Atomic Upgrading of Distributed Systems

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

Eelco Dolstra
Delft University of Technology,
The Netherlands
e.dolstra@tudelft.nl

Merijn de Jonge
Philips Research,
The Netherlands
merijn.de.jonge@philips.com

Abstract

Upgrading distributed systems is a complex process. It requires installing the right services on the right computer, configuring them correctly, and so on, which is error-prone and tedious. Moreover, since services in a distributed system depend on each other and are updated separately, upgrades typically are not *atomic*: there is a time window during which some but not all services are updated, and a new version of one service might temporarily talk to an old version of another service. Previously we implemented the *Nix package management system*, which allows atomic upgrades and rollbacks on single computers. In this paper we show an extension to Nix that enables the deployment of distributed systems on the basis of a declarative deployment model, and supports atomic upgrades of such systems.

1. Introduction

A distributed system can be described as a collection of independent computers that appear to a user as one logical system (Nadiminti et al. 2006). Services in a distributed system are working together to reach a common goal by exchanging data with each other. Services have dependencies on components on the same computer, the so-called *intra-dependencies*, but also on services on other computers in the network, the *inter-dependencies*.

Upgrading distributed systems is usually a semi-automatic process. Some tasks are executed manually and for some tasks specific deployment tools are used. The problem with this approach is that it requires people with knowledge about these upgrade tasks and up-to-date documentation. The upgrade process becomes problematic when the documentation is missing or outdated, or if people with necessary skills are not available.

Another problem of upgrading distributed systems is integrity. If we want to change the deployment state of a distributed system to a new state, for instance by adding, removing or replacing services on computers or migrating services from one computer to another we do not want to end up with a system in an *inconsistent* state, i.e., a mix of the old and new configurations. Thus, if we upgrade a distributed system, there should never be a point in time where a new version of a service on one computer can talk to an old version of a service on another computer.

To make the upgrade process reproducable and more efficient, the deployment and upgrade process should be automatic. Therefore, we need to capture the services and network, as well as the assignment of services to computers in the network, in a model. This model allows us to deterministically replicate a deployment scenario in a particular environment. The second advantage of such a model is that we can reason about its correctness. Finally, with a model of the deployment, we always know the deployment state of the distributed system. To guarantee the correctness of the deployment we need some notion of a transactional, atomic commit similar to database systems. That is, the transition from the current deployment state to the new deployment state should be performed completely or not at all, and it should not be observable half-way through.

We previously developed the Nix package manager (Dolstra 2006), a software deployment system that builds software packages from purely functional models. It guarantees that dependency specifications are complete and supports atomic upgrades and rollbacks. However, its models and atomic upgrades deal with single computers. In this paper we propose an extension to Nix, called *Disnix*, which extends the Nix language to model distributed deployments. We also show how distributed systems can be upgraded atomically.

2. The Nix package manager

Conventional package managers such as RPM (Foster-Johnson 2003) do not support atomic updates of packages. This is because updating a package requires replacing all the files belonging to the package with newer versions, which is not something that can be done in an atomic transaction on most file systems. Thus, during the update, there is a time window during which the system is in an inconsistent state: the file system contains some files belonging to the old ver-

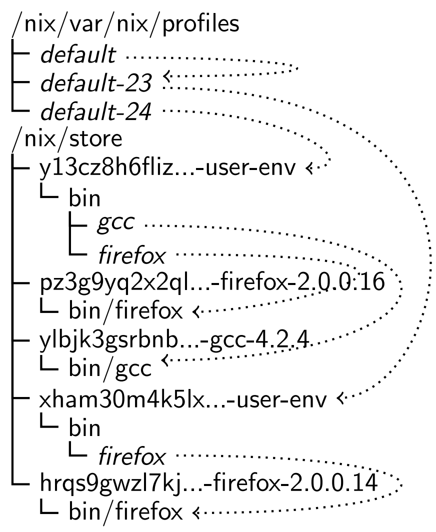

Figure 1. Nix store and atomic upgrades using symlink indirection

sion of the package, and some belonging to the new version. The effect of using the package during this time window (e.g., starting a program in the package) is undefined.

The Nix package manager (Dolstra et al. 2004; Dolstra 2006) solves this problem by building and storing packages in a purely functional manner: packages are built from source using descriptions in a simple purely functional language, and build results are immutable. This ensures that multiple versions of a package can coexist on a system, that upgrades can be performed in an atomic manner, and that it is possible to roll back to previous configurations. It is used among other things as the basis for NixOS (http://nixos.org/), a Linux distribution with a purely functional configuration management model (Dolstra and Löh 2008).

Figure 1 shows how Nix stores packages. Rather than storing packages in global namespaces such as the /usr/bin directory on Unix, each package is stored in isolation of others in a subdirectory of the *Nix store*, the directory /nix/store. For instance, the directory /nix/store/-pz3g9yq2x2ql...-firefox-2.0.0.16 contains a particular instance of Mozilla Firefox. The string pz3g9yq2x2ql... is a 160-bit cryptographic hash of the inputs to the build process of the package: its source code, the build script, dependencies such as the C compiler, etc. This scheme ensures that different versions or variant builds of a package do not interfere with each other in the file system. For instance, a change to the source code (e.g., due to a new version) or to a dependency (e.g., using another version of the C compiler) will cause a change to the hash, and so the resulting package will end up under a different path name in the Nix store.

Nix builds packages from a description in a purely functional language called the *Nix expression language*. Nix expressions describe dependency graphs of build actions, called *derivations*, that each build a path in the Nix store.

```
rec {
  HelloService = derivation {
    name = "HelloService-1.0";
    src = fetchurl {
      url = http://nixos.org/.../HelloService.tar.gz;
      md5 = "de3187eac06baf5f0506c06935a1fd29";
    };
    buildInputs = [ant jdk axis2];
    buildCommand = ''
      tar xf $src
      cd HelloService
      ant generate.service.aar
      mkdir -p $out/webapps/axis2/WEB-INF/services
      cp HelloService.aar $out/webapps/axis2/WEB-INF/services
    '';
  };

  HelloWorldService = derivation { ... };
  stdenv = ...
  firefox = import ...
  ... # other package definitions
}
```

Figure 2. pkgs.nix, an example of a Nix expression.

Figure 2 shows a Nix expression defining a number of derivations bound to variables that can refer to each other. For instance, the value of the variable HelloService is a derivation that builds an Apache Axis web service. The function derivation is the primitive operation that produces a build action from a set of *attributes*, such as the name of the package, its source (which is produced by the build action returned by the call to the function fetchurl), a shell script that performs the actual build action (buildCommand), and so on. All attributes are passed through environment variables to the build command, e.g., the variable src will contain the path in the Nix store of the downloaded sources. The environment variable out contains the target path in the store for the package, e.g., /nix/store/*hash*-HelloService-1.0.

The user can install packages using a command such as

```
$ nix-env -f pkgs.nix -i firefox
```

which builds the derivation resulting from the evaluation of the firefox variable in Figure 2, along with all its dependencies. To upgrade Firefox, the user updates the Nix expression in question (which is typically automated through a number of mechanisms, such as an automated download mechanism), and performs

```
$ nix-env -f pkgs.nix -u firefox
```

which builds the new Firefox package and makes it available in the user's PATH. This upgrade is atomic: at any point in time, issuing the command firefox either runs the old or the new version of Firefox, but never an inconsistent mix of the two. Figure 1 shows an example where, in an atomic action, we have upgraded Firefox *and* installed a new package, GCC. The user's PATH environment variable contains the path /nix/var/nix/profiles/default/bin, which — via some indirections through symbolic links — resolves to a directory containing symlinks to the currently "installed" programs. Thus, /nix/var/nix/profiles/default/bin/firefox resolves to the currently active version of Firefox. By flipping the symlink /nix/var/nix/profiles/default from default-23 to default-24, we can switch to the new configuration. Since replacing a symlink can be done atomically on Unix, upgrading can be done atomically.

Packages in the store are deleted by a *garbage collector* that determines liveness by following package runtime dependencies from a set of *roots*, namely, the symlinks in /nix/var/nix/profiles. Thus, only when the symlink default-23 is removed can the garbage collector delete firefox-2.0.0.14. Running processes and open files are also roots. This make it safe to run the garbage collector at any time.

Thus, Nix achieves atomicity of package upgrades due to two properties: first, the new version of the package is built and stored separately, not interfering with the currently active version; and second, the new version is activated in an atomic step by updating a single symlink. In the remainder of this paper, we show an extension to Nix that allows atomic upgrades of services and distributed deployments.

3. Disnix

Just as distributed systems appear to a user as one logical system, we also would like to deploy software on a distributed system as if it were a single system. However, this ideal is hard to achieve with conventional deployment tools. First, the deployment system necessarily has to know to what computer each service should be deployed. Second, it must know the dependencies between services on different computers. So in addition to local dependencies between packages on a single computer, the *intra-dependencies*, there are also dependencies between services on different computers, the *inter-dependencies*.

To make distributed deployment possible we need to extend the Nix deployment system. We call this extension *Disnix*. The Disnix deployment system contains an interface which allows another process or user to access the Nix store and Nix user profiles remotely through a distributed communication protocol, e.g. SOAP. The Disnix system also consists of tools that support distributed installing, upgrading, uninstalling and other deployment activities by calling these interfaces on the nodes in the distributed system.

To deploy packages on a single computer Nix just needs to know the compositions of each package and how to build them. Disnix needs some additional information for deploying services on multiple computers. Therefore we introduce three models: a *services model*, an *infrastructure model* and a *distribution model*. The *services model* describes the services that can be distributed across computers in the network. Each service is essentially a package, except that it has an extra property called *dependsOn* which describes the *inter-dependencies* on other services. Figure 3 shows a Nix expression that describes a model for two services: HelloService, which has no dependencies, and HelloWorldService, which depends on the former.

The *infrastructure model* is a Nix expression that describes certain attributes about each computer in the net-

```
rec {
    pkgs = import ./pkgs.nix;

    HelloService = {
        pkg = pkgs.HelloService;
        dependsOn = [];
    };
    HelloWorldService = {
        pkg = pkgs.HelloWorldService;
        dependsOn = [ HelloService ];
    };
}
```

Figure 3. services.nix

```
{   itchy = {
        hostname = "itchy";
        targetEPR = http://itchy/.../DisnixService;
    };
    scratchy = {
        hostname = "scratchy";
        targetEPR = http://scratchy/.../DisnixService;
    };
}
```

Figure 4. infrastructure.nix

```
{services, infrastructure}:

[   { service = services.HelloService;
      target = infrastructure.itchy; }
    { service = services.HelloWorldService;
      target = infrastructure.scratchy; }
]
```

Figure 5. distribution.nix

work. Figure 4 shows an example of a network with two computers. In the model we describe the hostname and the target endpoint references (targetEPR) of the Disnix interface. The targetEPR is a URL that points to the Disnix webservice that can execute operations on the remote Nix store.

The *distribution model* is a Nix expression that connects the services and infrastructure model, mapping services to computers. It contains a function which takes a services and infrastructure model as its inputs and returns a list of pairs. Each pair describes what service should be distributed to which computer in the network. Figure 5 shows an example. It is also possible to specify a specific component multiple times in the model. In this case the same variant of the service will be deployed on another machine. It is also possible to use multiple variants of a specific component by defining their compositions in the *services* model.

Figure 6 shows an overview of the Disnix system. It consists of a distribution function that takes a distribution model as input. After building the services in the model it distributes them to the target computers and finally activates them. Services that are no longer in use are deactivated. Activation and deactivation are performed via the Disnix interfaces on the target computers.

4. Distributed deployment states

A distributed system has certain services installed on certain computers in its network. We can describe this as a *deploy-*

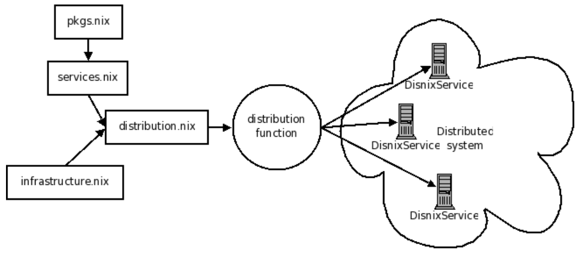

Figure 6. Concepts of the Disnix deployment system

ment state of the distributed system. If we want to change this deployment state, e.g. by installing new services, uninstalling old services, replacing services with newer versions, or by migrating services from one computer to another, we do this by defining a new distribution model. By calculating the intersection of the outputs of the current distribution model and the new distribution model we can derive for each computer in the network what services should be activated and what services should be deactivated.

Let's take $O = \{D \mid D \in (service, computer)\}$ as the output of the previous distribution model and $N = \{D \mid D \in (service, computer)\}$ as the output of the new distribution model. Let's take $I = O \cap N$. Then the services we should uninstall in the network are: $O - I$ and the services we should install in the network are: $N - I$. The distribution actions in set I remain unchanged.

The transition from the old state to the new state can be done by passing the outputs of the current and the new distribution model to the distribution function. By keeping older versions of the outputs it is also possible to switch back to older generations of deployment states. This can be done by passing the output of the current distribution model and the output of an older generation to the distribution function.

Observe that it is always possible to reconstruct the current distribution model and deployment state by capturing the global state of the distributed system. Obviously, this can become a costly operation in a large distributed environment with a lot of computers and slow network connections.

5. Atomic upgrading

To make the transition from the current deployment state to the new deployment state in the distributed system an atomic operation, we use a variant of the *two-phase commit protocol* (Skeen and Stonebraker 1987). This transition has to be atomic because we do not want to end up in an inconsistent state in case of a failure. Either all services should be activated and deactivated as described in the distribution model or the deployment state should stay as it currently is.

The two-phase commit protocol is a distributed algorithm that lets all nodes in a distributed system agree to commit a transaction. One node is called the *coordinator*, which initiates both phases. The rest of the nodes in the network are called the *cohorts*.

The first phase of the algorithm is the *distribution* or *commit-request phase*. In this phase all the nodes execute the transaction until the point that the modifications should be commited. The second phase of the algorithm is the *commit phase*. If all the cohorts succeed in executing the steps in the commit-request phase then the changes will be committed by each cohort. If one or more of the cohorts fail in the commit phase, then every cohort will do a rollback.

In the *commit-request* or distribution phase the derivations in the distribution model are built in the Nix store on the computer of the coordinator. Once all services have been built, the distribution function decides what service to distribute to which target computers by comparing what new services are defined in the new distribution compared to the previous one. The services and all its intra-dependencies are transferred to the computers in the network through the Disnix interface. After receiving the services, the interface will import the services in the Nix store on the target computer.

If all steps in the commit-request phase succeed then the *commit* or transition phase will start. In this phase each cohort acquires an exclusive lock so that no other process can modify the target profile (e.g. /nix/var/nix/profiles/default) on the target computer. All obsolete services are uninstalled from, and new services are installed into the profiles of the target computers. Because the profiles are garbage collector roots, the services will not be garbage collected. If the commit phase succeeds, each cohort releases the lock.

If the commit-request phase fails then there is nothing we have to do. It is not necessary to undo the changes, because there are no files overwritten due to the non-destructive model of the Nix package manager. The closures that are transferred to the target computers are not activated in the profiles. The transferred services do not have a garbage collector root, so they are still garbage and will be deleted by the garbage collector.

Of course, merely installing new versions of each service is not enough to achieve atomicity. We must also *block* access to services while the commit phase is in progress. This can be done by wrapping services in a proxy that does the following. In the commit-request phase, the proxy "drains" current connections, i.e., it waits for current connections to the old version of the wrapped service to finish. Once all connections are finished, it acknowledges the commit request. Second, once the commit request has been received, it blocks new incoming connections (i.e., connections are accepted, but are kept inactive). Thus, the commit phase can only start whenever no connections to old versions of services exist. In the commit phase, the old version of the service is stopped, the new version is started, and the blocked connections are unblocked and forwarded to the new version of the service. From this point, all connections will be to the new versions of the services in the system. There is no time window in which one can simultaneously reach the old version of some service *and* the new version of another.

6. Distributed profiles

Nix supports a static and a dynamic mechanism for binding intra-dependencies. Dynamic binding is supported by means of Nix profiles. Essentially, Nix profiles are lookup tables which map file names to Nix store locations (see Figure 1). They can be automatically upgraded and downgraded, and they enable dynamic binding of intra-dependencies. Consequently, a user does not need to remember the store location of a program `HelloWorldService` herself, but by having `nix-profile/bin` in her search path, typing `HelloWorldService` will map to the proper Nix store location of the version of HelloWorldService that is active in the current profile.

Static binding of intra-dependencies is supported by means of Nix expressions. In this case, our example program `HelloWorldService` which has an inter-dependency on `HelloService` receives a reference to the Nix store location of `HelloService` from Nix at build-time. This reference cannot be changed afterward. Only by rebuilding `HelloWorldService` a different binding to `HelloService` can be realized, but this will result in another instance of `HelloWorldService` in the Nix store.

Static binding through Nix expressions is also supported for *inter-dependencies*. The locations of services can be hard-coded in executables or be stored in configuration files. Whenever a dependency changes, Nix ensures that the corresponding bindings are updated accordingly, while the transaction mechanism of Disnix ensures that all required upgrades to the distributed environment are performed together as an atomic upgrade action. A disadvantage of static binding is that even if only the location of a service changes, all services that transitively depend on it will have to be (partially) rebuild and redeployed.

Alternatively, we extend the concept of Nix profiles to distributed profiles to also support dynamic binding. A distributed profile is similar to a Nix profile in that it maps names to locations. In contrast to Nix profiles, names and locations correspond to service names and network locations, respectively, rather than to file names and Nix store locations. Actually, these mappings correspond directly to a distribution model (see Figure 5). A special lookup service provides remote access to a distributed profile. A distribution model thus serves as input for Disnix to realize a distributed deployment, and as distributed profile to dynamically bind services to their locations. Like Nix profiles, updating and downgrading of distributed profiles are atomic actions. In contrast to static binding, there is no need to update all transitive dependent services when the location of a service changes, only the corresponding mapping in the distributed profile needs to be updated.

7. Concluding Remarks

In this paper we demonstrated that we can extend the Nix approach of upgrading single computers to upgrading distributed systems. This is done by introducing additional models and tools that use the Nix primitives of software deployment which form the *Disnix* deployment system. We also demonstrated that many analogies of Nix can be mapped to a distributed variant of Nix, called Disnix.

Currently, the distribution of services to computers in the network is static. We have to specify to which computer a service should be distributed. We are currently developing a more dynamic approach by extending the Disnix deployment system with a tool which generates a distribution expression dynamically based on Quality of Service models. With this solution we can support load balancing and we no longer have to provide a distribution model explicitly.

We assume that we only need to remove and install services in the profiles of the target computers in order to activate and deactivate them. For some services this process is sufficient. For instance Apache Axis2 supports hot deployment (Apache Software Foundation 2008). Some services however need to be started and stopped explicitly. There are even services that run inside containers, for instance Enterprise Java Beans (EJB) (Dearle 2007). To activate these services we need to restart the container. Currently, the only way to activate and deactivate processes explicitly is by calling a remote process through the Disnix interface.

This research is supported in part by NWO-JACQUARD project PDS: Pull Deployment of Services.

References

Apache Software Foundation. Apache Axis2 installation guide, 2008. URL http://ws.apache.org/axis2/1_4/installationguide.html.

Alan Dearle. Software deployment, past, present and future. In *FOSE '07: 2007 Future of Software Engineering*, pages 269–284, Washington, DC, USA, 2007. IEEE Computer Society.

Eelco Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Faculty of Science, Utrecht University, The Netherlands, January 2006.

Eelco Dolstra and Andres Löh. NixOS: A purely functional Linux distribution. In *ICFP 2008: 13th ACM SIGPLAN Intl. Conf. on Functional Programming*. ACM Press, September 2008.

Eelco Dolstra, Eelco Visser, and Merijn de Jonge. Imposing a memory management discipline on software deployment. In *Proc. 26th Intl. Conf. on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.

Eric Foster-Johnson. *Red Hat RPM Guide*. John Wiley & Sons, 2003. Also at http://fedora.redhat.com/docs/drafts/rpm-guide-en/.

K. Nadiminti, M. Dias De Assuncao, and R. Buyya. Distributed systems and recent innovations: Challenges and benefits. *InfoNet Magazine*, 16(3):1–5, 2006.

Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. In *Concurrency control and reliability in distributed systems*, pages 295–317, New York, NY, USA, 1987. Van Nostrand Reinhold Co.