Automated Deployment of a Heterogeneous Service-Oriented System

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

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

***Abstract*—Deployment of a service-oriented system in a network of machines is often complex and labourious. In many cases components implementing a service have to be built from source code for the right target platform, transferred to the right machines with the right capabilities and activated in the right order. Upgrading a running system is even more difficult as this may break the running system and cannot be performed atomically. Many approaches that deal with the complexity of a distributed deployment process only support certain types of components or specific environments, while general solutions lack certain desirable non-functional properties, such as atomic upgrading. This paper shows *Disnix*, a deployment tool which allows developers and administrators to reliably deploy, upgrade and roll back a service-oriented system consisting of various types of components in a heterogeneous environment from declarative specifications.**

I. Introduction

The service-oriented computing (SOC) paradigm is nowadays a very popular way to rapidly develop, low-cost, interoperable, evolvable, and massively distributed applications [1]. The key in realizing this vision is the Service Oriented Architecture (SOA) in which a system is decoupled into "services", which are autonomous, platform-independent entities that can be described, published, discovered, and loosely coupled and perform functions ranging from answering simple requests to sophisticated business processes.

While a service-oriented system provides all kinds of advantages such as exposing a system to the Internet infrastructure, deploying it is often very labourious and error-prone because many deployment steps are performed manually and in an ad-hoc fashion. Upgrading is even more complex. Replacing components may break the system, and upgrading is not an atomic operation; i.e. while upgrading it is observable from the outside that the system is changing. This introduces all kinds of problems to end users, such as error messages or inaccessible features.

Because the software deployment process of a service-oriented system in a network is very hard, this is also a major obstacle in reaching their full potential. For instance, a system administrator might want to run every web service of a system on a separate machine (say, for privacy reasons), but refrain from doing so because it would take too much deployment effort. Similarly, actions such as setting up a test environment that faithfully reproduces the production environment may be very expensive.

Existing research in dealing with the complexity of the software deployment process in distributed environments is largely very context-specific; e.g. designed for a particular class of components [2], [3] or specific environments (such as Grid Computing [4]). Other approaches illustrate the use of component models and languages [5], [6] which can be used to make deployment easier. While existing research approaches provide useful features to reduce the complexity of a particular software deployment process and also support various non-functional aspects, few of them deal with *complete* heterogeneous systems consisting of multiple platforms, various types of components and *complete* dependencies, i.e. services and *all* their dependencies including their infrastructure, such as database back-ends of which service-oriented systems may be composed.

The contribution of this paper is a deployment tool, *Disnix*, which is used to *model* the components of a system including *all* its dependencies, along with the machines in the heterogeneous network in which the system is to be deployed. These are then used to *automatically* install or upgrade the complete system efficiently and reliably, to support various types of services and protocols by using an extensible approach, and to *upgrade* or rollback the system almost atomically.

As a case study, we used Disnix to automate the deployment of SDS2, a SOA-based system for asset tracking and utilization services in hospital environments developed at Philips Research.

II. Motivation

In this paper we will use as a case study a SOA-based system developed at Philips Research, called the Service Development Support System (SDS2) [7]. SDS2 is a platform that provides data abstractions over huge data sets produced by medical equipment in a hospital environment.

Background: Medical devices produce very large amounts of data such as status logs, maintenance logs, patient alarms, images, and measurements. This data often has a poorly-defined structure, e.g. logfiles. In SDS2 web

services provide the role of transformers, creating abstraction layers for the data sets stored in the data repositories. By combining transformers, they turn the data sets into useful and concrete information with a well-defined interface. Data abstractions, such as workflow analysis results, can be presented by one of the web application front-ends to the stakeholders.

The services of SDS2 may be distributed across several locations. For instance, the web service providing access to the log records should be located within the hospital environment, since medical data may not be stored outside the hospital for privacy reasons. The web service that performs certain analysis jobs over data sets may be located within the Philips Enterprise environment, since Philips has a more powerful infrastructure to perform analysis tasks and may want to use analysis results for other purposes.

Implementation: All the data that is produced by the devices are stored in MySQL databases. Events that are produced by medical devices are broadcasted through one or more Ejabberd servers. All the web service components are implemented in the Java Programming language using the Apache Axis2 library. For hosting the web application components Apache Tomcat is used. The web application front-ends are implemented using the Google Web Toolkit.

Deployment process: Deploying the SDS2 platform is a labourious process. First, a global configuration file needs to be created, which contains URLs of all the services of which SDS2 consists. Second, all the platform components have to be built and packaged from source code (Apache Maven is used for this). For every machine that provides data storage, MySQL databases have to be installed and configured. For every Ejabberd server, user accounts have to be created and configured through a web interface. Finally, the platform components have to be transferred to the right machines in the network and activated. This process has several disadvantages. Many steps are performed manually, and are therefore time-consuming and subject to errors. This complexity is proportional to the number of target machines in the network.

Moreover, some steps have to be performed in the *right order*; e.g. a web service that provides access to data in a MySQL database, requires that the database instance is started before the web service starts. Performing these tasks in the wrong order may bring the web service in an inconsistent state, e.g. due to failing database connections that are not automatically restored.

Upgrading is difficult and expensive: when changing the location of a service, the global configuration file needs to be adapted and all the dependent services need to be updated. Furthermore, it is not always clear what the dependencies of a service are and how we can upgrade them reliably and efficiently, i.e. we only want to replace the necessary parts without breaking the entire system. Upgrading is also not *atomic*; when changing parts of the system it may be observable by end users that the system is changing, as the web front-ends may return incorrect results.

In *heterogeneous* networks (i.e. networks consisting of systems with various architectures) the deployment process is even more complex, because we have to compile and test components for multiple platforms. Finally, since the deployment process is so expensive, it is also expensive to deploy the platform in a test environment and perform test-cases on them.

The technology used to implement SDS2 is very common in realizing service-oriented systems. The problems above are also applicable to other systems using the same or similar technology.

To deal with the complexity of the software deployment process of SDS2 and similar systems, we require a solution that *automatically* deploys a service-oriented system in an environment taking dependencies into account. Performing the deployment process automatically is usually less time-consuming and error-prone. Moreover, with a solution that takes all dependencies into account we never have a breaking system due to missing dependencies and we can also deploy the dependencies in the right order derived from the dependency graph. We also want to *efficiently* upgrade a service-oriented system by only replacing the changed parts and taking the dependencies into account so that all dependencies are always present and deployed in the right order.

In order to automate the installation and upgrade, we have to capture the services of a system and the infrastructure in *declarative specifications*. Using these specifications, we can derive the deployment steps of the system, and reproduce a previous deployment scenario in an environment of choice, such as a test environment.

Furthermore, we require features to make the deployment process *atomic*, so that we never end up with an inconsistent system in case of a failure and can guarantee that it is not observable to end users that the system is changing.

Since the underlying implementation of a service could be developed for various platforms with various technologies, we should be able to build the service for *multiple platforms* and *integrate* with the existing environment.

While there are solutions available that deal with these requirements, few have a notion of *complete* dependencies, and most are only applicable within a certain class of component types or environments and lack desirable non-functional properties such as atomic upgrades. Therefore, a new approach is required.

III. Disnix

To solve the complexities of the deployment process of distributed systems such as SDS2, we designed *Disnix* [8] (http://nixos.org/disnix), a distributed deployment extension to the Nix deployment system [9], [10].

Nix is a package manager which stores components in isolation from each other and provides a purely functional

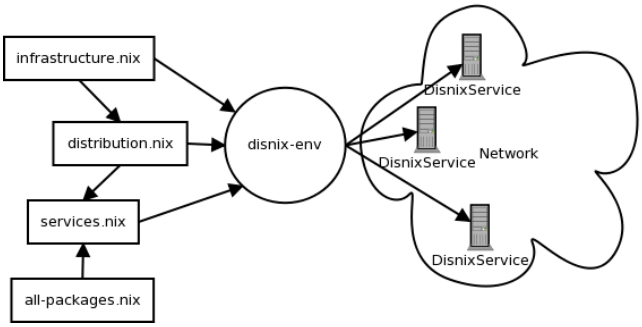

Figure 1. Overview of Disnix

language to specify build actions. Nix only deals with *intra-dependencies*, which are either run-time or build-time dependencies residing on the same machine. Disnix provides additional features on top of Nix to deal with distributed systems, including management of *inter-dependencies*, which are run-time dependencies between components residing on different machines. In this chapter we give an overview of Disnix and show how it is used to automate the software deployment process of SDS2.

A. Overview

Figure 1 shows an overview of the Disnix system. It consists of the disnix-env tool that takes three models as input. The *services model* describes the services of which the system is composed, their properties and their inter-dependencies. Usually this specification is written by the developers of the system. This model also includes a reference to all-packages.nix, a model in which the build functions and intra-dependencies of the services are specified.

The *infrastructure model* describes the target machines in the network on which the services can be deployed. Usually this specification is written by system administrators or can be generated by using a network discovery service.

The *distribution model* maps the services to machines in the network. Usually this specification is written by system administrators, or generated automatically.

The described models are implemented in the *Nix expression language*, a simple purely functional language used to describe component build actions. For example, to deploy SDS2, the developers and administrators must write instances of the above models (shown below) that describe the SDS2 services and the machines on which SDS2 is to be deployed. The following command then suffices to deploy SDS2:

```
$ disnix-env -s services.nix -i infrastructure.nix \
    -d distribution.nix
```

This command *builds* all components of SDS2 from source code for the intented target platforms, including all their intra-dependencies, *transfers* them to the selected target machines, and *activates* all services in the right order.

```
{javaenv, config, SDS2Util}: 1
{mobileeventlogs}: 2

let
  jdbcURL = "jdbc:mysql://"+
    mobileeventlogs.target.hostname+":"+
    toString mobileeventlogs.target.mysqlPort+"/"+
    mobileeventlogs.name; in
javaenv.createTomcatWebApplication rec { 3
  name = "MELogService";
  contextXML = '' 4
    <Context>
      <Resource
        name="jdbc/sds2/mobileeventlogs" auth="Container"
        type="javax.sql.DataSource"
        username="${mobileeventlogs.target.mysqlUsername}"
        password="${mobileeventlogs.target.mysqlPassword}"
        url="${jdbcURL}" />
    </Context>'';
  webapp = javaenv.buildWebService rec { 5
    inherit name;
    src = ../../../WebServices/MELogService; 6
    baseDir = "src/main/java";
    wsdlFile = "MELogService.wsdl";
    libs = [ config SDS2Util ]; 7
  };
}
```

Figure 2. Build expression of MELogService

To upgrade SDS2 or change its configuration, the user modifies the models, and runs disnix-env again. Disnix will rebuild and transfer any components that have changed. After the transfer phase, a *transition* phase is started in which obsolete services that are no longer in use are deactivated and new services are activated. (In this phase only the changed parts of the system are updated taking the inter-dependencies and their activation order into account.) If the activation of a particular service fails, a *rollback* is performed in which the old configuration is restored.

Since the machines in the network could be of a different type than the machine on which the deployment tool is running, the coordinator might have to delegate the build to a machine capable of building the service for the target platform (e.g. an i686-linux machine cannot compile a component for an i686-freebsd system). When there is no dedicated build machine available for this job, Disnix also provides the option to perform the build on the selected target machine in the network. Disnix performs actions on remote machines (such as transferring, building and activating) through a web service, the *DisnixService*, that must be installed on every target machine.

B. Building a service

Since Disnix is built around Nix, we need to specify for every component including its dependencies how it should be derived from source code.

Figure 2 shows a Nix expression for the MELogService component of SDS2. The MELogService is an Apache Tomcat web application containing a web service archive built from Java source code, providing an interface to log records stored in a MySQL database, which can reside on a

```
{system, distribution}: 8

rec {
 MELogService = import ../WebServices/MELogService { 9
  inherit javaenv config SDS2Util;
 };
 config = import ../libraries/config { 10
  inherit javaenv distribution;
 };
 javaenv = import ../tools/javaenv {
  inherit stdenv jdk;
 };
 jdk = ... { inherit stdenv; ... }
 stdenv = ... { inherit system; ... } 11
 mobileeventlogs = ...
 SDS2Util = ...
 ...
}
```

Figure 3. all-packages.nix: Intra-dependency composition

different machine. In order to allow the service to connect to the database server, the expression also generates a so-called *context XML file* that specifies connection settings of the MySQL database.

In order to build and configure this service, we need to derive the service from its source code, intra-dependencies (libraries, compilers) and inter-dependencies (the database backend). To specify what the intra-dependencies of a service are, we define the build expression in Figure 2 as two nested *functions*. The outer function 1 takes intra-dependencies as input parameters. In this case these are the packages config and SDS2Util, which are libraries of the SDS2 platform, and javaenv, which is used to compile and package Java source code. The inner function 2 takes the inter-dependencies as input parameters. Here, the service passed through the argument mobileeventlogs represents the MySQL database where data is stored.

The remainder of the expression consists of the function call 3 that composes an Apache Tomcat web application from a web service archive and a context XML file. The context XML file is generated by using the target property from the inter-dependency argument mobileeventlogs that provides the connection settings of the database server. The web service archive is derived in 5 by a function that compiles and packages the Java source code using the source code defined in 6 and libraries 7, which are provided by the intra-dependency arguments as inputs. By using a different build function other types of services can be built using the same dependency scheme.

Although this expression specifies how to derive the component from source code, we still cannot build this service directly. We have to *compose* the component by calling this expression with the expected function arguments. First we need to compose the service locally, by calling the function with the needed intra-dependency arguments; later, we provide the inter-dependency arguments as well.

Figure 3 shows a partial Nix expression that composes

```
{distribution, system}: 12

let pkgs = import ../top-level/all-packages.nix { 13
 inherit distribution system;
}; in
{
 mobileeventlogs = {
  name = "mobileeventlogs";
  pkg = pkgs.mobileeventlogs;
  type = "mysql-database";
 };
 MELogService = { 14
  name = "MELogService"; 15
  pkg = pkgs.MELogService; 16
  dependsOn = { 17
   inherit mobileeventlogs;
  };
  type = "tomcat-webapplication"; 18
 };
 SDS2AssetTracker = {
  name = "SDS2AssetTracker";
  pkg = pkgs.SDS2AssetTracker;
  dependsOn = {
   inherit MELogService ...;
  };
  type = "tomcat-webapplication";
 };
 ...
}
```

Figure 4. A partial services model for SDS2

the intra-dependencies of the SDS2 services. In 9 the MELogService expression in Figure 2 is imported and called with the right arguments (all the intra-dependencies of the MELogService, such as SDS2Util, are composed in this expression as well). The distribution parameter provided in 8 determines the distribution of services to machines in the network. It is used to compose the config component in 10, which is a registry library providing the locations of every web service. The system parameter defined in 8 specifies the platform for which the service has to be built using identifiers such as i686-linux and x86_64-freebsd. The system parameter is passed to 11, which is a component used virtually by every other component; e.g. passing i686-linux as system argument to stdenv will build every component for that type of platform.

C. Services model

Apart from specifying how each individual service should be derived from source code, intra-dependencies and inter-dependencies, we must also model what services constitute a system, how they are connected to each other (inter-dependency relationships) and how they can be activated or deactivated. This is captured in a services model, illustrated in Figure 4.

The expression is an attribute set in which every attribute represents a service with its properties, such as 14 defining the MELogService. 15 specifies an identifier for MELogService. 16 denotes the intra-dependency composition to be used, imported in 13 and defined in Figure 3. 17 refers to an attribute set in which each attribute points to a service in

```
{
 test1 = { [19]
  hostname = "test1.net";
  tomcatPort = 8080;
  mysqlUser = "user";
  mysqlPassword = "secret";
  mysqlPort = 3306;
  targetEPR = http://test1.net/.../DisnixService; [20]
  system = "i686-linux"; [21]
 };
 test2 = {
  hostname = "test2.net";
  tomcatPort = 8080;
  ...
  targetEPR = http://test2.net/.../DisnixService;
  system = "x86_64-linux";
 };
}
```

Figure 5. Infrastructure model for SDS2

the services model, each representing an inter-dependency of the MELogService. The attribute set provides a flexible way of composing services together, e.g. mobileeventlogs = othermobileeventlogs allows the user to compose a different inter-dependency relationship of the MELogService. [18] specifies what module has to be used for activation/deactivation of the service. Examples of types are: tomcat-webapplication, axis2-webservice, process and mysql-database. The Disnix interface will invoke the appropriate activation module on the target platform, e.g. the tomcat-webapplication will activate the given service in an Apache Tomcat web application container.

At [12] the distribution and system arguments are specified. The former argument is the distribution model and the latter agument is the system architecture of a target in the infrastructure model. These arguments are passed in [13] to the composition expression.

D. Infrastructure model

In order to deploy services in the network, we also have to specify what machines are available in the network, how they can be reached to perform deployment steps remotely, what architecture they have (so that services can built for that type of platform) and other relevant capabilities, such as authentication credentials and port numbers so that services can be activated or deactivated. This information is captured in the infrastructure model, illustrated in Figure 5.

The infrastructure model is an attribute set in which each attribute captures a system in the network with its relevant properties/capabilities, such as [19] specifying a machine called test1. Some attributes have reserved use such as [20] specifying the URL of the *DisnixService* and [21] specifying the system architecture so that services are built for that particular platform. The other attributes in [19] specify other properties such as how the MySQL server can be reached, required for deploying a database.

```
{infrastructure}: [22]

{
 mobileeventlogs = [ infrastructure.test1 ]; [23]
 MELogService = [ infrastructure.test2 ];
 SDS2AssetTracker = [
  infrastructure.test1 infrastructure.test2
 ]; [24]
 ...
}
```

Figure 6. Partial distribution model for SDS2

E. Distribution model

Finally, we have to specify to which machines in the network we want to distribute a specific service. This is defined in the distribution model illustrated in Figure 6.

The distribution model is an attribute set in which every attribute name represents a service in the service model and the attribute value is a list of machines from the infrastructure model, which is provided through a function argument in [22]. For instance, at [23] we specify that the mobileeventlogs database should be built, distributed and activated on machine test1. By specifying more machines in a list it is possible to deploy multiple redundant instances of the same service, for tasks such as load balancing. An example of this is [24] in which the SDS2AssetTracker is deployed on both test1 and test2.

IV. Implementation

In order to deploy a system from the models we described earlier, we need to *build* all the services that are mapped to a target machine in the distribution model for the right target, then *transfer* the services (and all its intra-dependencies) to the target machines and finally *activate* the services (and eventually deactivate obsolete services from the previous configuration). In this section we will explain how this process is implemented.

A. Building the services

The result of a build action defined in the Nix expressions are stored in a so called *Nix store*, a special directory in the file system, usually /nix/store. Each entry in the directory are components. A notable feature of the Nix store are the component names. The first part of the file name, e.g. y2ssvzcd86... is a SHA256 cryptographic hash of all inputs passed to the function that builds the component. Each component is stored in isolation, that is because there are no files that share the same name in the store. An example of a component in the Nix store is: /nix/store/y2ssvzcd86...-SDS2EventGenerator.

Nix can detect *runtime dependencies* from a build process, by scanning a component for the hash codes that uniquely identify components in the Nix store. The ELF header of the java executable contains a path to the standard C library which is: /nix/store/nqapqr5cyk...-glibc-2.9/lib/libc.so.6.

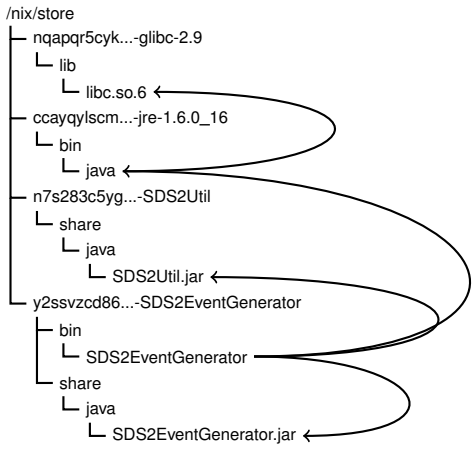

Figure 7. Runtime dependencies of the SDS2EventGenerator

For instance we know by scanning that a specific glibc component in the Nix store is a *runtime dependency* of java. Figure 7 shows the runtime dependencies of the SDS2EventGenerator.

Nix guarantees *complete deployment*, which requires that there are no missing dependencies. Thus we need to deploy *closures* of components under the "depends on" relation. If we want to deploy component X which depends on component Y, we also have to deploy component Y before we deploy component X. If component Y has dependencies we also have to deploy its dependencies first and so on.

B. Transferring closures

The second step is transferring the intra-dependency closures of the services to every machine in the distribution model. Since Nix has a purely functional deployment model we know that the build result of a component is always the same if the input parameters are the same, regardless on what machine the build is performed. By using this approach we can compare the store paths of the closure of the component to be transferred with the store paths present on the target system. Only the missing paths need to be transferred, making the transfer phase as efficient as possible.

C. Transition phase

The third step is the transition phase in which new services are activated and obsolete services are deactivated. During this phase the connection to the system can be blocked/queued so that end-users cannot observe that the system is changing.

First, all the service distributions in the old configuration are marked as active and all the services distributions in the new configuration not defined in the old configuration as inactive (if no previous configuration exists all service distributions will be marked as inactive).

In the next step all the service distributions that are no longer in the new configuration are deactivated, including all interdependent services to prevent breaking an inter-dependency relationship. If the deactivation of a particular service fails, the services previously deactivated are activated again. Finally, it will recursively activate all the service distributions (and its inter-dependencies) defined in the new configuration that are marked inactive. If a failure occurs the newly activated services are deactivated and the deactivated services are activated again.

Using this method to upgrade a distributed system gives us two benefits. By traversing the inter-dependency graph of a service we always have the inter-dependencies of a service activated before activating the service itself. This gives us no failing services due to breaking inter-dependency relationships. Moreover, this method only deactivates obsolete services and activates new services, which is more efficient that deploying a new configuration entirely from scratch.

D. Service activation and deactivation

Since services can have basically any form, Disnix provides *activation types* which can be connected to activation modules. In essence, an activation module is a process that takes 2 arguments, in which the former argument is either 'activate' or 'deactivate' and the latter is the Nix store path of the service that has to be activated/deactivated.

Moreover, an activation module often needs to know certain properties of the system, such as authentication credentials for a database or a port number on which a certain daemon is running. Therefore, Disnix passes all the properties defined in the infrastructure model as environment variables, so that it can be used by the activation module.

The activation module for an Apache Tomcat web application on Linux creates a symlink of the WAR file into the webapps/ directory of Tomcat, automatically triggering a hot deploy operation. On deactivation the symlink is removed, triggering the hot undeploy operation.

Similar activation modules are developed for other types, such as axis2-webservice which will hot deploy a web service in an Axis2 container, mysql-database which will initialise a MySQL database schema on startup or process which will start and kill a generic process. A developer or system administrator can also implement a custom activation module used for other types of services or use the wrapper activation module, which will invoke a wrapper process with a standard interface included in the service. (Observe that different platforms may require a different implementation of an activation module, e.g. activating a web service on Microsoft Windows consists of different steps to be performed than on UNIX based systems).

E. Atomic upgrading

To make the actual deployment process atomic we mapped concepts of the two-phase commit [11] algorithm

onto Nix primitives. 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. It consists of building the services from source code and transfering the services (and intra-dependencies) to the target machines in the network.

If all steps in the commit-request phase succeed then the *commit* or *transition phase* will start. In this phase all the obsolete services from the previous deployment state are deactivated and the services in the new distribution model are activated. During this phase access to the system can be blocked/queued so that users are not able to observe that the system is changing. After the transition phase is finished, the lock is released and the services and the new configuration are registered as used. Moreover, the deployment configuration is stored on the coordinator machine, so that it has a reference to the configuration for future upgrades.

If the commit-request phase fails then there is not much to be done. No files are overwritten due to the concept of unique filenames in the Nix store. The services that are transferred to the target computers are not activated yet and thus do not affect the running system. If the commit-phase fails, we roll back the newly activated services and reactivate the deactivated services from the old configuration.

V. Results

We modeled all the SDS2 components (databases, web services, batch processes and web application front-ends) to automatically deploy the platform in a small network of consisting of four 64-bit and 32-bit Linux machines. The initial deployment process (building the source code of all SDS2 components, transfering components and its dependencies and activating the services) took about 15 minutes. At Philips, this previously took hours in the semi-manual deployment process on a single machine. (Manually deploying an additional machine took almost the same amount of time and was usually too much effort.)

We were also able to upgrade (i.e. replacing and moving services) a running SDS2 system and to perform rollbacks. In this process only the changed services were updated and deactivated/activated in the right order, which only took a couple of seconds in most cases. In all the scenarios we tested no service ran into a consistent state due to breaking inter-dependency connections. The only minor issue we ran into was that during the upgrade phase, the web application in the user's browser (which is not under Disnix' control) was not refreshed, which may break certain features.

Although Disnix can activate databases, we currently cannot migrate them to other machines dynamically. Disnix activates a database by using a static representation (i.e. dump) but is not able to capture and transfer the state of a deployed database. This requires more investigation in dealing with mutable state.

VI. Related work

A number of approaches to distributed software deployment limit themselves to specific types of components, such as the BARK reconfiguration tool [2], which only supports the software deployment life-cycle of EJBs and supports atomic upgrading by changing the resource attached to a JNDI identifier; and [3], which implements a custom infrastructure on top of the JBoss application server to automatically deploy Java EE components.

In [12] an embedded software architecture is described that supports upgrading parts of the system as well as having multiple versions of a component next to each other. A disadvantage is that the underlying technology, such as the infrastructure and run-time system cannot be upgraded and that the deployment system depends on the technology used to implement the system.

Other approaches use component models and language extensions, such as [5] which proposes a conceptual component framework for dynamic configuration of distributed systems. In [6] the authors developed the GILGUL extension to the Java language for dynamic object replacement. Using such approaches requires the developers or system administrators to use a particular framework or language.

Approaches for specific environments also exist. GoDIET [4] is a utility for deployment of the DIET grid computing platform. It writes configuration files, stages the files to remote resources, provides an appropriately ordered and timed launch of components, and supports management and tear-down of the distributed platform. In [13] the Globus toolkit is described for performing 3 types of deployment scenarios using predefined types of components compositions in a grid computing environment. CODEWAN [14] is a Java-based platform designed for deployment of component-based applications in ad-hoc networks.

Finally, several generic approaches have been developed. The Software Dock is a distributed, agent-based deployment framework to support ongoing cooperation and negotiation among software producers and consumers [15]. It emphasises the delivery process of components from producer to consumer site, rather than *complete* deployment. In [16] a dependency-agnostic upgrade concept with running distributed systems is described in which the old and new configurations of a distributed system are deployed next to each other in isolated runtime environments. Separate middleware forwards requests to the new configuration at a certain point in time. TACOMA [17] uses agents to perform deployment steps and is built around RPM. This approach has limitations such as the inability to perform atomic upgrades or safely install multiple variants of components next to each other. Disnix overcomes these limitations. An earlier version of Disnix was described in [8], which only supported web services, did not support heterogeneous environments and used an implicit activation model.

In practice many software deployment processes are partially automated by manually composing several utilities together in scripts and use those to perform tasks. While this gives users some kind of specification and reproducibility, it cannot ensure properties such as correct deployment.

VII. Conclusion

We have shown *Disnix*, a distributed deployment extension to Nix, used to automatically deploy, upgrade and roll back a service-oriented system such as SDS2 consisting of components of various types in a network of machines running on different platforms. Because Disnix is extensible, takes inter-dependencies into account, and builds on the purely functional properties of Nix, we can safely upgrade only necessary parts, and deactivate and activate the required services of a system, making the upgrade process efficient and reliable. Although we deployed SDS2 in a small heterogeneous network, we need to run experiments in larger networks and more research is needed to deal with mutable state and with upgrading web applications.

Acknowledgements: This research is supported by NWO-JACQUARD project 638.001.208, *PDS: Pull Deployment of Services*. We wish to thank the contributors and developers of NixOS and SDS2, in particular Merijn de Jonge, who also contributed significantly to the development of Disnix.

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, vol. 40, pp. 38–45, 2007.

[2] M. J. Rutherford, K. M. Anderson, A. Carzaniga, D. Heimbigner, and A. L. Wolf, "Reconfiguration in the Enterprise JavaBean Component Model," in *CD '02: Proc. of the IFIP/ACM Working Conf. on Component Deployment*. Springer-Verlag, 2002, pp. 67–81.

[3] A. Akkerman, A. Totok, and V. Karamcheti, "Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments," in *CD '05: Proc. of the 3rd Working Conf. on Component Deployment*. Springer-Verlag, 2005, pp. 17–32.

[4] E. Caron, P. K. Chouhan, and H. Dail, "GoDIET: A Deployment Tool for Distributed Middleware on Grid 5000," Laboratoire de l'Informatique du Parallélisme (LIP), Tech. Rep. RR-5886, Apr. 2006. [Online]. Available: http://www.inria.fr/rrrt/rr-5886.html

[5] X. Chen and M. Simons, "A component framework for dynamic reconfiguration of distributed systems," in *CD '02: Proc. of the IFIP/ACM Working Conf. on Component Deployment*. Springer-Verlag, 2002, pp. 82–96.

[6] P. Costanza, "Dynamic Replacement of Active Objects in the Gilgul Programming Language," in *CD '02: Proc. of the IFIP/ACM Working Conf. on Component Deployment*. Springer-Verlag, 2002, pp. 125–140.

[7] M. de Jonge, W. van der Linden, and R. Willems, "eServices for Hospital Equipment," in *5th Intl. Conf. on Service-Oriented Computing (ICSOC 2007)*, B. Krämer, K.-J. Lin, and P. Narasimhan, Eds., Sep. 2007, pp. 391 – 397.

[8] 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.

[9] 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 Computer Society, May 2004, pp. 583–592.

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

[11] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," in *Concurrency control and reliability in distributed systems*. New York, NY, USA: Van Nostrand Reinhold Co., 1987, pp. 295–317.

[12] M. Mikic-Rakic and N. Medvidovic, "Architecture-level support for software component deployment in resource constrained environments," in *CD '02: Proc. of the IFIP/ACM Working Conf. on Component Deployment*. Springer-Verlag, 2002, pp. 31–50.

[13] G. v. Laszewski, E. Blau, M. Bletzinger, J. Gawor, P. Lane, S. Martin, and M. Russell, "Software, component, and service deployment in computational grids," in *CD '02: Proc. of the IFIP/ACM Working Conf. on Component Deployment*. Springer-Verlag, 2002, pp. 244–256.

[14] H. Roussain and F. Guidec, "Cooperative component-based software deployment in wireless ad hoc networks," in *CD '05: Proc. of the 3rd Working Conf. on Component Deployment*. Springer-Verlag, 2005, pp. 1–15.

[15] R. S. Hall, D. Heimbigner, and A. L. Wolf, "A cooperative approach to support software deployment using the software dock," in *ICSE '99: Proc. of the 21st Intl. Conf. on Software Engineering*. New York, NY, USA: ACM, 1999, pp. 174–183.

[16] T. Dumitras, J. Tan, Z. Gho, and P. Narasimhan, "No more HotDependencies: toward dependency-agnostic online upgrades in distributed systems," in *HotDep'07: Proc. of the 3rd workshop on Hot Topics in System Dependability*. Berkeley, CA, USA: USENIX Association, 2007, p. 14.

[17] N. P. Sudmann and D. Johansen, "Software deployment using mobile agents," in *CD '02: Proc. of the IFIP/ACM Working Conf. on Component Deployment*. Springer-Verlag, 2002, pp. 97–107.