A Generic Approach for Deploying and Upgrading Mutable Software Components

Sander van der Burg
Department of Software Technology, Delft University of Technology, Netherlands, s.vanderburg@tudelft.nl

***Abstract*—Deploying and upgrading software systems is typically a labourious, error prone and tedious task. To deal with the complexity of a software deployment process and to make this process more reliable, we have developed Nix, a purely functional package manager as well as an extension called Disnix, capable of deploying service-oriented systems in a network of machines. Nix and its applications only support deployment of *immutable* components, which never change after they have been built. However, not all components of a software system are immutable, such as databases. These components must be deployed by other means, which makes deployment and upgrades of such systems difficult, especially in large networks. In this paper, we analyse the properties of mutable components and we propose *Dysnomia*, a deployment extension for *mutable* components.**

I. Introduction

Many software systems these days are large, complex, consist of many components and are rarely self contained. As a consequence, the *deployment* process of software systems has become very tedious, complex and error prone [1]. In order to make a system available for use, many tasks must be performed, such as building software components from source code, delivering artifacts from the producer site to consumer site and activation of the components in the right order. Each task has a number of challenges that must be dealt with, such as the fact that dependencies may be broken or incorrect, which could lead to a phenomenon known as the DLL-hell.

Software systems are nowadays not only composed of components residing on single systems, but may also be composed of *services* deployed on various machines in a network (or in the cloud), working together to achieve a common goal. The complexity of the deployment process of distributed systems is even bigger.

Upgrading systems is a more difficult and risky process than performing a clean installation, because it is not always obvious which parts of a system can be replaced and upgrade steps may break a system, e.g. by removing or replacing a critical file. Furthermore, systems are temporarily inconsistent, because there is a time window in which files are in use belonging to both the old version and new version.

To deal with the complexity of deployment processes, we have developed Nix [2], [3], a purely functional package manager borrowing concepts from purely functional programming languages, such as Haskell. Nix provides a number of distinct features compared to conventional deployment tools, such as a purely functional domain-specific language to make builds reproducible by removing many side effects, a Nix store which safely stores multiple versions and variants of components next to each other, atomic upgrades and rollbacks, and a garbage collector which safely removes components which are no longer in use.

As Nix only manages the deployment of packages on a single system, we have developed *Disnix* [4], [5], as an extension providing features to deploy service-oriented systems in a network of machines.

In the purely functional deployment model of Nix, all components are stored as *immutable* file system objects in the Nix store, which never change after they have been built. Making components immutable ensures purity and reproducibility of builds. However, not all components of a system can be managed in such a model, because they may change at run-time, such as databases. Currently, Nix-related tools manage *mutable* state in an ad-hoc manner and outside the Nix store, e.g. by creating scripts which initialise a database at first startup. As a consequence, system administrators or developers who deploy systems using Nix-related tools still need to manually deploy and upgrade mutable components, which can be very tedious in large networks of machines.

In this paper, we give a brief introduction to Nix and Disnix, we explore the properties of mutable components and we propose *Dysnomia*, a generic deployment extension using concepts close to Nix, to manage the deployment and upgrades of mutable components. This tool is designed to be used in conjunction with Disnix and other Nix-related tools.

II. The Nix Deployment System

Nix is a package manager (with similar purposes as RPM [6]) designed to automate the process of installing, upgrading, configuring and removing software packages on a system. Nix offers a number of distinct features to make deployment more reliable.

Nix stores packages in *isolation* in a Nix store. Every component in the Nix store has a special filename, such as: `/nix/store/324pq1...-firefox-9.0.1` in which the former part: `324pq1...` represents a cryptographic hash-code derived from all build-time dependencies. If, for example, Firefox is build with a different version of GCC or linked against a different version of the GTK+ library, the hash code will differ and thus it is safe to store multiple variants next to each other, because components do not share the same name.

```
rec {
  stdenv = ...
  fetchurl = ...

  gtk = stdenv.mkDerivation {
    name = "gtk-2.24.1";
    src = ...
  };

  firefox = stdenv.mkDerivation {
    name = "firefox-9.0.1";
    src = fetchurl {
      url = http://../firefox-9.0.1-source.tar.bz2;
      md5 = "195f6406b980dce9d60449b0074cdf92";
    };
    buildInputs = [ gtk ... ];
    buildCommand = ''
      tar xfvj $src
      cd firefox-*
      ./configure
      make
      make install
    '';
  };
  ...
}
```

Figure 1. A partial Nix expression

After components have been built, they are made immutable by removing the write permission bits.

Nix packages are derived from declarative specifications defined in the purely functional Nix expression language. An example of a partial Nix expression is shown in Figure 1, containing an attribute set in which each attribute refers to a function building the component from source code and its dependencies. For instance, the firefox attribute value refers to a function call which builds Mozilla Firefox from source code and uses GTK+ as a build time dependency, which is automatically appended to the PATH environment variable of the builder. The output of the build is produced in a unique directory in the Nix store, which may be: /nix/store/324pq1...-firefox-9.0.1.

While performing a build of a Nix package, Nix tries to remove many side effects of the build process. For example, all environment variables are cleared or changed to unexisting values, such as the PATH environment variable. We have also patched various essential build tools such as gcc in such a way that they ignore impure global directories, such as /usr/include and /usr/lib. Optionally, builds can be performed in a chroot environment for even better guarantees of purity.

Because Nix removes as many side effects as possible, dependencies can only can be found by *explicitly* setting search directories, such as PATH to Nix store paths. Undeclared dependencies, which may reside in global directories such as /usr/lib cannot accidentally let a build succeed.

The purely functional deployment model of Nix has a number of advantages compared to conventional deployment tools. Because most side-effects are removed, a build function always gives the *same* result regardless on what machine the build is performed, giving a very high degree of reproducibility. Furthermore, we can *efficiently* upgrade systems, because components with the same path, which are already present do not have to be rebuilt or redownloaded. Upgrades of systems are also always *safe*, because newer versions and variants of components can be safely coexist with components belonging to an older configuration. Finally, there is never a situation in which a package has files belonging to an old and new version at the same time.

```
{pkgs, system, distribution}:

rec {
  testDatabase = {
    name = "TestDatabase";
    type = "mysql-database";
    pkg = pkgs.testDatabase;
  };

  testWebApplication = {
    name = "TestWebApplication";
    dependsOn = {
      inherit testDatabase;
    };
    pkg = pkgs.testWebApplication;
    type = "tomcat-webapplication";
  };
}
```

Figure 2. A partial Disnix services model

III. Disnix

Disnix extends Nix by providing additional models and tools to manage the deployment of service-oriented systems in a network of machines. Disnix uses a *services* model to define the distributable components available, how to build them, their dependencies and their types used for activation. An *infrastructure* model is used to describe what machines are available and their relevant properties. The *distribution* model maps services defined in the services model to machines defined in the infrastructure model.

Figure 2 shows an example of a service model, describing two distributable components (or services). testDatabase is a MySQL database storing records. testWebApplication is a Java web application which depends on the MySQL database.

Each service defines a number of attributes. The name attribute refers to its canonical name, dependsOn defines the dependencies of the service on other services (which may be physically located on another machine) and pkg defines how the service must be built (not shown here). The type attribute is used for the activation of the components by invoking a script attached to the given type. For example, the tomcat-webapplication type indicates that the given component should be deployed in an Apache Tomcat container. The mysql-database type is used to deploy an initial MySQL database from a given SQL file containing table definitions. Disnix supports a number of predefined activation types as well as an extension interface, which can be used to implement a custom activation type.

By running the disnix-env command with the three models as command-line parameters, the *complete* deployment process of a service-oriented system is performed, which consists of two phases. In the *distribution* phase, components are built from source code (including all their dependencies) and transferred to the target machines in the network. Because Nix components can be safely stored next to existing components of older configurations, we can safely perform this procedure without affecting an existing system configuration. After this phase has succeeded, Disnix enters the *transition* phase, in which obsolete components of the old configuration are deactivated and new components are activated in the right order. Optionally, end-user connections can be blocked/queued in this phase, so that end-users will not notice that the system is temporarily inconsistent.

In case of a change in a model, an *upgrade* is performed in which only necessary parts of the system are rebuilt and reactivated. Disnix determines the differences between configurations by looking at the hash code parts of Nix store components. If they are the same in both configurations, they do not have to be deployed again.

IV. Mutable Components

The components managed by Nix and Disnix are immutable. Mutable components, such as databases, cannot be managed in such a model, apart from the initial deployment step from a static schema. If a component in the Nix store would change, we can no longer guarantee that a deterministic result is achieved when it is used during a build of another component, because hash codes may no longer uniquely represent a build result. Apart from the fact that mutable components change over time, they have a number of other distinct characteristics compared to immutable components managed in the Nix store.

First, many mutable component types are hosted inside a *container*, such as a database management system or an application server. In order to deploy mutable components, file system operations are typically not always enough; Also the state of the container must be adapted. For example, to deploy a database component, the DBMS must be instructed to create a new database with the right authorisation credentials, in order to deploy a Java web application on a Servlet container, the Servlet container must be notified.

Second, state is often stored on the file system in a format which is close to the underlying implementation of the container and the given platform. These files may not be portable and they may have to reside in an unisolated directory reserved for the container. For example, it may be impossible to move database state files from a 32-bit instance of a container to a 64-bit instance of a container, or to transfer state data from a little endian machine, to a big endian machine. Furthermore, state files may also contain locks or may be temporarily *inconsistent*, because of unfinished operations.

Because of these limitations, many containers have utilities to dump their state data in a portable format (e.g. mysqldump) and in a consistent manner (e.g. single transaction) and can reload these dumps back into memory, if desired. Because such operations may be expensive, many containers can also create incremental dumps. In this paper, we refer to the actual files used by a container as *physical* state. A dump in a portable format is a *logical* representation of a physical state.

V. Dysnomia

Nix and Disnix provide automated, reliable deployment of static parts of software systems and networks of software systems. In this section, we explore how we can manage mutable software components in conjunction with Nix-related tools, in an extension which we call *Dysnomia*.

A. Managing Mutable Components

Deployment operations of Nix components are typically achieved by performing operations on the filesystem. As mentioned in Section IV, deployment of mutable components cannot be typically done by filesystem operations. Therefore, we must interact with tooling capable of activating and deactivating mutable components and we need to interact with tools capable of generating snapshots representing the logical representation of the state. Because these tools are different for each component type, we need to provide an abstraction layer, which can be used to uniformly access all mutable components.

To deploy mutable components, Dysnomia defines an *interface* with the following operations:

- activate. Activates the given mutable component in the container. Optionally, a user can provide a dump of its initial state.
- deactivate. Deactivates the given mutable component in the container.
- snapshot. Snapshots the current physical state into a dump in a portable format.
- incremental-snapshot. Creates an incremental snapshot in a portable format.
- restore. Brings the mutable component in a specified state by restoring a dump.

The actual semantics of the operations depend on the mutable component type. For example, the activate operation for a MySQL database implies creating a database and importing an initial schema, whereas the activate operation for a Subversion repository means creating a Subversion repository. The snapshot operation of a MySQL database invokes mysqldump which stores the current state of database in a SQL file and the snapshot operation for a Subversion repository invokes svnadmin dump to create a Subversion dump.

B. Identifying Mutable Components

Nix and Nix applications use unique names of components to make deployment reproducible and to determine how to efficiently upgrade systems. For example, if a particular component with a hash code already exists, we do not have to build it again because its properties are the same.

In order to reproduce state in conjunction with static parts of the system, we need to snapshot, transfer and restore the logical representation of state of mutable components and we must be capable of distinguishing between these snapshots based on their names so that we can decide whether we have to deploy a new generation or whether it can remain untouched. Unfortunately, we cannot use a hash of build-time dependencies to uniquely address mutable component generations.

To uniquely identify logical state of mutable components several properties are important. First, we need to know the name of the mutable component, which refers to the database name or Subversion repository name. We need to know in which container it is hosted, which we can identify by a container name and an optional identifier (if there are more instances of the same container type). Third, we need an attribute to make a distinction between generations, which must be derived from the container that is being used. For example, a generation number can be a MySQL binary log sequence number or a Subversion revision number. Also a time stamp could be used if a container does not have facilities to make a distinction between generations, although this may have practical implications, such as synchronisation issues. For example, the logical state of a particular Subversion revision can be identified by: `/dysnomia/svn/default/disnix/34124`.

C. Extending Disnix

We have replaced the Disnix activation module system by Dysnomia and extended the deployment procedure of `disnix-env` described in Section III, to include deployment steps for mutable components. In the transition phase, in which access from end users can be temporarily blocked, we first determine the names of the logical state snapshots of mutable components. Then we create dumps of all mutable components which have not been snapshotted yet (which becomes obvious by checking whether a dump with the same filename already exists) and we transfer these dumps to the coordinator machine. After deactivating obsolete services in the old configuration and activating new services in the new configurations, the state for the mutable components is transferred from the coordinator to their new locations in the network and activated. Finally, end user access is restored.

Because creating snapshots of mutable components may be expensive (especially for large databases) and because access to end users may be blocked, this procedure may give a large time window in which the system inaccessible, which is not always desirable. Therefore, we have also implemented an optimised version of this procedure using incremental snapshots (for mutable components capable of doing this). In this improved version, we create dumps of mutable components before the transition phase, allowing end users to still access the system. After the dumps have been transferred to their new locations, we enter the transition phase (in which end-user access is blocked) and we create and transfer incremental dumps, taking the changes into account which have been made while performing the upgrade process. In this procedure we still have downtime, but this time window is reduced to a minimum.

VI. Experience

We have developed Dysnomia modules for a number of mutable component types, which we have listed in Table I. Each column represents a separate aspect of the module, such as the container for which it has been designed, the semantics of the activation, deactivation, snapshot and restore operations and which attribute is used to make a distinction in naming the logical state snapshots. Most of the modules are designed for deployment of databases, which instruct the DBMS to create or drop a database and create snapshots and restore snapshots using a database administration tool. Additionally, we have implemented support for Subversion repositories which use a revision number to distinguish between generations. Ejabberd uses a database for storing user account settings, but has no support for incremental backups and a timestamp is the only way to make a distinction between generations. We have also implemented support for several other types of components, such as web applications and generic UNIX processes. These components typically do not change, but they still need to be activated in a container, which state must be modified.

We have applied Dysnomia in conjunction with Disnix to a number of case studies, such as two toy systems, ViewVC (http://viewvc.tigris.org) and an industrial case study from Philips Research. We have performed various redeployment scenarios, in which the state of mutable components had to be captured and transferred. For the Philips case study, redeployment took a long time in certain scenarios as the size of a number of databases was quite large (the total size of the entire system is 599 MiB).

VII. Related work

Many deployment tools support deployment of mutable state in some degree, although they have no consensus about mutable components. Conventional package managers such as RPM, modify state of parts of a system by using *maintainer scripts* invoked during the installation of a package. The purpose of maintainer scripts is to "glue" files from a package to files already residing on the system. The disadvantage is that they imperatively modify parts of the system, which cannot be trivially undone, and the operations depend on the current state of the system. As a consequence,

Table I
A COLLECTION OF DYSNOMIA TYPES AND THEIR PROPERTIES

Type	Container	Activation	Deactivation	Snapshot	Restore	Ordering
mysql-database	MySQL	Create database	Drop database	mysqldump	mysql import	Binlog sequence
postgresql-database	PostgreSQL	Create database	Drop database	pg_dump	psql import	Binlog sequence
subversion-repository	Subversion	Create repository	Drop repository	svnadmin dump	svnadmin load	SVN revision
ejabbard-database	Ejabberd server	Init database	Drop database	ejabberdctl backup	ejabberdctl restore	Timestamps
tomcat-webapplication	Apache Tomcat	Symlink directory	Remove symlink	-	-	-
apache-webapplication	Apache HTTPD	Symlink WAR file	Remove symlink	-	-	-
process	init/Upstart	Symlink job	Remove symlink	-	-	-

they may produce different results on other machines. The Mancoosi project investigates how to provide transactional upgrades of packages with maintainer scripts, by means of a system model and DSL language to implement maintainer scripts [7]. By simulating whether an upgrade may succeed, they can be made transactional. To perform rollbacks, inverse operations can be derived.

Also other model-driven deployment frameworks such as Cfengine [8] (and derivatives) and the IBM Alpine project [9] are capable of dealing with mutable state, although in an imperative and ad-hoc manner.

An earlier attempt of managing mutable state with Nix has been done in [10] in which a Copy-On-Write (COW) filesystem is used to capture mutable state of containers in constant time. Although this is more efficient, only the *physical* state can be captured, with all its disadvantages, such as the fact that it may be inconsistent and non portable. Also, the state of a container is captured as a whole, while it may also be desirable to treat parts of the container state as separately deployable mutable components (e.g. databases).

VIII. DISCUSSION

Dysnomia provides a generic manner for deploying mutable components and abstracts over physical and logical state. The major disadvantage of this approach is the use of the filesystem as an intermediate layer. For certain types of mutable components this works acceptably. For large databases, a distributed deployment scenario can be very costly, because it may take a long time to write a snapshot to a disk, transfer the snapshot and to restore it again, even for incremental dumps. Many modern database management systems have replication features, which can perform these tasks more efficiently. However, database replication is difficult to integrate in the Nix deployment model.

IX. CONCLUSION

In this paper, we have analysed issues with deployment and upgrades of mutable components in conjunction with Nix and we have described Dysnomia, an extension for deployment of mutable components. With Dysnomia, mutable components can be conveniently deployed in a generic manner. A drawback is that redeployment may be very costly for certain component types, such as very large databases.

ACKNOWLEDGEMENTS

This research is supported by NWO-JACQUARD project 638.001.208, *PDS: Pull Deployment of Services*. We wish to thank Eelco Dolstra: the author of Nix.

REFERENCES

[1] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf, "A characterization framework for software deployment technologies," University of Colorado, Tech. Rep. CU-CS-857-98, 1998.

[2] E. Dolstra, E. Visser, and M. de Jonge, "Imposing a memory management discipline on software deployment," in *Proc. 26th Intl. Conf. on Software Engineering (ICSE 2004)*. IEEE, May 2004, pp. 583–592.

[3] E. Dolstra, "The purely functional software deployment model," Ph.D. dissertation, Faculty of Science, Utrecht University, The Netherlands, Jan. 2006.

[4] S. van der Burg, E. Dolstra, and M. de Jonge, "Atomic upgrading of distributed systems," in *First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp)*, T. Dumitraş, D. Dig, and I. Neamtiu, Eds. ACM, Oct. 2008.

[5] S. van der Burg and E. Dolstra, "Automated deployment of a heterogeneous service-oriented system," in *36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, September 2010.

[6] E. Foster-Johnson, *Red Hat RPM Guide*. John Wiley & Sons, 2003, also at http://fedora.redhat.com/docs/drafts/rpm-guide-en/.

[7] A. Cicchetti, D. Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli, "A model driven approach to upgrade package-based software systems," in *Evaluation of Novel Approaches to Software Engineering*. Springer Berlin Heidelberg, 2010, vol. 69, pp. 262–276.

[8] M. Burgess, "Cfengine: a site configuration engine," *Computing Systems*, vol. 8, no. 3, 1995.

[9] T. Eilam, M. Kalantar, A. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal, "Managing the configuration complexity of distributed applications in internet data centers," *Communications Magazine, IEEE*, vol. 44, no. 3, pp. 166 – 177, march 2006.

[10] W. den Breejen, "Managing state in a purely functional deployment model," Master's thesis, Faculty of Science, Utrecht University, The Netherlands, Mar. 2008.