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 certain non-functional requirements from the domain are supported. In this paper we describe the *Disnix* toolset, which enables 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 build a distributed deployment architecture that takes non-functional requirements of the domain into account. Disnix has been under development for almost two 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 software deployment process of a distributed system, which consists of steps such as building components from source code, transferring the components from producer side to consumer side and the activation of the system, is usually a very difficult, expensive and time-consuming process. In many cases, there is a lot of labour involved, 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). Furthermore, there are non-functional properties from the domain that must be supported, such as security, privacy and performance.

In order to reduce the 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 [1], or designed for environments such as Grid Computing [2]. Furthermore, some general approaches have been developed, such as [3].

While these tools are useful for its purpose, some distributed systems are not homogeneous (i.e. not implemented by a single component technology or for a single platform/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.

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

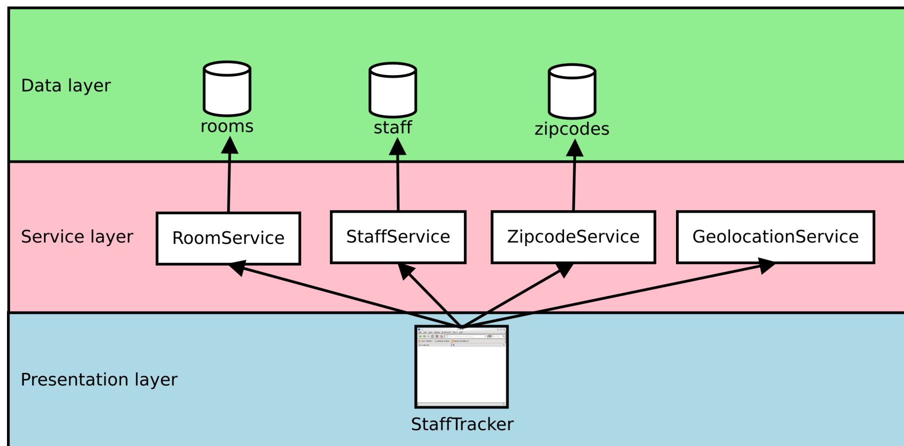

Figure 1: Architecture of the StaffTracker example system. Objects represent distributable components. Arrows represent dependency relationships between components.

Therefore we have developed Disnix, a toolset built on top of the Nix package manager [4, 5] used to automatically deploy 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. Moreover, 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 a service-oriented system designed for hospital environments [6].

In this paper 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. In Section 5, we explain the architecture of Disnix, such as its tools, libraries and design choices. Section 6 explains some extensions built on top of the Disnix toolset to make deployment more convenient for specific domains. Section 7 explains our experiences with the development and testing of the toolset, an open source community in which Disnix is being developed and some use cases. Finally, Section 8 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 publically used. The StaffTracker is a 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 [7] 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 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 this paper), 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.

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 publically 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 seperate 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 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. Existing research and solutions are able to make the deployment easier for specific components and environments, but cannot manage a hetereogeneous system such as the StaffTracker on multiple platforms *and* offer properties such as dependency completeness and atomic upgrades. Moreover, they also do not offer the ability to guarantee non-functional requirements, such as privacy.

3. Background

Disnix is a toolset used to perform distributed deployment tasks and is built on top of Nix [4, 5], a package manager with some distinct features compared to conventional package managers (e.g. RPM [8]) 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 filename in the Nix store. This approach makes it safe to store multiple versions and variants 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

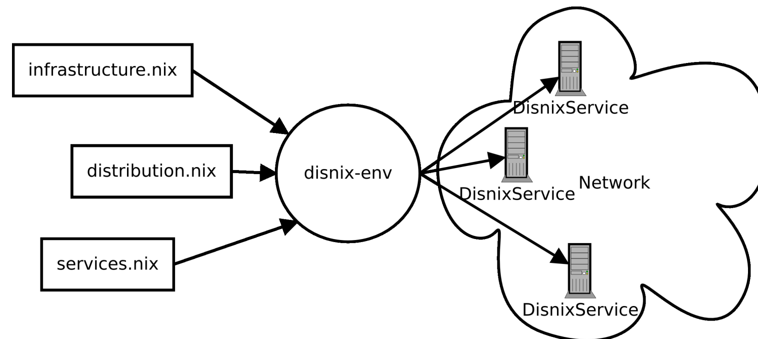

Figure 2: Overview of the disnix-env tool

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. 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 reproducable. 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 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. An *infrastructure* model is used to specify what machines are available

```
{distribution, system}:

let pkgs = import ../top-level/all-packages.nix { inherit system; }; in [1]
rec {
### Databases
  rooms = { [2]
    name = "rooms";
    pkg = pkgs.rooms;
    dependsOn = {};
    type = "mysql-database";
  };
  ...
### Web services
  RoomService = {
    name = "RoomService"; [3]
    pkg = pkgs.RoomService; [4]
    dependsOn = { [5]
      inherit rooms;
    };
    type = "tomcat-webapplication"; [6]
  };
  ...
### Web applications
  StaffTracker = {
    name = "StaffTracker";
    pkg = pkgs.StaffTracker;
    dependsOn = {
      inherit GeolocationService RoomService StaffService ZipcodeService;
    };
    type = "tomcat-webapplication";
  };
}
```

Figure 3: Partial services.nix model for the StaffTracker

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.

Figure 3 shows a partial service model for the StaffTracker system. This expression is a set of attributes in which every attribute (such as [2]) represents a component from the architecture described earlier in Figure 1. Every attribute is itself an attribute set defining the relevant properties of the service, such as the name of the service [3], a reference to the function that builds the service from source code [4], the inter-dependencies of the service [5] (which correspond to the arrows shown in Figure 1) and its type [6]. The functions that build every component from source code and its intra-dependencies (such as libraries and compilers) are defined in a seperate file, which is imported at [1] (not covered in this paper).

Figure 4 shows an infrastructure model used to capture the machines in the network and their relevant properties/capabilities. This model is also an attribute set. Here, every attribute represents a system in the network, such as a machine called test2 [7]. For each machine relevant properties are defined such as the architecture of the system [10], so that Disnix can build a service for the desired type of platform, and a hostname attribute [8] so that Disnix knows how to reach a target machine to perform remote deployment operations. The other attributes are optional and are used to activate particular types of services. For instance, the tomcatPort attribute [9] specifies through which port the Apache Tomcat server can be reached.

A distribution model, shown in Figure 5, maps services to machines in the network. In this

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

  test2 = { [7]
    hostname = "test2.testdomain.net"; [8]
    tomcatPort = 8080; [9]
    mysqlPort = 3306;
    mysqlUsername = "user";
    mysqlPassword = "secret";
    system = "x86_64-linux"; [10]
  };
}
```

Figure 4: An infrastructure.nix model for the StaffTracker

```
{infrastructure}:

{
  GeolocationService = [ infrastructure.test1 ]; [11]
  RoomService = [ infrastructure.test2 ];
  StaffService = [ infrastructure.test1 ];
  StaffTracker = [ infrastructure.test2 ];
  ZipcodeService = [ infrastructure.test1 infrastructure.test2 ]; [12]
  rooms = [ infrastructure.test2 ];
  staff = [ infrastructure.test2 ];
  zipcodes = [ infrastructure.test2 ];
}
```

Figure 5: A distribution.nix model for the StaffTracker

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, [11] 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 [12], which deploys the ZipcodeService on both test1 and test2. More details on how these models can be specified can be found in [9].

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.

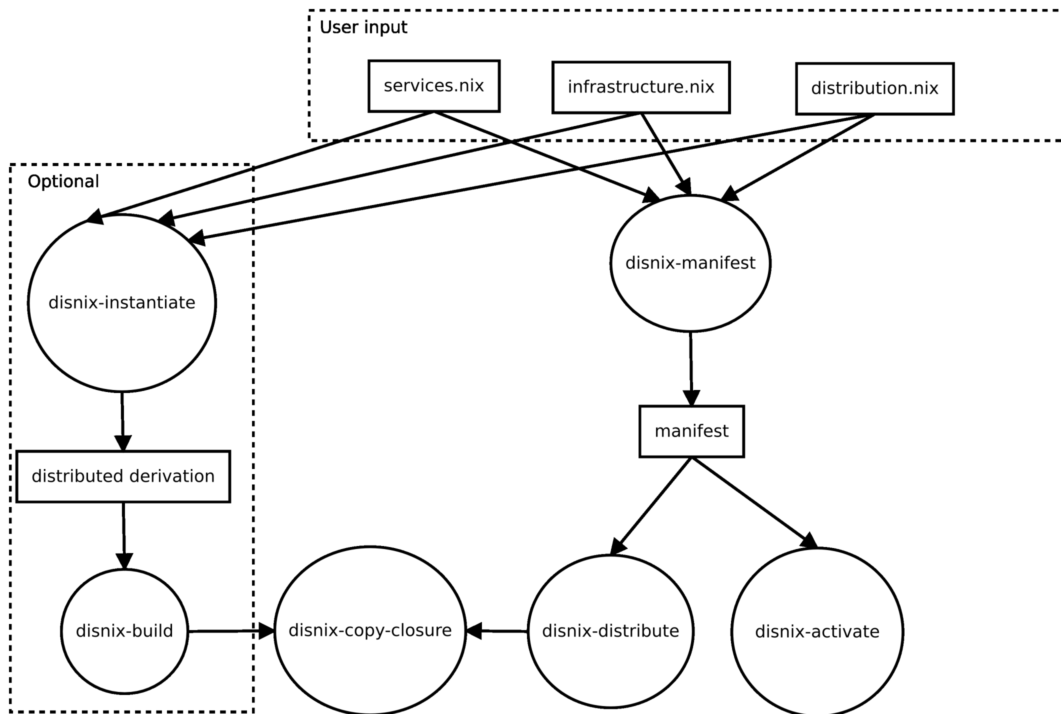

Figure 6: Architecture of disnix-env. Rectangles represent objects that are manipulated by the tools. The ovals represent a tool from the toolset performing a step in the deployment process.

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 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. More details on this can be found in [9, 10].

5. Architecture

The disnix-env tool, which performs the complete deployment process, is composed of several tools each performing a step in the deployment process, shown in Figure 6.

Firstly, the disnix-manifest tool is invoked with the three models as input parameters. This tool generates 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. This tool iterates over the distribution model and builds the services and all their dependencies for the given target machine and produces the manifest file referring to the build results.

The generated manifest file is then used by the disnix-distribute tool. This 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. Moreover, 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.

Finally, the disnix-activate tool is executed, which performs the *activation* phase. In this phase all the obsolete services are deactivated and all the new services are activated in the right order, by comparing the manifests of the current deployment state and the new deployment state. During this 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. This allows the upgrade process to be completely atomic. More details about this process can be found in [9, 10].

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 transfer the store derivation files to the target machines in the network, then 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-instantiate 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.

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.

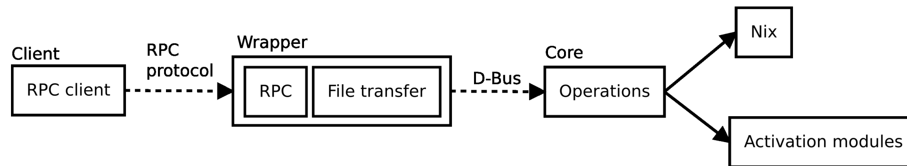

Figure 7: Architecture of the *DisnixService*

- 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 [11].

5.1. Disnix service

Some of the tools explained in the previous subsection 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 7. 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 choose 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 7, the Disnix core service can be reached through the D-Bus protocol [12], 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.

5.2. 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 a SSH connection. 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.

5.3. 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 7. In essence, an activation module is a process that takes two arguments: the first is either 'activate' or 'deactivate' 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 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. Moreover, there is a wrapper activation module, which will invoke a wrapper process with a standard interface included in the service.

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

5.4. 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.

5.5. 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.
- 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 [13] 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 the infrastructure model in equal portions and in circular order.

6. Extensions

Although the Disnix toolset itself provides useful features to make automated deployment of a distributed system possible and properties to make this process efficient and reliable, the toolset itself has a very generic approach. For instance, some non-functional properties cannot be addressed in a generic manner, since they are specific to the domain in which the system has to be used. We have to extend the toolset architecture to make this process more convenient, by integrating modules dealing with such non-functional properties.

In this section, we illustrate some of the extensions that we have designed. These extensions are implemented, but are still works in progress and not as mature and usable as the basic toolset.

6.1. Virtualization

In some cases, especially while developing a system, users may want to test a certain deployment scenario in virtual machines, instead of performing a real deployment scenario, which requires having physical machines available somewhere with a preconfigured base system.

While Disnix manages the services constituting a distributed system, Disnix does not manage the system configurations of the target machines. This basically means that the virtual machines for testing have to be configured by other means (i.e. manually, by following the installation procedure of the operating system), which still could take a lot of effort.

In [14] we have demonstrated an approach in which we can automatically and efficiently instantiate a network of virtual machines from a declarative specification and automatically perform test cases on those virtual machines. This work builds upon NixOS [15], a GNU/Linux distribution which uses the Nix package manager to manage all packages, including the Linux kernel, system services and configuration files. Because Nix uses a purely functional approach for packages, we can share the Nix store components across virtual machines and the host system, which makes virtual machine instances relatively cheap.

To overcome the burden of creating virtual machines to test a deployment scenario with Disnix, we developed a tool called disnix-vm-env. This is a wrapper around the regular disnix-env that instantiates and launches virtual machines to simulate the network as defined by the models.

To do this, the infrastructure model is replaced by a *network model*, which defines the configuration of a network of NixOS machines. Figure 8 shows a partial network model, which can be used to deploy the StaffTracker in a virtual environment. This model is an attribute set in which every attribute represents a system, such as [13] representing machine test1. Each attribute refers to a NixOS configuration that describes the complete system configuration, such as the running services, configuration files and system properties. For instance at [14], the configuration of the Apache Tomcat container is specified, which is tuned with a specic Java virtual machine option to increase the heap space at [15]. Other required services such as MySQL are defined at [16]. This

```
{
  test1 = {pkgs, ...}: [13]
    {
      services.openssh.enable = true;
      services.disnix.enable = true;
      services.tomcat = { [14]
        enable = true;
        catalinaOpts = "-Xms64m -Xmx256m"; [15]
        ...
      };
      ...
    };
  test2 = {pkgs, ...}:
    {
      services.openssh.enable = true;
      services.disnix.enable = true;
      services.mysql.enable = true; [16]
      services.tomcat = {
        enable = true;
        catalinaOpts = "-Xms64m -Xmx256m";
        ...
      };
      ...
    };
}
```

Figure 8: Partial network.nix model for the StaffTracker

file can be used to automatically build a network of NixOS systems conforming to the given system configurations. More details about this model and its applications can be found in [14].

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 specifcation succeeds, we know that the specification matches the given configuration. Unfortunately, this property only works because we built upon NixOS; other operating systems such as FreeBSD or Windows cannot use this approach.

The following instruction from the command line spawns a network of machines defined in the network Nix expression and then deploys the StaffTracker system into the virtual machines:

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

Figure 9 shows the architecture of the disnix-vm-env tool, which is a composition of several tools similar to the basic Disnix toolset. The disnix-generate-vms tool uses the network model to instantiate and launch a network of virtual machines running NixOS with the given configurations. The disnix-vm-generate-infra tool generates an infrastructure model from the network model which the disnix-env tool can use. Finally, the service model, generated infrastructure model and distribution model are used by disnix-env to perform the actual deployment on the virtual machines.

In order to connect to the *DisnixService* instances on the virtual machines a disnix-backdoor-client has been developed, which connects to each virtual machine through a UNIX domain

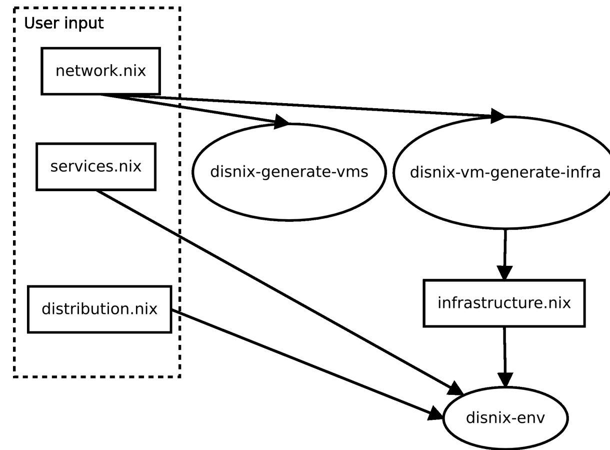

Figure 9: Architecture of disnix-vm-env

socket. This offers developers the advantage that no network settings have to be configured on the host system such as IP or port forwarding.

A limitation of this approach is that currently only NixOS configurations are supported. Deploying on a FreeBSD or Microsoft Windows system still requires the user to manually configure a base system.

6.2. Dynamic deployment

The deployment specifications required by Disnix are very *static*; a user must explicitly define all the machines and their relevant properties in the infrastructure model and has to explicitly define for each service to which machines they should be distributed. Maintaining such specifications by hand for large networks is impractical. Therefore a more dynamic approach is required. In [16] we have outlined the requirements for such an approach in which we want to *generate* an infrastructure and distribution model. Thus, we treat the infrastructure as a *dynamic cloud*, in which we automatically deploy components implementing a service to the right machines in the network, taking capabilities and quality of service attributes into account.

Although the Disnix toolset includes a very simple distribution generator called disnix-gendist-roundrobin, we would rather have a more sophisticated distribution generator that copes with non-functional requirements from the domain that we want to support.

By having a dynamic infrastructure and distribution generator, we can construct a service that continuously monitors the infrastructure and installed services and can react upon a change, e.g. perform a redeployment to fix the system if a machine breaks or to increase the capacity if a machine is added to the network.

Figure 10 shows an architecture of a dynamic deployment tool. The user provides a services model and a quality-of-service model, taking non-functional properties from the domain into account. A discovery service generates an infrastructure model. A distribution model generator maps the services to machines in the generated infrastructure model taking the constraints in the QoS model into account. Finally the services model, generated infrastructure and generated distribution model are passed to the disnix-env tool, which performs the actual deployment.

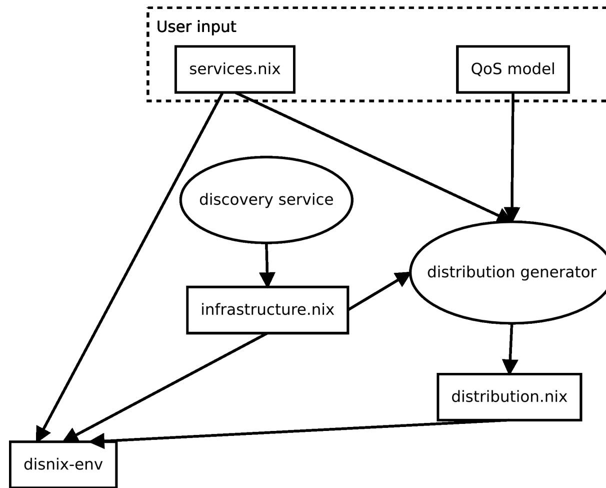

Figure 10: Architecture of a dynamic distributed deployment tool

Currently, we are working on a conceptual extension framework with an infrastructure model generator using Avahi [17] and a distribution model generator that uses a single variable (typically a system resource such as memory usage) to find a suitable distribution. Moreover, we are investigating non-functional requirements in the medical domain to build a more sophisticated dynamic deployment extension.

6.3. Service design abstraction

While Disnix provides properties to make the deployment of a distributed system safe and reliable and offers integration with modules to take non-functional requirements of the domain into account, not every non-functional property can be supported in the architecture of Section 6.2, because they are *implicit* in the design of a service.

An example of these issues are reliability issues, such as failing connections that are not automatically restored. If, for instance, the database connection with the rooms database fails and is not restored again, the RoomService will be inaccessible forever. Another possible issue is *static lookup* of inter-dependencies, which requires a user to deploy a new version of the service if a dependency is moved to another location. For example, if the zipcode database moves to another location all the inter-dependent services (ZipcodeService and the StaffTracker front-end) must be redeployed, since their configurations have to be adapted. Often service developers do not want to manually implement such constraints or even think about these issues, because they are too complex and too much effort.

We have developed ideas for higher level abstractions hiding such implementation details. One of our experiments is integrating WebDSL [18], a domain specific language for developing web applications with a rich data model into the dynamic deployment architecture.

By using an architecture integrated with a service abstraction layer (such as WebDSL for web applications), a developer only has to provide a high level specification of a web application and (optionally) some QoS attributes. The deployment architecture will make the resulting system available for use in a complex environment taking the QoS attributes into account.

7. Experience

7.1. Development

Disnix has been under development for almost 2 years. The initial prototype was designed for use with SDS2, our first industrial case study, and was implemented by using the same technology, such as the Java programming language and the Apache Axis2 library to create a SOAP interface exposing deployment operations.

Later on, Disnix was applied to different types of distributed systems implemented with different technologies. Some initial implementation choices turned out to be impractical and therefore a new version of Disnix was developed.

Java was an impractical dependency, because it requires a Java Runtime Environment, which is quite large and not every platform has a decent Java Runtime implementation available. Moreover, it was impractical for systems not using Java, such as PHP web application systems, to deploy an entire Java environment to make deployment possible. Furthermore, it was also desirable for developers to execute certain tasks separately and to use different protocols than SOAP.

In the second version of Disnix all the Java modules were rewritten in C. This was not a very huge effort since their purposes were already well explored and the libraries used in the C version were almost as convenient as the Java ones. This also eliminated the large Java dependency, since the libraries used in the C versions were much smaller and quite common on Linux systems.

Moreover, a more modular architecture was designed in which communication was performed by an external process instead of using a Java interface. This choice gave us the option to directly invoke remote machine operations from the command-line and to use a programming language of choice to implement the interface (e.g. the disnix-soap-client was implemented in Java, since the Java SOAP libraries were more convenient to use and the disnix-ssh-client was implemented as a shell script not requiring a complex dependency such as Java).

Furthermore, the concept of activation types was developed in which types were mapped to external processes with a standard interface performing the activation and deactivation. Process composition was very practical in making the system more modular, since the modules can be implemented in various technologies and invoked directly by end-users.

In the third version (which became release 0.1) support for *heterogeneous* networks (networks consisting of machines with different architectures and operating systems) was implemented. The format of the models were changed in order to facilitate abstraction over architecture types.

7.2. Testing

Disnix and its extensions are continuously built and tested by *Hydra*, our continuous build and integration server built around Nix [19]. Testing is done by using the approach described in [14] (also used for the virtualization extension described in Section 6.1), in which we declaratively instantiate cheap networks of virtual machines and perform automatic test-cases on them. By using this approach we continuously test the deployment operations of Disnix for various protocols.

7.3. Community

Disnix is released as free and open source software under the GNU Lesser General Public License version 2.1 or higher. Extensions such as the activation modules and DisnixWebService are released under the MIT license.

Disnix is part of the *Nix project* (`http://nixos.org/`), in which solutions based around the Nix package manager are developed, such as NixOS, a GNU/Linux distribution using the Nix package manager and Hydra, our continuous build and integration server. All Nix related software is released under free and open source licenses.

The Nix project has a small community of developers and contributors consisting of researchers from various universities, developers from companies and enthusiastic individuals. Having a community offers us useful contributions such as compilers, libraries and end user applications in *Nixpkgs* [20], a repository consisting of over 2,500 packages that can be automatically deployed by the Nix package manager. These can be used to construct and deploy services with Disnix. Another major contribution is cross-compilation support which can also be used with Disnix. Maintaining all the parts in the Nix project ourselves is too much effort and would limit ourselves in achieving new results.

7.4. Use cases

We have used Disnix for automating the deployment of various types of distributed systems consisting of diverse kinds of services, such as MySQL databases, PHP web applications, Java based web applications, Java based web services and UNIX processes. Moreover, we have applied Disnix to an industrial case study from Philips Research called SDS2 [6], which is explained in [9].

Our repository contains several examples with full source code to demonstrate possible applications of Disnix. Some of these examples are:

- A PHP/MySQL web application system with multiple databases.
- The StaffTracker application described in this paper: a service-oriented system implemented mostly in Java, consisting of 3 layers (databases, web services, web application front-ends)
- Another Java-based service-oriented system to demonstrate how concepts such as load balancing and lookup can be implemented and used with Disnix.
- A distributed system consisting of processes communicating through TCP sockets, which can be adapted to block/queue connections to make the upgrade process atomic. This example is explained in [10].
- A network of NixOS configurations.

These examples can be deployed either on physical machines or tested in virtual machines by using the virtualization extension from Section 6.1. The examples are released under the MIT license.

8. Conclusion

In this paper we have given an overview of the Disnix toolset, which can be used to automatically deploy a distributed system into a network of machines. We have explained extensions on top of the basic toolset to deal with certain non-functional requirements from the domain that cannot be solved generically.

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. Moreover, we illustrated several extensions implemented on top of the Disnix toolset by using the process composition approach.

Furthermore, we have discussed our experiences developing Disnix, such as an open-source community that offers many benefits and the testing process in which we automatically instantiate a distributed test environment to test Disnix.

We have applied Disnix to various types of distributed systems consisting of various components, including an industrial case study in the healthcare domain designed by Philips Research.

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/`.

In the future we intend to maintain the Disnix toolset and further develop the extensions on top of the Disnix toolset. A dynamic deployment framework will be developed to support quality-of-services attributes in the medical domain. Moreover, a service abstraction model will be developed dealing with non-functional deployment requirements implicit in the design of a service. Furthermore, more evaluation is needed in large environments. Dynamic deployment support is a precondition to support this. Finally, another industrial case study is in progress, which is much larger than SDS2.

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] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] MaxMind - GeoIP — IP Address Location Technology, `http://www.maxmind.com/app/ip-location`, 2010.

[8] E. Foster-Johnson, Red Hat RPM Guide, John Wiley & Sons, 2003.

[9] S. van der Burg, E. Dolstra, Automated deployment of a heterogeneous service-oriented system, in: I. Crnkovic, R. Mirandola (Eds.), 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society, 2010. To appear.

[10] 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.

[11] E. S. Raymond, The Art of Unix Programming, Thyrsus Enterprises, 2003. Also available at: `http://www.faqs.org/docs/artu`.

[12] freedesktop.org - software/dbus, `http://www.freedesktop.org/wiki/Software/dbus`, 2010.

[13] Graphviz, `http://www.graphviz.org`, 2010.

[14] 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. To appear.

[15] E. Dolstra, A. Löh, NixOS: A purely functional Linux distribution, in: ICFP 2008: 13th ACM SIGPLAN Intl. Conf. on Functional Programming, ACM, 2008.

[16] S. van der Burg, E. Dolstra, M. de Jonge, E. Visser, Software deployment in a dynamic cloud: From device to service orientation in a hospital environment, in: K. Bhattacharya, M. Bichler, S. Tai (Eds.), First ICSE 2009 Workshop on Software Engineering Challenges in Cloud Computing, IEEE Computer Society, 2009.

[17] Avahi, `http://avahi.org`, 2010.

[18] E. Visser, WebDSL: A case study in domain-specific language engineering, in: R. Lammel, J. Saraiva, J. Visser (Eds.), Generative and Transformational Techniques in Software Engineering (GTTSE 2007), Lecture Notes in Computer Science, Springer, 2008.

[19] E. Dolstra, E. Visser, The Nix Build Farm: A Declarative Approach to Continuous Integration, in: K. Mens, M. van den Brand, A. Kuhn, H. M. Kienle, R. Wuyts (Eds.), International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008).

[20] Nix Packages collection, `http://nixos.org/nixpkgs`, 2010.