Nix: A Safe and Policy-Free System for Software Deployment

(Extended Abstract)

Eelco Dolstra*, Eelco Visser and Merijn de Jonge
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
{eelco, visser, mdejonge}@cs.uu.nl

April 20, 2004

Abstract

Existing systems for software deployment are neither safe nor sufficiently flexible. Primary safety issues are the inability to enforce reliable component dependency specification, and a lack of support for multiple versions or variants of a component. This renders deployment operations such as upgrading or deleting components dangerous and unpredictable. A deployment system must also be flexible (i.e., policy-free) enough to support both centralised and local package management, and to allow a variety of mechanisms for transferring components. In this paper we present Nix, a deployment system that addresses these issues through a simple technique of using cryptographic hashes to compute unique paths for component instances.

Note: if accepted, the authors would prefer to present the paper as a "long talk".

1 Introduction

Software deployment is the act of transferring software to the environment where it is to be used. This is a surprisingly hard problem: a number of requirements make effective software deployment difficult in practice, as most current systems fail to be sufficiently *safe* and *flexible*.

The main safety issue that a software deployment system must address is *consistency*: no deployment action should bring the system into an inconsistent state. For instance, an installed component (software package) should never be able to refer to any component not present in the system; and upgrading

*Primary contact. Daytime telephone: +31-30-2533261; evening telephone: +31-30-6933000.

or removing components should not break other components or running programs [6]. It must also be possible to have multiple versions and variants of the same component installed and operational at the same time.

Deployment systems must be flexible. They should support both *centralised* and *local package management*: it should be possible for both site administrators and local users to install applications, for instance, to be able to use different versions and variants of components. Finally, it must not be difficult to support deployment both in source and binary form, or to define a variety of mechanisms for transferring components. In other words, a deployment system should provide flexible *mechanisms*, not rigid *policies*.

Many deployment systems have been created (e.g., RPM [5], Depot [7, 9], Deployme [8]). However, each fails to satisfy some or all of the above requirements. For instance, a LISA summary of twelve years of research in this field indicates that many existing tools ignore the problem of multiple versions of components and that end-user customisation has only been slightly examined [3]. Thus, there are still many hard outstanding deployment problems, and there seems to be no general deployment system available that satisfies all requirements. Most existing tools only consider a small subset of these requirements and ignore the others.

Contribution In this paper we present Nix[1], a safe and flexible deployment system providing mechanisms that can be used to define a great variety of deployment policies. The prime features of Nix are: i) concurrent installation of multiple versions and variants; ii) atomic upgrades and downgrades; iii) multiple user environments; iv) safe dependencies; v) complete deployment; vi) transparent source and binary deployment; vii) safe removal of components through garbage collection; viii) multi-level package management (i.e., different levels of centralised and local package management); ix) portability. The focus of the paper is on the principles and techniques underlying our system.

2 Safe deployment

Global unique path names The aforementioned features follow from the simple technique of using cryptographic hashes to compute unique paths for component instances. That is, each component installed on a system resides in a unique path, e.g., /nix/store/5e1454b4a9438756e961595c8275caff-firefox-0.8 for an instance of Mozilla Firefox. The hexadecimal number is a cryptographic hash of *all* inputs involved in building the component, such as the sources, the compiler, the C library used by it, the platform on which the build takes place, and so on.

This technique allows reliable determination of component dependencies through a technique borrowed from conservative garbage collection: to determine the set of components that a component depends on, we can *scan* the files

[1] Available as free software from http://www.cs.uu.nl/groups/ST/Trace/Nix.

of the component for hashes of other components. (This technique is described in detail in a previous paper [4].) Thus, we can reliably deploy a component by also deploying all components directly or indirectly referenced by it. For instance, since Firefox depends on GTK, the path of GTK is passed as an input to the build process of Firefox. As a result, the path ends up in the rpath field of the Firefox executable. This enables the scanning approach to detect that GTK is a run-time dependency. Consequently, when Firefox is deployed, the Nix system will also deploy GTK.

Safe coexistence of component variants Component variants can safely coexist, since if any input to the build of a component differs (e.g., the source or the compiler flags), a different hash will result. Since components with different hashes are stored in different locations, different variants cannot clash.

Consider, for instance, the (real-life) scenario where we install an application that requires GTK 2.4. Although GTK 2.4 is claimed to be binary-compatible with GTK 2.2, in practice it is not. For example, the library wxGTK does not work with version 2.4 because it relies on some undocumented functions of 2.2. (This shows the ineffectiveness of version-based notions of component compatibility). In many deployment systems the upgrade of GTK would typically overwrite the old version. This would break wxGTK. In Nix, on the other hand, this is not a problem, since the installed wxGTK continues to refer to the old version of GTK.

Nix expressions It may appear at first glance that the use of these cryptographic hashes in path names is inconvenient to developers and users. Fortunately, this is not the case because developers do not explicitly specify hashes. Instead, hashes are generated automatically from *Nix expressions*. This is a small language used to describe components, the variability in those components, and the composition thereof.

User environments Likewise, users are shielded from the hash paths through *user environments*, which are (on Unix) just sets of symlinks to those applications that have been "activated" (similar to GNU Stow [1]). User environments are not updated destructively; rather, they are just regular components (i.e., stored at a hashed path). Each installation action (through commands such as nix-env -i firefox) results in the construction of a new user environment based on the current one. Switching between environments is an atomic action (implemented by replacing a single symlink on Unix): at no point in time can the user observe that a component is only "half" installed or upgraded. Also, previous environments are retained to enable rollbacks.

Automatic garbage collection User environments also serve as roots of the *garbage collector*, which allows safe removal of components. That is, a component can be automatically removed when it is no longer reachable in the component dependency graph from the currently existing user environments.

E.g., GTK 2.2 will be deleted automatically only if there are no applications left that directly or indirectly refer to it.

Transparent source and binary deployment Nix expressions in principle describe the construction of components from source. Thus, an installation action causes the component (and its dependencies) to be built. This is generally undesirable for efficiency reasons. However, since a component is uniquely described by its hash, a "caching" scheme can be used safely to implement regular binary deployment: if we install a component with hash h, and a mapping on some installation medium (such as an FTP site) specifies that a pre-built package is available for hash h, we can use that in lieu of building it. Thus, we obtain binary deployment. On the other hand, if the user requests the installation of a variant for which no pre-built package is known, it will be built locally. Thus, binary (RPM-style) deployment gracefully "degrades" into source (FreeBSD Ports/Gentoo-style) deployment, and we get the best of both worlds.

3 Policy-free deployment

Nix is policy-free. That is, the ingredients introduced above are *mechanisms* for implementing software deployment. A wide variety of *policies* can be based on these mechanisms.

For instance, depending on the type of organisation it may or it may not be desirable or possible that users install applications. In an organisation where homogeneity of workspaces is important, the selection and installation of applications can be restricted to system administration. This can be achieved by restricting all the operations on the store, and the composition of user environments to system administration. They may compose several prefab user environments for different classes of users. On the other hand, for instance in a research environment, where individual users have very specific needs, it is desirable that users are capable of installing and upgrading applications themselves. In this situation environment operations and the underlying store operations can be made available to ordinary users as well.

Similarly, Nix enables deployment at different levels of granularity, from a single machine, a cluster of machines in a local network, to a large number of machines on separate sites. By composing environments for all relevant pairs of machine and user requirements, a cache of pre-built components can be filled. This cache can be used to automatically install components on a large number of machines.

4 Experience

Nix is currently being used for software deployment. We have created Nix expressions for some 160 components, ranging from basic components such as glibc to large applications such as Firefox (including *all* its dependencies). The system

has proven its worth in supporting the evolution of software installations. For instance, application upgrades that would lead to breakage of other applications in conventional systems due to mutually incompatible shared dependencies do not cause a problem with Nix.

We find that Nix is also very valuable for server systems where the ability to upgrade and rollback efficiently is important. For instance, our department's Subversion [2] server is administered using Nix, which allows atomic upgrades and, if necessary, rollbacks of the server configuration.

5 Outline of Full Paper

1. *Introduction*: provides a brief overview of the main issues in software deployment, lists the paper's contributions, and outlines the remainder of the paper.

2. *Motivation*: discusses deployment issues in greater detail (i.e., the requirements to be placed on a mature deployment system), including the need for reliable dependency identification, for supporting co-existing components variants, for ensuring consistency of an installation, for atomicity of deployments actions, and for supporting a wide range of deployment policies.

3. *Overview*: gives a bird's eye view of Nix from the perspective of its users (i.e., creators of Nix expressions, administrators, and end-users) to provide an intuition of the system.

4. *Implementation*: discusses in detail the underlying techniques used to ensure safe deployment.

5. *Deployment policies*: shows how a variety of deployment scenarios can be realised. This deals with the transmission of Nix expressions and pre-built packages from the producer to the consumer.

6. *User environment policies*: shows different ways in which user environments (a user's view on the set of available components) can be synthesised, from simple scenarios (e.g., all components come from the same source, and work on a single platform), to more complex ones (e.g., component come from multiple "channels", and must be kept synchronised between several platforms).

7. *Experience*: relates our experiences with the Nix system to show its effectiveness. We intend to continue experimenting with Nix for the deployment of our large collection of software components. These components need to be deployed in many compositions and variants, and they should be compiled on a number of different platforms. Consequently, deployment is complex due to the large number of software variants that we have to face.

8. *Related work*.

9. *Conclusion*: summarises the paper's main contributions and ends with some ideas for future work.

References

[1] GNU Stow. http://www.gnu.org/software/stow/.

[2] Subversion. http://subversion.tigris.org/.

[3] E. Anderson and D. Patterson. A retrospective on twelve years of LISA proceedings. *Proceedings of the 13th Systems Administration Conference (LISA 1999)*, pages 95–107, November 1999.

[4] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*. To appear, May 2004.

[5] Eric Foster-Johnson. *Red Hat RPM Guide*. John Wiley and Sons, 2003.

[6] John Hart and Jeffrey D'Amelia. An analysis of RPM validation drift. In *Proceedings of the 16th Systems Administration Conference (LISA-2002)*, pages 155–166. USENIX Association, November 2002.

[7] K. Manheimer, B. A. Warsaw, S. N. Clark, and W. Rowe. The Depot: A framework for sharing software installation across organizational and UNIX platform boundaries. *Proceedings of the 4th Systems Administration Conference (LISA 1990)*, pages 37–46, October 1990.

[8] K. Oppenheim and P. McCormick. Deployme: Tellme's package management and deployment system. *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, pages 187–196, December 2000.

[9] W. C. Wong. Local disk depot: customizing the software environment. *Proceedings of the 7th Systems Administration Conference (LISA 1993)*, pages 49–53, November 1993.