

Figure 4.8: A generic script for services (part 4)

```
#!@bash@/bin/bash

source @initscripts@/functions

RETVAL=0
prog="sshd"
softdeps="@softdeps@"
deps="@deps@"

KEYGEN=@sshd@/bin/ssh-keygen
SSHD=@sshd@/sbin/sshd
DSA_KEY=/etc/ssh/ssh_host_dsa_key
PID_FILE=/var/run/sshd.pid
OPTIONS="-h $DSA_KEY

do_keygen() {
        ...
}
```

Figure 4.9: A service specific script for `sshd` (part 1)

```
startService() { 54
    # Create keys if necessary
    do_keygen

    echo -n $"Starting $prog:"
    $SSHD $OPTIONS
    RETVAL=$?
    [ "$RETVAL" = 0 ] && @coreutils@/bin/touch \
       /var/lock/subsys/sshd
}

stopService() { 55
    echo -n $"Stopping $prog:"
    @coreutils@/bin/kill `@coreutils@/bin/cat \
      /var/run/sshd.pid`
    RETVAL=$?
    [ "$RETVAL" = 0 ] && @coreutils@/bin/rm -f \
       /var/lock/subsys/sshd
}
```

Figure 4.10: A service specific script for `sshd` (part 2)

56 57 58 59 Various calls are forwarded to the generic script.

### 4.3.5 Switching services

The start/stop scripts presented here don't have any mechanism to switch between different services (either a different version, or a completely newer version). Services should therefore be managed by a toplevel script which first stops the old service, builds the start/stop scripts for the new service if necessary, updates the profile to the new service and then starts the new service.

### 4.3.6 Improvements

The scripts presented here are "hackish" and can be improved quite a bit. The fact that inside the scripts Nix commands are executed is quite dubious. As a proof of concept these scripts work, but they are not recommended for production use.

```
case "$1" in
    start)  [56]
        start
        ;;
    stop)
        stop  [57]
        ;;
    restart)
        stop
        start
        ;;
    status)  [58]
        status
        ;;
    name)  [59]
        name
        ;;
    *)
        echo $"Usage: $0 start|stop|restart|status|name"
        RETVAL=1
esac
exit $RETVAL
```

Figure 4.11: A service specific script for `sshd` (part 3)

## Chapter 5

# Special configurations

Many programs on Linux systems use configuration information, which defines how a program should run. This configuration information is commonly found in a file or directory in a system wide configuration directory, often `/etc`. As an example, the configuration for the `sshd` service can be found here, or the central password file, or file system configuration. The type of configuration can vary greatly between packages.

While most configuration information is just part of one package, there are packages where the configuration is not part of one package, but it is spread over more packages. The various programs the configuration is part of often use some sort of plug-in mechanism, where programs can add configuration information in one common directory, which the main program can read.

When these packages are all installed in the same prefix, as on conventional Linux systems, the configuration information ends up in the same place. With Nix this information will be scattered over the Nix store. To make the programs work correctly with all configuration in different places extra work has to be done. Configurations from the various packages have to be combined, similar to combining kernel modules from an installed kernel with external kernel modules.

One example that uses configurations in multiple packages is the Linux hotplugging system. A hotplugging system gives users the possibility to add hardware dynamically to the running system through hardware buses like USB, IEEE 1394 (FireWire) or hot-swappable PCI. Adding hardware to a running system makes it mandatory that the operating system can load and unload drivers, make new device nodes, set permissions on the device nodes and so on.

The default Linux hotplug code is organised in such a way that programs can add configuration for various devices themselves. Programs like `sane`, for using scanners, or `gphoto2` for communicating with digital cameras, come with hotplug configuration for new scanners and cameras.

The Linux hotplug tools expect most configuration to reside in a single configuration directory. Like with the kernel modules described elsewhere in this thesis this is a problem. Once a program has been installed in the store other programs should not touch the installed files.

One tool that make Linux hotplugging work is `udev`, a daemon that creates and modifies `/etc`, the directory with device nodes (files representing hardware devices). This daemon is commonly started by the `init` program at system startup. The configuration for `udev` is located in `/etc/udev/`. Other programs normally add configuration for `udev` by writing their configuration during install in this directory.

Another program that is used in the Linux hotplugging system is `haldaemon`, a program that is part of the Hardware Abstraction Layer system for Linux.

The `haldaemon` is a normal service (normally started from System V initscripts), like `sshd` or `sendmail`, but with the added complication of the configuration that is spread over the Nix store.

The solution for this problem is to make a wrapper package around the main program (for example `udev` and `hal`). All packages which configuration is needed are passed as a dependency to the Nix expression for this package. The wrapper package with all configuration is passed as a dependency to another wrapper package, which consists of the configuration package and the original program. The wrapper makes sure that the correct configuration will be loaded by the program.

## Chapter 6

# NixOS Installer

An important subgoal of this project is to be able to install NixOS on real hardware to prove NixOS works as a standalone distribution and that no other distribution is needed to be able to use Nix.

### 6.1 Linux installers

Linux distributions are installed on a computer using an installer. The task of the installer is not simply to transfer packages from one medium (the installation CD) to another (the harddisk), but also to perform various configuration tasks, like partitioning and formatting of harddisks, adapting configuration files with hardware specific information (for example for the graphical environment), and so on.

Depending on the distribution this installer can be very minimalistic, or very feature rich. Some installers have completely automated the install process, can determine a lot of configuration options by themselves and require very little user interaction. Well known installers in this category are `anaconda` (Red Hat, Fedora Core and derivates) and `YaST` (SUSE LINUX). Other installers, like the Gentoo installer, are not more than a shell prompt and a set of shell commands that have to be entered in the right order by the administrator.

### 6.2 NixOS installer implementation

NixOS can only be installed from a CD. A small bootloader on the CD loads a Linux kernel. This Linux kernel starts the installation program. The installation program and related tools are located inside an initial ramdisk. This ramdisk is unpacked to memory by the kernel and the `init` program inside the ramdisk is started as the first process.

The initial ramdisk contains a minimal Nix store but no Nix database. There are just enough tools inside the installer to set up a suitable environment to do the real install. Included are `bash`, `coreutils` and `util-linux` and all their run time dependencies. The paths to these tools are set in the `PATH` environment variable for the first installer scripts. On a normal system the `PATH` variable would translate to a user's Nix profile. Using Nix profiles inside the installer would not make much sense. Only one specific instance of the tools is present on the installer CD and none of the tools used by the installer during installation need to be upgraded during the install itself.

Before the actual installation is done a suitable environment is set up. To make this environment the following things are done:

- Making device nodes, such as device nodes for the harddisk and its partitions, various `tty` character devices for extra installtime "rescue shells" and other special devices such as `/dev/null`. Other device nodes will be created dynamically later on.
- Mounting special filesystems which are used in Linux: `/proc`, `/sys` and `/dev/pts`
- Discovering and mounting the installation CD, which contains all the packages we want to install and setting links to it accordingly.
- Launching the script that does that actual installation of the packages onto the target drive.

The 'real' installer has the following tasks:

1. partitioning of disks
2. formatting of partitions
3. initializing the Nix store and Nix database on the target drive
4. installing Nix packages from the CD onto the target drive
5. setting up an initial environment
6. installing the bootloader

Not all of the steps are implemented by the current NixOS installer. Partitioning of disks is not performed. The NixOS installer simply expects two partitions to be present, one which is used to store data, the other to use as a swap partition. A full installer would make this configurable.

The partitions of the disks are formatted for a certain filesystem. Various filesystems are available for Linux, but the NixOS installer only uses the `ext2` filesystem. A full installer would have support for other filesystems as well and format

these accordingly.

Before the packages are installed a Nix store and database are created on the target drive. Packages are installed by copying over a minimal set of packages from the Nix store on the CD and registering the paths as valid in the database on the target drive. Otber packages on the CD are installed using Nix manifests. Nix manifests are a way to tell Nix where pre-built packages can be downloaded or copied from to be installed. Nix manifests work by registering the packages into the Nix database first. After registration the packages can be installed using the normal Nix installation tools.

The last step is installation of the bootloader and the bootloader configuration. As bootloader `grub` is used. The bootloader is configured to boot the kernel directly from the Nix store on the harddisk. This is only possible if the bootloader can read the filesystem the Nix store is stored on on the harddisk.

## 6.3 Installation CD

The NixOS installer CD uses `syslinux`. The `syslinux` tool is a small bootloader specifically meant for booting Linux-based systems from harddisk, CD or over a network via PXE. For NixOS we use `isolinux`, which is meant for CD installers and which is part of `syslinux`. It is the de facto standard for bootloaders for Linux installation CDs for the x86 platform. The `isolinux` program has a few limitations which have an impact on how to prepare the installation CD.

One limitation of `isolinux` that conflicts with Nix is that it can use file names and directory names up to 31 characters. This makes it impossible to boot the kernel directly from a Nix store on the installation CD. The character limit is a limit of the underlying ISO 9660 filesystem.

The task of `isolinux` is to boot the Linux kernel with the right parameters, such as the initial ramdisk which is used during the installation.

## 6.4 Generating the installer and CD image

The installer scripts and bootable CD image are generated by a shell script. In this build script packages are taken from the Nix store – and built first if the packages are not already present in the Nix store – and copied to a temporary location. This temporary location will eventually be the root directory inside the ISO 9660 filesystem that is created for the installer CD.

Inside this root filesystem are the initial ramdisk and a Nix store (but again, no Nix database) with packages that the full compiler will use. In this store there are more packages than in the Nix store that is inside the initial ramdisk.

This Nix store is mounted over the Nix store from the initial ramdisk. Inside the root filesystem there is also `isolinux` and its configuration and the Linux kernel that `isolinux` boots. Also present are the installer scripts and a copy of a statically linked `bash`.

For the packages that are used in the installer a "binary only" deployment is done. That is, only the run time dependencies and not the build time dependencies are put in the Nix closure. This is done to save space in the installer and keep the install CD under the size limit of a standard CD-ROM.

Paths to executables in the installer scripts are replaced by the full path to that executable in the Nix store in either the initial ramdisk or on the CD.

The initial ramdisk contains, apart from the Nix store, also a directory with special device nodes and empty directories which will serve as mount points for special filesystems such as `sysfs`, `procfs` and so on.

As a final step everything is packed into a bootable ISO 9660 image, which can be burned to a CD and booted. This image is not kept in the Nix store, due the massive size (over 400 megabyte).

The NixOS build scripts have three external dependencies: `bash` (to execute the scripts), `nix` and `which` (to determine the path to the Nix tools). For the rest all necessary tools come from the Nix store and are built first if necessary.

## 6.5 Installations NixOS using only Nix manifests

In an ideal situation the installation would be done in a cleaner way than it is done now, by using only Nix manifests. If only Nix manifests are used in the Nix installer it eliminates the need to explicitely register the copied packages as valid, as is done now.

The difficulty is that during installation time the Nix store in `/nix` is not the Nix store on disk that packages should be installed into, but the Nix store in the ramdisk of the installer. The Nix store for the target system is on the mounted disk in the ramdisk, and can be found for example in the directory `/tmp/mnt/nix/`.

If a package is installed in the Nix store in the temporary location, the temporary location (in `/tmp/mnt/nix/` and not the final location (in `/nix/` will be recorded into the Nix database.

When the environment variable `NIX_ROOT` is set Nix will perform the `chroot` system call. This system call uses the value of `NIX_ROOT` and makes this the new `/` directory for all commands that are executed by Nix. This ensures that

packages are registered in the Nix database with the right path and not with the path to the temporary path in the installer.

In theory this should work, but practice is harder. When a package is registered into the store from a Nix manifest using `nix-pull`, an entry is made into the Nix database that notes that the program has to be downloaded from a cache. When the program is installed, for example wit `nix-env`, a helper program, `download-using-manifests.pl` is started. This is small Perl script expects to find `perl` and several utilities from `nix` in the store to execute them. When these packages are not yet installed `dowload-using-manifests.pl` will fail to run. This is a bug in Nix (bug NIX-50 in the Nix bug tracking system).

Chapter 7

# Building packages and running programs on NixOS

## 7.1 Building packages on NixOS

After NixOS is installed and the network is enabled on it it can be used to build packages. Since NixOS is a completely pure environment it is impossible that build scripts will find anything in fixed paths like `/bin`, `/usr/bin` and `/usr/local/bin` and accidentily pick this up as an unidentified dependency.

A build on NixOS will reveal these dependencies, either because the package will fail to build or because the result of the build is different than on a normal Linux machine.

One example of a package which builds cleanly in Nix on a machine with a regular Linux distribution (Fedora Core 3), but not in NixOS is the OpenOffice.org office suite. During the build it fails when at some point `xargs` is executed.

On Linux `xargs` defaults to using `/bin/echo` if no specific command is given. The Single Unix Specification merely states that `echo` should be executed, but does not specify where this tool has to be installed [9]. Because no default command is given, `xargs` tries to execute `/bin/echo`, which is not present on NixOS and thus the build fails. On a normal Fedora Core 3 installation this tool is present and the build works.

## 7.2 Running programs on NixOS

Running programs on NixOS is usually not a problem, since the locations of many of the dependencies a program needs, such as libraries, are hardcoded into the program. Many programs will run perfectly even if there are no programs, not even `/bin/sh`, present in the well-known search paths at all.

But that doesn't mean that all programs will run without a hitch. Apart from configuration issues, where programs will look for their configuration in fixed locations, there are programs, especially shell scripts, that assume that programs they need can always be found in the default search path, which is simply not always true on NixOS. It is also very common to set the interpreter of a shell script on Linux systems to `/bin/bash`, but on NixOS, this path does not exist.

Even normal C programs can have problems during execution. Some programs use `dlopen` to dynamically load a library at runtime, others have paths hard-coded in scripts that are executed. Some programs, such as the ISC DHCP suite, even go as far as defining the complete environment for child processes to run in, including the `PATH` environment variable to look for other programs.

All these problems will only show up at runtime and are hard to detect at buildtime.

## Chapter 8

# Future work

There are still many areas where NixOS can be improved a lot, but which were outside the scope of this project. Many of the concepts introduced in NixOS have only been implemented as a proof of concept. While they work, they are not very user friendly and require a lot of knowledge about how the system is designed and how Nix works.

## 8.1 NixOS installer

### 8.1.1 Partitioning

NixOS expects a fixed harddisk layout: `/dev/hda1` for installing all data and `/dev/hda2` for a swap partition. This is very unrealistic since many computers do not have this layout, or users want to be able to install NixOS on other partitions of their harddisk.

### 8.1.2 Graphical installer

One major improvement would be to make a graphical installer for NixOS, including functionality for hardware detection and configuration, harddisk partitioning, package selection and so on.

Two popular graphical installers for Linux are `anaconda`[11] and `YaST`. The `anaconda` programma is primarily used in Red Hat Linux and Red Hat Linux derivates, such as Fedora, but it has also been ported to Progeny Linux, a Debian GNU/Linux offshoot, which uses a different package management system than Red Hat Linux. So far, `YaST` has only been used in SUSE Linux, mostly due to licensing. Not too long ago the sources for `YaST` were released under the GNU General Public License, but it has yet to be picked up by distributions other than SUSE Linux.

Both programs perform similar tasks during installation time, but differ in how they do it and in the used technology.

| | anaconda | YaST |
|---|---|---|
| textual interface | yes | yes |
| GUI toolkit | GTK | Qt |
| Usage | installtime | installtime and runtime |
| Default packages | RPM and DEB (Progeny) | RPM |

Many tasks that these tools perform during install time are package management system agnostic. The functionality for installing packages should be rewritten to use Nix. Because the tools are platform agnostic it should be fairly straightforward to take out the RPM specific bits and replace them with calls to Nix. The configuration the installer writes to the filesystem (hardware configuration, and so on) should also be kept in a Nix expression or profile. This way the configuration becomes an input for building the configuration scripts in a pure way.

The graphical installer itself should also have to be built with Nix and run in a pure Nix environment from the installation CD.

### 8.1.3 Network installs

Currently the installer scripts do not support installing a whole Nix environment via a network. Since installing packages via a network is one of the strong points of Nix, being able to do a full network install is, eventually, a must for NixOS.

Nix has the concept of a "network cache"[1], which is built with `nix-push`. At installation time the packages from the network cache are registered in the Nix database with `nix-pull`, which tells Nix certain versions of packages can be installed from the network cache.

Being able to install from a network cache can make it a lot easier to customize the distribution at install time. To be able to do installs from a network cache there is one important additional step that needs to be performed in the installer, which is to bring up networking in the installer so packages can be downloaded from the network (if the cache resides on a local disk or installer CD, as is done in the current version of NixOS, this step is of course not needed). For this to work properly some sort of hardware detection should be implemented to determine what network card is in the machine and to load the right kernel modules for that networkcard.

[1] The term "network cache" is a bit misleading, because the cache can also exist on a CD or harddisk partition which is mounted at installation time. However, to be in sync with the terminology, we'll refer to this as a "network cache" as well

## 8.2 Better startup scripts

The startup scripts that are in NixOS are very simple. No configuration is done, except launching an emergency shell, with which it is possible to install programs, install and launch services, and so on.

## 8.3 Support for more filesystems

Right now the support for filesystems is rather limited. This is due to the fact that we load the kernel straight from the Nix store at boot time. The bootloader, GRUB, has limited support for filesystems (`ext2`, `ext3`, `minix`, `fat`, `ffs`, `iso9660`, `jfs`, `reiserfs`, `ufs2`, `vstafs` and `xfs`). Many distributions are moving to filesystems which have data on disks managed by Logical Volume Management (LVM), of which GRUB has no knowledge. If the Nix store is on a LVM partition, GRUB cannot boot the kernel directly from the Nix store. Linux distributions that use LVM, such as Red Hat Enterprise Linux or Fedora (starting with version 3) keep a small `ext3` partition, about 100 MB in size, on which the kernel is stored.

There are a few approaches to tackle this limitation:

1. force users to stick to supported filesystems. This is not very realistic if we want NixOS to be deployed in large scale environments.
2. keep a small Nix store on the bootpartition. This is undesirable in our view, since at one point things will have to be thrown out of the Nix store because of diskspace problems.
3. copy the kernel-image (and possibly an initial ramdisk) from the Nix store to the small boot partition. This is not very elegant from a Nix point of view, but probably the most workable solution. This copy action should be done when a new version of the kernel is added to the boot configuration. The risk is that the small boot partition will be filled with copies of unused kernels.

## 8.4 Ports to other operating systems

Right now NixOS is very Linux centric, but since Nix itself has been ported to other operating systems so can NixOS. For many Unix(-like) operating systems the sourcecode is available (*BSD, OpenSolaris) and many packages are shared or similar between these operating systems and Linux. The challenge in porting NixOS to these systems is that there is a stronger coupling between the various components in the operating system, such as the kernel and the C library, than on Linux. On other systems these components are expected to be

deployed together. Decoupling these components is no easy task, but it would be interesting to see if Nix would still work and how much work it would take to do so.

## 8.5 User authentication and security

One of the weak points in NixOS currently is user authentication. The reason for this lies in one of the the security trade-offs that was implemented in Nix, namely the removal of support for so called "setuid" binaries. If a program has the "setuid" bit set it will be running with all the privileges of the owner of the program. An example of using "setuid" in user authentication is the `passwd` program. This program has to read the `/etc/passwd` file (and also `/etc/shadow` if shadow passwords are used). This file has to be protected from write access by normal users, otherwise every user can add accounts, or delete accounts, which only the superuser should be able to do. Yet every user has to be able to update his/her password. Many other system components are also "setuid" and owned by the superuser.

Binaries that are "setuid" are in general owned by users that have more access to certain parts of the system than normal users. These binaries have been a prime target for exploits. The reason "setuid" is not desirable in Nix is that in general a package is not deleted from the Nix store when a package is upgraded, unless it is explicitely deleted using the garbage collection mechanism. There is no "destructive upgrade" as in conventional package management systems. If "setuid" binaries would be allowed in Nix this would pose a serious threat to system security.

For example, say version 1 of a program is "setuid" and owned by `root` and contains a local exploit. Exploiting this hole by a local user would lead to full access to the system for this user. Installation with Nix of a new version without the exploit would not delete the vulnerable version, so the exploit would still exist on the system and would still be exploitable.

The only good option to make "setuid" work in Nix and NixOS is to allow destructive upgrades, where a vulnerable version of a package is explicitely deleted from the Nix store when a new version is installed. No work has currently been done to implement this feature.

## Appendix A

# Reducing the NixOS installer size

The default C library in NixOS is `glibc`. The size of a fully compiled `glibc` is about 78 MB, of which a very large part will never be used by the NixOS installer. There is a lot of unnecessary cruft which can be left out, without losing any functionality, such as local language support.

Many Linux distributions try to reduce the size of their installer by either statically linking in a small subset of `glibc` into the installer, or by using one of the alternative C libraries, such as `uClibc` or `dietlibc` or the more recent `klibc`, which was developed specifically by the Linux kernel programmers for building programs such as installers.

Tests with various C libraries have shown that the NixOS installer does not really have issues when `glibc` is used for the installer on a pretty recent PC. A 40 MB ramdisk (it is 40 MB in size when compressed) still loads fine on a PC with a memory configuration which would these days is considered 'low end': 256 MB. On other systems that are not equiped with a large amount of memory (such as embedded systems) using a different C library would make more sense. The reason why in NixOS a different C library is used to build some programs in the installer is to save space on the installation CD. A smaller installer disk it takes less time to load the initial ramdisk. This makes the install process a little bit faster, and there is also more room on the CD to include other precompiled software packages.

The `dietlibc` C library helps to decrease the size of the install disk. Tools statically linked to `dietlibc` are a bit larger than tools that are dynamically linked to `glibc`, but the size of all tools combined is not larger than the size of the dynamically linked tools plus a full `glibc`.

Some programs (such as `bash`) don't compile nicely with `dietlibc`. In these cases the binaries that are used in NixOS are staticaly linked with `glibc` instead.

```
source $stdenv/setup

ensureDir "$(dirname $out/bin/diet)"

cat > $out/bin/gcc << END
#! $SHELL -e
export NIX_GLIBC_FLAGS_SET=1
exec $dietlibc/bin/diet $gcc/bin/gcc "$@"
END

chmod +x $out/bin/gcc

ln -s $out/bin/gcc $out/bin/cc
```

Figure A.1: A wrapper around `diet`.

## A.1 Support for `dietlibc` in Nix

The `dietlibc` library provides a wrapper around `gcc`, which sets the right flags to link with `dietlibc` instead of `glibc`. The wrapper is called `diet` and is followed by a normal invocation of `gcc`:

```
$ diet gcc -o foo foo.c
```

The `diet` wrapper expects `gcc` to be in the `PATH`. To be able to use the `diet` wrapper in Nix a small wrapper around `diet` is used which is parameterized with the version of `gcc` that should be used (see figure A.1).

Invocations of this wrapper (which is called `gcc` and behaves just like `gcc` itself) will execute the `diet` wrapper with the right instance of `gcc` and link with `dietlibc`. To prevent that `gcc` will still use the header files and libraries from `glibc` the `NIX_GLIBC_FLAGS_SET` variable is set to 1. The `diet` wrapper itself will make sure that `dietlibc` is used.

The Nix expression for `dietlibc` in Nix is very straightforward (see figure A.2).

In Nix compiling and linking is done using a wrapper around `gcc` which sets, amongst others, includepaths to find header files and various linking options to link with the right instance of the C library. A wrapper for using `dietlibc` with another version of `gcc` is also easily created.

Some symbols, like `u_short`, are present in `glibc` but are not present by default in `dietlibc`. There are some programs, like `coreutils` which will fail to compile because of this. To be able to use these definitions an addition flag has to be passed to the compiler, namely `NIX_CFLAGS_COMPILE="-D_BSD_SOURCE=1"`,

```
dietgcc = (import ../build-support/gcc-wrapper) {
  nativeTools = false;
  nativeGlibc = false;
  gcc = (import ../os-specific/linux/dietlibc-wrapper) {
    inherit stdenv dietlibc;
    gcc = stdenv.gcc;
  };
  inherit (stdenv.gcc) binutils glibc;
  inherit stdenv;
};
```

Figure A.2: Nix expression for `diet`.

which will take care of defining these symbols in the correct way.

The tools that are needed for the NixOS installer build without any significant problems, or only need small patches.

## A.2 Statically linking with `glibc`

During the development of the NixOS installer scripts the following code to take the closure of a statically linked version of `bash` is being used:

```
bashStatic=$($NIX/nix-store -qR $(nix-store -r \
  $(echo '(import ./pkgs.nix).bashStatic' | \
    $NIX/nix-instantiate -)))
```

Since `bash` is linked statically, it is to be expected that there are no dependencies: the binary can run perfectly standalone. But in this case `glibc` is still marked as a dependency and returned in the closure.

As it turns out the store path for `glibc` is still present in the statically linked `bash`. This happens because during compile time the binaries are linked with a part of `glibc` which has these paths recorded in it.

The tool in Nix that scans for references of store paths in binaries finds these paths and will therefore conclude that `glibc` is a dependency that is needed at run time, when in fact it is not, which can be shown easily (the output was slightly edited):

```
$ pwd
/nix/store/alrwlshj20lksprj52b1ivgk8s64xg8k-bash-3.0/bin
$ file bash
bash: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.0.0, statically linked, stripped
```

The executable does contain unnecessary references to `glibc`:

```
$ strings bash | grep glibc
/nix/store/1mgfgy3ga4m9z60747s0yzxl0g6w5kxz-glibc-2.3.6/lib/locale
/nix/store/1mgfgy3ga4m9z60747s0yzxl0g6w5kxz-glibc-2.3.6/etc/localtime
/nix/store/1mgfgy3ga4m9z60747s0yzxl0g6w5kxz-glibc-2.3.6/share/zoneinfo
/nix/store/1mgfgy3ga4m9z60747s0yzxl0g6w5kxz-glibc-2.3.6/libexec/getconf
/nix/store/1mgfgy3ga4m9z60747s0yzxl0g6w5kxz-glibc-2.3.6/lib/gconv
/nix/store/1mgfgy3ga4m9z60747s0yzxl0g6w5kxz-glibc-2.3.6/lib/
/nix/store/1mgfgy3ga4m9z60747s0yzxl0g6w5kxz-glibc-2.3.6/etc/ld.so.cache
```

The files that are referenced are not used at all. To be able to still make a valid closure using Nix the references to the Nix store can be replaced by a dummy value:

```
$ strings bash | grep glibc
/nix/store/ffffffffffffffffffffffffffffffff-glibc-2.3.6/lib/locale
/nix/store/ffffffffffffffffffffffffffffffff-glibc-2.3.6/etc/localtime
/nix/store/ffffffffffffffffffffffffffffffff-glibc-2.3.6/share/zoneinfo
/nix/store/ffffffffffffffffffffffffffffffff-glibc-2.3.6/libexec/getconf
/nix/store/ffffffffffffffffffffffffffffffff-glibc-2.3.6/lib/gconv
/nix/store/ffffffffffffffffffffffffffffffff-glibc-2.3.6/lib/
/nix/store/ffffffffffffffffffffffffffffffff-glibc-2.3.6/etc/ld.so.cache
```

This storepath is not used in the Nix store and the closure will not suck in `glibc`.

# Bibliography


[1] LWN Distributions List. http://lwn.net/Distributions/

[2] RPM Package Manager. http://www.rpm.org/

[3] http://www.linuxjournal.com/article/7034

[4] http://www.oreillynet.com/onlamp/blog/2006/01/rpm_rollback_in_fedora_core_45.html

[5] FreeBSD ports collection. http://www.freebsd.org/ports/index.html

[6] http://portdowngrade.sourceforge.net/

[7] http://ezine.daemonnews.org/200406/ports-things-go-wrong.html

[8] FreeBSD handbook. http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/makeworld.html

[9] http://www.opengroup.org/onlinepubs/007908799/xcu/xargs.html

[10] Eelco Dolstra, Martin Bravenboer and Eelco Visser. Service Configuration Management. In *12th International Workshop on Software Configuration Management (SCM-12)*, September 2005.

[11] Fedora Project. Anaconda installer http://fedora.redhat.com/projects/anaconda-installer/

[12] Eelco Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Universiteit Utrecht, Utrecht, The Netherlands, 2006

[13] http://www.cs.uu.nl/groups/ST/Trace/Nix, 2005

[14] http://www.stratego-language.org/Stratego/ContinuousDistribution

[15] Linux kernel. http://www.kernel.org/

[16] GNU Mach microkernel. http://www.gnu.org/software/hurd/gnumach.html

[17] LILO bootloader. http://home.san.rr.com/johninsd

[18] Free Software Foundation, GNU GRUB. http://www.gnu.org/software/grub/

[19] GNU stow, http://www.gnu.org/software/stow/stow.html