Disnix: A toolset for distributed deployment

Sander van der Burg, Eelco Dolstra

Delft University of Technology, The Netherlands

Abstract

The process of deploying a distributed system in a network of machines is often very complex, labourious and time-consuming, while it is hard to guarantee that the system will work as expected and that specific non-functional deployment requirements from the domain are supported. In this paper we describe the *Disnix* toolset, which provides system administrators or developers with automatic deployment of a distributed system in a network of machines from declarative specifications and offers properties such as complete dependencies, atomic upgrades and rollbacks to make this process efficient and reliable. Disnix has an extensible architecture, allowing the integration of custom modules to make the deployment more convenient and suitable for the domain in which the system is to be used. Disnix has been under development for almost four years and has been applied to several types of distributed systems, including an industrial case study.

Keywords: Software deployment, Distributed systems, Service-oriented systems

1. 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]. Service-oriented systems are composed of distributable components (also called *services*) using various technologies, such as web services, databases, web applications and batch processes, working together to achieve a common goal.

A service-oriented system eventually must be made available for use (or *deployed*) in a production or test environment by system administrators or developers. The software deployment process of a service-oriented system, which consists of steps such as building components from source code, transferring the components from producer side to consumer side and activating the system, is usually a very difficult, expensive and time-consuming process. In many cases, the process is quite labourious, it is hard to guarantee that a system will work as expected, the system may break completely and it is a process which cannot be performed atomically (i.e. during upgrading a user may observe that the system is changing). Apart from the deployment of service components of a service-oriented system, the underlying infrastructure, such as the machines in the network must be installed and configured with operating systems and system services.

Email addresses: s.vanderburg@tudelft.nl (Sander van der Burg), e.dolstra@tudelft.nl (Eelco Dolstra)

In order to reduce these complexities in the software deployment cycle, several solutions have been developed. Many of these tools are specifically designed for component technologies such as Enterprise Java Beans [2], or designed for environments such as Grid Computing [3]. Furthermore, some general approaches have been developed, such as [4].

While these tools are useful for their purpose, some distributed systems are not homogeneous (i.e. not implemented by a single component technology or for a single platform or architecture). For instance, many Java EE systems are composed of components implemented in the Java programming language, but also of non-Java components, such as databases, which cannot be completely and reliably deployed by existing deployment tools.

Because deployment of service-oriented systems is so hard, such processes are not performed frequently. Not being able to frequently deploy a system has many drawbacks, such as the inability to test the behaviour of the system in a distributed environment closely resembling the production environment during development.

For these reasons, we have developed Disnix, a toolset built on top of the Nix package manager [5, 6] that automatically deploys a distributed system in a network of machines. In contrast to other tools for distributed deployment, Disnix is designed to support the deployment of complete *heterogeneous* systems, which are composed of components using various technologies on various platforms/architectures *and* Disnix offers features to make the deployment process safe and reliable, such as *models* to describe services and machines in the network to *automatically* perform deployment steps, *complete* dependencies, *atomic* upgrades and rollbacks, *garbage collection* to safely remove obsolete components and a *modular* architecture to integrate with the domain in which the system is to be used. We have applied Disnix to several use cases, such as SDS2, a service-oriented system designed for hospital environments [7].

In previous work, we have developed a basic prototype of Disnix [8], to demonstrate the concept of atomic upgrading in distributed environments. Later, we have redeveloped Disnix to make the deployment of SDS2 possible [9], supporting stricter models and multiple types of components.

In this paper, we have extended Disnix with more features to support a wide range of service-oriented systems in heterogeneous networks and we have implemented a complementary extension called *DisnixOS* to support infrastructure deployment. Furthermore, we explain the concepts underlying Disnix, as well as its architecture, which consists of primitives to perform deployment steps in a generic manner and is *extensible* to make the deployment process suited to the target domain in which the system is to be used.

The paper is organised as follows. We first introduce a motivating example in Section 2. We then provide some background information in Section 3. Section 4 briefly discusses the purpose and concepts of the Disnix toolset. Section 5 explains the models which Disnix uses to perform deployment. In Section 6, we explain the architecture of Disnix, such as its tools, libraries and design choices. In Section 7, we describe DisnixOS, an extension providing infrastructure deployment. Section 8 explains how Disnix can be used and outlines some example cases. Section 9 explains some related work. Finally, Section 10 concludes and outlines future work.

2. Motivating example

Figure 1 shows the architecture of the StaffTracker system, a simple distributed system developed as one of the example cases for Disnix that can be publicly used. The StaffTracker is a

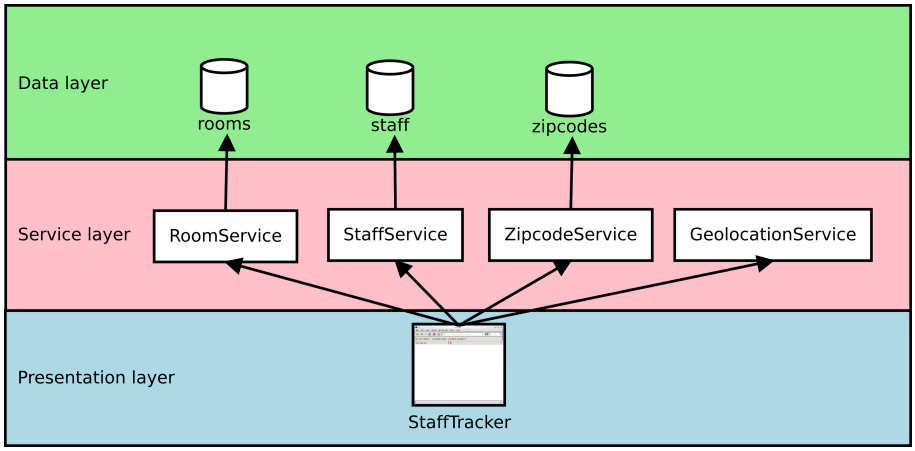

Figure 1: Architecture of the StaffTracker example system. Objects represent distributable components. Arrows represent dependency relationships between components.

system to manage staff of a university, and uses several web services to look up the location of a staff member from an IP address, a staff member's zipcode from a room number, and an address from a zipcode.

The architecture of the StaffTracker consists of three layers. The *data layer* contains components that actually store data in MySQL databases. The *service layer* contains web services, providing an interface to the data stored in the databases (the GeolocationService uses GeoIP [10] to track the country of origin from an IP address). The *presentation layer* contains the StaffTracker web application front-end, which can be used by end users to manage staff of the university. This web application invokes the web services in the middle layer to retrieve records or to modify the data stored in the databases.

All the components in Figure 1 are *distributable* components (which we call *services* later on), i.e. they can be distributed across various machines in a network. For instance, the zipcode database may be located on a different machine as the StaffTracker web application.

To deploy the StaffTracker system, a system administrator or developer must install the databases on one or more MySQL servers. Moreover, all the web services and web application front-end must be built from source code, packaged, and activated on one or more Apache Tomcat servers. Apart from the service components, also the *infrastructure* components such as the Apache Tomcat server and MySQL DBMS must be installed and properly configured, before any service component can be deployed.

There are various reasons why a developer or system administrator wants to deploy components from Figure 1 on multiple machines, rather than a single machine. For instance, a system administrator may want to deploy the StaffTracker web application on a machine which is publicly accessible on the Internet, while the data, such as the zipcodes, must be restricted from public access since they are privacy-sensitive. Moreover, each database may need to be deployed on a separate machine, since a single machine may not have enough disk space available to store all the data sets. Furthermore, multiple redundant instances of the same component may have to be deployed in a network, to offer load-balancing or failover.

Because of all these constraints, the deployment process of a system such as the StaffTracker

becomes very complex. This complexity increases for every additional machine on which components of the system are deployed. Most of the existing research and solutions are able to make the deployment easier for specific components and environments, but cannot manage both a heterogeneous system such as the StaffTracker on multiple platforms *and* offer properties such as dependency completeness and atomic upgrades.

3. Background

Disnix is a toolset used to perform distributed deployment tasks and is built on top of Nix [5, 6], a package manager with some distinct features compared to conventional package managers (e.g. RPM [11]) to make deployment safe and reliable, such as model-driven deployment, complete dependencies, atomic upgrades and rollbacks and a garbage collector.

Nix stores packages in isolation from each other in a directory called the *Nix store*. Every package has a special file name such as /nix/store/5am52fmn...-firefox-3.6.6, in which the first part 5am52fmn... is a cryptographic hash derived from all the inputs (e.g. libraries, compilers, build scripts) used to build the component. If a user decides to build the component with, say, a different compiler, this will result in a different hash and thus a different file name in the Nix store. This approach makes it safe to store multiple versions and variants of the same component next to each other, because no components share the same name.

Since every component is stored in isolation in the Nix store rather than a global directory such as /usr/lib, we have stricter guarantees that its dependencies are correct and complete. With conventional package managers the fact that a package builds successfully does not guarantee that the dependency specification is correct, since dependencies are stored in global locations and may still be accidentally found. In Nix, all packages reside in the Nix store and must therefore be explicitly specified. This guarantees that if a package builds correctly on one machine, it will build on other machines of the same type as well.

Nix supports atomic upgrades and rollbacks, because components are stored safely next to each other and are never overwritten or automatically removed. Thus there is no time window in which a package has some files from the old version and some files from the new version (which would be bad because a program may crash if it is started during that period).

Moreover, Nix uses a purely functional domain-specific language called the Nix expression language to specify build actions. This makes building a package deterministic and reproducible. A garbage collector is included to safely remove packages that are no longer in use.

Because of these features, Nix is a very good basis to build a distributed deployment tool to make the deployment process of a distributed system, such as the StaffTracker efficient, reliable and atomic. However, Nix only deals with *intra-dependencies*, which are either run-time or build-time dependencies residing on the same machine. In order to deploy a distributed system into a network of machines additional features are provided by Disnix. Most importantly, these include models to describe machines in the network, and management of *inter-dependencies* – the run-time dependencies between components residing on different machines.

4. Overview

Figure 2 shows an overview of the disnix-env tool, which is used to perform the complete deployment process of a distributed system automatically from models. Three types of models (specified in the Nix expression language) are required as input parameters to automate the deployment process, each capturing a specific concern. All the remote deployment operations are

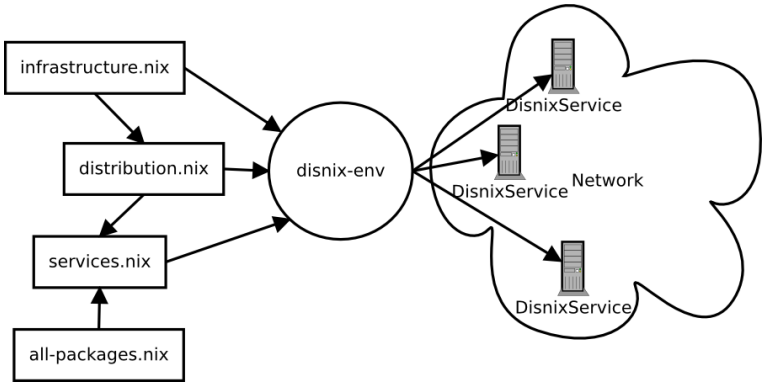

Figure 2: Overview of the disnix-env tool

performed by the *DisnixService*, a service that exposes deployment operations through a custom RPC protocol and must be installed on every machine in the network. All the deployment operations are initiated from a single machine, called the *coordinator*.

A *services* model is used to specify the available distributable components, how to build them, their *inter-dependencies*, and their *types*. The latter determine how a service is to be activated or deactivated. 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.

An *infrastructure* model is used to specify what machines are available in the network, how they can be reached in order to perform remote deployment operations, and other relevant capabilities and properties. A *distribution* model is used to specify a mapping of services to machines in the network.

By writing instances of the three models mentioned above, the user can deploy a complete system in a network of machines, by running:

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

This command performs all the deployment steps to make the system available for use; i.e. *building* all the services defined in the services model from source code, *transferring* the services and all their intra-dependencies to the machines in the network and finally *activating* them in the right order derived from the dependency graph. By adapting the model instances above and by running disnix-env again, an *upgrade* is performed instead of a full installation. In this phase, only the changed components are built from source code and transferred to the target machines and only obsolete services are deactivated and new services activated in the right order of activation. In case of a failure, Disnix will perform a *rollback*, which will bring the system back in its previous deployment state.

A machine in the network may be of a different architecture than the machine distributing the services and thus incapable of building a service for that particular platform. Therefore, Disnix also provides the option to build services on the target machines in the network, instead of the coordinator machine.

Disnix has several important features. Since it is built on top of the Nix package manager, it will inherit properties to ensure that all intra-dependencies are always complete, components

```
{ intraDep1, intraDep2 }: 1
{ interDep1, interDep2 }: 2

buildFunction { ... } 3
```

Figure 3: Structure of a Disnix expression

can be stored safely next to each other, and because Nix never overwrites files, an upgrade will not break the system in case of a failure. Moreover, Disnix can also be used to block or queue connections during the upgrade phase so that users will not be able to observe that the system is changing to make the upgrade process fully atomic.

Usually, there is only one coordinator machine involved in deployment processes. However, it is also possible to have multiple coordinators, which can either deploy the same configuration or a different configuration (which is made distinct by a profile identifier). Because of the purely functional deployment model of Nix, we can safely deploy multiple variants of packages next to each other without interference. However, at runtime it may be possible that a process from one configuration may conflict with another process from a different configuration. Disnix does not keep track of these runtime conflicts between different configurations, and in the case of a runtime error, it performs a rollback because the activation process fails. A system administrator has to manually check whether no conflicts occur. For reliability, it is not recommended to deploy multiple configurations in the same environment.

5. Models

In the previous section, we have explained that Disnix uses various models to perform the deployment of a distributed system. In this section, we explain how these models are constructed and what their purposes are.

5.1. Building a service

For every service defined in the architecture shown in Figure 1, we have to write a Disnix expression. The goal of a Disnix expression is to derive a service from source code and its build-time dependencies, such as a particular variant of a compiler and libraries. The build result is stored in a unique folder in the Nix store. Furthermore, the component must also be configured to be able to connect to its inter-dependencies, for example by generating a configuration file containing connection settings in a format that the component understands.

The structure of a Disnix expression is outlined in Figure 3. A Disnix expression is a nested function defined in the Nix expression language. The outer function header 1 defines the intra-dependencies arguments, which are build-time and run-time dependencies residing on the same system where the component is deployed. These dependencies could be libraries, compilers and configuration files. The inner-function header 2 defines the inter-dependency arguments, which are run-time dependencies on services which may be located on different machines in the network, such as web services or databases.

Every inter-dependency argument is a set of attributes, containing various properties of a service which are defined in the services model, shown later in Section 5.3. A special attribute is the targets property which contains a list of machines on which the inter-dependency is deployed

```
{ stdenv, apacheAnt, axis2 }: 4
{ zipcodes }: 5

let
  jdbcURL = "jdbc:mysql://"+zipcodes.target.hostname+":"+
    toString zipcodes.target.mysqlPort+"/"+zipcodes.name; 6

  contextXML = ''
    <Context>
      <Resource name="jdbc/ZipcodeDB" auth="Container" type="javax.sql.DataSource"
                maxActivate="100" maxIdle="30" maxWait="10000"
                username="${zipcodes.target.mysqlUsername}" password="${zipcodes.target.mysqlPassword}"
                driverClassName="com.mysql.jdbc.Driver"
                url="${jdbcURL}?autoReconnect=true" />
    </Context> 7
  '';
in
stdenv.mkDerivation { 8
  name = "ZipcodeService";
  src = ../../../../services/webservices/ZipcodeService;
  buildInputs = [ apacheAnt ];
  AXIS2_LIB = "${axis2}/lib";
  AXIS2_WEBAPP = "${axis2}/webapps/axis2";
  buildPhase = "ant generate.war";
  installPhase = ''
    ensureDir $out/conf/Catalina
    cat > $out/conf/Catalina/ZipcodeService.xml <<EOF 9
    ${contextXML}
    EOF
    ensureDir $out/webapps
    cp *.war $out/webapps
  '';
}
```

Figure 4: Disnix expression for the ZipcodeService component of the StaffTracker system

and their properties defined in the infrastructure model, shown later in Section 5.4. The target property points to the first targets element for convenience.

In the remainder of the Disnix expression 3, the component is built from source code, and the given intra-dependencies and inter-dependency parameters. In principle, any build tool can be invoked from a Disnix expression as long as it is pure. In the Nix packages repository, we have many types of build tools available, such as GNU Make, CMake, Apache Ant and SCons, which can be conveniently used by end-users.

Apart from compiling software from source code, it is also possible to deploy binary software by copying prebuilt binaries directly into the output folder. For some component types, such as ELF executables, patching is required, because normally these executables look into default locations, such as /usr/lib, to find their run-time dependencies, which does not work correctly because they are stored in isolated Nix store directories. We have developed a tool called PatchELF[1], which makes it possible to change the RPATH field of an ELF executable, so that the runtime dependencies of prebuilt binaries can be found.

Figure 4 shows a Disnix expression for a particular web service component of the StaffTracker system, called ZipcodeService, which provides access to records in the zipcode MySQL database, which may be located on a different machine.

[1] http://nixos.org/patchelf.html

```
{ system, pkgs }: 10

rec { 11

### Databases

  zipcodes = import ../pkgs/databases/zipcodes {
    inherit (pkgs) stdenv;
  };
  ...

### Web services + Clients

  ZipcodeService = import ../pkgs/webservices/ZipcodeService { 12
    inherit (pkgs) stdenv apacheAnt axis2;
  };
  ...

### Web applications

  StaffTracker = import ../pkgs/webapplications/StaffTracker {
    inherit (pkgs) stdenv apacheAnt axis2;
    inherit GeolocationServiceClient RoomServiceClient StaffServiceClient ZipcodeServiceClient;
  };
}
```

Figure 5: all-packages.nix: Partial intra-dependency composition of the StaffTracker system

This expression uses the following intra-dependency parameters 4: stdenv, an environment providing a collection of standard UNIX shell utilities, such as GNU Make and the GCC compiler. The apacheAnt parameter provides access to the Apache Ant tool. The axis2 parameter refers to Apache Axis2 library used to build web services. The zipcodes inter-dependency argument 5 refers to a MySQL database in which all the zipcodes are stored.

In the remainder of the Disnix expression, we generate a JDBC connection string 6 from the inter-dependency parameters so that the Java application can connect to the database. Furthermore, we call the stdenv.mkDerivation function 8 to build the service from source code and the given intra-dependencies. In order to allow the web service to connect to the database, we generate 7 and include 9 a so-called context XML file. This is a configuration file used by Apache Tomcat in order to configure web application specific settings, such as database connectivity settings.

5.2. Composing services

Although the expression in Figure 4 specifies how to derive the component from source code, we cannot build this service directly using the expression in Figure 4. We have to *compose* the service by calling this expression with the expected function arguments. First we need to compose the service locally, by calling the function with the required intra-dependency arguments; later, we provide the inter-dependency arguments as well.

Figure 5 shows a partial Nix expression that composes the intra-dependencies of the StaffTracker system. This composition expression is a function taking two arguments 10. The system argument is a system identifier, such as i686-linux defining a 32-bit Linux system, pkgs is a reference to the Nixpkgs collection, a repository containing 2500 packages which can be deployed by the Nix package manager and Disnix. In this repository, many expressions for common applications, libraries and tools can be found, such as Apache Ant, Apache Axis2, the Java

```
{ distribution, pkgs, system }:

let customPkgs = import ../top-level/all-packages.nix { inherit pkgs system; }; in 13
rec {
### Databases
  rooms = { 14
    name = "rooms";
    pkg = customPkgs.rooms;
    dependsOn = {};
    type = "mysql-database";
  };
  ...
### Web services
  RoomService = {
    name = "RoomService"; 15
    pkg = customPkgs.RoomService; 16
    dependsOn = { 17
      inherit rooms;
    };
    type = "tomcat-webapplication"; 18
  };
  ...
### Web applications
  StaffTracker = {
    name = "StaffTracker";
    pkg = customPkgs.StaffTracker;
    dependsOn = {
      inherit GeolocationService RoomService StaffService ZipcodeService;
    };
    type = "tomcat-webapplication";
  };
}
```

Figure 6: Partial services.nix model for the StaffTracker

Development Kit, GCC and GNU Make, for which we do not have to write expressions ourselves. This means that for the StaffTracker example, we only have to provide intra-dependency compositions for the 8 service components, shown earlier in Figure 9.

The rec keyword at 11 states that we have a mutually recursive attribute set in which attributes can refer to each other. This keyword is required, because we have to pass attributes as function parameters to every function imported in this file. At 12, the expression from Figure 4 is imported and called with the right intra-dependency arguments. All intra- and inter-dependencies are defined and composed in this model (such as the StaffTracker web front-end) or defined as packages in Nixpkgs (such as Apache Ant).

5.3. Services model

Figure 6 shows a partial services model for the StaffTracker system. This expression is a set of attributes, in which every attribute (such as 14) represents a distributable component (or service) from the architecture described earlier in Figure 1. Every attribute is itself an attribute set defining relevant properties of the service, such as the name of the service 15, a reference to the function that builds the service from source code 16, the inter-dependencies of the service 17 (which correspond to the arrows shown in Figure 1) and its type 18. The inter-dependencies defined at 17 are later augmented with targets and passed as inter-dependency arguments to the expression defined earlier in Figure 4. At 13, the intra-dependency composition expression shown in Figure 5 is imported.

```
{
  test1 = {
    hostname = "test1.testdomain.net";
    tomcatPort = 8080;
    system = "i686-linux";
  };

  test2 = { 19
    hostname = "test2.testdomain.net"; 20
    tomcatPort = 8080; 21
    mysqlPort = 3306;
    mysqlUsername = "user";
    mysqlPassword = "secret";
    system = "x86_64-linux"; 22
  };
}
```

Figure 7: An infrastructure.nix model for the StaffTracker

```
{ infrastructure }:

{
  GeolocationService = [ infrastructure.test1 ]; 23
  RoomService = [ infrastructure.test2 ];
  StaffService = [ infrastructure.test1 ];
  StaffTracker = [ infrastructure.test2 ];
  ZipcodeService = [ infrastructure.test1 infrastructure.test2 ]; 24
  rooms = [ infrastructure.test2 ];
  staff = [ infrastructure.test2 ];
  zipcodes = [ infrastructure.test2 ];
}
```

Figure 8: A distribution.nix model for the StaffTracker

5.4. Infrastructure model

Figure 7 shows an infrastructure model used to capture the machines in the network and their relevant properties and capabilities. This model is also an attribute set. Here, every attribute represents a system in the network, such as a machine called test2 [19]. For each machine, relevant properties are defined such as the architecture of the system [22], so that Disnix can build a service for the desired type of platform, and a hostname attribute [20] so that Disnix knows how to reach a target machine to perform remote deployment operations. The other attributes are optional and used to activate particular types of services. For instance, the tomcatPort attribute [21] specifies through which port the Apache Tomcat server can be reached. These attributes are specific to the activation types attached to the service defined in the services model.

5.5. Distribution model

A distribution model, shown in Figure 8, maps services to machines in the network. In this model, the name of each attribute corresponds to a service defined in the services model, while its value is a list of machines defined in the infrastructure model. For instance, [23] specifies that the GeolocationService must be deployed on the test1 machine defined in the infrastructure model. It is also possible to deploy multiple redundant instances of the same service by specifying multiple machines in a list, such as [24], which deploys the ZipcodeService on both test1 and test2.

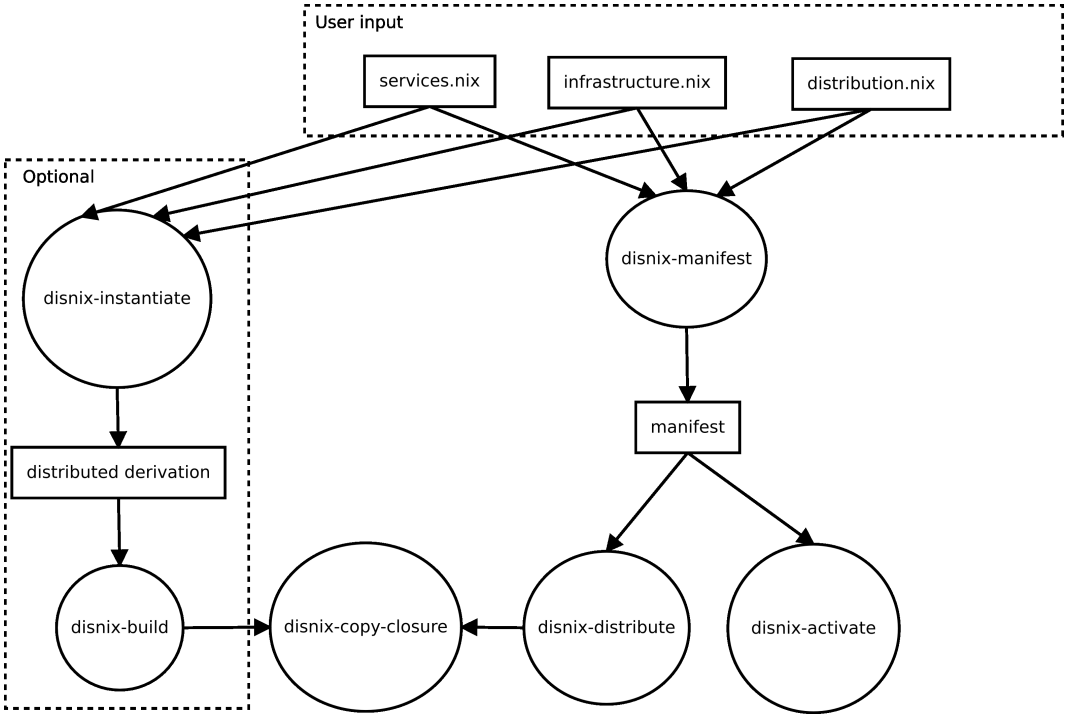

Figure 9: Architecture of disnix-env. Rectangles represent objects that are manipulated by the tools. The ovals represent a tool from the toolset performing a step in the deployment process.

6. Architecture

In order to deploy a system from the models we have 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). The disnix-env tool, which performs the complete deployment process, is composed of several tools each performing a step in the deployment process. The architecture is shown in Figure 9.

The disnix-env process is a composition of several other processes, for the following reasons:

- A user should be able to perform steps in the deployment process separately. For instance, a user may want to build all the services defined in the model to see whether everything compiles correctly, while he does not directly want to activate them in the network.
- Testing and debugging is relatively easy, since everything can be invoked from the command line by the end user. Moreover, components can be invoked from scripts, to easily extend the architecture.
- Components are implemented with different technologies. The disnix-manifest and disnix-instantiate tools are implemented in the Nix expression language, because it is required to build components by the Nix package manager. Other tools are implemented in the

C programming language, such as the disnix-distribute and disnix-activate tools and shell scripts: disnix-env.

- Prototyping is relatively easy. A tool can be implemented in a high level language first and later reimplemented in a lower level language.

The principles above are quite common for developing UNIX applications and inspired by Raymond [12].

6.1. Building the services

The Disnix models are *declarative* models. Apart from the build instructions of the services, the complete deployment process including the order is defined from these models and their defined intra- and inter-dependencies.

In the first step of the deployment process, the disnix-manifest tool is invoked with the three Disnix models as input parameters. The result of this tool is a *manifest* file, an XML file that specifies what services need to be distributed to which machine and a specification of the services to be activated. The manifest file is basically a concrete version of the abstract distribution model. While the distribution model specifies which services should be distributed to which machine, it does not reference the actual service that is built from source code for the given target machine under the given conditions, i.e. the component in the Nix store, such as /nix/store/x39a1...-StaffTracker. Because the concrete Nix store path names have to be determined, this results in a side effect in which all services are built from source code for the given target platforms.

6.2. Transferring closures

The generated manifest file is then used by the disnix-distribute tool. 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 knowledge, we can compare the store paths of the closure of the component to be transferred with the store paths present on the target system.

Nix can detect *runtime dependencies* from a build process, by scanning a component for occurrences of hash codes that uniquely identify components in the Nix store. For example, 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. We know by scanning that a specific glibc component in the Nix store is a *runtime dependency* of java.

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.

The disnix-distribute tool will efficiently copy the components and their intra-dependencies to the machines in the network through the remote interfaces. Only the components that are not yet distributed to the target machines are copied, while keeping components that are already there intact. This process is also non-destructive, as no existing files are overwritten or removed. To perform the actual copy step to a single target, the disnix-copy-closure tool is invoked.

6.3. Transition phase

Finally, the disnix-activate tool is executed, which performs the *transition* phase. In this phase, all the obsolete services are deactivated and all the new services are activated. The right order is derived from the inter-dependencies defined in the services model. By comparing the manifests of the current deployment state and the new deployment state, an upgrade scenario can be derived.

During the transition phase, every service receives a lock and unlock request to which they can react, so that optionally a service can reject an upgrade when it is not safe or temporarily queue connections so that end users cannot observe that the system is changing. While the transition phase is ongoing, other Disnix clients are locked out, so that they cannot perform any deployment operation, preventing that the upgrade process is disturbed.

6.4. Distributed building

Disnix provides an optional feature to build services on the target machines instead of the coordinator machine. If this option is enabled, two additional modules are used. The disnix-instantiate tool will generate a distributed derivation file. This file is also an XML file similar to the manifest file, except that it maps store derivation files (low-level specifications that Nix uses to build components from source code) to target machines in the network.

It then invokes disnix-build with the derivation file as argument. This tool transfers the store derivation files to the target machines in the network, builds the components on the target machines from these derivation files, and finally copies the results back to the coordinator machine. The disnix-copy-closure tool is invoked to copy the store derivations to a target machine and to copy the build result back.

Finally, it performs the same steps as it would do without this option enabled. In this case disnix-manifest will not build every service on the coordinator machine, since the builds have already been performed on the machines in the network. Also, due to the fact that Nix uses a purely functional deployment model, a build function always gives the same result if the input parameters are the same, regardless of the machine that performed the build.

6.5. Atomic upgrading

To make the actual deployment process atomic, we mapped concepts of the two-phase commit [13] 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 committed. This phase consists of building the services from source code and transferring 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 file names 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 deactivate the newly activated services and reactivate the deactivated services from the old configuration.

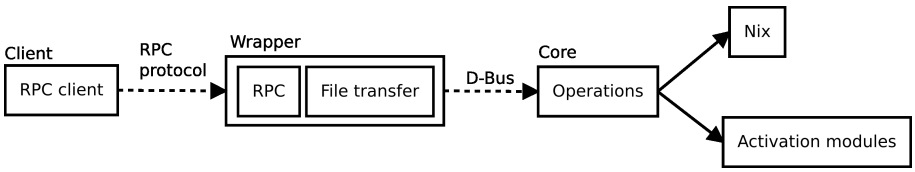

Figure 10: Architecture of the *DisnixService*

In rare cases, it may be possible that the coordinator crashes and that the locks are not released. In such cases the administrator has to manually unlock all the target machines in the network. Basically, these limitations are the same as the original two phase commit algorithm.

6.6. Disnix service

Some of the tools explained in the previous subsections have to connect to the target machines in the network to perform deployment steps remotely, e.g. to perform a remote build or to activate/deactivate a service. Remote access is provided by the *Disnix Service*, which must be installed on every target machine.

The Disnix service consists of two major components, shown in Figure 10. We have a component called the *core* that actually executes the deployment operations by invoking the Nix package manager to build and store components and the activation modules package to activate or deactivate a service. The *wrapper* exposes the methods remotely through an RPC protocol of choice, such as an SSH connection or through the SOAP protocol provided by an external add-on package called DisnixWebService.

There are two reasons why we have chosen to make a separation between the interface component and the core component:

- *Integration.* Many users have specific reasons why they choose a particular protocol e.g. due to organisation policies. Moreover, not every protocol may be supported on every type of machine.
- *Privilege separation.* To execute Nix deployment operations, super-user privileges on UNIX-based systems are required. The wrapper may need to be deployed on an application server hosting other applications, which could introduce a security risk if it is running as super user.

Custom protocol wrappers can be trivially implemented in various programming languages. As illustrated in Figure 10, the Disnix core service can be reached through the D-Bus protocol [14], used on many Linux systems for inter-process communication. In order to create a custom wrapper, a developer has to map RPC calls to D-Bus calls and implement file transport of the components. D-Bus client libraries exist for many programming languages and platforms, including C, C++, Java and .NET and are supported on many flavours of UNIX and Windows.

6.7. Disnix interface

In order to connect to a machine in the network, an external process is invoked by the tools on the deployment coordinator machine. An interface can be configured by various means, such as specifying the DISNIX_CLIENT_INTERFACE environment variable. The Disnix toolset includes

two clients: disnix-client, a loopback interface that directly connects to the core service through D-Bus; and disnix-ssh-client, which connects to a remote machine by using an SSH connection (which is the default option).

The separate DisnixWebService package includes an additional interface called disnix-soap-client, which uses the SOAP protocol to invoke remote operations and performs file transfers by using MTOM, an XML parser extension to include binary attachments without overhead.

A custom interface can be trivially implemented by writing a process that conforms to a standard interface, calling the custom RPC wrapper of the Disnix service.

6.8. Activation modules

Since services can represent many things, such as processes or databases, and cannot be activated generically, Disnix provides *activation types*, which can be connected to activation modules as illustrated in Figure 10. In essence, an activation module is a process that takes four arguments: the first is either 'activate', 'deactivate', 'lock' or 'unlock' and the second is the Nix store path of the service that has to be activated/deactivated. Disnix passes all the properties defined in the infrastructure model as environment variables, so that system properties such as port numbers can be used.

The lock and unlock options can be used to notify the component that the transition phase starts or ends so that it can optionally block or queue incoming connections. A service can also reject a lock, which aborts the transition phase, if the upgrade is not safe.

The Disnix toolset includes a set of activation modules that can be used for activating commonly found services such as Apache Tomcat web applications, Apache HTTP server web applications, MySQL databases, NixOS configurations and generic UNIX processes. 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 deployment operation. On deactivation the symlink is removed, triggering a hot undeployment operation. Moreover, there is a wrapper activation module, which will invoke a wrapper process with a standard interface included in the service. This type can be used to provide a component specific activation interface.

Custom activation modules can be trivially implemented by a process that conforms to the activation module interface.

6.9. Libraries

Many of the tools in the Disnix toolset require access to information stored in models. Since this functionality is shared across several tools we implemented access to these models through libraries. The libmanifest component provides functions to access properties defined in the manifest XML file, libinfrastructure provides functions to access to properties defined in the infrastructure model and libdistderivation provides functions to access properties in a distributed derivation file.

We rather use library interfaces for these models, since they are only accessed by tools implemented in the C programming language and do not have to be invoked directly by end users.

6.10. Additional tools

Apart from disnix-env and the tools that compose it, the Disnix toolset includes several other utilities that may help a user in the deployment process of a system:

- disnix-query. Shows the installed services on every machine defined in the infrastructure model.

- disnix-collect-garbage. Runs the garbage collector in parallel on every machine defined in the infrastructure model to safely remove components that are no longer in use.
- disnix-visualise. Generates a clustered graph from a manifest file to visualise services, their inter-dependencies and the machines to which they are distributed. The dot tool [15] can be used to convert the graph into an image, such as PNG or SVG.
- disnix-gendist-roundrobin. Generates a distribution model from a services and infrastructure model, by applying the round robin scheduling method to divide services over each machine in equal portions and in circular order.

7. Infrastructure deployment

Although the Disnix toolset provides useful features to make automated deployment of a service-oriented system possible and properties to make this process efficient and reliable, the toolset itself has a very generic approach. Some desirable deployment properties cannot be addressed in a generic manner, since they are specific to the domain in which the system has to be used. For instance, the underlying infrastructure, such as the configurations of machines in the network, are not managed by the basic toolset, because a network may be composed of systems running various operating systems and hardware architectures. The purpose of Disnix is to provide automatic and reliable deployment for the *service* components of a system.

Management of *infrastructure* is also a complex and labourious process. For example, installing an operating system on a machine may take some time. Moreover, a machine must be configured to have the right properties and capabilities, such as running a database or application server with the right settings.

In this section, we explain *DisnixOS*, a complementary toolset for Disnix, providing *infrastructure* deployment using NixOS. NixOS is a GNU/Linux distribution built on top of Nix, which can be used to deploy a complete system configuration from a declarative specification, including networks of machines.

7.1. Background

NixOS is a GNU/Linux distribution [16], which uses the Nix package manager to manage all packages, system components (such as the Linux kernel) and configuration files. In NixOS, system configurations are derived from declarative specifications describing properties of the system, such as kernel modules, partitions, services and system packages.

The Nix expression in Figure 11 shows an example of a NixOS configuration. A NixOS configuration is essentially a Nix expression defining a function, having an argument called pkgs, which refers to Nixpkgs, a collection of 2500 packages which can be deployed by the Nix package manager. The remainder of the function is an attribute set defining properties of a system.

The boot attribute set [26], describes the settings of the GRUB bootloader, such as the partition containing the Master Boot Record (MBR) is /dev/sda. At [27], the settings of hard drive partitions are specified, such as the root partition and the swap partition. System services are specified at [28], such as an OpenSSH server providing remote SSH access. Furthermore, system packages are defined at [29], which can be used by all users on the machines, such as the Lynx web browser and Midnight Commander.

In Figure 12, a network model is shown, which combines several NixOS configurations in an attribute set. For example, the test1 attribute [30] refers to the NixOS configuration defined

```
{ pkgs, ... }: 25

{
  boot.loader.grub.device = "/dev/sda"; { 26

  fileSystems = 27
    [ { mountPoint = "/";
        device = "/dev/sda2";
      }
    ];

  swapDevices = [ { device = "/dev/sda1"; } ];

  services.openssh.enable = true; 28

  environment.systemPackages =
    [ pkgs.mc pkgs.subversion pkgs.lynx ]; 29
}
```

Figure 11: /etc/nixos/configuration.nix: A NixOS configuration file

```
{
  test1 = import ../configurations/openssh.nix; 30
  test2 = import ../configurations/tomcat.nix; 31
  test3 = import ../configurations/httpd.nix;
}
```

Figure 12: network.nix: A network of NixOS configurations

earlier in Figure 11. Similarly, other NixOS configurations can be defined in this model, such as 31 defining a NixOS configuration running Apache Tomcat.

This network specification can be used by NixOS to deploy a network of NixOS machines or to create a network of virtual machines closely matching the system configurations defined in the network model. A virtual network can be instantiated cheaply, because components in the Nix store can be accessed from the host machine through a network file system, a technique described in previous work [17]. DisnixOS combines the network model and infrastructure deployment features of NixOS as an addition to Disnix, to provide complementary infrastructure deployment next to service deployment.

7.2. Overview

The usage of DisnixOS is similar to the basic Disnix toolset. By running the following command:

```
$ disnixos-env -s services.nix -n network.nix -d distribution.nix
```

both the services of which the system is composed and the configurations of network of machines are built from source code. Then the NixOS configurations in the network are upgraded, and finally the service-oriented system is deployed the same way the regular disnix-env command does.

Similarly, DisnixOS provides the disnixos-vm-env command which can be invoked with the same parameters as disnixos-env. In this case a network of virtual machines is generated and launched, closely matching the configurations of the machines defined in the network model.

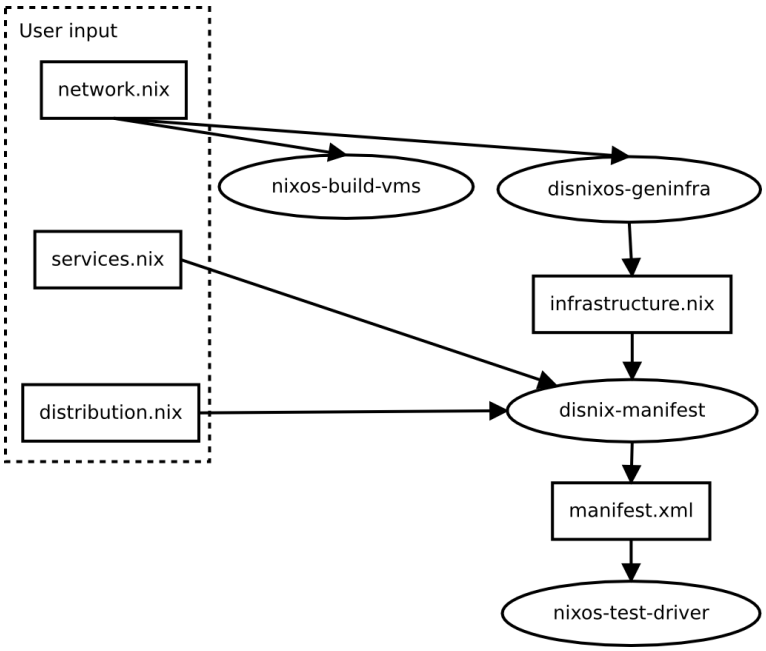

Figure 13: Architecture of the disnixos-vm-env tool

The services are automatically deployed in the virtual network. This tool is useful for testing a configuration, without having physical machines available. Moreover, tweaking NixOS configurations to completely match the actual systems is not required, as NixOS takes care of the virtual machine characteristics. Furthermore, a virtual network can also be generated from *any* Linux distribution (not only NixOS) using the Nix package manager and the Kernel-based Virtual Machine (KVM).

7.3. Architecture

Figure 13 shows the architecture of the disnixos-vm-env tool. First, the network model is used by a NixOS tool called nixos-build-vms, which builds a network of virtual machine configurations closely matching the configurations described in the network model.

Then the disnixos-geninfra tool is used to convert a network model into an infrastructure model, which the basic Disnix toolset understands. The main difference between the infrastructure model and the network model is that the infrastructure model is an *implicit* model: it should reflect the system configurations of the machines in the network, but the system administrator must make sure that the model matches the actual configuration. Moreover, it only has to capture attributes required for deploying the services. The network model is an *explicit* model, because it describes the *complete* configuration of a system and is used to deploy a complete system from this specification. If building a system configuration from this specification succeeds, we know that the specification matches the given configuration.

After generating the infrastructure model, the disnix-manifest tool is invoked, which generates a manifest file for the given configuration. Finally, the nixos-test-driver is started, which launches

the virtual machines for testing.

The architecture of the disnixos-env tool is similar, except that it does not generate VM instances and instead it distribute closures of NixOS configurations and activates the NixOS configurations, instead of launching virtual machines.

7.4. Limitations

DisnixOS is built on top of NixOS, which is a Linux based system. For many operating systems (especially proprietary ones), system services are so tightly integrated with the operating system that they cannot be managed through Nix (upon which Disnix is based). The DisnixOS extension is not suitable for deployment of networks using other types of Linux distributions or other operating systems, such as FreeBSD or Windows, because they have system configurations which cannot be completely managed by Nix.

On proprietary infrastructure, such as Windows, the basic Disnix toolset can still be used for deploying service components, but infrastructure components have to be deployed by other means, e.g. manually or by using other deployment tools.

8. Usage

Disnix and its extension DisnixOS, as well as other Disnix extensions can be obtained from the Disnix website: `http://nixos.org/disnix`. The source code of example cases is also available, demonstrating various features of Disnix and allowing users to do experiments. Disnix and related projects are available under free and open-source software licenses.

8.1. Examples

The following examples are available:

- *StaffTracker (PHP/MySQL version)*, a distributed system composed of several MySQL databases and a PHP web-application front-end. This is a simplified version of the system we have used as an example throughout this paper.
- *StaffTracker (Web services version)*. A more complex variant of the previous example implemented in Java, in which web services provide interfaces to the databases. This is the example system we have used throughout this paper.
- *StaffTracker (WCF version)*. An experimental port of the previous web services example to .NET technology using C# as an implementation language, Windows Communication Foundation (WCF) as underlying technology for web services. This example is supposed to be deployed in an environment using Microsoft IIS and SQL server as infrastructure components.
- *ViewVC*. An example case using ViewVC (`http://viewvc.tigris.org`), a web-based CVS and Subversion repository viewer implemented in Python. The Disnix models can be used to automatically deploy ViewVC, a MySQL database storing commit records, and a number Subversion repositories.
- *Hello World example*. A trivial example case demonstrating various ways to compose web services. In this example, a basic composition is demonstrated, an alternative composition by changing an inter-dependency, and two advanced compositions, using a lookup service and load balancer, which *dynamically* compose services.

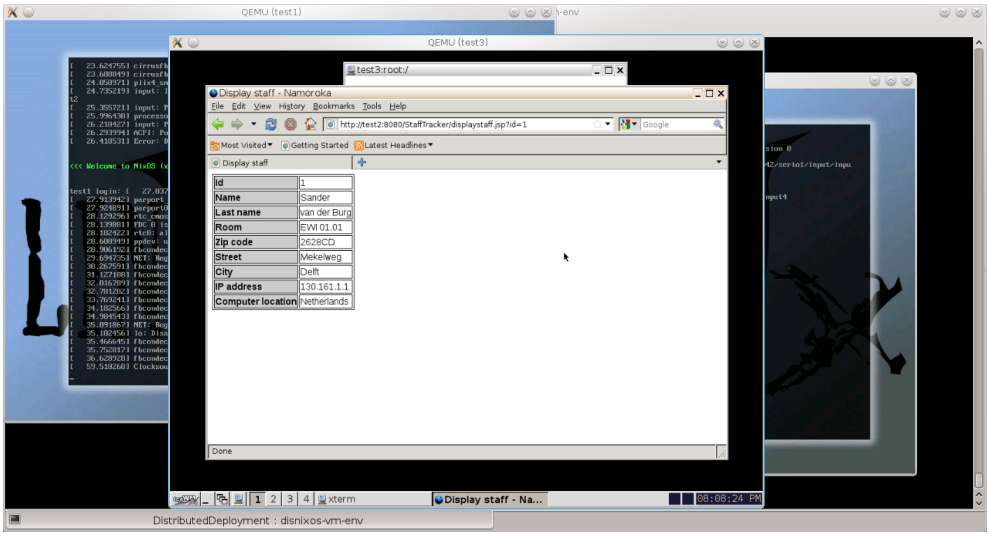

Figure 14: Using disnixos-vm-env to deploy the StaffTracker in a network of three virtual machines

- *Disnix proxy example*. A trivial example case demonstrating the use of a TCP proxy which temporarily queue network connections during the transition phase. This is the example we used in one of our previous papers about atomic upgrading [8]. Furthermore, this is the most trivial example case using self-hosted service components, which do not require any infrastructure components.
- *NixOS configurations*. In this example, we show that we can also describe complete NixOS systems as services and that we can upgrade complete machine configurations in the network.

8.2. Usage scenarios

There are three ways to try Disnix with the examples, as described in the previous section. The first option is to deploy the examples in a heterogeneous network, which also takes the most effort. This requires users to manually install a couple of machines in a network running required system services, such as a MySQL server or a web server. The Disnix service and its dependencies must also be installed manually. The configuration script detects which system services are supported and automatically configures the activation scripts. Furthermore, a developer or system administrator must manually write an infrastructure model resembling the properties of the machines.

Another way of trying out the examples is to use DisnixOS. This requires users to install a network of NixOS machines with a basic configuration, which includes the Disnix service. In the examples, NixOS configuration files can be found, which only require some small adaptations (such as the hard drive partition configuration settings) to work on an arbitrary machine.

The easiest and most convenient way to try the examples is by using disnixos-vm-env. This command generates, as explained previously, a network of NixOS virtual machines closely resembling the network model and automatically deploys the service-oriented system. An impres-

sion of this is shown in Figure 14 in which the StaffTracker system is deployed in a network of three virtual machines.

8.3. Example usage

In this section, we give a short walkthrough, showing how Disnix can be used to deploy our most trivial example: the Disnix TCP proxy, consisting of a server and a client communicating through TCP sockets. This example system consists of two services: a hello-world-service component sending a ”Hello world!” message and a hello-world-client component, which connects to the server to retrieve and display the message.

Optionally, another service called disnix-tcp-proxy can be used as a proxy for the hello-world-service. Before the transition phase starts, the proxy will ”drain” all existing connections until all connections are closed. This proxy also temporarily queues incoming connections during the upgrade phase and forwards them to the new version after the upgrade, to make the transition process completely atomic.

8.3.1. Installing a target machine

Each target machine in the network requires you to install two packages: disnix-activation-scripts provides a collection of activation scripts which can be attached to service types. The disnix package contains the Disnix service and client utilities used to perform deployment steps. Both can be configured and installed by either using the Nix package manager, or by compiling it from source code.

Apart from installing the package, you also have to run the Disnix daemon as a system service. Configuration of system services typically differ among Linux distributions and other operating systems, such as Microsoft Windows. The Disnix manual has more information about the installation process and the configuration of system services.

8.3.2. Installing the coordinator machine

The coordinator machine also needs an installation of the disnix package so that the client utilities can be found. No system services have to be configured. Furthermore, the Disnix TCP proxy example tarball must be extracted on the coordinator machine, so that the deployment can be performed.

8.3.3. Performing deployment

The Disnix TCP proxy example package contains the source code for the client and service as well as a collection of Disnix expressions, which can be used for deployment. The Disnix models can be found in the DistributedDeployment/ folder. The distribution model is preconfigured to deploy the server and client component on separate machines in the network. You can adapt the distribution model to change this scenario. For example, it is also possible to map all the services to a single machine. The infrastructure model also needs to be adapted to match the configuration of the actual network. For example, the hostname attribute must match the hostname or IP address of a target machine in the network.

The deployment process can be performed by running the following command-line instruction on the coordinator machine:

```
$ disnix-env -s services-without-proxy.nix \
  -i infrastructure.nix \
  -d distribution-without-proxy.nix
```

By running the command-line instruction above, the complete deployment process is performed, which consist of building all services, transferring them to the target machines in the network and by activating the services. The result is a complete running system. After deploying the system, the following instruction can be invoked from the command-line on the client machine:

```
$ /nix/var/nix/profiles/disnix/default/bin/hello-world-client
```

The client returns: "Hello world!", a message sent by the server. By replacing the models with services-with-proxy.nix and distribution-with-proxy.nix, the variant with the TCP proxy can be used.

9. Related work

There is quite an amount of work on deployment of services in distributed environments. A number of approaches limit themselves to specific types of components, such as the BARK reconfiguration tool [18], which only supports the software deployment life-cycle of EJBs and supports atomic upgrading by changing the resource attached to a JNDI identifier.

Akkerman et al [2] have implemented a custom infrastructure on top of the JBoss application server to automatically deploy Java EE components. In their approach they use a component description language to model components and their connectivity and an ADL for modeling the infrastructure, to automatically deploy Java EE components to the right application servers capable of hosting the given component. The goals of these models are similar to the Disnix services and infrastructure models; however, they are designed specifically for Java EE environments. Moreover, the underlying deployment tooling do not have capabilities such as storing components safely in isolation and performing atomic upgrades.

In [19] 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 [20] which proposes a conceptual component framework for dynamic configuration of distributed systems. In [21] 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 [3] 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 [22] the Globus toolkit is described for performing three types of deployment scenarios using predefined types of components compositions in a grid computing environment. CODEWAN [23] 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 [4]. It emphasises on the delivery process of components from producer to consumer site, rather than *complete* deployment. In [24] 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 [25] 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.

In [26], a model-driven application deployment architecture is described, containing various models capturing separate concerns in a deployment process, such as the application structure, network topology of the data center, and transformation rules. These models are provided by various stake holders involved in the deployment process of a distributed application in data centers, such as developers, domain experts and data center operators. These models are transformed and optimised by various tools into a concrete deployment plan which can be used to perform the actual deployment process, taking various functional requirements and non-functional requirements into account. These techniques are applied in various contexts, for example to automatically compose services in a cloud environment [27] and can be bridged to various tools to perform the deployment and provisioning [28].

In this paper, the models used by Disnix and DisnixOS are merely executable specifications used to perform the actual deployment process and not designed for deployment planning. Moreover, the semantics of the Nix expression language that Disnix uses are close to the underlying purely functional deployment model. However, in our recent work [29], we have implemented an extension on top of Disnix, providing additional models to specify a policy of mapping services to machines in the network based on functional properties defined in the service and infrastructure models. Also, various deployment planning algorithms described in literature can be used for this purpose. In conjunction with a discovery service service-oriented systems can be made self-adaptable. It could also be possible to integrate Disnix components into the architecture of [26] so that their deployment planning techniques can be used and our deployment mechanics to make a software deployment process reliable.

Another notable aspect that sets Disnix apart from other deployment tools, is that every service component must be managed through the Nix store (although there are ways to refer to components on the host system, which is not recommended). Disnix cannot be used to manage the deployment of existing service components deployed by other means. If we would be able to refer to components outside this model, we lose all the advantages that the purely functional deployment model of Nix offers, such as the fact that we cannot uniquely distinguish between variants of components, store them safely next to each other and that we can safely and atomically upgrade components. If the DisnixOS extension is used, also the underlying infrastructure components are managed by Nix and thus cannot be used to manage infrastructure components deployed by other means. In some cases, upgrades with Disnix can be more expensive than other deployment tools, since everything has to managed by the Nix package manager and components are bound to each other statically, but in return the Nix deployment model provides very powerful reliability aspects. In [29], we have used Disnix to dynamically upgrade distributed systems in the case of events, which shows that for most events the costs are actually not that high.

Regarding dynamic upgrades: in [30], a dynamic update system called POLUS is described that runs multiple versions of a process simultaneously and redirects function calls from the old version to the new version using the debugging API. Moreover, it can also recover tainted state by using callback functions. The system is not used in a distributed environment however. In our approach, we only run one instance of a particular component and the proxy queues incoming connections, while the transition process is in progress. However, it may be possible to make

the proxy more sophisticated to support redirection from old versions to new versions of two instances running simultaneously. Furthermore, Disnix and Nix cannot deal with mutable state.

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.

10. Conclusion

In this paper we have given an overview of the Disnix toolset, which can be used by system administrators or developers to automatically deploy a service-oriented system into a network of machines. We have shown that we are using process composition to create a modular deployment architecture. This offers us various benefits, such as the ability to perform deployment steps separately, supporting tools implemented with various technologies, easy integration and easy prototyping.

Furthermore, we have described DisnixOS, a complementary extension to Disnix for infrastructure management. This extension is built on top of NixOS, a Nix-based Linux distribution. Because it is also built on top of Nix, this extension makes it possible to deploy the underlying infrastructure next to services and to easily deploy a system in a network of virtual machines for testing.

Disnix, its extensions and examples as well as other Nix-related projects such as NixOS and Hydra are released as free and open source software, available from `http://nixos.org`. This web site also offers publicly available example cases demonstrating the features of Disnix.

Acknowledgments. 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, F. Leymann, Service-oriented computing: State of the art and research challenges, Computer 40 (2007) 38–45.

[2] A. Akkerman, A. Totok, 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.

[3] E. Caron, P. K. Chouhan, H. Dail, GoDIET: A Deployment Tool for Distributed Middleware on Grid 5000, Technical Report RR-5886, Laboratoire de l'Informatique du Parallélisme (LIP), 2006.

[4] R. S. Hall, D. Heimbigner, 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, ACM, New York, NY, USA, 1999, pp. 174–183.

[5] E. Dolstra, E. Visser, M. de Jonge, Imposing a memory management discipline on software deployment, in: Proc. 26th Intl. Conf. on Software Engineering (ICSE 2004), IEEE Computer Society, 2004, pp. 583–592.

[6] E. Dolstra, The Purely Functional Software Deployment Model, Ph.D. thesis, Faculty of Science, Utrecht University, The Netherlands, 2006.

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

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

[9] S. van der Burg, E. Dolstra, Automated deployment of a heterogeneous service-oriented system, in: 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society, 2010.
[10] MaxMind - GeoIP — IP Address Location Technology, `http://www.maxmind.com/app/ip-location`, 2010.
[11] E. Foster-Johnson, Red Hat RPM Guide, John Wiley & Sons, 2003.
[12] E. S. Raymond, The Art of Unix Programming, Thyrsus Enterprises, 2003. Also available at: `http://www.faqs.org/docs/artu`.
[13] D. Skeen, M. Stonebraker, A formal model of crash recovery in a distributed system, in: Concurrency control and reliability in distributed systems, Van Nostrand Reinhold Co., New York, NY, USA, 1987, pp. 295–317.
[14] freedesktop.org - software/dbus, `http://www.freedesktop.org/wiki/Software/dbus`, 2010.
[15] Graphviz, `http://www.graphviz.org`, 2010.
[16] E. Dolstra, A. Löh, NixOS: A purely functional Linux distribution, in: ICFP 2008: 13th ACM SIGPLAN Intl. Conf. on Functional Programming, ACM, 2008.
[17] S. van der Burg, E. Dolstra, Automating system tests using declarative virtual machines, in: 21st IEEE International Symposium on Software Reliability Engineering (ISSRE 2010), IEEE Computer Society, 2010.
[18] M. J. Rutherford, K. M. Anderson, A. Carzaniga, D. Heimbigner, 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.
[19] M. Mikic-Rakic, 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.
[20] X. Chen, 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.
[21] 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.
[22] G. v. Laszewski, E. Blau, M. Bletzinger, J. Gawor, P. Lane, S. Martin, 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.
[23] H. Roussain, 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.
[24] T. Dumitras, J. Tan, Z. Gho, 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, USENIX Association, Berkeley, CA, USA, 2007, p. 14.
[25] N. P. Sudmann, 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.
[26] T. Eilam, M. Kalantar, A. Konstantinou, G. Pacifici, J. Pershing, A. Agrawal, Managing the configuration complexity of distributed applications in internet data centers, Communications Magazine, IEEE 44 (2006) 166 – 177.
[27] A. V. Konstantinou, T. Eilam, M. Kalantar, A. A. Totok, W. Arnold, E. Snible, An architecture for virtual solution composition and deployment in infrastructure clouds, in: Proceedings of the 3rd international workshop on Virtualization technologies in distributed computing, VTDC '09, ACM, New York, NY, USA, 2009, pp. 9–18.
[28] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, A. V. Konstantinou, Model driven provisioning: bridging the gap between declarative object models and procedural provisioning tools, in: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware, Middleware '06, Springer-Verlag New York, Inc., New York, NY, USA, 2006, pp. 404–423.
[29] S. van der Burg, E. Dolstra, A self-adaptive deployment framework for service-oriented systems, in: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), ACM, 2011.
[30] H. Chen, J. Yu, R. Chen, B. Zang, P.-C. Yew, Polus: A powerful live updating system, in: Proceedings of the 29th international conference on Software Engineering, ICSE '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 271–281.