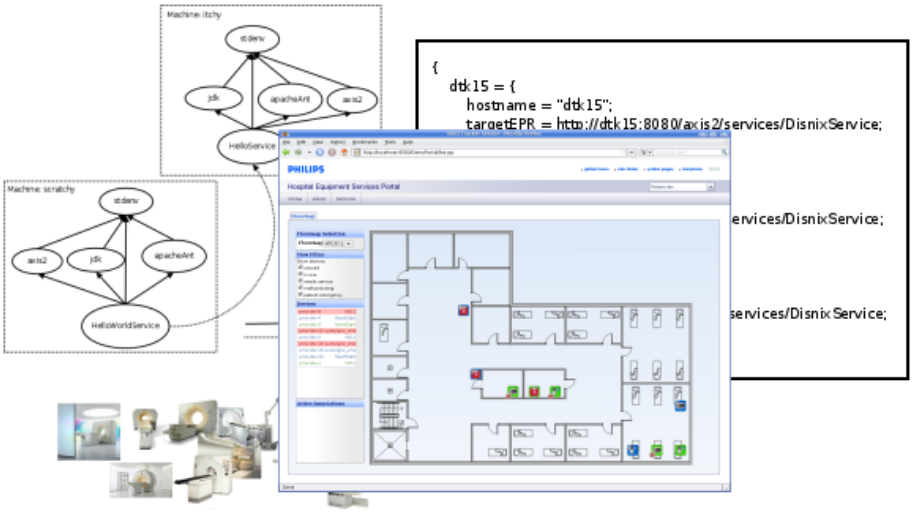
Model-driven Distributed Software Deployment

Master's Thesis



Sander van der Burg

Model-driven Distributed Software Deployment

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Sander van der Burg
born in Numansdorp, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Philips Research
Healthcare Systems Architecture
High Tech Campus 37, 5656 AE
Eindhoven, the Netherlands
www.extra.research.philips.com/swa

Model-driven Distributed Software Deployment

Author: Sander van der Burg
Student id: 1274856
Email: svanderburg@gmail.com

Abstract

The *software deployment* process, which involves all steps to make a software system available for use, is a complex process. Software systems can operate on just one machine but there are also many software systems with components running on multiple computers in a network that are working together to reach a common goal, which are called *distributed systems*. The software deployment process for distributed systems is more complex than for single systems, since we have dependencies which are on the same machine in the network, but also dependencies on components on other machines in the network. Components of systems in hospital environments are distributed and its deployment process is error prone and tedious. In this thesis we extend a deployment system called *Nix* for single systems with new tools and models to make software deployment in distributed environments possible. The extension is called *Disnix*, which is a framework to make distributed deployment possible by reusing the Nix primitives. This thesis also contains a case study where we used an existing distributed system for hospital environments developed at Philips Research which is called *SDS2*. We also adapted the SDS2 platform to fit in the Disnix framework and we identified some design constraints that should be met in order to make a system automatically deployable.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
University co-supervisor:	Dr. E. Dolstra, Faculty EEMCS, TU Delft
Company supervisor:	Dr. M. de Jonge, Philips Research
Committee Member:	Dr. Ir. D.H.J. Epema, Faculty EEMCS, TU Delft

Preface

From the moment I started using Linux distributions, I was always trying to build customized systems and I noticed that it was a very complex process in order to maintain system configurations efficiently. I used to work on a project called *Linux from Scratch* [25] in the past which is a book that explains how to build your own Linux system from all the original sources. Developing a system from scratch was complex, error prone and very time consuming. When I met Eelco Visser and when I heard about the Nix project that deals with these issues and hearing about an application at Philips Research, this project quickly captured my interest.

I started working on Nix in order to make automatic deployment of WebDSL applications possible. Therefore I would like to thank the WebDSL developers: Danny Groenewegen, Zef Hemel, Jippe Holwerda, Lennart Kats, Sander Vermolen, Ruben Verhaaf, Wouter Mouw and Michel Weststrate. They share my interest in deploying web applications. However, I got tired of their countless jokes about Nix and Disnix ("Dit is ook niks!").

I worked several days in the week at the Healthcare Systems Architecture department of Philips Research to work on my case study. The HSA department was a very nice environment to work in. I learned a lot about medical systems and their software applications, e.g. Clinical Desktops. Many colleagues took interest in my project. Therefore I would like to thank Harm Buisman, Georgio Mosis, Aleksandra Tesanovic and Jin Min about our nice "off-topic", but also technical related discussions, at the coffee corner. Especially that story about a bar in London.

I would like to thank Luuk Kuiper who was my roommate at HSA. He helped me with some issues on the SDS2 platform. We also had many interesting discussions about developing complex systems and software architectures. I would like to thank Wim van der Linden who is the project leader of the SDS2 platform. He helped me with providing information about technical details of SDS2 and by making some modifications in order to fit SDS2 in my deployment extension. Furthermore, I would like to thank Krystyna Milian, who was a fellow intern at the HSA department. She was kind enough to let me explain the details about my project to her and to look at a demonstration of my work. She let me notice some bugs I forgot to solve. Unfortunately she was not able to show me much of her work on Clinical Desktops at the time I was there.

I also would like to thank my supervisors. Eelco Visser is my university supervisor

and offered me this project. He shared many ideas about software deployment and about making ideas and software available as Open Source. We both believe that anyone that has interest in the research that we do should be able to participate and be able to use our work.

Eelco Dolstra is my other university supervisor and the author of the Nix deployment system. I had many interesting discussions with him about Nix, Linux distributions, our old computer systems and other topics. He helped me thinking about some details and improving some parts on my extension to Nix. He was also a co-author of a paper about this thesis which we published to the HotSWUp'08 workshop.

Merijn de Jonge is my supervisor at Philips Research. He helped me with providing details about hospital environments and details of the SDS2 platform. He also helped me with making modifications to the SDS2 platform. He is a "believer" in the Nix deployment system. Discussing with him about Nix always gave me inspiration for new ideas which I used to develop my extension to the Nix deployment system. He was the other co-author of the paper we have submitted.

Finally I want to thank my brother and parents for their support. I explained them a couple of times what this research is about, and although they claim that my explanation was clear, they are still not able to explain it to others.

Sander van der Burg
Delft, the Netherlands
March 3, 2009

Contents

List of Figures

Chapter 1

Introduction

Many software systems that are in use today are modular and usually consist of many software components. To make a software system available for use we have to build all the software components from source code, transfer them from the producer site to the customer site and activate the software system on the customer site. A software component is almost never self-contained. It usually depends on other components, such as libraries or compilers to make it work. These components are called *dependencies*. In order to make a software system available for use, all the necessary dependencies have to be present, should be correct (i.e. the right version or variant of the component) and the system should be able to find them.

Software systems can operate on just one machine but there are also many software systems that consist of components running on multiple computers in a network that are working together to reach a common goal. These systems are called *distributed systems*. There are many distributed systems in use today. For instance many systems in hospital environments are distributed systems. There are a wide range of devices available in hospitals ranging from workstations to medical equipment e.g. MRI scanners. Systems are connected through wired and wireless networks with complex topologies. Devices work together by exchanging messages, for instance the exchange of patient information.

This thesis is about making software systems which consists of many software components available for use in distributed environments. In this thesis we extend the *Nix software deployment* system with new models and tools, which we call *Disnix* to make automatic and correct software deployment possible in distributed environments. As a case study, we use an existing distributed system for use in hospital environments called *SDS2*, which is developed at Philips Research.

1.1 Software deployment

This thesis is about distributed software deployment. The definition of software deployment is ambiguous. To avoid confusion we define the defintion of software deployment used in this thesis.

The basic meaning of deployment may be described as a mapping. The term deployment

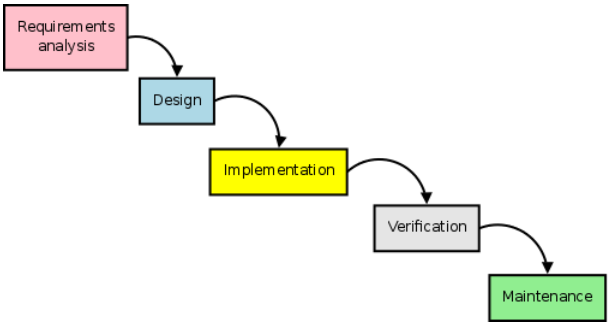


Figure 1.1: A software development process represented in a waterfall model

can be used in various contexts, e.g. military, software, grid computing etc. For instance the military deployment process consists of steps of moving units and materials to strategic locations.

Some definitions of software deployment that are used are: "the installation of software components on a machine" or "transferring software components from the producer site to the consumer site". In this thesis we define software deployment as *all of the activities* that make a software system available for use [1]. A software deployment process consists of several activities such as compiling the source code, release management, transferring components to the site of the customer, installation, uninstallation, activation and deactivation of components. The definition of software deployment in this thesis is more complete than definitions used by others, because it consists of *all* steps that make a software system available for use and not just the delivery and installation steps.

The software deployment process is a crucial part of the *software development* process. A software development process is a structure imposed on the development of a software product and is also known as a software life cycle [42]. There are several models and methodologies for software development processes such as waterfall and iterative methods. Each method consists of a variety of tasks or activities that take place during the process, such as requirements analysis, design, implementation, verification, maintenance and so on. Figure 1.1 illustrates an example of a software development process.

In some software development methodologies the software deployment process usually starts after the code is appropriately tested and ready to be transferred to the customer site, however it also fits in other activities of a software deployment process such as in a verification phase e.g. deploying components in a special testing environment and during the development phase e.g. running continuous integration tests.

The software deployment process should be a simple problem, but it turns out to be much more complex. There are many issues that we can categorize as *environment issues* and *manageability issues* [15].

Environment issues concern essentially correctness. The software system has all sorts of demands in the environment in which it executes. For instance it requires other components to be present in the system, that some configuration files exist and so on. If some of these

requirements do not hold it could result in a software system that will not operate the way the developer intended or might not work at all. Some issues are:

- A software component is almost never self-contained, but it depends on other components to make it work. These components are called *dependencies*. Some examples of dependencies are libraries, configuration files and so on. Not all dependencies are needed during runtime. Some dependencies are only needed during compile time and deployment time, e.g. in order to compile a component from source code we need compilers, linkers etc. and in order to fetch the source code from a repository we also need tools to download them.
- A software system should be able to find its dependencies, for instance by setting environment variables such as the CLASSPATH environment variable for Java applications
- Some software components require specific hardware e.g. a specific processor type
- In order to run some components you have to deploy them in a specific container, for instance an application server which manages the resources for the application [13]

Manageability issues concern the ability to properly manage the deployment process. It is essentially about properly executing the steps of the deployment process for instance installing, upgrading, uninstalling components and so on. Some of these issues are:

- If we want to uninstall a component we want this action to be safe, e.g. we do not want to remove a component which is still in use.
- We also want to keep a software system up to date. Upgrading components in a software system should be safe and atomic. We do not want to have a mix of an old configuration and a new configuration of a system. We also want to have atomic rollbacks.

1.2 Distributed systems

Distributed systems can be described as a collection of computers that appear to a user as one logical system [38]. Services in a distributed system are working together to reach a common goal by exchanging messages with each other. A distributed system is built on top of a network and tries to hide the existence of multiple autonomous computers.

The software deployment process of distributed systems is even more challenging than software deployment on single systems. On single machines we have dependencies on components that should be available on the same machine. In this thesis we will call these dependencies *intra-dependencies*. Intra-dependencies can be runtime dependencies, compile time dependencies and deployment time dependencies. Some examples of intra-dependencies are: libraries, compilers, linkers, download tools, configuration files and so on. The format of addressing an intra-dependency is usually a path to the local filesystem where the component is stored.

In distributed environments components also have dependencies on components running on other systems in the network. In this thesis we will call these runtime dependencies between components on different systems *inter-dependencies*. For instance, many web applications store data in a database backend. The web application service and the database service can be located on different machines in the network. The web application will not work if the database management system is not running. So the web application has an *inter-dependency* relationship with the database. The format of addressing an inter-dependency is typically a URL which contains a hostname or IP address, an optional port number of the machine and a pathname which identifies a component on a particular machine in the network. Sometimes there is also a specific RPC protocol involved that is used for the communication between components. Inter-dependencies are related to intra-dependencies, since the pointer address of an inter-dependency (e.g. a URL) is usually stored on the local filesystem in a configuration file which is stored on the filesystem, which is a way to adress an intra-dependency.

On single systems we do not have an inter-dependency relationship because we assume that all components are running on the same machine. The only thing we have to do on single systems is making sure that all needed components are activated in the right order, which is usually done by the service manager of the operating system. Thus, the deployment process in distributed systems is more challenging because we have two-dimensions of dependency relationships, the *intra-dependencies* and the *inter-dependencies*. Both dependency relations should be satisfied in order to make a software system working.

1.3 Hospital environments

The research in this thesis was done at the Healthcare Systems Architecture department of Philips Research. Many systems in a hospital environment are distributed. Hospital environments consist of many devices ranging from workstations to medical equipment such as MRI scanners. Devices can be connected through wired and wireless networks with complex topologies. All these devices work together, e.g. by exchanging patient information from and to medical equipment. Hospitals are also complex organizations. Many hospitals consist of several sites each with its own devices, have their own software running on it, and has its own organization policies. A typical deployment scenario of a hospital environment is illustrated in Figure 1.2.

The software deployment process in hospital environments is usually a semi-automatic process. There are tools to assist administrators with deploying software in a hospital, but a large part of the deployment process is done manually with deployment documentation. A semi-automatic software deployment process requires up to date documentation which describes how to execute the deployment steps, and people with necessary skills to execute deployment steps.

If people with the appropriate skills disappear or due to the lack of up to date or complete documentation, the software deployment process becomes more time consuming, more error prone, risky and more expensive. This is not very unlikely for many big organizations. It also becomes more difficult or even impossible to reproduce a previous deployment sce-

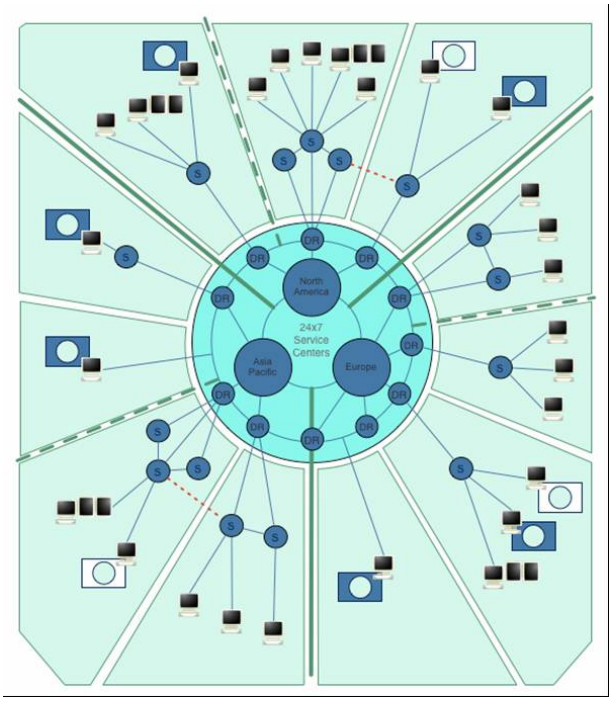


Figure 1.2: A distributed deployment scenario in a hospital environment

nario and reason about its correctness. Because the software deployment process in hospital environments is so complex, they upgrade their IT infrastructure only once or twice a year.

1.4 Vision on software deployment

The software deployment process should be a simple process and not a complex one. Since most deployment processes are semi-automatic, there is always a risk on human errors. Therefore the software deployment process should be fully automatic instead of semi-automatic. An automatic software deployment process is faster and more efficient. To make automatic software deployment possible we have to capture the system configuration in models.

With a correct model of the system configuration and by using a reliable deployment system we never have systems that result in an *incorrect* deployment state, e.g. a mix of old and new configurations or a configuration that does not match the specified specification in a model. Another benefit of automatic deployment is that we never lose documentation of the configuration of a system because it is captured in the model and we are able to reproduce the configuration in an environment of choice.

Just as distributed systems appear to a user as one logical system, we also would like to deploy software on a distributed system as it it were a single system. So in our vision we want the deployment process of an entire distributed system modeled in one declarative

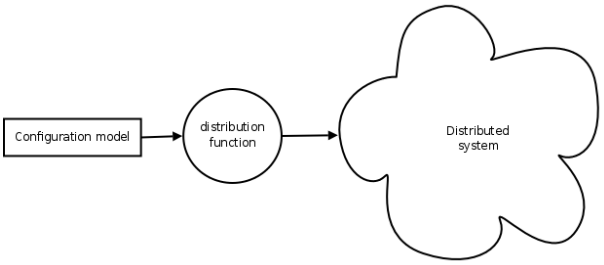


Figure 1.3: Deployment vision

specification that can be applied to a particular network by using a distribution function that executes the necessary deployment steps, which is illustrated in Figure 1.3.

1.5 Realizing distributed deployment

The Nix package manager [15] is a software deployment system that builds software packages from purely functional models. It guarantees that dependency specifications are complete and supports atomic upgrades and rollbacks. However, its models and atomic upgrades deal with single systems. The Nix deployment system solves some of the issues required to reach our vision on software deployment.

In this thesis we propose the Disnix deployment system. Disnix extends the Nix deployment system to make distributed deployment of software systems possible. This is done by mapping the deployment concepts of Nix for single systems to distributed systems.

We will use the Service Development Support System (SDS2) developed at Philips Research, as a case study. The SDS2 system is a service oriented architecture for asset tracking and utilisation services in hospital environments. In this thesis project we adapted the SDS2 platform to make it automatically deployable with Nix and Disnix.

1.6 Research questions

The research question investigated in this thesis are:

- How can we map the concepts of the Nix package manager for software deployment of single systems to distributed systems?
- What is required to turn the semi-automatic deployment process of an existing distributed software system into a fully automatic deployment process by using the Nix approach of software deployment?
- What is required to make the automatic deployment process of a distributed system efficient and reliable?

1.7 Approach

In this thesis project we use an existing distributed system as a case study, which is called SDS2 developed at Philips Research. The first step is exploring the concepts of the Nix package manager and the SDS2 platform. The second step is automating the software deployment process of SDS2 by creating Nix expressions of SDS2 platform components and adapting the SDS2 platform components. The third step is extending the Nix deployment system with distributed deployment features. The final step is adapting the SDS2 platform for automatic distributed software deployment.

By using SDS2 as case study, we can derive features that are needed to extend the Nix deployment system with distributed deployment operations. We can also derive other requirements that are needed to turn a semi-automatic deployment process of a distributed system into a fully automatic deployment process. Finally, we can generalize the derived features and requirements in the case study to other distributed systems.

1.8 Success criteria

In order to succeed this thesis project, we need a working prototype implementation that supports distributed deployment of an adapted SDS2 platform that will meet our derived requirements. It should demonstrate that the complexity of the software deployment process of SDS2 is reduced.

1.9 Outline of this thesis

In Chapter 2 we explain the *Nix deployment system* with an overview of the concepts and features of Nix. The deployment concepts are reused later in this thesis to make a mapping to the concepts of the Disnix deployment system which is used for distributed systems. In Chapter 3 we explain the *Service Development Support System* (SDS2) which is used as case study in this thesis. We give an overview of the ideas behind SDS2, the architecture of SDS2 and what technologies are used. Chapter 4 explains the modifications made to SDS2, how the components of SDS2 are modelled and necessary infrastructure components in the Nix expression language in order to make it automatically deployable with Nix. In Chapter 5 we give an overview of the idea and features of the *Disnix deployment system*. Chapter 6 explains the service component of the Disnix deployment system in detail which provides remote access to Nix operations. Chapter 7 explains how the upgrade process of Disnix works. In Chapter 8 we explain what modifications we made to SDS2 in order to make it automatically deployable with Disnix. Chapter 9 lists related work and Chapter 10 concludes this thesis.

Chapter 2

The Nix Deployment System

In this research project we use the Nix deployment system as a basis for distributed software deployment. Nix solves a number of software deployment problems needed to reach our vision on software deployment. Conventional package managers have some limitations which the Nix deployment system overcomes [15, 19]. In this chapter, we explain the ideas and features of the Nix deployment system.

2.1 Motivation

Conventional package managers such as RPM [24] have implicit dependency specifications. The dependency specifications are not validated and are often inexact (e.g. nominal). It is possible to specificy what libraries, compilers are needed to build a component but it is not specified where to find them or what version or variant we exactly need. Thus due to these inexact dependency specifications it is possible to have build errors.

For instance a RPM spec file contains a line such as: Requires: openssl $>=$ 0.9.8, which specifies that a component depends on another component named openssl and the version should be 0.9.8 or higher. There still could be several variants of this component. If for instance the openssl component is compiled with GCC version 2.95.2 and another openssl component is compiled with GCC version 4.2.3 they are still the same according the RPM spec. There are differences between this variants of components however. One problem is that components compiled with GCC 2.95.2 cannot be linked together with components compiled with GCC version 4.2.3 because they have a different ABI. In Nix all the dependency specifications are *exact*, so except for a name and version number, Nix uses hashcodes based on all inputs (including the used compiler variant) to identify components.

Files are often stored in global namespaces. For instance on Linux based systems, executables of programs are often stored in /usr/bin and shared libraries are often stored in /usr/lib. By using these global namespaces it is possible that components can interfere with each other, because it is not known where components get their dependencies from. Another issue of these global namespaces is that it is not possible to deploy multiple versions or variants of components side-by-side.

Conventional package managers do not support atomic updates to packages. This is

because updating a package requires replacing all the files belonging to the package with newer versions, which is not something that can be done in an atomic transaction on most file systems. Thus, during the update, there is a time window during which the system is in an inconsistent state. The file system contains some files belonging to the old version of the package, and some belonging to the new version. The effect of using the package during this time window (e.g., starting a program in the package) is undefined.

Replacing files is a destructive operation. If the update fails at some point we could have a inconsistent state of dependencies which could result in a unusable system. Because most conventional package managers use a destructive update model it is also difficult or even impossible to roll back to a previous configuration.

2.2 The Nix store

Nix stores components in a so called *Nix store*. The Nix store is a special directory in the file system, usually /nix/store. Each entry in the directory are components. Each component is stored in isolation, that is because there are no files that share the same name in the store. An example of a component in the Nix store is: /nix/store/bwacc7a...-hello-2.1.1. A notable feature of the Nix store are the component names. The first part of the file name, e.g. bwacc7a... is a SHA256 *cryptographic hash* in base-32 notation of *all inputs* involved in building the component. The hash is computed over all inputs, including:

- Sources of the components
- The script that performed the build
- Any command-line arguments or environment variables passed to the build script
- All build time dependencies, e.g. the compiler, linker, libraries and standard UNIX command-line tools such as cat, cp and tar

The cryptographic hashes in the store paths serve two main goals:

- *Preventing interference*. The hash is computed over all inputs involving the build process of the component. Any change, even if it is just a white space in the build script, will be reflected in the hash. For all dependencies of the component the hash is computed recursively. If two component compositions are different, they will have different paths in the Nix store. Because the hash code reflect all inputs to build the component the installation or uninstallation step of a component will not interfere with other configurations.
- *Identifying dependencies*. The hash in the component file names also prevents the use of undeclared dependencies. Instead of searching for libraries and other dependencies in global name spaces such as /usr/lib, we have to give paths to the components in the Nix store explicitly. For instance the following compiler instruction:

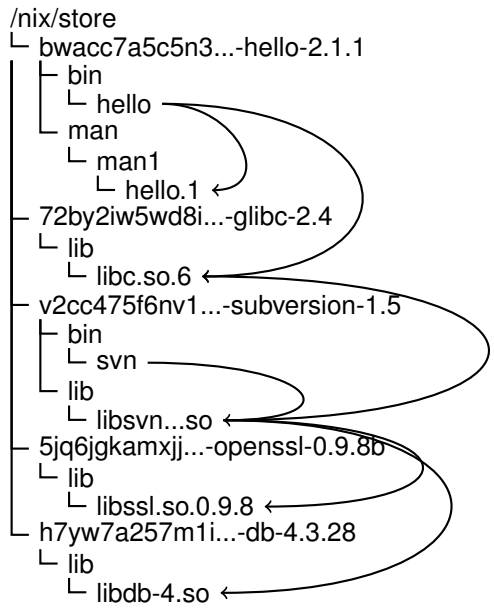


Figure 2.1: Runtime dependencies of the GNU Hello component

```
gcc -o test test.c lib.c -lglib
```

will fail because we have to specify the explicit path to the glib library. If we pass the -L/nix/store/72a81...-glib-2.8.1/lib argument to the compiler the command will succeed because it is able to find a specific glib library.

Nix can detect *runtime dependencies* from a build process. This is done by scanning a component for the hashcode that uniquely identify a component in the Nix store. For instance libraries that are needed to run a specific executable are stored in the RPATH field in the header of the ELF executable, which specifies a list of directories to be searched by the dynamic linker at runtime. The header of the GNU Hello executable contains a path to the standard C library which is: /nix/store/72by2iw...-glibc-2.4/lib/libc.so. By scanning the hello component we know that a specific glibc component in the Nix store is a *runtime dependency* of GNU Hello. Figure 2.1 shows the runtime dependencies of the GNU Hello component in the Nix store.

Nix also 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. By scanning hashcodes in components we always know the *runtime dependencies* of a component and by using hashcodes

themselves we always know the *buildtime dependencies*. So Nix always knows the entire dependency graph of a specific component.

2.3 Purely functional model

The deployment model of Nix is also known as the *purely functional deployment model* because it borrows its concepts from purely functional programming languages such as Haskell [30].

In a purely functional programming language the result of a function call depends exclusively on the definition of the function and the arguments. So it should always give the same result based on its input parameters. Because the result of a function call is always the same based on its input parameters we only have to execute a specific function call once. The next time the same function is called we can return the result from a cache. Lazy evaluation is another feature used particularly in purely functional languages. A function is only executed when it is needed. A purely functional programming language is also a programming language that excludes destructive modifications or updates. In a functional programming language there are no mutable variables but instead there are identifiers referring to immutable, persistent values.

In the Nix deployment model a component depends exclusively on the build inputs. If we build a component on a specific machine and we build the same component with the same inputs on an other machine it should result in exactly the same component. So every build action is fully *deterministic*[1]. This feature is very useful to distribute components to other machines, i.e. distributed builds and gives us guarantees that if it will work on one machine it will also work on other machines. It also gives us other guarantees such as non-interference. A feature of Nix is that because a build action is deterministic we do not have to build the component again. If the same component is already in the Nix store then we can return that component instead of building it again. Another feature is *lazy evaluation*; a component will only be built if it is needed. So if we want to build a certain application component, only the application component and its dependencies are built. Components in the Nix store are also *immutable*. Once a component is realized it can never be changed again. This is done by making all the files and directories read-only in the Nix store [2].

2.4 The Nix expression language

Nix components are built from *Nix expressions*. The Nix expression language is a domain specific language (DSL) for describing component build actions. The Nix expression language is also a purely functional language.

The expression in Figure 2.2 shows a Nix expression which describes the build action for the GNU Hello component.

[1] It has some caveats, see [17].

[2] This is done by giving directories and executable files UNIX permission mode 555 and ordinary files UNIX permission mode 444

```
{stdenv, fetchurl, perl}: 1

stdenv.mkDerivation { 2
  name = "hello-2.1.1"; 3
  builder = ./builder.sh; 4
  src = fetchurl { 5
    url = mirror://gnu/hello/hello-2.1.1.tar.gz;
    md5 = "70c9ccf9fac07f762c24f2df2290784d";
  };
  inherit perl; 6

  meta = {
    description = "GNU Hello, a classic computer science tool";
    homepage = http://www.gnu.org/software/hello/;
  };
}
```

Figure 2.2: pkgs/applications/misc/hello/ex-1/default.nix: Nix expression for GNU Hello

1 States that the expression is a *function* that should be called with three arguments: stdenv, fetchurl and perl. These arguments are dependencies needed to build the component, but we do not know how to build them here. stdenv is a component that nearly every component uses. It contains a standard UNIX environment with a C compiler (gcc), the bash shell and other standard tools such as: cp, gzip, tar, ld, and grep. The function also takes the fetchurl function as its input to download files and perl, the Perl interpreter. The remainder of the file is the body of the function.

2 In the body of the function we call the stdenv.mkDerivation function. Basically this function is a high level wrapper around the builtin Nix derivation function which is more low-level and requires more arguments. The stdenv.mkDerivation function fills a lot of boilerplate code for this low-level function. This function derives the component from its inputs. The derivation function takes an attribute set as its input, which is just a list of key/value pairs where each value is an arbitrary Nix expression.

3 The *name* attribute specifies the symbolic name and version of the component. The name becomes part of the path of the component in the Nix store.

4 The *builder* attribute specifies a script that actually builds the component. Omitting this attribute will result in a standard autotools build action which is usually: ./configure; make; make install

5 The *src* attribute specifies the sources of the component for the builder. In this case the src attribute is bound to the result of a call to the fetchurl function. The fetchurl

```
source $stdenv/setup [7]

PATH=$perl/bin:$PATH [8]

tar xvfz $src [9]
cd hello-*
./configure --prefix=$out [10]
make
make install
```

Figure 2.3: pkgs/applications/misc/hello/ex-1/builder.sh: Builder script for GNU Hello

function takes a URL as its input where it can download the source code. It uses the MD5 hash code to verify whether the downloaded source tarball matches the expected version in order to guarantee that the execution of the function is pure.

[6] The derivation requires a Perl interpreter, so we have to pass the value of the perl function to the builder. The inherit perl; statement is syntactic sugar for perl = perl; which looks a bit silly in this context.

The builder script of the hello world component is illustrated in Figure 2.3, which describes how the build should actually be performed.

[7] When Nix invokes a builder script it clears all environment variables. The only environment variables available are the attributes declared in the derivation. For instance the PATH environment variable is empty.[3] Clearing the environment is done to prevent undeclared inputs being used in the build process. For instance if the PATH variable is set to /usr/bin then the builder might use /usr/bin/gcc instead of the specified GCC variant.

[8] Because the hello component needs Perl we have to put it in the PATH. The perl environment variable points to the location of the Perl component in the Nix store.

[9] We have to unpack the downloaded tarball. The src variable is bound to the result of the fetchurl function which downloads the source tarball. The result of this function is the location of the downloaded tarball in the Nix store.

[10] We have to compile and install the source code. GNU Hello is a typical autotools-based package [36]. The out variable points to the target location of the component in the Nix store. After executing the ./configure, make and make install statements the code of GNU Hello will be compiled and stored in the Nix store.

[3] Actually it is set to /path-not-set

```
rec { 11
    hello = import ../applications/misc/hello { 12
        inherit fetchurl stdenv perl; 13
    };

    perl = import ../development/interpreters/perl { 14
        inherit fetchurl stdenv;
    };

    fetchurl = import ../build-support/fetchurl {
        inherit stdenv; ...
    };

    stdenv = ...;
}
```

Figure 2.4: pkgs/top-level/all-packages.nix: Composition of the Hello package and some other components

Since the Nix expression in figure Figure 2.2 is a function we cannot use it to build and install the component directly. We need to *compose* it, by calling the function with the needed arguments.

Figure 2.4 shows a partial Nix expression which composes all available Nix components. In this file all Nix expressions which describe build actions are imported and called with its expected arguments. This is a large file and in order to make the expression more clear we only show components relevant to the GNU Hello component.

11 This file defines a set of attributes that evaluate to derivations. With the keyword rec we define a *mutually recursive* set of attributes so that attributes can refer to each other in order to compose them together.

12 In this line we import the Nix expression for the Hello component.

13 In this part of the code we compose the component by passing the needed arguments to the function defined in the Nix expression in Figure 2.2. These arguments are defined in the same mutually recursive attribute set.

14 Also the other components are composed by calling the Nix expressions that build them with the appropriate arguments.

In this thesis we call this composition expression described in Figure 2.4 the *packages model*, because it contains a collection of packages that we can deploy on a specific system.

2.5 Store derivations

Components are not built from Nix expressions but from *store derivation files*. The Nix expression language is not used directly because it is fairly high level and subject to evolution. The language might change in the future to support new features. We have to deal with all the variability which is very inflexible. The second thing is that the richness of the language is nice for users but complicates the operations we want to perform e.g. building and installing. Third, it is difficult to uniquely identify a Nix expression. Nix expressions can import other Nix expression that are located on the filesystem. It is not a trivial operation to generate an identifier, e.g. a cryptographic hash that identifies this expression.

For all these reasons Nix expressions are not built directly but translated to *store derivation* files which is encoded in a more primitive language. A store derivation describes the build action of one single component. Store derivation files are also placed in the Nix store and have a store path as well. The advantage of keeping store derivation files inside the Nix store gives us a identification of source deployment objects.

```
Derive(
  [("out", "/nix/store/a36clhh1b3f3pmh8w1fxnzkp3q1l6zny-hello-2.3", "", "")] 15
, [ ("/nix/store/0n7qbzi7k3rblfkayifdwmkz5waxxzar-hello-2.3.tar.bz2.drv", ["out"]) 16
  , ("/nix/store/71i28yc3qyavldq30n9bh73k5jqxbfbx-bash-3.2-p39.drv", ["out"])
  , ("/nix/store/wkqapywzvfshm1rk9zmj70mzbmsld54z-stdenv-linux.drv", ["out"])
  ]
, ["/nix/store/9krlzvny65gdc8s7kpb6lkx8cd02c25b-default-builder.sh"] 17
, "i686-linux" 18
, "/nix/store/mm631h09mj964hm9q04l5fd8vw12j1mm-bash-3.2-p39/bin/sh" 19
, ["-e", "/nix/store/9krlzvny65gdc8s7kpb6lkx8cd02c25b-default-builder.sh"] 20
, [ ("builder", "/nix/store/mm631h09mj964hm9q04l5fd8vw12j1mm-bash-3.2-p39/bin/sh") 21
  , ("name", "hello-2.3")
  , ("out", "/nix/store/a36clhh1b3f3pmh8w1fxnzkp3q1l6zny-hello-2.3")
  , ("src", "/nix/store/vgaqghyhgk3apb503q5483v8cmn32ggm-hello-2.3.tar.bz2")
  , ("stdenv", "/nix/store/w8gq89c4asjlvhzsy9xvss55c13g020v-stdenv-linux")
  , ("system", "i686-linux")
  ]
)
```

Figure 2.5: Store derivation file of the GNU Hello component

Figure 2.5 illustrates the store derivation file for the GNU Hello component.

15 Specifies the Nix store path that will be built by this derivation.

16 Specifies all the paths of the input derivations. These derivations should be built first before we can build this component.

17 Contains all the paths of all sources in the Nix store. For instance the builder script mentioned in Figure 2.2 is copied in the Nix store, since everything has to be in the Nix store to prevent undeclared dependencies.

18 This attribute specifies the hardware and operating system on which the derivation should be built.

19 Specifies the builder program to be executed and 20 specifies the command-line arguments that should be passed to it. In this case we start a shell that will execute the builder script in the Nix store.

21 Specifies all the environment variables that should be passed to the builder process.

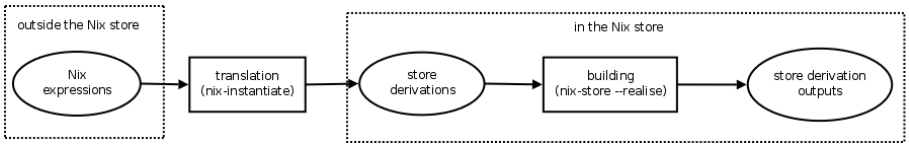


Figure 2.6: Two-stage building of Nix expressions

Figure 2.6 illustrates the two-stage build process. The Nix expressions are translated into a set of store derivation files that are put in the Nix store. These store derivation files can be realized, which results in that the build action described in the store derivation is executed. The outputs of this build action also live in the Nix store.

By calling the nix-instantiate command it is possible to translate a specific Nix expression into store derivation files. For instance:

```
$ nix-instantiate hello.nix
/nix/store/1ja1w63...-hello-2.1.1.drv
```

After executing the nix-instantiate command it will return a list of store derivation files that it has generated from it. By executing nix-store –realise command it is possible to execute the build action which is described in a store derivation file. For instance:

```
$ nix-store --realise /nix/store/1ja1w63...-hello-2.1.1.drv
/nix/store/bwacc7a...-hello-2.1.1
```

By executing the instruction above the build operations for the GNU hello component are executed. If the build action is finished it will return the store path to the resulting component that it has built.

2.6 Nix profiles

Storing components in a Nix store in isolation from each other is a nice feature, but not very user friendly to end users. For instance, if a user wants to start the program hello it has to specify the full path to the hello executable in the Nix store, e.g. /nix/store/bwacc7a5c5n3...-hello/bin/hello.

To make components more accessible for users Nix provides the concept of *Nix profiles* which are specific user environments that abstract over store paths. By adding the

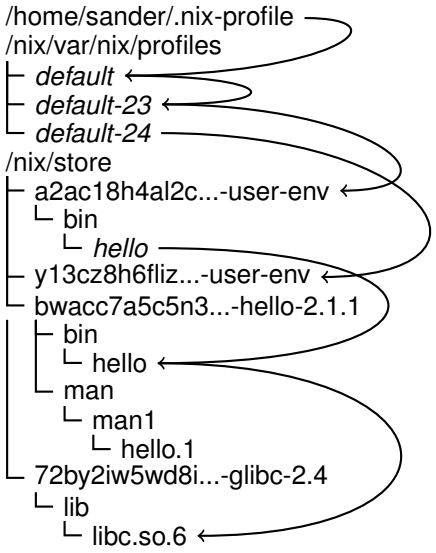


Figure 2.7: An example Nix profile that contains the GNU Hello component

/home/user/.nix-profile/bin directory to the PATH environment variable which resolves indirectly to the store paths of each component the user can start the hello component activated in its profile by just typing: hello. This is illustrated in Figure 2.7.

The set of available components in the environment of the user, can be modified using the nix-env command. To add a component to the Nix environment, for instance the Hello component, the user can type:

```
$ nix-env -f all-packages.nix -i hello
```

By executing the command above the component will be build and in the Nix profile of the user symlinks are made that point to the files of the GNU Hello component.

In order to upgrade the hello component the user can type:

```
$ nix-env -f all-packages.nix -u hello
```

In order to uninstall the hello component the user can type:

```
$ nix-env -f all-packages.nix -e hello
```

The uninstall operation mentioned above does not delete the specific Hello component from the Nix store. It just removes the symlinks to the hello components files from the environment. The Hello component could be in use by another Nix profile of another user or still being a running process, thus it is not safe to remove them. The Nix garbage collector, which has to be called explicitly, checks whether a component is in use and if it is not in use it will safely deletes them from the Nix store.

If it turns out that the new version of the GNU Hello component does not meet our requirements or we regret that we have uninstalled the GNU Hello component, we can rollback to a previous version of the user environment, by typing:

```
$ nix-env --rollback
```

2.7 Atomic commits

The deployment steps in Nix are atomic: at any point in time, issuing the command hello either runs the old or the new version of GNU Hello, but never an inconsistent mix of the two. The user's PATH environment variable contains the path /nix/var/nix/profiles/default/bin, which — via some indirections through symbolic links — resolves to a directory containing symlinks to the currently "installed" programs. Thus, /nix/var/nix/profiles/default/bin/hello resolves to the currently active version of GNU Hello. By flipping the symlink /nix/var/nix/-profiles/default from default-23 to default-24, we can switch to the new configuration. Since replacing a symlink can be done atomically on UNIX, upgrading can be done atomically.

Thus, Nix achieves atomicity of package upgrades due to two properties: first, the new version of the package is built and stored separately, not interfering with the currently active version; and second, the new version is activated in an atomic step by updating a single symlink.

2.8 Garbage collection

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

The garbage collector can be invoked by typing:

```
$ nix-collect-garbage
```

It is also possible to remove all older derivations of the user profile and garbage with it, by typing:

```
$ nix-collect-garbage -d
```

This command also removes the older default-23 symlink and also removes all the garbage with it.

2.9 Nixpkgs

The Nix packages collection (Nixpkgs) is a distribution of Nix expressions for many existing Unix and Linux components [22]. The Nix packages collection contains many compo-

nents found in other Linux distributions ranging from fundamental components such as the Bash shell, GCC and the C library Glibc to user applications like Firefox and Eclipse.

In this project we have reused as many Nix expressions for open source components as possible. We also contributed several Nix expressions for open source components that the SDS2 platform uses, such as the *Google Web Toolkit*, *ejabberd* and *Apache Tomcat* which were not part of the Nixpkgs collection yet.

2.10 NixOS

The Nix package manager is also the basis for an experimental Linux distribution called NixOS [17].

In NixOS all components are stored in the Nix store including all operating system specific parts, such as the Linux kernel, additional kernel modules, bootloader and configuration files such as the httpd.conf which is a configuration file for the Apache webserver. Nix has almost no impure directories. NixOS has no /usr, /lib, /opt and a very minimal /bin and /etc. The /bin/sh symlink is a notable exception, because the standard C library system() function requires to have a shell at /bin/sh. NixOS has a minimal /etc directory which only contains configuration files which should be shared across multiple components. For instance the /etc/passwd file which contains all user accounts is shared across multiple applications. For the non-operating system parts NixOS uses packages from the Nixpkgs collection.

On ordinary systems such as regular Linux distributions it is still possible to call components outside the Nix store. For instance it is still possible to call /usr/bin/gcc from a builder script which makes a build impure. Because all packages on NixOS are pure and we have only a minimal set of impure files outside the Nix store, we have better guarantees that if our package works on NixOS that it is also pure.

Just as packages in the Nix package manager can be built from one declarative specification, NixOS allows us the build an entire system from one declarative expression.

The entire configuration of NixOS is built from a configuration top-level expression which is illustrated in Figure 2.8.

[22] Specifies on what partition the GRUB bootloader should be installed. In this case this is the master boot record of this first harddrive /dev/sda.

[23] Specifies what filesystems should be mounted on a specific mount point. In this case partition /dev/sda2 will be mounted as the root partition.

[24] Specifies all available swap devices. The /dev/sda1 device is a swap partition.

[25] Specifies all network settings. In this example we add an extra entry to the /etc/hosts configuration file.

[26] Specifies all services settings. In this example we enable a SSH server, a X Window System server and a MySQL server. The X Window System starts a KDE desktop session.

```
{
  boot = {
    grubDevice = "/dev/sda"; 22
  };

  fileSystems = [ 23
    { mountPoint = "/";
      device = "/dev/sda2";
    }
  ];

  swapDevices = [ 24
    { device = "/dev/sda1"; }
  ];

  networking = { 25
    extraHosts = "127.0.0.2 nixos.localhost nixos";
  };

  services = { 26
    sshd = {
      enable = true;
    };

    xserver = {
      enable = true;
      sessionType = "kde";
    };

    mysql = {
      enable = true;
    };
  };
}
```

Figure 2.8: configuration.nix: A top level configuration file for NixOS

Configuration changes are also non-destructive just like ordinary Nix packages because everything is stored side-by-side in the Nix store. With this approach one can easily roll back to a previous configuration. The GRUB bootloader in NixOS allows the user to boot into any previous system configuration that has not been garbage collected. The command nixos-rebuild builds and activates a new system configuration.

For service activation and deactivation NixOS uses the upstart daemon, which is also used by other Linux distributions such as Ubuntu and Fedora [44]. Upstart is an event-based replacement for the tradional init daemon on UNIX systems. Traditional init usually starts processes sequentially, whereas upstart spawns every process based on events.

The /etc/event.d directory of NixOS contains service description files that can be spawned by the upstart daemons. Service description files are built from Nix expressions [16].

Figure 2.9 illustrates a Nix expression that generates a Upstart service file for MySQL:

27 is a function which takes the configuration parameters and an attribute set that describes all the package compositions as inputs.

```
{pkgs, config}: 27

let
  cfg = config.services.mysql;
  mysql = pkgs.mysql;
  pidFile = "${cfg.pidDir}/mysqld.pid";
  mysqldOptions =
    "--user=${cfg.user} --datadir=${cfg.dataDir} " +
    "--log-error=${cfg.logError} --pid-file=${pidFile}";
in

{
  name = "mysql"; 28

  users = [ 29
    { name = "mysql";
      description = "MySQL server user";
    }
  ];

  extraPath = [mysql];

  job = '' 30
    description "MySQL server"

    stop on shutdown

    start script 31
        if ! test -e ${cfg.dataDir}; then
            mkdir -m 0700 -p ${cfg.dataDir}
            chown -R ${cfg.user} ${cfg.dataDir}
            ${mysql}/bin/mysql_install_db ${mysqldOptions}
        fi

        mkdir -m 0700 -p ${cfg.pidDir}
        chown -R ${cfg.user} ${cfg.pidDir}
    end script

    respawn ${mysql}/bin/mysqld ${mysqldOptions} 32

    stop script 33
        pid=$(cat ${pidFile})
        kill "$pid"
        ${mysql}/bin/mysql_waitpid "$pid" 1000
    end script
  '';
}
```

Figure 2.9: mysql.nix: Nix expression for generating the MySQL service job

28 Specifies the name of the Upstart job. This name becomes part of the name of the service component in the Nix store.

29 The MySQL server should run as a non-privileged user instead as super user for security reasons. This attribute specifies the user account under which the MySQL server runs.

30 This attribute is a string which describes the definition of the Upstart service file.

31 In order to run MySQL we also need *state data*, e.g. an initial empty database. This part of the job creates an initial empty database for MySQL if there is no initial state data available. By using this construction services are *self initializing*.

32 This line actually starts the MySQL server daemon. By using the respawn statement the Upstart daemon will start the process again if it is terminated.

33 We also have to stop some subprocesses that the MySQL server daemon spawned. This part will clean all the subprocesses after the MySQL server upstart job is stopped.

Chapter 3

The Service Development Support System (SDS2)

In this research project we use as a case study an existing distributed system which is developed at the Healthcare Systems Architecture [35] department. The system is called SDS2 which is an acronym for Service Development Support System. In this chapter we explain the ideas behind SDS2, of what parts SDS2 consists of, and what technologies are used.

3.1 Information as a service

Hospital environments consist of many devices ranging from workstations to medical equipment such as MRI scanners as illustrated in Figure 3.1. Medical devices produce lots of data such as status logs, maintenance logs, patient alarms, images, and measurements. The amount of data that is produced by these devices is often huge and data structures have an implicit structure, e.g. logfiles.

The stakeholders in hospitals often need valuable information from these huge data sets. A solution to provide valuable information from these datasets is making specific types of information available through a service. So basically a service is an abstraction of all these data sets that will turn these datasets in something more useful. Some of these abstraction services are:

Figure 3.1: A broad range of medical devices

- *Analysis and prediction services*. Find patterns in data and use these to e.g., predict occurrences of interesting events
- *Data composition services*. Synthesize useful information by combining different data sources, e.g., preventive maintenance indicator from event log and maintenance log.
- *Location aware services*. Provide information based on location or movements of artifacts.

3.2 Service oriented architecture

A service oriented architecture (SOA) addresses the requirements of loosely coupled standards-based and protocol independent distributed computing [39]. In a SOA software resources are presented as "services". Services are well defined, self-controlled modules that provide business functionality and are independent of the state or context of other services. In a SOA, all services are autonomous. Their operation is perceived as opaque by external components. This guarantees that external components do not have to care about how the operation is executed by the services. It is also irrelevant in a SOA whether a component is local or remote because their interfaces can be invoked trough a network or loopback interface, but performance typically differs.

Some technologies that are used to build a SOA are:

- *SOAP*. SOAP originally stood for Simple Object Access Protocol (SOAP) and it is a light-weight protocol allowing RPC-like calls which are formatted in XML over the internet by using various transport protocols. Examples of transport protocols that can be used with soap are HTTP which is also used to serve websites, SMTP which is used for sending e-mail and XMPP which is used by a Jabber chat application. The HTTP protocol has wider acceptance as it works well with today's Internet infrastructure.
- *WSDL*. The Web Services Description Language (WSDL) is a XML-based interface description language to model a Web service, which is a software system designed to support interoperable machine-to-machine interaction over a network.
- *UDDI*. Universal Description Discovery and Integration is a platform-independent, XML-based registry for businesses to list themselves on the Internet. It is designed to be interrogated by SOAP messages and to provide access to WSDL documents.

The technologies listed above are widely used, but other technologies can be used to build SOAs as well.

In the SDS2 platform, web services provide the role of transformers. Each transformer creates a specific abstraction layer of the huge implicit datasets stored in the data repositories, such as logfiles. By combining these transformers they will produce more useful and specific information that can be presented by one of the front-end applications to the end user. This approach is illustrated in Figure 3.2.

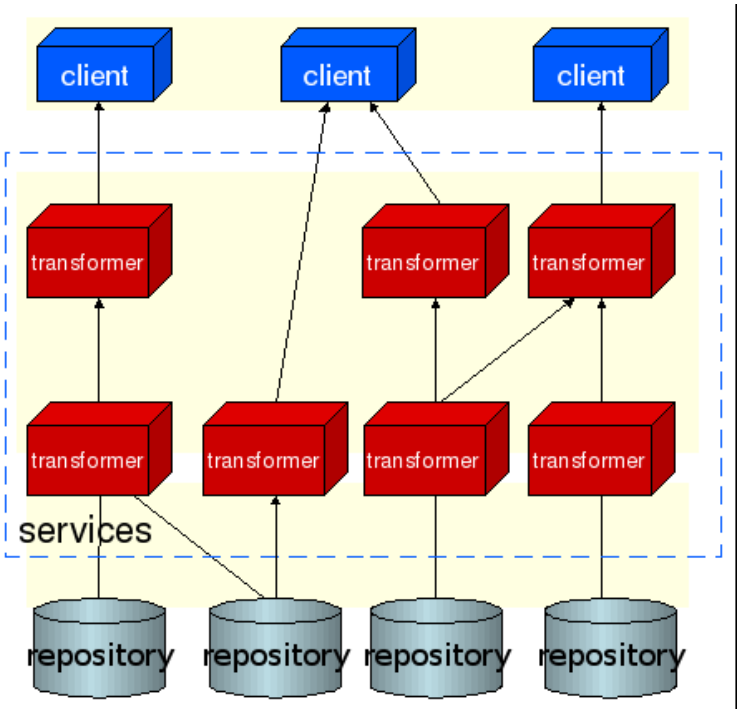


Figure 3.2: Transformers in SDS2

3.3 Tracking devices in a hospital environment

In SDS2 hospital devices are generating location events and status events. These events are sent to a nearby XMPP server or possibly more XMPP servers. These events are broadcasted in communities, which are channels where events are broadcasted on. There are also transformers connected to the XMPP server that listen to events in these communities. Transformers store these events in the database or respond on the events as illustrated in Figure 3.3.

3.4 Stakeholders

The SDS2 front-end applications provide specific abstractions on location and status events logs of medical devices in a hospital environment based on questions the stakeholders have. Currently the SDS2 provides information for the *nurse*, the *field service engineer* and the *hospital administrator*. Some questions for each stakeholder are:

- *Nurse*. Where is the device when I need one? Why is it not starting up? Who do I contact when I need assistance?
- *Field service engineer*. What is the current status of the device? How is the device being used/abused? What errors did the device report? When?

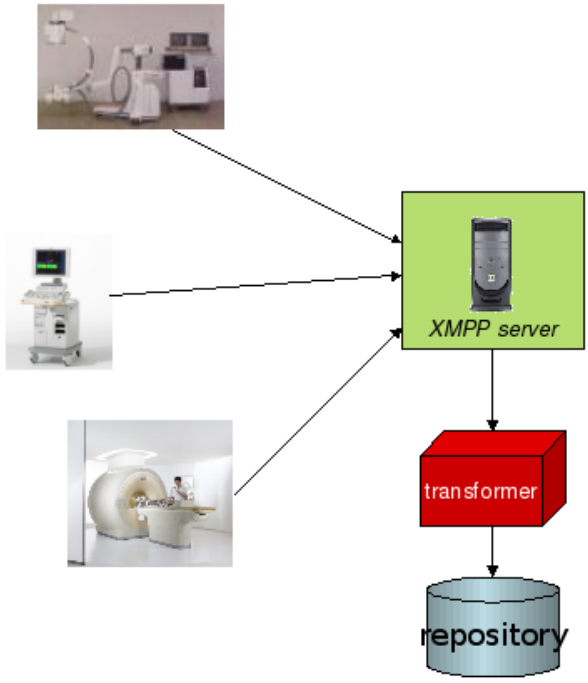


Figure 3.3: Live data in SDS2

- *Hospital administrator*. How does the performance compare to other hospitals? How often has the device been used? How efficient is the workflow of my devices?

3.5 Front-end applications

The SDS2 system currently provides two front-end applications for the stakeholders; the *Asset Tracker Service* and the *Utilisation Service*.

3.5.1 Asset Tracker Service

One of the front-end applications is the Asset Tracker Service. This application can be used by two stakeholders: the *nurse* and the *field service engineer*.

The asset tracker shows floor maps with medical devices on it. The user can pick a floor from the menu and the application shows all the devices that are on the floor with their status. The locations and status of devices are updated constantly by listening to location and status events that are broadcast by devices on the XMPP channel.

An impression of the Asset Tracker Service can be seen in Figure 3.4. Currently it shows the floormap with identifier HTC37.1 (which is the first floor of the Philips Research building with number 37 on the High Tech Campus) and devices that are located on that floor. It also shows us that three devices are broken and need maintenance. They are marked

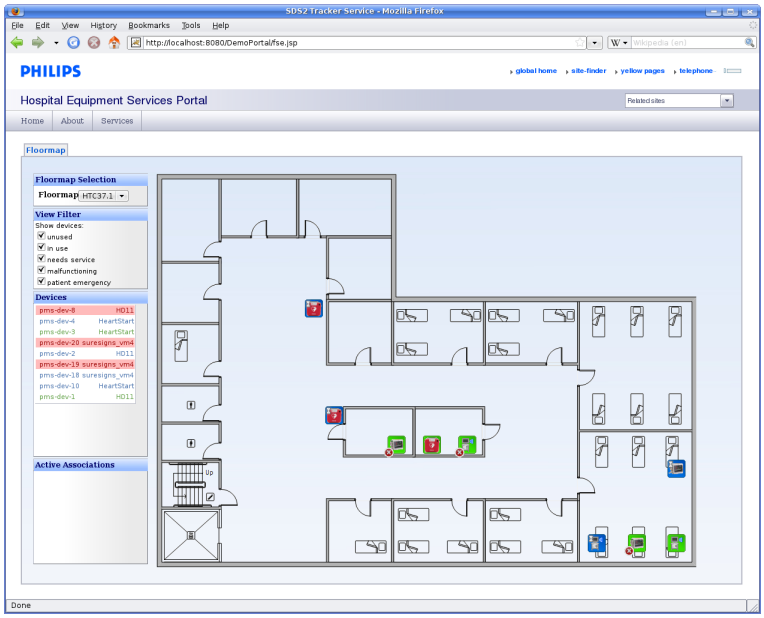


Figure 3.4: The Asset Tracker Service in action

with a red color in the list and are marked with a red icon with a cross on it on the floor map.

The field service engineer can also pick a device from the floor map and query a maintenance log and event log for a particular device. Figure 3.5 shows a maintenance log for a device with identifier pms-dev-3 on the floor map.

3.5.2 Utilisation Service

The other front-end application is the Utilisation Service. This application can be used by the *hospital administrator*.

The Utilisation Service also shows a floormap which the user can pick from the menu. For each floor the user can query specific statistics for a particular device, a group of devices or for all devices. In Figure 3.6 we queried all followed paths of all the HeartStart devices from 2007-01-01 till 2007-01-02. Figure 3.7 shows all the usage hours in each room of the device with identifier pms-dev-3 from 2007-01-01 till 2007-01-02.

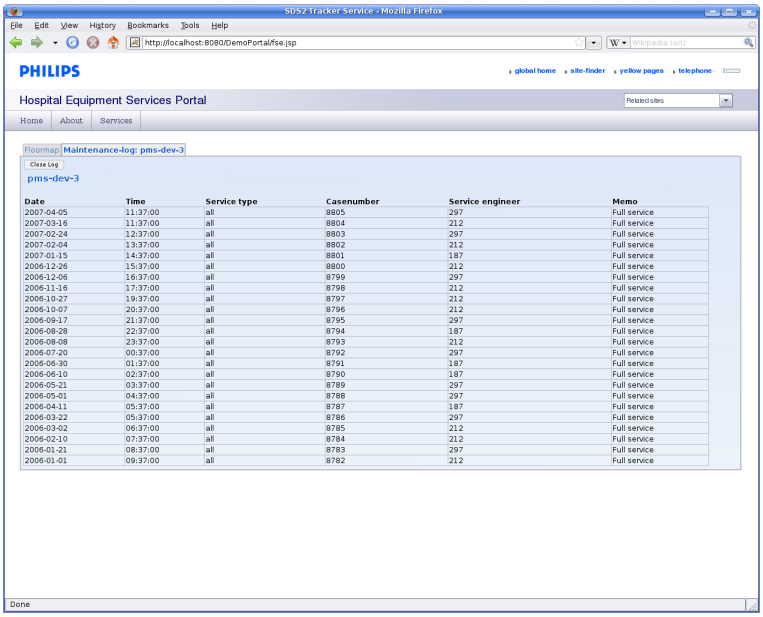


Figure 3.5: Maintenance log of the pms-dev-3 device

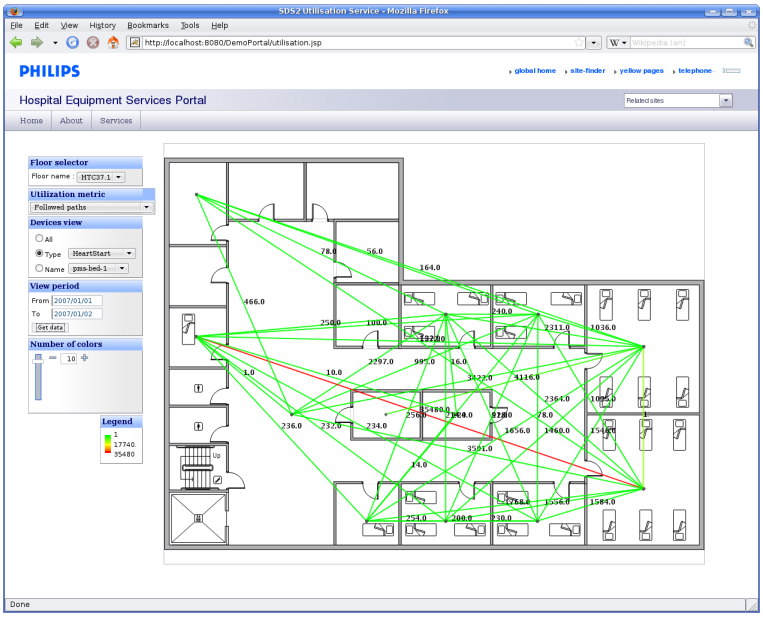


Figure 3.6: Followed paths of all HeartStart devices

Figure 3.7: Usage hours in each room for the device with identifier pms-dev-3

3.6 Deployment view

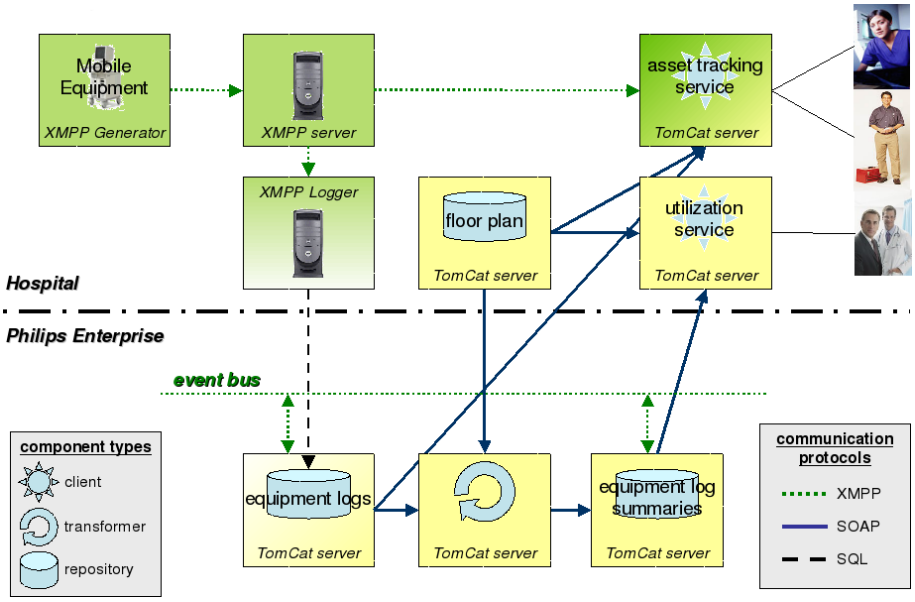


Figure 3.8: Deployment view of SDS2

Figure 3.8 shows the deployment view of SDS2 with the major components. The components run on different machines on different sites. Some components are located in the hospital and some components are located in the Philips Enterprise environment.

All live data, i.e. status and location events, are generated by the mobile equipment in the hospital, which is the *XMPP Generator* component at the top left in Figure 3.8. Mobile equipment devices are generating location and status events which are sent to a nearby XMPP server.

The Extensible Messaging and Presence Protocol (XMPP) is an open protocol, which uses XML to encode messages and uses HTTP as transport protocol [43]. It is the core protocol of the Jabber Instant Messaging and Presence technology. One strength of the XMPP protocol is decentralization. There is no central master server and anyone can run their own XMPP server. It is an open system where anyone who has a domain name and a suitable internet connection can run their own XMPP server and talk to users on other servers, which is similar to e-mail.

SDS2 uses the XMPP protocol for sending events. XMPP is a location independent protocol. In a hospital environment there could be several XMPP servers on different locations that capture location and status events. All these XMPP servers are connected to each other. By using this approach it is possible to address every device in the hospital independent of its location.

The *XMPP Logger* component listens to status and location events on the XMPP server and stores them in the *equipment logs* repository. The XMPP logger uses a database to store

all the events and updates them by sending SQL instructions through a TCP connection.

From the huge repository of stored events summaries are generated by a transformer which uses the *Floorplan service* to map location events to floors. The generated summaries are stored in the *equipment log summaries* repository.

The *asset tracking service* front-end application displays events by listening to status and location events on the XMPP server and can display event logs by invoking the *equipment logs* service. The *utilisation service* displays utilisation statistics by invoking the *equipment log summaries* service. All communication with the services is done by using the *SOAP* protocol which is a standard protocol for communicating with web services. In SDS2 the HTTP protocol is used as a transport protocol, because it is the most widely used transport protocol for SOAP.

3.7 Technologies

The SDS2 platform uses several technologies:

- *Java*. All the components of SDS2 are written in the Java programming language.
- *Apache Tomcat*. All the web applications are hosted on a servlet container.
- *Apache Axis2*. All the web services are hosted in a web service container. The Axis2 web service container itself is hosted on *Apache Tomcat*.
- *Ejabberd*. An open-source XMPP server where all the location and status events are broadcasted.
- *Smack*. XMPP client library written in Java.
- *MySQL*. Open source database which is used for storing the mobile events, summaries, maintenance logs and hospital installed base records.
- *Google Web Toolkit (GWT)*. An open source Java software development framework to create Ajax applications in Java. The asset tracker and utilisation services are GWT applications.

3.8 Architecture of SDS2

The SDS2 system consist of several major components, that we categorize in three groups, *applications* which are non-interactive programs that generate and store data, *web services* which provides abstractions over the huge implicit datasets and *web applications* which are the front-ends for the stakeholders. The components and its dependencies are visible in Figure 3.9.

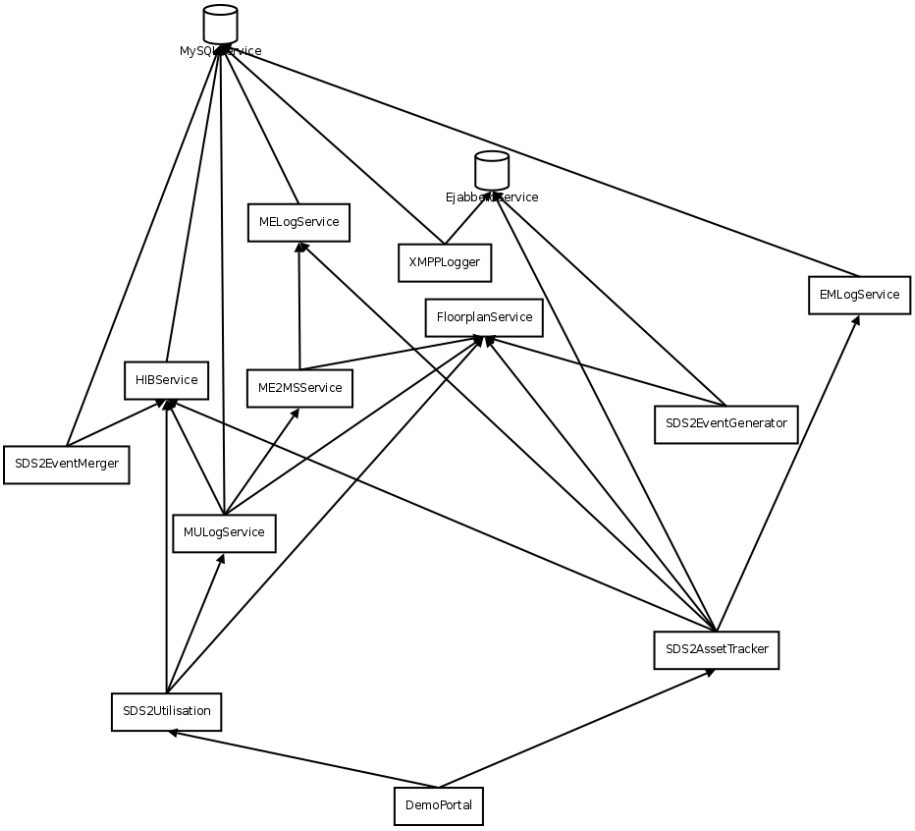


Figure 3.9: Inter-dependency graph of SDS2

Applications:

- *Event generator* (SDS2EventGenerator). Simulates the behavior of hospital equipment and their movements on a hospital floor. The status and location events that the devices generate are broadcasted on an XMPP channel. It uses the Floorplan Service to calculate position information of the assets.
- *XMPP logger* (XMPPLogger). Captures location and status events on the XMPP channel and stores them in a MySQL database.
- *Event merger* (EventMerger). Associates for each device a status event to a location event in the MySQL database, so that it is known where a status change happended. It invokes the *HIBService* to get a list of available devices.

Web services:

- *Hospital Installed Base Service* (HIBService). Provides a description of a device in a hospital based on its identification string. All device records are stored in a MySQL database.

- *Floorplan Service* (FloorPlanService). Translates world coordinates to floor plan coordinates and provides floor names and floormap images.
- *Mobile Event Log Service* (MELogService). Queries a specific set of events of a device in a given time period in the MySQL database.
- *Mobile Utilisation Log Service* (MULogService). Queries a specific set of summaries of a device in a given time period and caches the summaries in a MySQL database. Summaries are based on a set of events, which are generated by the *ME2MSService* component. It invokes the *HIBService* and *FloorPlanService* components to calculate various statistics, for instance the traveled paths.
- *Mobile Event to Mobile Utilisation Service* (ME2MSService). Generates summaries of sets of mobile events. It uses the *FloorPlanService* to translate world coordinates to floorplan coordinates.
- *Equipment Maintenance Log Service* (EMLogService). Provides a log of maintenance records of each device which are provided by the field service engineer. The maintenance logs are stored in a MySQL database.

Web applications:

- *Asset Tracker* (SDS2AssetTracker). Shows a floormap where the user can see the location and status of devices in the hospital. The floormaps are provided by the *FloorPlanService* component and a list of available devices is provided by the *HIBService* component. The locations and statuses are updated constantly by listening to status and location events that are broadcasted by the XMPP server. The field service engineer can query the maintenance and event logs for each device, which are provided by the *EMLogService* and *MELogService* components.
- *Utilisation* (SDS2Utilisation). Shows a floormap where the user can query various usage statistics of hospital devices e.g. the followed paths, hours of usage and so on. The floormaps are provided by the *FloorPlanService* component, the utilisation statistics are provided by the *MULogService* component and a list available devices is provided by the *HIBService* component.
- *DemoPortal*. A webportal where the stakeholder can pick its application to use which is either the Asset Tracker or the Utilisation service.

Chapter 4

Modeling the SDS2 platform in Nix

The first part of this research project is modeling all the SDS2 components and all its dependencies in Nix. Making the SDS2 platform automatically deployable with Nix on single systems is a precondition in this project in order to make it automatically deployable on distributed systems.

Before adapting SDS2 and creating Nix expressions for SDS2, the software deployment process of SDS2 was a *semi-automatic* process. Some steps were done automatically, e.g. compiling the source code and packaging the files and some steps were done manually, e.g. deploying the web application on Apache Tomcat.

In this chapter we explain how we used the Nix deployment system to automate this deployment process and what modifications we made to SDS2 in order to make it automatically deployable on NixOS. In this project we use NixOS as the deployment platform because it is a good way to create and test Nix packages because it provides better guarantees that a package build action is pure.

4.1 The original deployment process

The SDS2 platform is developed using the Eclipse IDE [21]. The semi-automatic deployment process of SDS2 consists of the following steps, which are described in the SDS2 installation manual [12]. The manual describes all steps to install the development platform on which SDS2 is developed, how to build all the components and how to activate them. All steps in this document are supposed to be executed on the Microsoft Windows XP operating system.

4.1.1 Deployment steps

The first part of the document describes how to install the Eclipse IDE with its required plugins, the Apache Tomcat server and the Apache Axis2 container. It consists of the following steps:

- Installing the Java Development Kit by executing the steps of the installation wizard
- Installing the Eclipse IDE by unpacking the Eclipse IDE zip file

- Installing the Cypal Studio Eclipse plug-in which enables IDE features for GWT applications
- Installing Apache Tomcat by executing the steps of the installation wizard
- Installing Apache Axis2 on Apache Tomcat by copying the axis2.war web application archive to the webapps folder of Apache Tomcat

The second part of the document describes how to install and configure the MySQL server. This process consists of the following steps:

- Installing the MySQL server by executing the steps of the installation wizard. Some of these steps also let the user configure the MySQL server settings, such as the user accounts and the host and port where the server should be listening on.
- Installing the MySQL GUI front-end by executing the steps of the installation wizard.
- Filling the database with the initial database schema of SDS2 by importing a MySQL dump with the MySQL GUI front-end

The third part of the document describes how to install and configure the ejabberd server, which basically consists of the following steps:

- Installing the ejabberd server by executing the steps of the installation wizard.
- Enabling the shared_roster setting by editing the conf/ejabberd.cfg configuration file
- Importing all user accounts by restoring an ejabberd dump file

The fourth part is building all the SDS2 platform components. Most of these steps are executed from within the Eclipse IDE. Many steps have to be performed by hand since most tasks such as generating application archives should by executed explicitly by the user. Only the compilation of Java source files to Java bytecode files is done automatically. The process consists of the following steps:

- Unpacking all the library archives in the packages directory.
- Compiling the config component. This component contains all configurations settings such as the location of the MySQL server, ejabberd server and web services where SDS2 consists of. It also contains user account settings and so on. All SDS2 platform components have a intra-dependency on this config component. The SDS2 platform provides a few predefined config components. For instance one configuration is called devlocalhost which is a configuration that can be used to deploy all components on one single system.
- Compiling and installing Axis2 archives for all web services which are mentioned in Section 3.8, which consists of the steps:

  - Executing generate.service.aar target the Apache Ant build file which generates an AAR file. This step is done by using a Apache Ant file, since there is no feature in the Eclipse IDE that supports generating AAR files at the time writing the installation document.
  - Copying the resulting AAR file in the webapps/axis2/WEB-INF/services directory of Apache Tomcat

- Compiling and installing web application archives for all web applications mentioned in Section 3.8, which consists of the steps:
  - Open the GWT hosted mode dialog
  - Press the compile button in the dialog
  - Right click on the project and export it as a web application archive (WAR) file
  - Copy the exported WAR file to the webapps directory of Apache Tomcat

- Compiling and installing the console applications for all applications mentioned in Section 3.8, which consists of the steps:
  - Right click on the project and select 'export FAT jar'
  - Pick the main class from the project
  - Removing the _fat postfix from the filename
  - Finally click on the 'Finish' button to generate the JAR file

Finally we have to *activate* the entire SDS2 platform. This process consists of the following steps:

- Start the ejabberd server
- Start the MySQL server
- Start the Apache Tomcat server
- Start the three console applications by executing the Windows batch files

4.1.2 Complexity

Figure 4.2 shows the number of tasks that should be executed manually by the user. In order to make the *entire* SDS2 platform available for use a total number of 51 tasks should be performed where each task consists of a few mouse clicks and filling some data fields by the user. Deploying the entire SDS2 platform from scratch took me about 3,5 hours on my computer at HSA site.

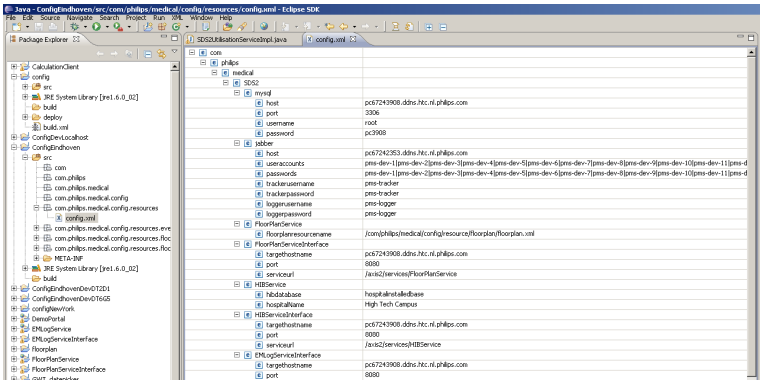


Figure 4.1: The semi-automatic deployment process of SDS2 by using the Eclipse IDE

Section	Description	Number of manual tasks
1	Installing Eclipse IDE, Apache Tomcat, Axis2	5
2	Installing MySQL server, GUI and schema	3
3	Installing and configuring ejabberd	3
4	Building SDS2 platform components	2
4a	Building web service components	12
4b	Building web application components	8
4c	Building console application components	12
5	Activation of SDS2 platform	6
	Total number of tasks	51

Figure 4.2: Number of manual tasks of the semi-automatic deployment process of SDS2

4.2 Automating the deployment process

The original deployment process of SDS2 is a *complex* process. It requires the user to execute many manual deployment tasks which should be executed in the right order and conform to the specification in the installation document. It is difficult to move a component from one machine to another and still have a working system and so on. It is also very *time consuming* to make the entire SDS2 platform ready for use, since many tasks have to be performed manually. The manual deployment process is also *error prone*, as many steps are done manually, we also have a chance on errors. Finally, we have to keep the installation document up to date in order to get a working system configuration. The deployment process becomes more difficult when this document is lost or outdated.

In order to make the deployment process of SDS2 fully automatic using Nix, we have to create Nix components for all SDS2 platform components, related libraries and service components. We have to modify SDS2 so that it can be built in a non-interactivate way, e.g. creating Apache Ant files for each project so that we do not have to build them by using

the Eclipse IDE. SDS2 needs some minor modifications in the code of, such as porting Windows specific features so that SDS2 will run on NixOS. Finally we have to create Nix expressions for all SDS2 components.

In the next sections we will explain these steps in detail.

4.3 Modeling infrastructure components

The infrastructure components that the SDS2 platform uses are:

- *ejabberd*. The XMPP server where all status and location events generated by devices in the hospital are broadcasted on.
- *Apache Tomcat*. A servlet container in which the web applications are running.
- *Apache Axis2*. A Java-based SOAP library and web service container in which all web service components are running.
- *MySQL*. Stores all the mobile event logs, mobile utilisation logs, hospital installed base records and equipment maintenance logs.

All the infrastructure components are open-source and the Nix expressions we developed for these components are part of the Nixpkgs project.

4.3.1 Ejabberd

ejabberd is an open-source XMPP application server, written in the Erlang programming language [23]. In order to get a working ejabberd service we have to make Nix packages for the Erlang compiler and for the ejabberd server.

Creating Nix expressions for these components is a trivial task because the build process is Autotools based. The only thing we had to do is fix some hardcoded references such as the erlang interpreter, which is hardcoded to a unspecified `erl` command which is impure. The location of the erlang interpreter is fixed by running a replace command with the sed editor on the configuration template [34]. Another non-standard behavior step is that ejabberd is autotools based, but the source code resides in the `src` directory of the source code tree. In order to enter that directory first we had to implement a pre-configure hook which is described in [35].

In order to run ejabberd we have to create an Upstart [44] service component for NixOS that activates and deactivates it. The service component also specifies the locations of the state data of ejabberd which are the `/var/lib/ejabberd` and `/var/log/ejabberd` directories. The log files and account settings database should be stored outside the Nix store because parts in the Nix store are immutable.

4.3.2 Apache Tomcat and Apache Axis2

Apache Tomcat is an implementation of the Java Servlet and JavaServer Pages technologies [5]. The Apache Tomcat server is a container server in which Java web applications

```
{stdenv, fetchurl, expat, erlang, zlib, openssl}:

stdenv.mkDerivation {
  name = "ejabberd-2.0.0";
  src = fetchurl {
    url = http://www.process-one.net/downloads/ejabberd/2.0.0/ejabberd-2.0.0.tar.gz;
    sha256 = "086e105cb402ef868e3187277db1486807e1b34a2e3e3679f0ee6de1e5fd2e54";
  };
  buildInputs = [ expat erlang zlib openssl ];
  patchPhase = ''
    sed -i -e "s|erl \\\|${erlang}/bin/erl \\\|" src/ejabberdctl.template 34
  '';
  preConfigure = '' 35
    cd src
  '';

  meta = {
    description = "Open-source XMPP application server written in Erlang";
    license = "GPLv2";
    homepage = http://www.ejabberd.im;
  };
}
```

Figure 4.3: Nix expression for Ejabberd

can be deployed and Tomcat provides the resources for them. The filesystem structure of the Apache Tomcat component is listed in Figure 4.4.

bin	Contains all executables to start and stop the Apache Tomcat server
conf	Contains all configuration files for the Apache Tomcat server
lib	Contains all the Java libraries that are needed to run the Servlet container
shared/lib	Contains Java libraries that are shared by all web applications that are deployed on this server
common/lib	Contains Java libraries that are shared by both the Servlet container and all web applications deployed on this server
logs	Contains all logfiles
temp	Contains all temporary files
webapps	Directory in which web applications should be stored. By putting WAR files in this directory the WAR archive will be unpacked and the web application will be activated.
work	Contains all compiled JSP files

Figure 4.4: Directory structure of Apache Tomcat

Apache Axis2 is a library for building and a container for running web services [4]. In order to use the Apache Axis2 container we have to deploy the container web application on Apache Tomcat. The filesystem structure of the Apache Axis2 component is listed in Figure 4.5.

Creating Nix components for the Apache Tomcat and Axis2 services is done by copying

axis2-web	Contains the webpages, images and other files to display the web front-end
META-INF	Contains a manifest file
WEB-INF/classes	Contains Java class files which can be used by the Axis2 container
WEB-INF/conf	Contains Apache Axis2 configuration files
WEB-INF/lib	Contains all Java libraries which can be used by the Axis2 container
WEB-INF/modules	Contains extension modules for Apache Axis2
WEB-INF/services	Directory in which web services should be stored. By putting AAR files in this directory the AAR archive will be unpacked and the web service will be activated.

Figure 4.5: Directory structure of the Apache Axis2 container web application

the files from the tarball to the Nix store. It is not possible to use this component to run an Apache Tomcat service. After building a Nix component, it can never be changed and all files and directories in the store are made read-only. Some directories of Apache Tomcat and Apache Axis2 need to be writable, for instance in order to deploy a WAR file we have to put it in the webapps directory or we have to upload it using the web interface of Axis2.

We also want to build Java web applications and Axis2 web services the same way as ordinary applications by storing them in isolation from each other in the Nix store. In order to make web applications and web services deployable in the Nix store we have to build a wrapper around Apache Tomcat.

The Upstart service component of Apache Tomcat creates a separate Apache Tomcat instance directory which resides in /var/tomcat which is a writable directory. Before starting the Apache Tomcat server the service component executes the following steps:

- Make a common/lib symlink to /nix/var/nix/profiles/default/common/lib
- Make a shared/lib symlink to /nix/var/nix/profiles/default/shared/lib
- Make a webapps directory. In this case we cannot create a symlink to the /nix/var/nix/profiles/default/webapps folder. All the WAR files will be unpacked in the same directory so the webapps folder needs to be writable as well. In this case we create an empty webapps folder. Then we symlink all the contents of the /nix/var/nix/profiles/default/webapps into the webapps folder.
- Starting the Apache Tomcat startup script which activates the Servlet container

By using this strategy it is also possible to store shared libraries, common libraries, web application and web services in the Nix store in isolation from each other. All library components, web services and web applications that are activated in the Nix profile which Apache Tomcat uses, are activated and available for use in the Servlet container.

4.3.3 MySQL

There was already a MySQL package and service available in Nixpkgs and NixOS. It also stores its state data outside the Nix store in /var/mysql.

4.4 Modeling library components

The library components that the SDS2 platform uses are:

- *Smack*. An XMPP client library written in Java.
- *MySQL JDBC driver*. Allows connections to a MySQL database in Java programs through the JDBC interface.
- *Google Web Toolkit (GWT)*. An open source Java software development framework to create Ajax applications in Java. It compiles Java code to JavaScript code.

All the library components are open-source just like the infrastructure components and the Nix expressions we developed for these components are part of the Nixpkgs project.

4.4.1 Smack

The Smack API is an XMPP client library written in Java [31]. It is used by the event generator to send status and location events to the XMPP server, by the XMPPLogger to listen to status and location events and by the Asset Tracker service to display the events.

The source code archive already contains a prepackaged JAR file, thus it is not necessary to compile the source code again since Java class files are the same for every platform. This component is created by copying the prepackaged Smack JAR file into the Nix store.

4.4.2 MySQL JDBC driver

The source code archive already contains a prepackaged JAR file, thus it is not necessary to compile the source code. This component is created by copying the MySQL JDBC JAR file into the Nix store. We also have to make it available for Apache Tomcat. The Apache Tomcat server manages a pool of database connections which are requested by web applications running on Apache Tomcat through the JNDI interface. So the JDBC driver has to be available for both the Servlet container and the web applications. Therefore the JDBC JAR file should be stored in the common/lib directory of Apache Tomcat.

The MySQL JDBC driver component in the Nix store does not store the JAR file in the common/lib directory because it can be used by other applications or another application server such as JBoss, which have different locations of storing libraries. In order to make the JDBC driver working on Tomcat we also created a tomcat-mysql-jdbc component which is a wrapper around the MySQL JDBC driver library. This component creates a symlink to the MySQL JDBC JAR file in the common/lib directory.

4.4.3 Google Web Toolkit (GWT)

The Google Web Toolkit is an open source Java software development framework to create Ajax applications in Java [26]. With GWT an Ajax-based web application front-end is written in the Java programming language. GWT then compiles the Java code into optimized JavaScript that should work with all major browsers such as Microsoft Internet Explorer, Mozilla Firefox, Opera, Safari and so on.

The Google Web Toolkit provides two major components:

- *GWT compiler*. This tool compiles Ajax based Java code to optimized JavaScript code.
- *GWT shell*. This tool is an interpreter for the Ajax based Java code. This tool can be used by developers to test and debug applications.

The Google Web Toolkit components are written in Java as well. The *GWT shell* and *GWT compiler* provides a GUI to the end user which is built in SWT. SWT is a GUI component library that wraps around the native GUI toolkit of the platform. On Linux platforms SWT widgets are implemented using GTK+.

In order to get the GWT tools working the Java Virtual Machine needs to find the GTK+ libraries and all its dependencies. The Java Virtual Machine seeks for native libraries in global namespaces such as /usr/lib, but since we do not have those directories on NixOS we have to specificy the location of GTK+ and all its dependencies explicitly. Therefore we built wrapper scripts around the GWT compiler and GWT shell that provide this.

4.5 Modeling the SDS2 platform components

Previously, all the SDS2 platform components were packaged by using the Eclipse IDE and installed by copying the resulting archives to the Apache Tomcat container by hand. The only exception is the generation of Axis2 archives. The reason why the SDS2 developers provide Apache Ant build files for these components is because the Eclipse IDE provides no features to generate an AAR file.

The *Nix deployment system* requires that a component can be built in a non-interactive way. So in order to make the SDS2 platform componentens automatically deployable with Nix we have to provide build scripts for all the components of the SDS2 platform.

4.5.1 Creating build scripts and Nix expressions

In order to make all the SDS2 components deployable with *Nix* we have to create build scripts that compile the source code and generate archives from it. Since we use Nix as the deployment system and all components are stored in isolation from each other we also have to make all the build scripts parametrized, so that we can specify the locations of the dependencies of each component during build time.

We choose to use the Apache Ant tool [3] to automate the build of all the SDS2 components since that is a tool that is designed for Java projects and that the SDS2 developers are

familiar with. We have created an Apache Ant build script for each SDS2 platform component. Basically each Apache Ant compiles all Java source code and creates a package from it which could be one the following:

- Each web service is packaged in an AAR file which is a JAR file with specific web service properties for Axis2. The AAR file contains a directory called lib which contains specific libraries that can be used by the Webservice. All library dependencies of the web service are stored in this folder.
- Each web application is packaged in a WAR file which is a JAR file with specific web application properties that can be deployed on a Servlet container. The WAR file contains a directory called WEB-INF/lib which contains libraries that can be used by the web application. All library dependencies of the web service are stored in this folder.
- Each library and application component is packaged in a JAR file which contain a set of compiled Java classes.

The example Ant file in Figure 4.6 shows the build script for the HIBService component, which executes the following steps:

36 Specifies the location in which all the compiled class files and other files should be stored that end up in a package.

37 Imports all environment variables.

38 In this section we copy the library JAR files of all dependencies. The location of this JAR file is specified by setting the AXIS2UTIL_LIB environment variable which can refer to a library that is stored in the Nix store. If this environment variable is not set then it will use the version that is available in the SDS2 repository. This is useful when the user is invoking the Apache Ant file from within the Eclipse IDE.

39 Sets all the Java libraries in the CLASSPATH so that the Java compiler is able to find them.

40 This section of the code copies all library JAR files to the build directory so that the Java compiler can find them.

41 This line executes the Java compiler which compiles all Java source files to Java bytecode.

42 Copies all other files to deployment directory so that they will be packaged as well.

43 This statement creates an Axis2 archive (AAR file) which packages all files in the deployment directory.

```
<project basedir="." default="generate.service.aar">
    <property name="deploybuild.dir" value="deploy/build"/> 36
    <property environment="env"/> 37
    <condition property="AXIS2UTIL_LIB" value="${env.AXIS2UTIL_LIB}" 38
               else="${basedir}/../../libraries/Axis2Util/deploy/">
        <isset property="env.AXIS2UTIL_LIB"/>
    </condition>
    <condition property="CONFIG_HOME" value="${env.CONFIG_HOME}"
               else="${basedir}/../../config/">
        <isset property="env.CONFIG_HOME"/>
    </condition>
    ...
    <path id="service.classpath"> 39
        <fileset dir="${AXIS2_LIB}">
            <include name="*.jar"/>
        </fileset>
        <fileset dir="${deploybuild.dir}/classes/lib">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="compile">
        <mkdir dir="${deploybuild.dir}"/>
        <mkdir dir="${deploybuild.dir}/classes"/>
        <mkdir dir="${deploybuild.dir}/classes/lib"/>

        <copy toDir="${deploybuild.dir}/classes/lib" failonerror="false"> 40
            <fileset dir="${AXIS2UTIL_LIB}/">
                <include name="*.jar"/>
            </fileset>
        </copy>
        <copy toDir="${deploybuild.dir}/classes/lib" failonerror="false">
            <fileset dir="${CONFIG_HOME}">
                <include name="*.jar"/>
            </fileset>
        </copy>
        ...
        <javac debug="on" fork="true" destdir="${deploybuild.dir}/classes"
               srcdir="${basedir}/src" classpathref="service.classpath" /> 41
    </target>

    <target name="generate.service.aar" depends="compile">
        <copy toDir="${deploybuild.dir}/classes" failonerror="false"> 42
            <fileset dir="${basedir}/resources">
                <include name="**/*.xml"/>
                <include name="**/*.wsdl"/>
            </fileset>
        </copy>
        <jar destfile="${deploybuild.dir}/../HIBService.aar"> 43
            <fileset dir="${deploybuild.dir}/classes"/>
        </jar>
    </target>
    ...
</project>
```

Figure 4.6: build.xml: Example of a partial Apache Ant file for the SDS2 HIBService component

The libraries that are packaged in the WAR and AAR files do not interfere with each other. Each web application is running in its own process and uses its own libraries, except for the libraries that are stored in the shared library folders which are mentioned in Section 4.3.2.

By using these Apache Ant files, we can create Nix expressions for each component that set the environment variables to its dependencies in the Nix store and then executes the compilation and packaging targets of the Apache Ant files. Finally an entry is created for the SDS2 component in the Nix store and the resulting archive is copied to it.

4.5.2 Accessing the Subversion repository

The source code of the SDS2 platform is stored in a Subversion [11] repository which is hosted inside the Philips Research environment. Subversion provides various protocols to access subversion repositories, for instance: WebDAV, SSH and its own SVN protocol. At Philips Research the only method to access the Subversion repository is by using the SSH protocol. Other transport protocols cannot be used due to internal organization policies.

By using the SSH protocol all repository access is handled by the SSH server which requires either authentication by a username and password or by providing a public/private key pair. The Subversion client uses OpenSSH [41] to handle all SSH protocol operations.

The use of the SSH protocol makes automation of the build process difficult. OpenSSH provides no way to automate username and password authentication. Solving this by passing a username and password to the standard input of the OpenSSH client has no use because it ignores all data coming from the standard input due to security reasons.

Using a public/private key pair to automate the build process is also difficult. OpenSSH is very strict with handling key files. OpenSSH requires that a key file is stored in the home directory of the user which is executing the OpenSSH client. It also requires that only the owner has read and write access on the key file[1], thus there should be no read, write and execute on the group and other bits.

Since Nix build actions are executed in a clean environment with no home directory in order to make builds pure we cannot store a key file in the home directory. All Nix build actions are executed under a set of a special build users. We do not know which user the build runs as.

We fixed this authentication problem by creating a hacked Nix fetchsvn component, called fetchsvnssh. This component uses the expect tool which passes a username and password to the OpenSSH client. This solution is not very elegant because this authentication method is not always the preferred way and we have to specify the username and password in a Nix expression. The solution however is pure and it works.

4.5.3 Composing the SDS2 platform components

Just like every Nix package the SDS2 platform components also have to be *composed*. Figure 4.7 shows a partial Nix expression which describes the compositions of the SDS2 platform components:

[1] The file should have UNIX permission mode 600

```
let pkgs = import (builtins.getEnv "NIXPKGS_ALL") { }; in 44
with pkgs;

rec {
#### ACCOUNT SETTINGS
  username = "user"; 45
  password = "secret";

#### CONFIGS
  configs = rec {
    configDevlocalhost = import ../config {
      inherit stdenv fetchsvnssh apacheAnt jdk;
      inherit username password;
      configName = "devlocalhost";
    };
    config = configDevlocalhost;
  };
#### LIBRARIES
  SDS2 = recurseIntoAttrs (rec { 46
    libraries = recurseIntoAttrs (rec { 47
      Axis2Util = import ../libraries/Axis2Util {
          inherit username password;
          inherit stdenv fetchsvnssh apacheAnt jdk axis2;
      };
      ...
    });
#### WEBSERVICES
    webservices = recurseIntoAttrs (rec {
      HIBService = import ../SDS2WebServices/HIBService { 48
          inherit username password;
          inherit stdenv fetchsvnssh apacheAnt jdk axis2;
          inherit (SDS2.libraries) SDS2Util Axis2Util;
          inherit (configs) config;
          inherit (SDS2.webservices) HIBServiceInterface;
      };
      ...
    });
#### WEBAPPLICATIONS
    webapplications = recurseIntoAttrs (rec {
      SDS2AssetTracker = import ../SDS2WebApplications/SDS2AssetTracker {
          inherit username password;
          inherit stdenv fetchsvnssh apacheAnt jdk;
          inherit axis2 gwt gwtwidgets gwtdragdrop smack jetty61 tomcat6;
          inherit (configs) config;
          inherit (SDS2.libraries) Axis2Util SDS2Util gwtdatepicker NeedsServiceClient;
          inherit (SDS2.webservices) EMLogServiceInterface FloorPlanServiceInterface;
          inherit (SDS2.webservices) HIBServiceInterface MELogServiceInterface;
      };
      ...
    });
#### APPLICATIONS
    applications = recurseIntoAttrs (rec {
      EventMerger = import ../SDS2Applications/EventMerger {
          inherit username password;
          inherit stdenv fetchsvnssh apacheAnt jdk;
          inherit axis2 mysql_jdbc;
          inherit (configs) config;
          inherit (SDS2.libraries) Axis2Util;
          inherit (SDS2.webservices) HIBServiceInterface;
      };
      ...
    });
  });
}
```


Figure 4.7: Partial composition expression of SDS2

44 Imports the *packages model* of Nixpkgs, so that various packages we need are available, such as Apache Ant and Apache Axis2. The NIXPKGS_ALL environment variable is a special environment variable used on NixOS which contains the path to the all-packages.nix file of Nixpkgs. We need to include this in order to get compositions of various open source packages such as Apache Tomcat, Apache Axis2, Google Web Toolkit and so on.

45 Since we have no anonymous access to the Subversion repository in the Philips Research environment, we have to store the authentication to the repository in the composition Nix expression. The username and password are passed as input arguments to the Nix expressions that will build the SDS2 platform components.

46 All SDS2 platform components are grouped in the SDS2 attribute set. By using these groups it is also possible to deploy entire categories of SDS2 components instead of specifying every single individual component. The recurseIntoAttrs function is used to allow the Nix package manager to search deeper in the nested attribute sets. In this case we define an attribute that contains all SDS2 components.

47 It is also possible to define subgroups. In this case we define the SDS2.libraries attribute which contains all SDS2 platform libraries.

48 In this Nix expression we also import the Nix expressions functions that actually build the components. In this case we call the Nix expression that builds the HIBService component which uses the Apache Ant script illustrated in Figure 4.6.

4.5.4 Porting SDS2 components to NixOS

There were also some minor modifications we had to make to the SDS2 platform in order to run it on Linux systems and NixOS, which is also a Linux based system. On Windows platforms the directory separator symbol is a backslash: \, however on UNIX based platforms such as Linux the directory separator symbol is a forward slash: /. Some pathnames in the SDS2 code contains backslashes. We changed this inconsistency by replacing the backslash symbol with the result of the System.getProperty("file.separator") method call. This method provides a backslash symbol on Windows based systems and a forward slash on UNIX based systems.

4.6 Deploying SDS2

By making all these modifications to SDS2 platform and by writing Nix expressions for every SDS2 component, it is possible to deploy the entire SDS2 platform or parts of the SDS2 platform on NixOS by using the Nix package manager.

The following instruction deploys the *entire* SDS2 platform on one single machine:

```
nix-env -f all-packages.nix -i -A SDS2
```

It is also possible to deploy certain parts of the SDS2 platform. The following instruction for instance only deploys all SDS2 web service components:

```
nix-env -f all-packages.nix -i -A SDS2.webservices
```

It is even possible to deploy SDS2 components individually. The following instruction will only deploy the SDS2AssetTracker web application:

```
nix-env -f all-packages.nix -i SDS2AssetTracker
```

One limitation of the Nix package manager is managing state data for components. For instance MySQL stores database files and logfiles on disk which changes constantly. Ejabberd stores user accounts in a database and also stores logfiles on disk. State data is stored *outside* the Nix store, since a component in the Nix store can never be changed after it has been built. Therefore we are providing plain old shell scripts which initialize the initial database schemas of SDS2 and the initial configuration of ejabberd.

This step cannot be captured in Nix expressions since initialization of state data outside the Nix store is an impure operation. Nix currently provides no support for managing state data, except for an experimental branch of Nix which is called *S-Nix* (State Nix) [14]. S-Nix however is still a proof-of-concept tool and not yet ready for use[2].

The ejabberd and MySQL state data can be initialized by executing:

```
sh install-ejabberddata.sh
sh install-mysqldata.sh
```

After deploying all the necessary SDS2 platform components and state data the SDS2 system has to be activated. This can be done by starting the Apache Tomcat, MySQL, Ejabberd and SDS2 application components on NixOS by typing:

```
initctl start ejabberd
initctl start mysql
initctl start tomcat
sds2-eventgenerator
sds2-xmpplogger
sds2-eventmerger
```

After deploying the SDS2 platform components, installing the initial state data and activating the services we have a running SDS2 system on a single machine. The application can be accessed by opening a web browser and visiting the Demo Portal web application by typing http://localhost:8080/DemoPortal in the address bar.

After modeling the SDS2 platform and related libraries and services in Nix, the entire SDS2 platform can be built and activated by executing 9 manual tasks, which are all command line instructions to be performed on NixOS, which is significantly less than the semi-automatic deployment process described in Section 4.1. Also the time of deploying the entire SDS2 platform from scratch, which are all tasks mentioned in this section, took about 30 minutes on my machine at the HSA site.

[2]It might never be used at all, since it relies on the Ext3cow filesystem module which is no longer maintained and updated since Linux kernel release 2.6.20.3.

4.7 Dependency graph of SDS2

The deployment of the SDS2 platform components and all its dependencies is a complex case. Figure 4.8 shows the dependency graph of the SDS2AssetTracker and all its build time dependencies. A total number of 1817 components are needed, which consist of operating system components, libraries, servers, compilers, and download tools. All components are needed in order *to build an entire system* running the SDS2AssetTracker. Figure 4.9 shows a subset of the dependency graph where the SDS2AssetTracker component is visible. The Nix package manager guarantees *correct deployment*, so in this case if we want to deploy the SDS2AssetTracker component Nix *guarantees* that the SDS2AssetTracker component and all its 1816 dependencies are available in order to build the SDS2AssetTracker.

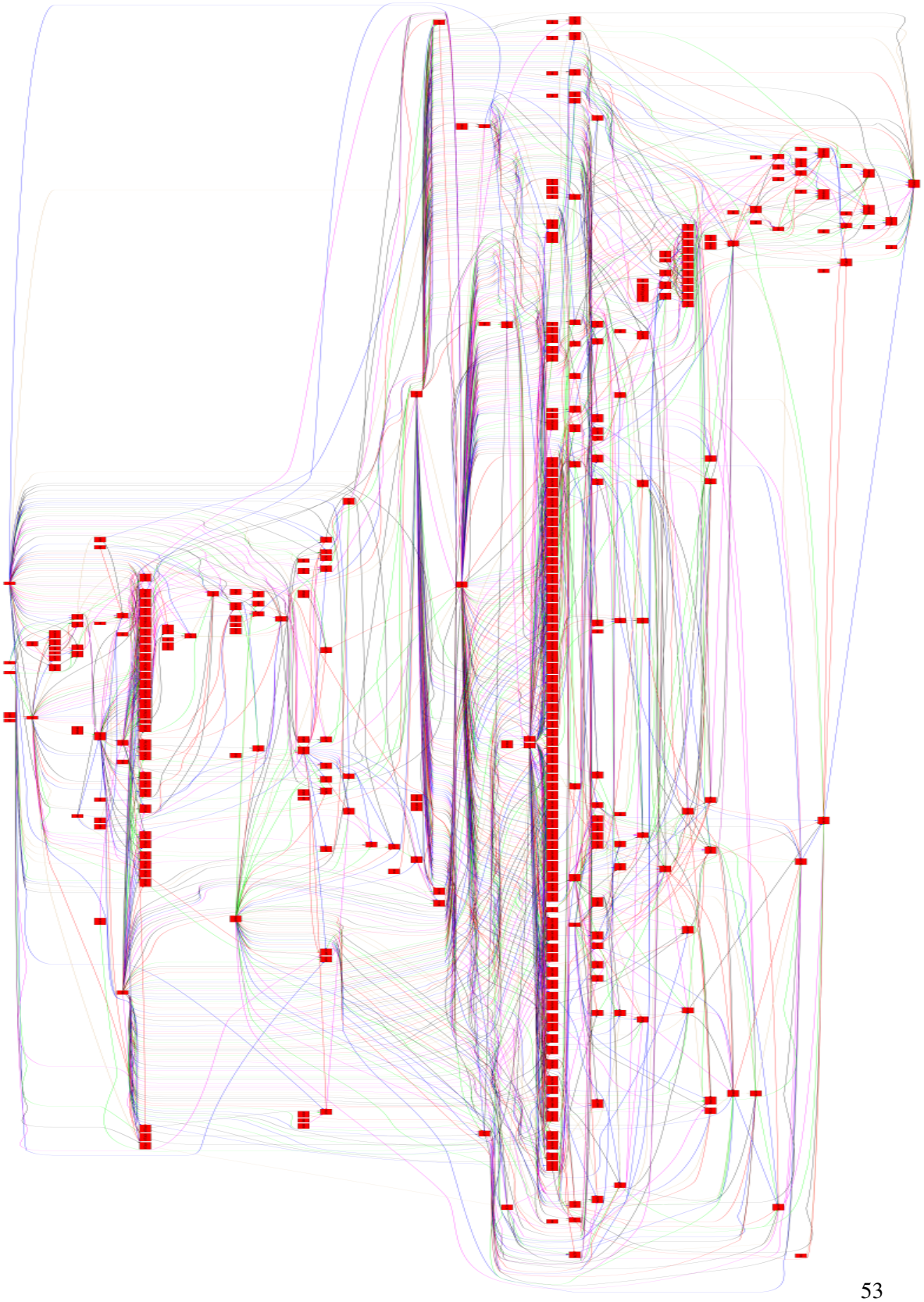


Figure 4.8: Dependency graph of the SDS2AssetTracker component

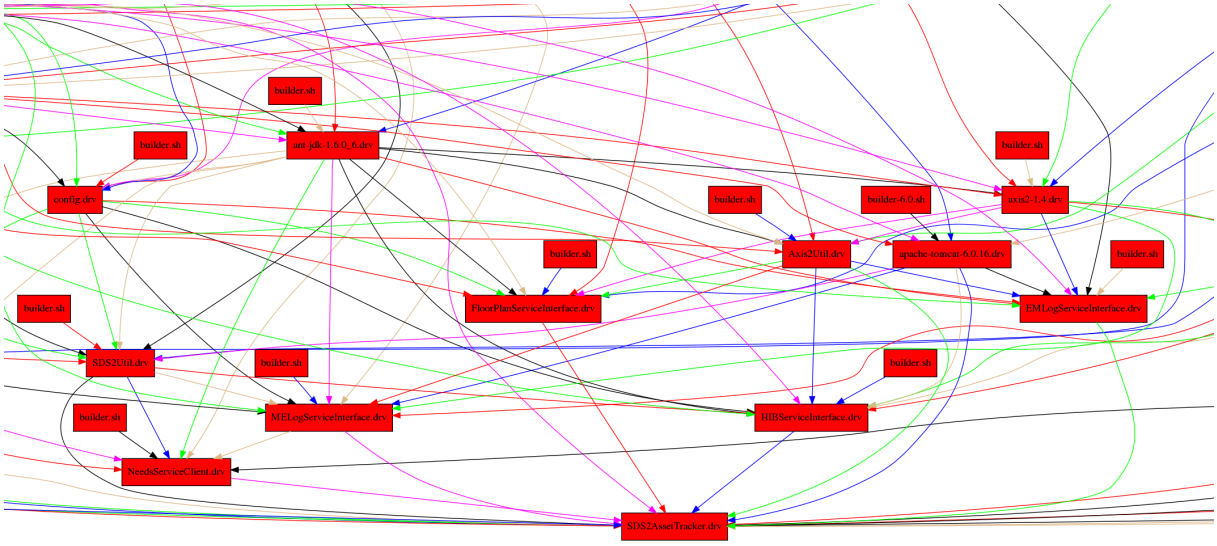


Figure 4.9: Subset of the dependency graph of the SDS2AssetTracker component

Chapter 5

The Disnix Deployment System

To make distributed deployment possible we have to extend the existing *Nix deployment system* with new models and tools. In this chapter we give an overview of the *Disnix deployment system* which extends the Nix deployment system with features that makes distributed deployment possible by using the Nix primitives of software deployment.

5.1 Motivation

Just as distributed systems appear to a user as one logical system, we also would like to deploy software on a distributed system as if it were one single system. However, this ideal is hard to achieve with conventional deployment tools. First, the deployment system necessarily has to know to what machine each service should be deployed. Second, it must know the dependencies between services on different machines. So in addition to local dependencies between packages on a single machine, the *intra-dependencies*, there are also dependencies between services on different machines, the *inter-dependencies*. The two dependency relationships are illustrated in Figure 5.1.

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

5.2 Example case

In this chapter we use a trivial Hello World example case to illustrate the models Disnix uses. The example system is a simple distributed system that consists of two web services as illustrated in Figure 5.2. One web service is called HelloService which has one method that just returns the string 'Hello'. The other web service is called HelloWorldService which invokes the HelloService to retrieve the 'Hello' string. The HelloWorldService then uses

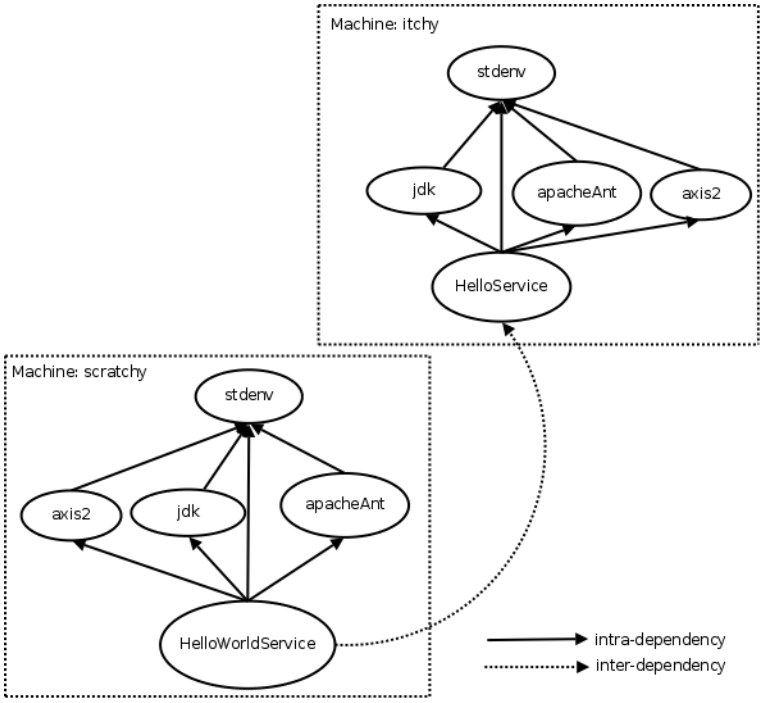


Figure 5.1: Intra-dependencies and inter-dependencies

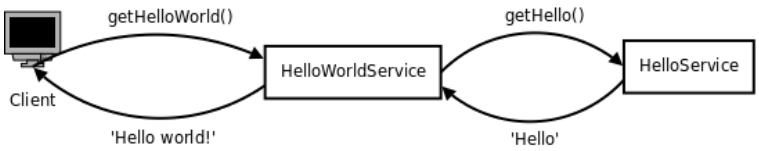


Figure 5.2: Hello World example case

this string to form the string 'Hello world!' which can be retrieved by invoking the getHelloWorld method.

The distributed system has a *packages model* which is a Nix expression that composes packages together, just like we mentioned earlier in Figure 2.4. This model specifies what components are available and how to compose them. So basically this model captures the *intra-dependencies* of a system.

Figure 5.3 illustrates the packages model for the Hello World example case. It shows two web services that are written in the Java programming language and make use of Apache Axis2 libraries.

49 Imports the *packages model* of Nixpkgs, so that various packages we need are available, such as Apache Ant and Apache Axis2. The NIXPKGS_ALL environment vari-

```
let pkgs = import (builtins.getEnv "NIXPKGS_ALL") { }; 49
in
with pkgs;

rec
{
    HelloService = import ../services/HelloService { 50
        inherit stdenv fetchsvn jdk apacheAnt axis2;
    };

    HelloWorldService = import ../services/HelloWorldService {
        inherit stdenv fetchsvn jdk apacheAnt axis2;
    };
}
```

Figure 5.3: top-level/all-packages.nix: Packages model for the Hello World example

able is a special environment variable used in NixOS which contains the path to the all-packages.nix file of Nixpkgs.

50 Calls the function that creates the store derivation of a component, which is in this case the build action of the HelloService component.

```
{stdenv, fetchsvn, jdk, apacheAnt, axis2}:

stdenv.mkDerivation {
  name = "HelloService-0.1";
  src = fetchsvn {
    url = https://svn.nixos.org/repos/nix/disnix/HelloWorldExample/trunk/HelloService;
    sha256 = "83421d756e261879cf7d01e9fe7baecfcb30d88c625350a7b477f616ed80eb5e";
    rev = 12935;
  };
  builder = ./builder.sh;
  buildInputs = [apacheAnt jdk];
  inherit axis2;
}
```

Figure 5.4: services/HelloService/default.nix: Nix expression for the HelloService

The Nix expression of the HelloService is a trivial one as illustrated in Figure 5.4. It basically downloads the source code of the component from a Subversion repository, then compiles all the Java files, then it generates an Apache Axis2 archive (AAR file) and copies the resulting AAR file into the Nix store. The compilation and packaging steps are described in the builder.sh script that is referenced in this Nix expression.

The Nix expression of the HelloWorldService is slightly more complex. The HelloWorldService has to know how to reach the HelloService. Figure 5.5 shows the Nix expression for

```
{stdenv, fetchsvn, jdk, apacheAnt, axis2}:
{HelloServiceHostname ? "localhost", HelloServicePort ? 8080}: 51

stdenv.mkDerivation {
  name = "HelloWorldService-0.1";
  src = fetchsvn {
    url = https://svn.nixos.org/repos/nix/disnix/HelloWorldExample/trunk/HelloWorldService;
    sha256 = "74f868564b7be5917a7d8adeed468cc12b892d8d977f192ef29d12a91688ec90";
    rev = 12935;
  };
  builder = ./builder.sh;
  buildInputs = [apacheAnt jdk];
  inherit axis2 HelloServiceHostname HelloServicePort; 52
}
```

Figure 5.5: services/HelloWorldService/default.nix: Nix expression for the HelloWorldService

the HelloWorldService component. In this case we have a nested function where the inner function takes the hostname and portname of the HelloService as its arguments as illustrated in 51. The default values of these arguments is a localhost connection on port 8080. These settings are targeting an Apache Tomcat service running on the same machine with its standard settings. Finally these values are inherited as illustrated in 52 so that the builder script can use these values.

This nested function strategy is used to make it possible to specify the connection settings from a seperate function. For instance by referencing the HelloWorldService attribute from the attribute set in Figure 5.3 a build action is executed with the default parameters: hostname localhost and port number 8080. By passing function arguments to this attribute the reference to the HelloService can be customized. For instance by referencing the same HelloWorldService attribute with: HelloWorldService { hostname = "itchy"; port = 80; } a HelloWorldService component will be built that connects to the HelloService on a machine with hostname itchy on port 80. This strategy is used later by an extension in order specify the locations of web services during deployment-time.

Figure 5.6 illustrates the builder script for the HelloWorldService component.

53 The code is extracted from a Subversion repository, which is stored as a separate component in the Nix store with read-only permissions. We need to have write access in the source code directory, thus this section of the script fixes the file permissions on the source code file in order to make it possible to build the component.

54 The AXIS2_LIB environment variable points to the directory in the Nix store in which the Apache Axis2 libraries are stored.

55 Generates a configuration file that contains the location of the HelloService. This configuration file is used by the HelloWorldService to connect to the HelloService.

56 Executes the Apache Ant target that builds the Java files and generates an Apache Axis2 archive from it.

```
source $stdenv/setup

# Fix the permissions of the source code [53]

cp -av $src/* .
find . -type f | while read i
do
    chmod 644 "$i"
done
find . -type d | while read i
do
    chmod 755 "$i"
done

# Set Ant file parameters

export AXIS2_LIB=$axis2/webapps/axis2/WEB-INF/lib [54]

# Generate connection settings for the HelloService

echo "helloservice.targetEPR=http://$HelloServiceHostname:$HelloServicePort\
/axis2/services/HelloService" \
 > src/org/nixos/disnix/example/helloworld/helloworldservice.properties [55]

# Compile, package and deploy

ant generate.service.aar [56]
ensureDir $out/webapps/axis2/WEB-INF/services
cp HelloWorldService.aar $out/webapps/axis2/WEB-INF/services [57]
```

Figure 5.6: services/HelloWorldService/builder.sh: Builder script for the HelloWorldService

[57] Copies the resulting AAR file into the Nix store.

By using these expressions it is possible to install the HelloWorldService and make it available to Apache Axis2 container by typing:

```
$ nix-env -f all-packages.nix -i HelloWorldService
```

Although it is possible to build the HelloWorldService component automatically and have correct intra-dependencies, we still have no guarantees that the system is working yet. We have to be sure that the HelloService component is activated *before* the HelloWorldService component is activated. Another limitation of using the standard Nix package manager is that we still cannot specify the location of the HelloService dynamically. It is fixed to localhost or to another specified host. In a real distributed deployment scenario we do not necessarily know in advance on what machine a component should be deployed. *Disnix* provides additional models and tools to overcome this limitions which is explained in the next sections.

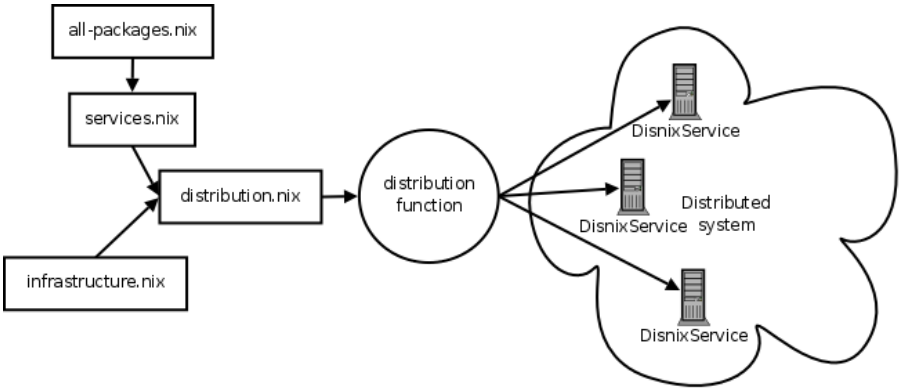


Figure 5.7: Overview of Disnix

5.3 Overview

Figure 5.7 shows an overview of the Disnix system. It consists of a *distribution function* that takes a *distribution model* as input which contains a mapping of services to machines in the network. The distribution model takes two models as input; a service model which describes all the available services and an infrastructure model which describes the machines in the network. The service model reuses the packages model to describe build actions. Another element in the illustration is the *DisnixService* which is a remote interface to the Nix store and Nix profiles on a machine in the network.

The distribution function is built from several primitives, in this case command-line tools, that perform certain steps of the distributed deployment process. The entire deployment process can be invoked at once or some steps can be invoked separately by calling the appropriate command-line tools.

By executing the entire distributed deployment procedure all services defined in the distribution model will be built from source code. After building the services the distribution function distributes them to the target machines and finally activates them. Services that are no longer in use are deactivated. Activation and deactivation are performed via the Disnix interfaces on the target machines. For the transition step a distributed transaction algorithm is used in order to make it an atomic operation.

5.4 Models for distributed systems

To deploy packages on a single machine Nix just needs to know the compositions of each package and how to build them. Disnix needs some additional information for deploying services on multiple machines. Therefore we introduce three models: a *services model*, an *infrastructure model* and a *distribution model*.

5.4.1 Services model

The *services model* describes the services that can be distributed across nodes in the network. Each service is essentially a package, except that it has an extra property called *dependsOn* which describes the *inter-dependencies* on other services. Figure 5.8 shows a Nix expression that describes a model for two services: HelloService, which has no inter-dependencies, and HelloWorldService, which depends on the former.

```
{distribution}: 58

let lib = import ../lib; 59
    pkgs = import ../top-level/all-packages.nix; 60
in

rec { 61
    HelloService = { 62
        name = "HelloService";
        pkg = pkgs.HelloService; 63
        dependsOn = [];
    };
    HelloWorldService = rec { 64
        name = "HelloWorldService";
        HelloMachine = lib.findTarget
           {inherit distribution; serviceName = "HelloService";}; 65
        pkg = pkgs.HelloWorldService 66
           {HelloServiceHostname = HelloMachine.hostname;
            HelloServicePort = HelloMachine.tomcatPort;};
        dependsOn = [ HelloService ]; 67
    };
}
```

Figure 5.8: services.nix: Services model

58 The services model is a function which takes a distribution model as its input argument. The distribution model is used to give parameters to services, so that services know about the distribution of itself and other services. A distribution model is defined in Section 5.4.3.

59 The services model also uses some helper functions which are defined in the lib directory.

60 The services model imports the *packages model* described earlier in Figure 5.3 which defines a set of attributes that evaluate to derivations. This model also specifies the *intra-dependencies* of a service.

61 The services model also contains a *mutually recursive* attribute set where services can refer to each other just like the *packages model*.

62 This attribute defines the properties of a service.

63 Refers to the intra-dependency closure of the HelloService. The *packages model* which is imported in 60 defines how to compose this package.

64 Also the other services of the distributed system are defined in the same expression.

65 Searches the target machine of the HelloService in the *distribution model* by using the lib.findTarget function. This attribute set is used in 66 to pass the *hostname* and *port* as arguments to the HelloWorldService component.

66 Refers to the intra-dependency closure of the HelloWorldService which is defined in the package model. In this case we pass the hostname and port of the HelloService as arguments to the function which is defined in Figure 5.5 so that the HelloWorldService is able to find the HelloService.

67 Describes the *inter-dependencies* of the service. The HelloWorldService has an inter-dependency on the HelloService component. In order to activate the HelloWorldService component in the distributed environment, we have to activate the HelloService first.

5.4.2 Infrastructure model

The *infrastructure model* is a Nix expression that describes certain attributes about each machine in the network. Figure 5.9 shows an example of a network with two machines. In the model we describe the hostname and the target endpoint references (targetEPR) of the Disnix interface. The targetEPR is a URL that points to the Disnix web service that can execute operations on the remote Nix stores and profiles.

```
{
    itchy = { 68
        hostname = "itchy"; 69
        targetEPR = http://itchy:8080/axis2/services/DisnixService; 70
        tomcatPort = 8080; 71
    };
    scratchy = {
        hostname = "scratchy";
        targetEPR = http://scratchy:8080/axis2/services/DisnixService;
        tomcatPort = 8080;
    };
}
```

Figure 5.9: infrastructure.nix: Infrastructure model

68 An attribute set which describes properties of a machine in the network

69 Specifies the hostname of the machine. This attribute can be used as arguments in the services model to generate for configuration files so that the services can find each other.

70 The target endpoint reference of the Disnix Service, which provides remote access to the Nix store and Nix profiles.

71 Specifies on what port where Apache Tomcat listens. This attribute is optional.

5.4.3 Distribution model

The *distribution model* is a Nix expression that connects the services and infrastructure model, mapping services to machines. It contains a function which takes a services and infrastructure model as its inputs and returns a list of pairs. Each pair describes what service should be distributed to which machine in the network. Figure 5.10 shows an example.

```
{services, infrastructure}: 72

[
    { service = services.HelloService;
      target = infrastructure.itchy; } 73
    { service = services.HelloWorldService;
      target = infrastructure.scratchy; }
]
```

Figure 5.10: distribution.nix: Distribution model

72 The distribution model is a function which takes a services model and a infrastructure model as its input arguments. We can use the services model defined in Section 5.4.1 and the infrastructure model defined in Section 5.4.2 for this.

73 This attribute set specifies a mapping of a service to a machine. The machine is defined in the services model and the target machine is defined in the infrastructure model.

5.5 Distribution export

Just as components are built from *store derivation files*, we also have a *distribution export file* which is also encoded in a more simple format and describes how to distribute components in the Nix store to machines in the network. The reason why we use a more primitive format for this is for the same reasons as we use a *store derivation file*, which is a low-level representation of a build action. The Nix expression Disnix uses also consists of multiple

files that are scattered across the filesystem. The distribution export is generated by a Nix expression that takes the distribution model as its input.

We have two variants of the distribution export file that are used by two transaction algorithm variants. One variant is used by the *transition by transfer* method. In this method we build all the services on the coordinator machine and finally we transfer and activate them on the target machines in the network. The other variant is used by the *transition by compilation method*. In this method we transfer all *store derivation files* of the services to the target machines in the network and finally the services are compiled and activated on the target machines. Both transition methods are explained later in Chapter 7.

5.5.1 Transition by transfer

```
{stdenv, distribution}: 74

let
  distributionExport = map (
                         entry:
                           { service = if entry.service ? pkg
                                       then entry.service.pkg.outPath
                                       else null;
                                target = entry.target.targetEPR; }
                       ) distribution; 75
in
stdenv.mkDerivation {
    name = "distribution-export";
    outputXML = builtins.toXML distributionExport; 76
    builder = ./builder.sh; 77
}
```

Figure 5.11: export.nix: Nix expression that generates a distribution export

Figure 5.11 illustrates one of the Nix expression variants that generates a distribution export file.

74 This is a function header that takes two input arguments. stdenv is the standard environment that provides basic UNIX utilities. distribution is a distribution model, which could be a model defined in Figure 5.10.

75 Applies a function to each member of the distribution model. The function translates the service attribute into service.pkg.outPath. This variable returns the path of the component in the Nix store. If the component is not built yet this call will trigger a component build action for the specified component. The function also translates the target attribute into target.targetEPR which is the URL of the *Disnix* service that is running on the target machine. It is also possible to omit the specification of an

intra-dependency closure in the services model. In that case the service is null and will not be built. This feature is sometimes needed to capture services that are not in the Nix store.

76 This function creates a XML presentation of the attribute set that the distributionExport function returns

77 The builder scripts writes the XML representation created in 76 to a file in the Nix store.

```
<?xml version="1.0" encoding="utf-8"?>
<expr>
  <list>
    <attrs>
      <attr name="service">
        <string value="/nix/store/gibwn64r9bj0a1v218j8a043ajfhy46p-HelloService-0.1"/>
      </attr>
      <attr name="target">
        <string value="http://itchy:8080/axis2/services/DisnixService"/>
      </attr>
    </attrs>
    <attrs>
      <attr name="service">
        <string value="/nix/store/20800czw8mn9c826vq4ih8i5sxa41z0n-HelloWorldService-0.1"/>
      </attr>
      <attr name="target">
        <string value="http://scratchy:8080/axis2/services/DisnixService"/>
      </attr>
    </attrs>
  </list>
</expr>
```

Figure 5.12: The transition by transfer distribution export file for the Hello World example

This variant of the distribution export file contains the locations of the component in the Nix store and the address of the Disnix interface that provides access to the Nix store and Nix profile of the machine which is illustrated in Figure 5.12. This distribution export file can be used by a Disnix tool to activate a new configuration of a distributed system or to upgrade an old configuration to a new configuration.

5.5.2 Transition by compilation

Another variant of the distribution export file contains the locations of the *store derivations* of the component in the Nix store and the address of the Disnix interface that provides access to the Nix store and Nix profile of the target machines. The Nix expression that generates this export file is similar to the Nix expression illustrated in 75 on Figure 5.11, except that it maps the service attribute to the service.pkg.drvPath. This variable returns the path of the store derivation file in the Nix store, which is illustrated in Figure 5.13.

```
<?xml version="1.0" encoding="utf-8"?>
<expr>
  <list>
    <attrs>
      <attr name="service">
        <string value="/nix/store/bw7dnwxip051msdainrjni2a6dybysyn-HelloService-0.1.drv"/>
      </attr>
      <attr name="target">
        <string value="http://itchy:8080/axis2/services/DisnixService"/>
      </attr>
    </attrs>
    <attrs>
      <attr name="service">
        <string value="/nix/store/2490znhi86a9ayff2sm96clg5miws6if-HelloWorldService-0.1.drv"/>
      </attr>
      <attr name="target">
        <string value="http://scratchy:8080/axis2/services/DisnixService"/>
      </attr>
    </attrs>
  </list>
</expr>
```

Figure 5.13: The transition by compilation distribution export file for the Hello World example

5.5.3 Deployment states

A distributed system has certain services installed on certain computers in its network. We can describe a *distribution export file* as a *deployment state* of the distributed system. The distribution export file is a representation of the intended *deployment state* of a distributed system because it specifies what service is deployed on which machine in the network.

5.6 Disnix toolset

The distribution function illustrated in Figure 5.7 consists of serveral primitives which can be invoked as command-line tools seperately or as one command-line tool. The Disnix toolset provides several tools for executing distributed software deployment steps. Also some additional tools are provided such as an garbage collector, a simple distribution model generator and an RPC interface. All the tools are prefixed with disnix-soap- because all the tools connect to DisnixService through the SOAP protocol. In theory it is also possible to support other protocols such as CORBA or RMI.

The Disnix toolset consists of the following command-line tools:

- disnix-soap-client. An interface to the DisnixService which allows remote access to the Nix store and Nix profiles of the target machine. Basically this tool is an RPC interface to the DisnixService.

- disnix-soap-collect-garbage. Executes the garbage collector on all machines on a given *infrastructure model*.
- disnix-soap-copy-closure. Efficiently copies an intra-dependency closure of a package to the target machine by packing the closure in a SOAP message.
- disnix-soap-instantiate. Creates a *store derivation file* of the Nix expression that creates a *distribution export file*.
- disnix-soap-deploy. Distributes and activates all components in a *distribution export file* which is illustrated in Figure 5.12. If two files are given it will compare the differences of the files and will only activate and deactivate the changes.
- disnix-soap-gendist. Generates a *distribution export file* from a given *services*, *infrastructure* and *distribution* model.
- disnix-soap-env. A composition of disnix-soap-gendist and disnix-soap-deploy. It will activate a new configuration described in a given *services*, *infrastructure* and *distribution* model directly. This command represents the distribution function described in Figure 5.7.
- disnix-soap-dist-roundrobin. Generates a *distribution* model from all or a subset of services and machines specified in a given *services* model and *infrastructure* model by using the Round-robin scheduling method. It assigns services to each machine in equal portions and in order. It will also include the entire *inter-dependency* closure of a service automatically.

5.7 Examples

To activate the entire Hello World example case, the user can type:

```
$ disnix-soap-env services.nix infrastructure.nix distribution.nix
```

To generate a distribution export file, the user can type:

```
$ disnix-soap-gendist services.nix infrastructure.nix distribution-empty.nix
```

To upgrade from the current distributed deployment state to a empty deployment state the user may use the disnix-soap-deploy command. The former argument is the new deployment state and the latter is the old deployment state:

```
$ disnix-soap-deploy /nix/store/0p7rplhgri...-distribution-export/output.xml \
    /nix/store/jg7brpf4p7...-distribution-export/output.xml
```

To remove all the garbage and old Nix profiles from the machines in the network the user can type:

```
$ disnix-soap-collect-garbage -d infrastructure.nix
```

Chapter 6

The Disnix Service

In order to extend the existing Nix deployment system with support for distributed systems we need to have access to the Nix stores and Nix profiles of a remote machine. One component of the Disnix deployment system is the *Disnix Service*, which provides remote access to the Nix store and Nix profiles through a web service interface. In this chapter we explain the structure of the Disnix Service.

6.1 Overview

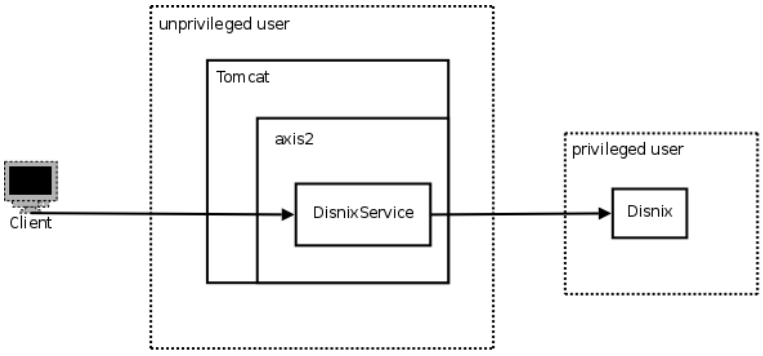


Figure 6.1: Overview of the Disnix interface

The Disnix service consists of two major components as illustrated in 6.1. We have one component that actually executes all operations which is the core Disnix component. We call this component *Disnix*. There is also another component that we call the *Disnix Service* which is in this case a web service interface to the core Disnix. The *Disnix Service* allows a user to execute operations on remote Nix stores and Nix profiles by using the SOAP protocol.

There are several reasons why we choose to make a separation between the interface component and the core component:

- *Integration*. A distributed system often uses a specific protocol that components use to communicate with each other. For instance in a service oriented architecture all components are exposed as "services" which can be invoked by using the SOAP protocol. We want our deployment service to integrate as much as possible with the existing distributed system, thus the deployment service should be invoked the same way as every other component of the distributed system. Many users have specific reasons why they choose a particular protocol e.g. due to organization policies. A distributed deployment system should also support this. That is why we separated the interface part and the core part. If we want to support another protocol such as CORBA we just have to build a CORBA interface which calls the Disnix core component.

- *Privilege separation*. To execute all Nix deployment operations on the Nix store and Nix profiles we need to have super user privileges on UNIX based systems. The web service interface is an application that runs on Apache Tomcat. It is not recommended to run the Apache Tomcat server as privileged user, thus we have to separate the interface component which has no privileges with the core component which has privileges.

6.2 The core Disnix service

The core Disnix service is the service that actually executes all deployment operations and has privileges to access the global Nix profile and to the Nix store. It is invoked by the Disnix Web service layer (or another RPC abstraction) that provides remote access to users.

6.2.1 Implementation

The core Disnix service is a service written in the C programming language. The reason why the core Disnix is implemented in the C programming language is that it can make use of the Nix and OpenSSL libraries which contain hashing algorithms and access to Nix operations. The core Disnix also has to run as a privileged user in order to allow access to the global Nix profile.

It also needs to be accessible through a local interprocess communication protocol and therefore it is implemented as a D-Bus service. D-Bus is a system for interprocess communication (IPC) which contains a daemon and wrapper libraries to make applications communicate with each other. [40]. Users can run several instances of it, each called a channel. There is a privileged channel which is called the *system bus*, and a private channel for each logged in user which is called the *session bus*. The private instances are required because the system bus has access restrictions. D-Bus is also more high level and supports authentication which makes it easier than writing a custom UNIX domain socket protocol. It also provides security features to restrict access to a specific user or to a specific user class [47].

Each application using D-Bus contains *objects*, which basically map to objects in a programming language of choice such as Java, C++, C# or GObject which is used in the C programming language if you are using the GLib interface of D-Bus. Each object has

members. Two kinds of members are *methods* and *signals*. *Methods* are operations that can be invoked on an object, with optional input arguments and output arguments. *Signals* are broadcasts from the object to any interested observers of the object. Signals may contain data payload. The methods and signals can be defined in an introspection XML file which is an interface definition file similar to a WSDL file in web services or an IDL file in CORBA. The introspection file for Disnix is illustrated in Figure 6.2.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-Bus Object Introspection 1.0//EN"
 "http://standards.freedesktop.org/dbus/1.0/introspect.dtd">

<node name="/org/nixos/disnix/Disnix">
    <interface name="org.nixos.disnix.Disnix">
        <method name="install">
            <arg type="s" name="file" direction="in" />
            <arg type="s" name="args" direction="in" />
            <arg type="b" name="isAttr" direction="in" />
            <arg type="s" name="pid" direction="out" />
        </method>
        <method name="uninstall">
            <arg type="s" name="derivation" direction="in" />
            <arg type="s" name="pid" direction="out" />
        </method>
        <method name="instantiate">
            <arg type="s" name="files" direction="in" />
            <arg type="s" name="attrPath" direction="in" />
            <arg type="s" name="pid" direction="out" />
        </method>

        ...

        <signal name="finish">
            <arg type="s" name="pid" direction="out" />
        </signal>
        <signal name="success">
            <arg type="s" name="pid" direction="out" />
            <arg type="s" name="path" direction="out" />
        </signal>
        <signal name="failure">
            <arg type="s" name="pid" direction="out" />
            <arg type="s" name="msg" direction="out" />
        </signal>
    </interface>
</node>
```

Figure 6.2: disnix.xml: Partial D-Bus introspection file for the Disnix server

D-Bus method calls are supposed to finish in a short time (in 30 seconds to be exact) or timeouts will occur. Some Nix operations however could take hours or even days to finish. For instance a rebuild of the entire Nixpkgs collection takes several hours. For that reason we implemented the Disnix server as an asynchronous daemon.

Each function call to the Disnix server will immediately return a process ID that consists of a SHA256 hashcode of the method name and all its inputs. The specified action will then

```
disnixInstall(file, args, isAttr) : 78
    pid ← printHash32(stringToSHA256("install:" + file + ":" + args + ":" + isAttr))79
    if pid ∉ runningProcesses : 80
        runningProcesses ← runningProcesses ∪ {pid}
        createThread(disnixInstallThreadFunc(pid, file, args, isAttr))81
    return pid82

disnixInstallThreadFunc(pid, file, args, isAttr) : 83
    fp ← popen("nix-env -f "+file+" "+args + ...)84
    while (fgets(line, sizeof(line), fp)) ≠ NULL : 85
        puts(line)
    if pclose(fp) = 0 :
        emitFinishSignal(pid)86
    else :
        emitFailureSignal(pid)87
    runningProcesses ← runningProcesses − {pid}88
```

Figure 6.3: Executing the install method on the core Disnix service

be executed in its own thread. If there is already a job running with the same process ID then it will just return the process ID without spawning a new thread. Because all Nix operations are purely functional we always have the same result based on the input arguments, thus there is no need to execute the same operation again. If the operation that the thread executes succeeds it will send a *success* or a *finish* signal over the system bus, so that the caller knows that its job has succeeded. If the operation fails it will send a *failure* signal over the system bus.

Figure 6.3 illustrates the invocation of the install method on the Disnix core service:

78 Is the install method which is invoked by the D-Bus daemon. It takes 3 input arguments which are file, args and isAttr. The file argument is the filesystem location of a Nix expression to use. The args argument are all parameters to the nix-env command and isAttr specifies whether the given argument is an attribute selection or not.

79 Generates a SHA256 hashcode in base 32 notation of the install function with all its input arguments.

80 In this line we check whether the input function is already running or not. Since the result of a Nix operation *exclusively* depends on its input arguments the results is always the same, thus we do not have to run the same job if it is already executing.

81 Adds the process to the list of running processes and finally spawns a thread that executes the install operation.

82 Finally the process ID of the job is returned to the client. The client can then wait for a signal that that contains this process ID, which is a notification that the job is finished and either succeeded or failed.

83 Is the thread that actually executes the install operation.

84 This line executes the nix-env command which installs a specific package in the Nix profile. The nix-env command is called with the specified input arguments.

85 Reads the output of the nix-env command and writes it to the standard output of the core Disnix service, which is a logfile.

86 If the operation succeeded a *signal* is sent over the system bus which contains the process ID so that clients are notified that the operation is finished and succeeded.

87 If the operation failed a *signal* is sent over the system bus which contains the process ID and an optional error message so that clients are notified that the operation has finished and failed.

6.2.2 Features

The core Disnix service provides the following operations:

- *install*. Installs a derivation in the global Nix profile or in a specified user profile.
- *upgrade*. Upgrades a derivation in the global Nix profile or in a specified user profile.
- *uninstall*. Uninstalls a derivation in the global Nix profile or in a specified user profile.
- *instantiate*. Instantiates a given Nix expression.
- *realise*. Realises a given store derivation file.
- *import*. Imports the given serialization of a closure in the Nix store.
- *printInvalidPaths*. Checks whether the given paths exist in the Nix store. It will return all the given paths that do not exist in the Nix store.
- *collectGarbage*. Runs the garbage collector.
- *activate*. Activates a given component on the machine. The activation script should be provided by the user which is set by the DISNIX_ACTIVATE_HOOK environment variable, since there is no generic way to activate a specific component.
- *deactivate*. Deactivates a given component on the machine. The deactivation script should be provided by the user which is set by the DISNIX_DEACTIVATE_HOOK environment variable, since there is no generic way to deactivate a specific component.

- *lock*. Locks the server so that one process gets exclusive access. If the DISNIX_LOCKMANAGER_HOOK environment variable is set, it will first invoke that process to check whether it is allowed to grant exclusive access. This hook can be used in combination with the disnix-tcp-proxy tool which is a proxy for TCP connections. The client of this proxy can grant a lock if and only if there is no open TCP connection. It is also possible to specify a custom built lock manager that copes with application specific settings.
- *unlock*. Unlocks the exclusive access.

The core Disnix server sends the following signals:

- *success*. A signal that is returned when an operation succeeds. The signal contains a process ID *and* a collection of resulting Nix store paths. For instance operations such as *instantiate* and *realise* return Nix store paths.
- *finish*. A signal that is returned when an operation succeeds. The signal contains only a process ID. For instance operations such as *collectGarbage* will not return anything.
- *failure*. A signal that is returned when an operation fails. The signal contains a process ID.

6.2.3 Extension support

As mentioned in Section 6.2.2 the core Disnix server also support extensions by specifying external processes in environment variables. The *activation* and *deactivation* methods of the core Disnix server are using an external process which should activate and deactivate a specific component which are specified by setting the DISNIX_ACTIVATION_HOOK and DISNIX_DEACTIVATION_HOOK environment variables. If the environment variables are set then the specified external processes are invoked and called with the Nix component name as command line argument. The external process, which for instance could be a shell script then executes the necessary steps to activate and deactivate the components.

```
#!/bin/sh
cp $1/webapps/axis2/WEB-INF/services/*.aar /var/tomcat/webapps/axis2/WEB-INF/services
```

Figure 6.4: Example activation script for the Hello World example case

Figure 6.4 shows the *activation* script for the Hello World example case. The $1 variable is the first command line argument which is the Nix store path of the component that should be activated. In this case all the Axis2 archives are copied from the Nix store to the Apache Axis2 deployment directory. Since Apache Axis2 supports *hot-deployment* this action will automatically activate the web service in the Apache Axis2 container.

Figure 6.5 shows the *deactivation* script for the Hello World example case. In this case all the Axis2 archives that are stored in a specific Nix store entry are removed from

```
#!/bin/sh
for i in `find $1/webapps/axis2/WEB-INF/services -name *.aar`
do
    rm -v /var/tomcat/webapps/axis2/WEB-INF/services/`basename $i`
done
```

Figure 6.5: Example deactivation script for the Hello World example case

the Apache Axis2 deployment directory. This action will automatically deactivate the web service in the Apache Axis2 container due to the *hot-deployment* feature.

If the DISNIX_LOCKMANAGER_HOOK environment variable is set, the *lock* method of the core Disnix server will invoke that process to determine whether it should grant a lock or not. The lock manager will invoke the external process and waits until it finishes. If the return status of the process is equal to 0 then the lock will be *granted*. If the return status of the process is not equal to 0 then it *rejects* the lock request. This hook can be used in combination with the disnix-tcp-proxy-client tool. By calling this tool without parameters it will invoke the disnix-tcp-proxy server and checks whether there are open TCP connections. If there are no open TCP connections the client finishes with exit status 0. If some error occurred the client finishes with a non zero exit status.

6.2.4 Examples

We also developed a simple command line interface to the core Disnix service which connects to the core Disnix service on the system bus and allows access to the operations the Disnix service provides.

This operation installs GNU Hello world by calling the *install* operation of the D-Bus service:

```
$ disnix-client -i hello
```

This operation executes the *garbage collector* method of the D-Bus service, which also removes all older Nix profiles:

```
$ disnix-client --collect-garbage -d
```

6.3 The Disnix Web service interface

The *Disnix Web service* component provides a layer on top of the core Disnix service that allows users to access the core Disnix methods through the SOAP protocol. The Disnix Web service is written in the Java programming language and uses the Apache Axis2 container. It connects to the core Disnix service by using the D-Bus Java library [32].

```
install(dbusInterface, signalHandler, file, args, isAttr) :
    pid ← dbusInterface.install(file, args, isAttr)
    signalHandler.addPid(pid, this)
    waitForNotificationToResume()
    if someError :
        throw Exception();
```

Figure 6.6: Executing the install method on the Disnix Webservice

6.3.1 Implementation

Web services running in the Apache Axis2 container are *synchronous* by default, that is all operations of the web services are executed in the same thread of execution. This is not desirable in our case because Nix operations can take hours to finish which mean we have to wait hours to execute another Nix operation through the same Webservice interface.

Apache Axis2 also provides a message listener that allows *asynchronous* execution of methods. By using this listener each method call is executed in its own thread. Each SOAP method call to the Disnix Webservice invokes a method of the Disnix D-Bus service. The way Axis2 provides asynchronous method invocation is quite different then D-Bus however. Therefore we have to build a mapping that provides this.

The mapping algorithm for the install method is illustrated in Figure 6.6. For each D-Bus method invocation the execution thread in the web service will block. After receiving the process ID of the job that the method returns we put the process ID and the thread of execution in a list and the thread sleeps until it receives a notification. After receiving a *success*, *finish* or *failure* signal from the D-Bus service, the signal handler notifies the thread in the list with the specified process ID. Finally the execution thread finishes and throws an exception if an error occurred.

6.3.2 File transfer support

The Disnix Webservice also supports file transfers by using the Message Transmission Optimization Mechanism (MTOM). MTOM allows more efficient sending of binary data in a SOAP request or response [6]. Binary content often has to be re-encoded to be sent as text data with SOAP messages. XML supports opaque data as content through the use of either base64 or hexadecimal text encoding. Both techniques increase the size of the data. For UTF-8 underlying text encoding, base64 encoding increases the size of the binary data by a factor of 1.33 of the original size, while hexadecimal encoding expands data by a factor of 2. MTOM overcomes this problem by using XOP (XML-binary Optimized Packaging) to transmit binary data. The Disnix Webservice uses MTOM to transfer files, such as a closure of a package, efficiently from one machine to another.

6.3.3 Examples

The Disnix toolset also provides a disnix-soap-client tool which is a command line interface to the Disnix Webservice Interface.

This operation installs the GNU Hello package in the global Nix profile of the target machine, which is called itchy:

```
disnix-soap-client -i hello http://itchy:8080/axis2/services/DisnixService
```

The following instruction instantiates a single Nix expression which is stored on the target machine which is called scratchy.

```
disnix-soap-client -i --remotefile test.nix \
    http://scratchy:8080/axis2/services/DisnixService
```

It is also possible to specify files that are stored on the client machine. The following instructions will create a serialization of the GNU Hello world closure on the client machine. Then the Disnix interface imports the given serialization in the Nix store of the target machine itchy. The closure is transferred to the target machine by using MTOM.

```
nix-store --export /nix/store/a36clhh1...-hello-2.3 > hello.closure
disnix-soap-client --import --localfile hello.closure \
    http://itchy:8080/axis2/services/DisnixService
```

6.4 Disnix Service on NixOS

NixOS provides an Upstart service for the core Disnix service, which can be enabled in the top-level configuration.nix file. The web service interface can be enabled by enabling the Apache Tomcat server on NixOS and by installing the DisnixService package in the global Nix profile.

Figure 6.7 shows a partial NixOS configuration file that enables the Disnix core service and Disnix web service.

89 This line imports a *packages model* that composes various application specific hooks.

90 This section contains all settings of the Disnix core service.

91 Defines the activation, deactivation and lock manager hooks which executes the activation, deactivation and locking steps of a specific component. In this case the hooks are defined in the Nix expression defined in 89.

92 This section contains all Apache Tomcat settings.

93 The Disnix Service uses the D-Bus Java library to connect to the core Disnix service. The D-Bus library needs some native libraries available that provides UNIX domain socket connection support. This attribute specifies the location in which Apache Tomcat can find its native libraries.

```
let pkgs = import /home/sander/hooks.nix; 89
in
{
  ...

  services = {
    disnix = { 90
      enable = true;
      activateHook = "${pkgs.activateHook}/bin/activateHook"; 91
      deactivateHook = "${pkgs.deactivateHook}/bin/deactivateHook";
      lockManagerHook = "${pkgs.lockManagerHook}/bin/lockManagerHook";
    };

    tomcat = { 92
      enable = true;
      javaOpts = "-Djava.library.path=/nix/var/nix/profiles/default/shared/lib"; 93
    };
  };
}
```

Figure 6.7: configuration.nix: Partial NixOS configuration that enables the Disnix core service and the Disnix web service

Chapter 7

Atomic upgrading

The transition from the current configuration of a distributed system to a new configuration of a distributed system should be an *atomic* operation. We either want the system in the current configuration or the new configuration, but never in an *inconsistent* state, i.e. a mix of an old and new configuration.

In this chapter we explain how the upgrade process works. We explain what *deployment states* are, how services are transferred from one machine to another and how the transitions between deployment states are done. We have implemented two distributed transaction algorithm variants to do this.

7.1 Distributed deployment states

A distributed system has certain services installed on certain computers in its network. We can describe this as a *deployment state* of the distributed system. If we want to change this deployment state, e.g. by installing additional services, uninstalling obsolete services, replacing services with newer versions, or by migrating services from one computer to another, we do this by defining a new distribution model. By calculating the intersection of the distribution export file of the current distribution model and the new distribution model we can derive for each computer in the network what services should be installed and what services should be uninstalled. The *distribution export* file described earlier in Figure 5.12 represents the intended deployment state of a distributed system.

Define $O = \{D \mid D \in (service, computer)\}$ as the output of the previous distribution model and $N = \{D \mid D \in (service, computer)\}$ as the output of the new distribution model. Define $I = O \cap N$. Then the services we should uninstall in the network are: $O - I$ and the services we should install in the network are: $N - I$. The mappings in set I remain unchanged, since they are the same in the new and the previous configuration.

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

```
copyClosure(path, disnixInterface) :
    storePath ← queryResolve(path)
    deps ← ε
    reqs ← queryRequisites(storePath)
    for each r ∈ reqs
        push(deps, r)
    invalidPaths ← printInvalidPaths(disnixInterface, deps)
    if length(invalidPaths) > 0 :
        s ← export(invalidPaths)
        importClosure(disnixInterface, s)
```

Figure 7.1: copyClosure: Efficiently copies the intra-dependency closure of a package from one Nix store to another by using the DisnixService

of the machine where the distribution function runs.

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

7.2 Transferring closures

To change the deployment state of a system, we have to transfer components from one machine to another, for instance to replace a specific service or to install an additional service into the network. To transfer a Nix component from one machine to another and still have correct dependencies, we have to copy the all the necessary components of an intra-dependency closure of a package. The algorithm of the copyClosure method is illustrated in Figure 7.1.

First we have to query what components of the closure are already installed on the target machine. This is done by invoking the printInvalidPaths method of the *Disnix Service*. All store paths of the closure that do not exist on the target machine are returned. Then we have to serialize all the missing components on the source machine, which are all Nix components of the intra-dependency closure that are not available on the target machine. Finally the serialization is sent to the *Disnix Service* and imported in the Nix store of the target machine by invoking the import method of the *Disnix Service*.

The disnix-soap-copy-closure command allows the user to efficiently copy a closure from the client machine to the target machine by sending a serialization of a closure in a SOAP message to the Disnix Service by using MTOM. For instance the following instruction copies the closure of the GNU Hello component from the client machine to the target machine which is called itchy:

```
disnix-soap-copy-closure /nix/store/a36clhh1...-hello-2.3 \
    http://itchy:8080/axis2/services/DisnixService
```

7.3 Two phase commit protocol

To make the transition from the current deployment state to the new deployment state in the distributed system an atomic operation, we use a variant of the *two-phase commit protocol* [46]. This transition phase has to be atomic because we do not want to end up in an inconsistent state in case of a failure or in the middle of the upgrade process. Either all services should be activated and deactivated as described in the distribution model or the deployment state should stay as it currently is. Also no other process should interfere in a transition phase by modifying the deployment state concurrently.

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

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

We have implemented two variants of the two-phase commit algorithm to execute an atomic upgrade, the *transition by distribution* method and the *transition by compilation* method.

7.4 Transition by distribution

In this variant of the two-phase commit protocol we build all the packages on the coordinator machine. After the packages are built the entire closure of the package will be serialized and imported in the Nix stores of the target machines. Finally the packages are activated in the Nix profile of the target machines.

In the *commit-request* or distribution phase the derivations in the distribution model are built in the Nix store on the machine of the coordinator.

If the resulting component is already in the Nix store then it will not be built again because the result of each build action is deterministic and will give the same result.

Once all services have been built, the distribution function decides what service to distribute to which target machine by calculating the intersection of the outputs of the current and the previous distribution. The closures of these services under the intra-dependency relation are transferred to the machines in the network through the Disnix interface. After receiving the closures, the interface will import the services in the Nix store on the target machine. This part of the algorithm is illustrated in Figure 7.2.

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

```
transitionByDistribution(newDistributionExport, currentDistributionExport) :
    N ← importDistributionExport(newDistributionExport)
    O ← {}
    if currentDistributionExport ≠ ε :
        O ← importDistributionExport(currentDistributionExport)
    I ← O ∩ N
    deactivate ← O − I
    activate ← N − I
    for each a ∈ activate :
        copyClosure(a.service, a.targetEPR)
    if lock(disnixInterface) = 0 :
        for each d ∈ deactivate :
            deactivate(d.targetEPR, d.service)
        for each d ∈ deactivate :
            uninstall(d.targetEPR, d.service)
        for each a ∈ activate :
            install(a.targetEPR, a.service)
        for each a ∈ activate :
            activate(a.targetEPR, a.service)
        unlock(disnixInterface)
```

Figure 7.2: transitionByDistribution: Transition to a new deployment state by copying closures of packages to the target machines

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

An advantage of this method is that all the packages are stored on one location, the coordinator machine, and in case of a reconfiguration of the machines in the network we do not have to build all packages that we have built before, unless the inputs of the build function changes. The disadvantage of this approach is that we depend on the resources of the coordinator machine to build all the packages, thus we need a coordinator machine which is powerful enough to compile all packages for the target machines. It is also possible to configure the Nix package manager to delegate builds elsewhere but currently Disnix provides no interface to support this feature.

7.5 Transition by compilation

In this variant of the two-phase commit protocol we build all the packages on the target machines. This is done by sending the closure of store derivation files of a package from the coordinator machine to the target machine. After that the packages will be built on the

```
transitionByCompilation(newDistributionExport, currentDistributionExport) :
    N ← importDistributionExport(newDistributionExport)
    O ← {}
    if currentDistributionExport ≠ ε :
        O ← importDistributionExport(currentDistributionExport)
    I ← O ∩ N
    deactivateDrvs ← O − I
    activateDrvs ← N − I
    for each a ∈ activateDrvs :
        copyClosure(a.service, a.targetEPR)
    deactivate ← {}
    activate ← {}
    for each d ∈ deactivateDrvs :
        result ← realise(d.service, d.targetEPR)
        deactivate ← deactivate ∪ result
    for each a ∈ activateDrvs :
        result ← realise(a.service, a.targetEPR)
        activate ← activate ∪ result
    if lock(disnixInterface) = 0 :
        for each d ∈ deactivate :
            deactivate(d.targetEPR, d.service)
        for each d ∈ deactivate :
            uninstall(d.targetEPR, d.service)
        for each a ∈ activate :
            install(a.targetEPR, a.service)
        for each a ∈ activate :
            activate(a.targetEPR, a.service)
        unlock(disnixInterface)
```

Figure 7.3: transitionByCompilation: Transition to a new deployment state by compiling packages on the target machines

target machine. Finally the packages are activated in the Nix profile of the target machines.

In the *commit-request* or distribution phase the derivations in the distribution model are instantiated and result in a set of store derivation files in the Nix store on the machine of the coordinator.

Once all services have been instantiated, the distribution function decides what store derivation to distribute to which target machine by calculating the intersection of the outputs of the current and the previous distribution. The closures of these store derivations under the intra-dependency relation are transferred to the machines in the network through the Disnix interface. After receiving the store derivations, the interface will import the store derivations in the Nix store on the target machine. After receiving all the store derivation files, the compilation of the services will be started on the target machines. This part of the

algorithm is illustrated in Figure 7.3.

If all steps in the commit-request phase succeed then the *commit* or transition phase will start which are same steps as in the transition by distribution method mentioned in Section 7.4.

An advantage of this method is that we can build multiple packages in parallel, since the compilation is done on each machine in the network and not on the coordinator machine. A disadvantage of this method is that packages that are shared between machines in a network will be built multiple times which could result in longer deployment times. For instance if package X is needed on 10 machines in the network, then we have to build that package 10 times. Another disadvantage is that not all customers in a commercial environment do not prefer compilation of source code on their machines, or sometimes compilation on target machines is not possible for small systems such as PDAs. Thus this transition method is probably not very suitable for use in hospital environments but it could have other purposes when there is no dedicated buildfarm available for use.

7.6 Blocking access

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

Disnix provides a disnix-tcp-proxy tool to drain TCP connections. The disnix-tcp-proxy listens on a specific TCP port which forwards the connection to the actual service. It monitors the number of active connections. The disnix-tcp-proxy-client can query the number of active connections. By using the DISNIX_LOCKMANAGER_HOOK environment variable it is possible to grant an exclusive lock if and only if there are no active connections by invoking the disnix-tcp-proxy-client.

The disnix-tcp-proxy tool is only useful for stateful TCP connections, for instance a connection with MySQL server. Some services have other bindings such as stateless connections by using the HTTP, XMPP or UDP protocols or it might even have application specific bindings. For instance one of the transport protocols SOAP uses is the HTTP protocol. The HTTP protocol connects only when executing a request, after that it closes the TCP connection. In that case using a TCP monitor is useless. In such cases the developer can implement a custom lock manger and make it available for use for the Disnix Service

by setting the DISNIX_LOCKMANAGER_HOOK environment variable.

Unfortunately blocking access is also a very *expensive* operation. In a network with many nodes and slow network connections this could result in very long delays. Blocking access can be avoided if the system being deployed is able to dynamically rebind the connections in such a way that a component in the new configuration will not "talk" to a component in the old configuration.

This solution is a *design constraint* on the system that is being deployed. The Disnix deployment system is not able to solve this in a generic way. The only feature the deployment system provides is sending a notification by invoking an external process defined in the DISNIX_LOCKMANAGER_HOOK environment variable. The system that is being deployed should solve atomicity in the transition by itself, e.g. by error detection. If the system being deployed can not guarantee atomicity then the only way the guarantee atomicity is blocking the entire system.

7.7 Distributed profiles

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

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

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

Alternatively, we extend the concept of Nix profiles to distributed profiles to also support dynamic binding. A distributed profile is similar to a Nix profile in that it maps names to locations. In contrast to Nix profiles, names and locations correspond to service names and network locations, respectively, rather than to file names and Nix store locations. Actually, these mappings correspond directly to a distribution model. A special lookup service provides remote access to a distributed profile, which is illustrated in Figure 7.4. A dis-

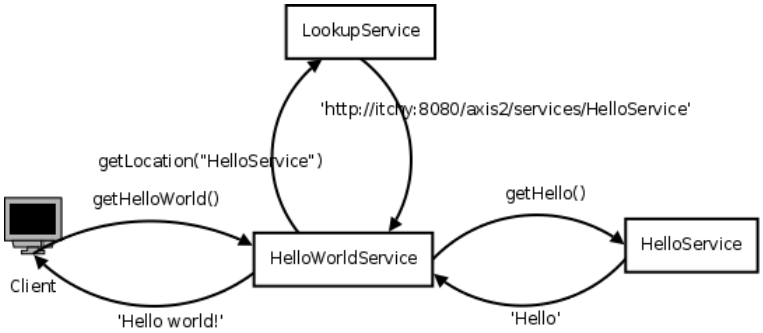


Figure 7.4: Dynamic binding of the hello world example case through a lookup service

tribution model thus serves as input for Disnix to realize a distributed deployment, and as distributed profile to dynamically bind services to their locations. Like Nix profiles, updating and downgrading of distributed profiles are atomic actions. In contrast to static binding, there is no need to update all transitive dependent services when the location of a service changes, only the corresponding mapping in the distributed profile needs to be updated.

There are, of course, more sophisticated methods available to make the deployment of web services more dynamic such as using UDDI as a lookup service, which provides lookup for more attributes then just URLs and can be distributed across multiple systems so that we no longer have a *single point of failure*. Another aproach is using XMPP as transport protocol. In this case the XMPP server guarantees the delivery of SOAP messages independend of the location of the web service.

Chapter 8

Modeling the SDS2 platform in Disnix

The final part of this research project is adapting the SDS2 platform to make it automatically deployable in distributed environments by creating models for the *Disnix deployment system*. There are some modifications we had to make to some components of SDS2, and we also had to introduce some new features to the SDS2 platform. The SDS2 platform had some weaknesses that makes the distributed deployment process hard and error prone.

In this chapter we explain the modifications we have to make to the SDS2 platform and how the Disnix models of SDS2 are constructed.

8.1 Generated config component

As mentioned in Chapter 4 all the SDS2 platform components have an intra-dependency on the config component which contains configuration information about the entire SDS2 platform, such as the locations of all web services. There are a few predefined configurations for instance devlocalhost which assumes that all services are installed on the developer machine.

To make distributed deployment possible we have to create a Nix expression that generates the XML config file that contains all locations of the services from the distribution model.

Figure 8.1 illustrates the Nix expression for the generated config:

94 Generates an attribute set that contains the name of the service, hostname of the target machine, Apache Tomcat port, MySQL port for each entry in the distribution model. The Apache Tomcat port defaults to 8080 and the MySQL port defaults to 3306 if they are not defined in the infrastructure model.

96 Generates an XML representation of the attribute set that is bound to the result of the distributionExport function which is defined in 94.

97 XSL stylesheet that transforms the XML representation of the attribute set to the structure of the SDS2 XML config file.

```
{stdenv, fetchsvnssh, apacheAnt, jdk, libxslt, configName, distribution, username, password}:

let
    distributionExport = map (
                           entry:
                           { service = entry.service.name;
                             target = entry.target.hostname;
                             tomcatPort = if entry.target ? tomcatPort
                                          then entry.target.tomcatPort else 8080;
                             mysqlPort = if entry.target ? mysqlPort
                                         then entry.target.mysqlPort else 3306; }
                         ) distribution; [94]
in
stdenv.mkDerivation {
  name = "config";
  src = fetchsvnssh {
    url = "gforge.natlab.research.philips.com/svnroot/public/sds2/branches/NixDeploy/config";
    md5 = "cf520a2b53fc76f0339cafd405db7dba";
    inherit username;
    inherit password;
  };

  src2 = fetchsvnssh {
    url = "gforge.natlab.research.philips.com/svnroot/public/sds2/branches/NixDeploy/configs";
    md5 = "f16e7ed3501f1e761d0b620deedaccf6";
    inherit username;
    inherit password;
  };

  builder = ./builder.sh; [95]
  distributionXML = builtins.toXML distributionExport; [96]
  transformXSL = ./transform.xsl; [97]

  inherit configName;
  buildInputs = [apacheAnt jdk libxslt];
}
```

Figure 8.1: Nix expression that build a generated config

[95] The builder script executes the transformation of the XML representation by using the XSL stylesheet. Finally it will compile the config component and packages the config library in a JAR file with the generated configuration file.

8.2 Lookup service

Using a generated config component is a *static* approach of binding SDS2 platform services together which is illustrated in Figure 8.2. A downside of this approach, in particular for the SDS2 platform, is that all locations are defined in the config component. If the distribution model changes we have a different build input argument of the function which builds the config component, which results in a different config component in the Nix store. Since every platform component of SDS2 has an intra-dependency on config, this results that *every* SDS2 component has to be rebuilt in case of a change in the distribution model. This approach is inflexible and expensive.

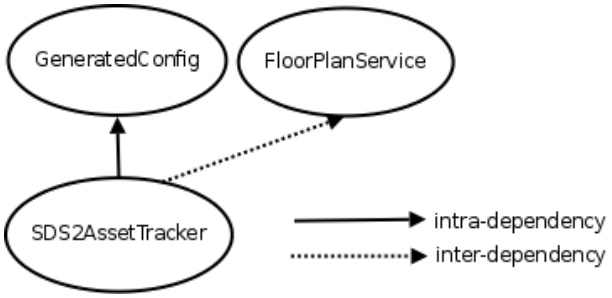


Figure 8.2: Static binding of platform services

In order to overcome the limition of static binding we have developed a *Lookup service* which is a *dynamic* approach to bind SDS2 platform services together. The additional SDS2 web service is called ConfigService. The ConfigService is basically a web service that returns properties from the configuration XML file in the config component. The ConfigService takes the generated config component mentioned in Section 8.1 as its input and returns properties from that generated config file.

We also developed a ConfigServiceInterface component which is a client to the ConfigService and a special config component that uses the ConfigServiceInterface to retrieve properties from the ConfigService instead of the XML configuration file itself which is stored inside the library. This approach is illustrated in Figure 8.3.

We have added an option to the SDS2 packages model that allows the user to use the special config component. So basically every component, except for the ConfigService has a dependency on the ConfigServiceInterface instead of a static config. As long as the location of the ConfigService does not change in the distribution model, we never have to rebuild all the SDS2 platform components, except for the ConfigService. This approach makes migrating services from and to machines in the network more flexible.

Another more sophisticated approach, that took interest by the SDS2 developers, is using XMPP as a transport protocol for SOAP. By using this approach each web service connects to an XMPP server which is connected with other XMPP servers. The XMPP server delivers each SOAP message to the specified web service connected to one of the XMPP servers in the network independent of its location. This approach will be implemented in SDS2 in the future, which is more sophisticated than using a single web service for lookup.

8.3 Service model

We also developed a *services* model for the SDS2 platform which can be used in combination with an arbitrary *infrastructure* model and *distribution* model to deploy the SDS2 platform with the *Disnix Deployment System*.

Figure 8.4 shows a part of the service model of the SDS2 platform. This model describes of what services the SDS2 platform consists and what the *inter-dependencies* of each service are. This model captures all the components and the connections of the components of

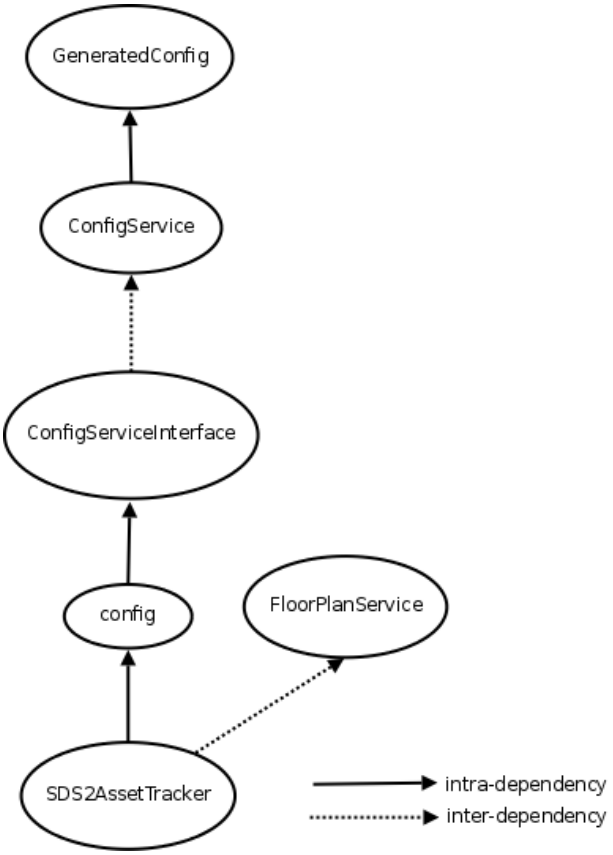


Figure 8.3: Dynamic binding of platform services by using the ConfigService

the SDS2 platform described in Section 3.8. It also imports the packages model of the SDS2 platform, that captures the intra-dependencies of the SDS2 platform. It passes the distribution model as an argument to the packages model so that the can generate a config component that contains all locations of the web services based on the distribution model. We also have to specify what type of config we want to use, which is either a generated config, that uses static binding of SDS2 services, or a lookup service to dynamically bind SDS2 services.

8.4 Activation hooks

In order to activate SDS2 platform components, putting components in the Nix profile of the target machine is not sufficient to get the components running. There is also an explicit *activation step* needed that activates the component. The SDS2 platform has three types of components: *applications* which are console applications that should be started by calling a shell script, *web services* and *web applications* which can be activated by using hot-

```
{distribution}:
let pkgs = import ../top-level/all-packages.nix
           { inherit distribution; configType = "generated"; };
in
rec {
    EMLogService = {
        name = "EMLogService";
        pkg = pkgs.SDS2.webservices.EMLogService;
        dependsOn = [ MySQLService ];
    };
    FloorPlanService = {
        name = "FloorPlanService";
        pkg = pkgs.SDS2.webservices.FloorPlanService;
        dependsOn = [ ];
    };
    HIBService = {
        name = "HIBService";
        pkg = pkgs.SDS2.webservices.HIBService;
        dependsOn = [ MySQLService ];
    };
    ME2MSService = {
        name = "ME2MSService";
        pkg = pkgs.SDS2.webservices.ME2MSService;
        dependsOn = [ FloorPlanService ];
    };
    MELogService = {
        name = "MELogService";
        pkg = pkgs.SDS2.webservices.MELogService;
        dependsOn = [ MySQLService ];
    };
    MULogService = {
        name = "MULogService";
        pkg = pkgs.SDS2.webservices.MULogService;
        dependsOn = [ MySQLService ME2MSService HIBService FloorPlanService ];
    };
    SDS2AssetTracker = {
        name = "SDS2AssetTracker";
        pkg = pkgs.SDS2.webapplications.SDS2AssetTracker;
        dependsOn = [ EjabberdService FloorPlanService HIBService EMLogService MELogService ];
    };
    SDS2Utilisation = {
        name = "SDS2Utilisation";
        pkg = pkgs.SDS2.webapplications.SDS2Utilisation;
        dependsOn = [ FloorPlanService MULogService HIBService ];
    };
    SDS2EventGenerator = {
        name = "SDS2EventGenerator";
        pkg = pkgs.SDS2.applications.SDS2EventGenerator;
        dependsOn = [ EjabberdService FloorPlanService ];
    };
    ...
}
```

Figure 8.4: services.nix: Services model for the SDS2 platform

deployment.

Apache Tomcat supports hot-deployment. Applications can be activated on a running Apache Tomcat instance by copying the WAR file into the webapps directory and can be deactivated by removing the WAR file from the webapps directory. Also Apache Axis2 supports hot-deployment by copying or removing AAR files from the WEB-INF/services directory.

```
#!/bin/sh

# Activate web applications in tomcat webapps directory

if [ -d $1/webapps ]
then
    for i in `find $1/webapps -name *.war`
    do
        cp -v $i /var/tomcat/webapps
    done
fi

# Activate web services in Axis2 services directory

if [ -d $1/webapps/axis2/WEB-INF/services ]
then
    for i in `find $1/webapps/axis2/WEB-INF/services -name *.aar`
    do
        cp -v $i /var/tomcat/webapps/axis2/WEB-INF/services
        cp -v $i /var/tomcat/temp/*-axis2/WEB-INF/services
    done
fi
```

Figure 8.5: SDS2 activation script

Figure 8.5 shows the SDS2 activation script and Figure 8.6 shows the SDS2 deactivation script which are invoked by setting the DISNIX_ACTIVATION_HOOK and DISNIX_DEACTIVATION_HOOK environment variables. The script is called with the Nix component name of the SDS2 platform component as its argument. It checks the type of component, either a *web service* or *web application* and finally it copies the archive in the right directory so that it will be hot deployed on the application server. Figure 8.6 is similar to the activation script, which removes the archives from the application service directories in order to deactivate it.

8.5 Inter-dependency relationships

Inter-dependency relationships are potential weaknesses in a distributed system. Since network links can be interrupted or failing we also have bindings between components that can be lost. In this case the system could malfunction or it might even fail completely.

We have various types of inter-dependency bindings in SDS2:

- *TCP connection*. The connection with the MySQL server is a TCP connection. If

```
#!/bin/sh

# Deactivate web applications in tomcat webapps directory

if [ -d $1/webapps ]
then
    for i in `find $1/webapps -name *.war`
    do
        rm -v /var/tomcat/webapps/`basename $i`
    done
fi

# Deactivate web services in Axis2 services directory

if [ -d $1/webapps/axis2/WEB-INF/services ]
then
    for i in `find $1/webapps/axis2/WEB-INF/services -name *.aar`
    do
        rm -v /var/tomcat/webapps/axis2/WEB-INF/services/`basename $i`
        rm -v /var/tomcat/temp/*-axis2/WEB-INF/services/`basename $i`
    done
fi
```

Figure 8.6: SDS2 deactivation script

the connection interrupts/fails it has to be created again. Usually Java programs will throw an exception in this case which should be handled properly.

- *HTTP connection*. All web services are using the SOAP protocol to communicate with each other. The SDS2 platform uses HTTP as transport protocol for SOAP messages. If the connection interrupts/fails it does not have to be created again. The HTTP protocol only creates a TCP connection when there is a message in transit. After that it will close the connection. Failures will only happen when a SOAP message is sent over a connection which is down.

- *XMPP connection*. All status and location events are sent over an XMPP channel. XMPP also uses the HTTP underlying transport protocol. XMPP also guarantees delivery of messages to clients connected to the server independent of its location. As long as we have a connection with an XMPP server a message will always be delivered to a node at some point. It is also possible to connect multiple XMPP servers to each other. In that case if a node is connected to another XMPP server the message will be sent to that other XMPP server, which is similar to e-mail.

The SDS2 platform also had some inter-dependency weaknesses, which we had to solve. For instance if a MySQL connections fails the connection will not be restored again. A failing connecting could result in a *failing system*. We solved some of these problems which are explained in this section.

8.5.1 MySQL connection handling

In the original version of SDS2 all *MySQL connections* are set up in the web services itself during initialization. If the MySQL connection fails the entire web service fails because MySQL connections are not restored in case of a failure.

Setting up MySQL connections was also a very inefficient process. In the constructor method of each web service, each web service registers the MySQL JDBC driver on the Driver Manager, connects to the MySQL database by using the settings defined in the config component. Finally all MySQL statements are sent over that connection.

```
try
{
    Class.forName("com.mysql.jdbc.Driver").newInstance();
    String conStr = Config.getInstance().getProperty("com.philips.medical.SDS2.database.url");
    Connection conn = DriverManager.getConnection(conStr);
}
catch (Exception ex)
{
    System.err.println(ex);
}
```

Figure 8.7: Connecting to a MySQL database by setting up the driver manager and connection

This solution is not very scalable because a web service relies on a single MySQL connection. It is also difficult to support features such as load balancing of database services unless the application developer implements support for it in the web service.

The Java EE standard also provides a few additional features to the standard JDBC API [49, 5]. It is also possible to manage JDBC connections on the application server. In this case we request a database connection from the application server by using the JNDI interface. The application server manages a pool of database connections which can support advanced features such as load balancing, clustering, pooling and so on. These features are hidden to the application developer. We have modified the SDS2 web service so that it will use JDBC connections which are managed by the Apache Tomcat server.

The approach mentioned in Figure 8.8 has several advantages over a creating manual database connection which is illustrated in Figure 8.7. The latter approach offers more *transparency*; we do not have to load a specific JDBC driver, in this case a MySQL JDBC driver, ourselves. The application server manages the database connections and drivers. With this approach we can build the application independent from the type of database that is being used. Another advantage of this approach is ease of *scalability*. The application server manages the database connections which is hidden from the application developer, thus there is no need for the application developer to implement complex features such as load balancing and so on.

```
try
{
    InitialContext ctx = new InitialContext();
    DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/sds2/mobileeventlogs");
    Connection conn = ds.getConnection();
}
catch (Exception ex)
{
    System.err.println(ex);
}
```

Figure 8.8: Connecting to a database by invoking the application server

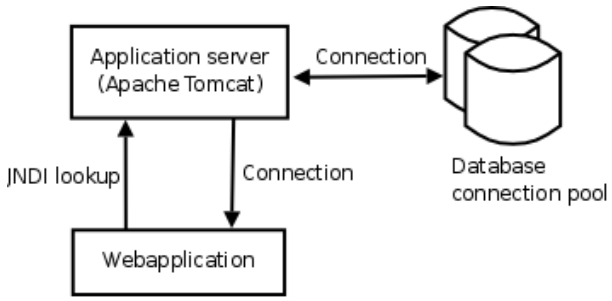


Figure 8.9: Managing database connection by the Application Server

8.5.2 Webservices binding

All the service clients which connect to the web services are initialized in the constructor of each web service and application. If the location of a specific web service changes or if a inter-dependend web service becomes unreachable the application is not able to reach it anymore, since it is not restored or modified after construction time. In that case we have to redeploy or restart that web service or application in order to make use of the new connection settings.

We modified all interfaces to the web services in SDS2 to connect to the target service before each request. By using the ConfigService component the locations of the web services can be changed dynamically without affecting the state of the web service. This is also a good design pattern which should be used in a SOA, since each service has to manage his own state [39].

8.5.3 Revised services model

By using these solutions we have less strict inter-dependencies. If a connection with a web service is (temporarily) not available only a specific feature of a web service is not available, but it will not end up in a *failing* system.

The advantage of making these dependencies less strict is that it is also easier to dynam-

ically change the deployment state of the system. We no longer have to deactivate an entire *inter-dependency closure* of services, but just the services that should be modified. Thus by modifying the SDS2 platform we have a more flexible architecture that can be installed and updated automatically.

8.6 Example

In this section we will illustrate an example deployment scenario of SDS2 in a specific test environment.

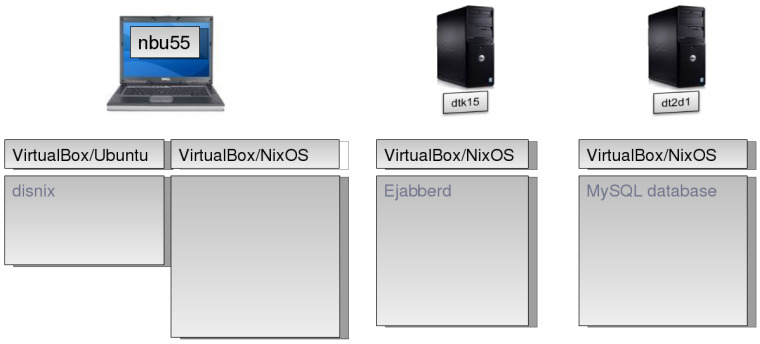


Figure 8.10: Initial deployment state of the test environment

Figure 8.10 shows a specific test environment with three machines identified by nbu55, dtk15 and dt2d1. Each machine is running NixOS inside a virtual machine with the Disnix Service on it. The nbu55 is running a second virtual machine which runs runs Ubuntu. It acts as the coordinator machine. All NixOS machines are not running any services yet, except for the *Disnix* service which provides remote access to the Nix stores and Nix profiles. The MySQL and ejabberd services are also configured in advance, because they have state data which can not be captured in the Nix deployment model and therefore it is also a limition for Disnix. For the same reason the MySQL and ejabberd service locations will never change either, because the state data can not be migrated automatically.

Figure 8.11 illustrates the infrastructure model for the environment illustrated in Figure 8.10.

In the first example deployment step we will deploy all the SDS2 platform services on the dtk15 and dt2d1 machines, which is illustrated in Figure 8.12. Figure 8.13 shows the *distribution* model that specifies the configuration illustrated in Figure 8.12.

Executing the following instruction will build all the SDS2 platform components, distribute them to the appropriate machines in the network and activates them, which results in a running SDS2 environment with services distributed across the two machines specified in Figure 8.13:

```
$ disnix-soap-env services.nix infrastructure.nix distribution.nix
```

```
{
    dtk15 = {
        hostname = "dtk15";
        targetEPR = http://dtk15:8080/axis2/services/DisnixService;
    };

    dt2d1 = {
        hostname = "dt2d1";
        targetEPR = http://dt2d1:8080/axis2/services/DisnixService;
    };

    nbu55 = {
        hostname = "nbu55";
        targetEPR = http://nbu55:8080/axis2/services/DisnixService;
    };
}
```

Figure 8.11: infrastructure.nix: Infrastructure model for the test environment

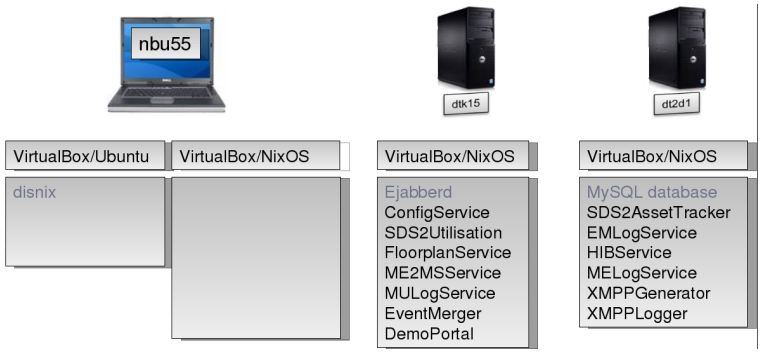


Figure 8.12: Deployment state over two machines in the test environment

In the next example we will move some services from the dtk15 and dt2d1 machines to the nbu55 machine, which is illustrated in Figure 8.14. The modifications to the distribution model are illustrated in Figure 8.15.

The following instruction will create a new distribution export file for the new configuration and returns the path to the distribution export file:

```
$ disnix-soap-gendist services.nix infrastructure.nix distribution-new.nix
/nix/store/0p7rplhgri...-distribution-export/output.xml
```

The following instruction will update the current configuration of the system to the new configuration of the system:

```
$ disnix-soap-deploy /nix/store/0p7rplhgri...-distribution-export/output.xml \
    /nix/store/jg7brpf4p7...-distribution-export/output.xml
```

```
{services, infrastructure}:

[
    { service = services.ConfigService; target = infrastructure.dtk15; }
    { service = services.EMLogService; target = infrastructure.dt2d1; }
    { service = services.FloorPlanService; target = infrastructure.dtk15; }
    { service = services.HIBService; target = infrastructure.dt2d1; }
    { service = services.ME2MSService; target = infrastructure.dtk15; }
    { service = services.MELogService; target = infrastructure.dt2d1; }
    { service = services.MULogService; target = infrastructure.dtk15; }
    { service = services.SDS2AssetTracker; target = infrastructure.dt2d1; }
    { service = services.SDS2Utilisation; target = infrastructure.dtk15; }
    { service = services.DemoPortal; target = infrastructure.dtk15; }
    { service = services.EventMerger; target = infrastructure.dtk15; }
    { service = services.SDS2EventGenerator; target = infrastructure.dt2d1; }
    { service = services.XMPPLogger; target = infrastructure.dt2d1; }
    { service = services.MySQLService; target = infrastructure.dt2d1; }
    { service = services.EjabberdService; target = infrastructure.dtk15; }
]
```

Figure 8.13: distribution.nix: Distribution model over two machines in the test environment

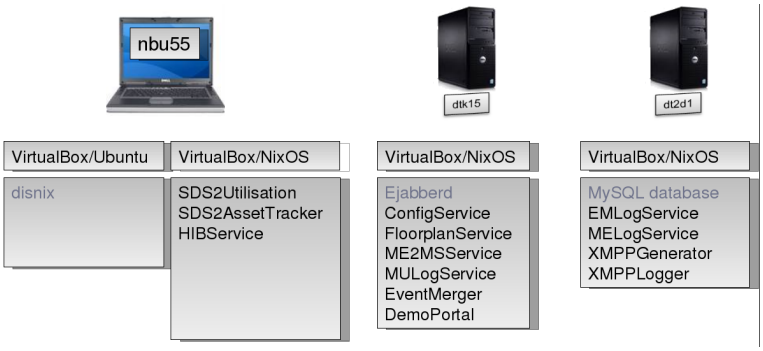


Figure 8.14: Deployment state over three machines in the test environment

The former argument is the distribution export file of the new configuration and the latter is the distribution export file of the old configuration. By executing this command only the HIBService, SDS2AssetTracker and SDS2Utilisation services will be deactivated on the dtk15 and dt2d1 machines and activated on the nbu55. All the other services keep running, which means that the entire SDS2 system stays operational, except that some features are not available during this transition phase.

In order to clean the entire SDS2 system the same strategy can also applied by using an empty distribution model which is illustrated in Figure 8.16. The deployment tool will then deactivate all services in the network. By calling the garbage collector the user can remove all obsolete services. The following instruction will remove all garbage from all the machines in the network:

```
{services, infrastructure}:

[
    { service = services.HIBService; target = infrastructure.nbu55; }
    { service = services.SDS2AssetTracker; target = infrastructure.nbu55; }
    { service = services.SDS2Utilisation; target = infrastructure.nbu55; }
    ...
]
```

Figure 8.15: distribution-new.nix: Updated distribution model which uses three machines

```
{services, infrastructure}:

[
]
```

Figure 8.16: distribution-empty.nix: Empty distribution model

```
$ disnix-soap-collect-garbage -d infrastructure.nix
```

8.7 Complexity

To deploy the entire SDS2 platform with Disnix, only 3 tasks must be performed by the user. The database schema of SDS2 should be imported in the database and the ejabberd server should be configured. Finally the entire SDS2 platform can be deployed by using the disnix-soap-env command on a network of machines.

Chapter 9

Related Work

Software deployment is not a very well explored domain in computer science, but some other research exists. This chapter discusses some solutions and approaches related to the software deployment process of distributed systems.

9.1 Dependency agnostic method

In [20] the authors propose a dependency-agnostic method of upgrading running distributed systems. This is done by deploying the old configuration and a new configuration of a distributed system next to each other in separate "worlds", which are runtime environments that are isolated from eachother. By using this method we have the old and the new configuration of a system running at the same time. A separate middleware component intercepts requests and can forward them to the new configuration at a certain point in time. The paper does not provide an implementation. Nix/Disnix partially supports this scenario. It is possible to describe two systems in one distribution model and thus deploying two configurations next to each other. Finally after activating that configuration the old configuration can be removed. The only thing that should be provided is extensions that allow running separate versions next to each other. For instance if we use Apache Tomcat we should avoid name clashes of web applications with the same name, or we have to run multiple Apache Tomcat instances on one machine. Also we have to provide a middleware component that intercepts and forwards requests from the old to the new configuration.

9.2 GoDIET

GoDIET [9] is a utility that provides automated deployment of a distributed DIET platform, which is a grid computing platform. Their definition of 'deployment' is different than the definition software deployment we use. In this case deployment is the mapping of middleware components to system resources in the grid. This definition has similarities however with software deployment which is also a mapping. An XML file is used to describe available storage and compute resources, the desired DIET agent and server hierarchy, and any add-on services to DIET. GoDIET writes all necessary 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. Disnix is much different since it is a framework focused on all kinds of distributed systems and is a more general solution whereas GoDIET is a more specific solution. Currently Disnix is only targeted at Service Oriented Architectures of which SDS2 is an example.

9.3 An architecture for distribution web services

In [2] is a case study where the authors propose an architecture for the distribution of web services by evaluating existing software distribution tools. In this paper the authors illustrated several UML models that can be used to describe the architecture of a web service that can be used by existing (local) deployment tools. They also propose to use the existing protocols of a distributed system just like Disnix does. Disnix, however, is not local-deployment tool agnostic. It relies on the Nix deployment system for local deployment. It is also not very useful to adapt Disnix for other local-deployment tools since it relies on the purely functional model Nix uses.

9.4 OMG specification for software deployment

The Object Management Group (OMG) also provides a specification for software deployment which can be applied to distributed systems as well. For instance [29] illustrates what this specification looks like. The OMG specification for software deployment uses UML, which is a modeling language developed by the OMG, to model software components and their relationships. It requires CORBA, which is a distributed communication protocol maintained by the OMG, to communicate with interfaces. Although the OMG provides a specification of how to model components and how the interfaces should communicate, it does not specify how the deployment steps are done. There are currently no systems available that use this OMG specification that I am aware of. Disnix does not use UML for modeling or CORBA as a distributed communication protocol. Since Disnix reuses the purely functional model of Nix, we also need a purely functional language to model configurations, which UML is not. We also want our deployment service to integrate with the system being deployed, therefore we do not want to be tied to CORBA as a communication protocol.

9.5 The Software Dock

The Software Dock is a distributed, agent-based deployment framework for supporting ongoing cooperation and negotiation among software producers and software consumers [27]. It uses a standardized deployment schema for describing software systems which is called the Deployable Software Description (DSD).

The Software Dock is similar to Nix/Disnix in the sense that is uses a model-driven approach to deployment and that there is an agent that acts as the coordinator of the deployment. The Software Dock however is more targeted at the delivery process of components

from producer to consumer site and not with *correct deployment*. Nix also uses the *purely functional deployment model* in order to guarantee correct deployment. The paper does not mention the mechanisms that are used in the Software Dock in order to provide correct deployment.

9.6 A SOA-based software deployment management system

In [10] the authors propose an architecture for the deployment of SOA based components by using the OSGi platform, which is a Java-based service platform that can be remotely managed. The architecture consists of four subsystems: a management agent, a deployment management server, a process management module and an operation portal. The solution proposed in this paper is not explained in detail. A similar approach is used in [33] which also uses the OSGi platform.

The scope of this solution is limited, since it is tied to Java application services and service oriented architectures. The Nix/Disnix approach of deployment is more general but requires some extensions to solve application specific settings.

9.7 Dynamic module replacement

There is also an approach illustrated in [7], where parts of a running distributed system can be replaced dynamically for components programmed in the Argus programming language system. Disnix does not use this approach since we do not target replacing parts of a running distributed system on code level, but we are focused on replacing components on deployment level.

9.8 Ad-hoc solutions

There are also many ad-hoc solutions that people use to configure distributed systems. For instance on Linux based systems, administrators create shell scripts that combine several tools with different purposes together, such as a tool to transfer files to download files, package managers and so on. For instance to transfer files from producer to consumer site tools are available, such as OpenSSH [41] or a webserver. To download software packages from the repositories of the Linux distribution author solutions are available such as APT [45] or Yum [34]. To install packages on target machines the local package manager of the Linux can be used such as RPM [24] or the Debian package manager [45]. Activation and deactivation can be done by executing commands on the target machine by using a SSH connection. Disnix tries to formalize these ad-hoc solutions and reuses the Nix primitives of software deployment for single systems.

Chapter 10

Conclusions and Future Work

This chapter gives an overview of the project's contributions. After this overview, we reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

10.1 Contributions

In this project we have developed an extension to the *Nix deployment system* for distributed systems. This extension is called the *Disnix deployment system* which is a toolset and framework that allows one to execute software deployment operations in distributed environments by using the Nix primitives of software deployment for single systems.

The *Disnix deployment system* contains many tools. One tool is a SOAP and D-Bus interface that provides remote access to the Nix store and Nix profiles of machines in the network through a standard RPC protocol, which is necessary since an RPC protocol is chosen due to internal organization policies/restrictions. Disnix also provides higher level tools that execute distributed deployment steps, such as distributed install, uninstall and activation operations by invoking the Disnix interfaces on the target machines in the network. It also provides some features to make operations *atomic*, but requires support by the system that is being deployed in order to do this efficiently.

We designed Disnix with extensibility in mind by providing integration with external processes that provide context specific solutions. The software deployment process of distributed system is *more challenging* than the software deployment of single systems. Not all software deployment activities can be solved in a generic manner. For instance the activation and deactivation of services is application specific, such as activating a web application in the Apache Tomcat container. Also blocking access to guarantee atomicity in the transition phase from an old configuration to a new configuration in a distributed system could be an expensive operation and should be avoided. This is only possible by letting the software system that is being deployed cooperate with the deployment system, and thus application specific extensions are needed.

We also showed in this thesis how to transform a semi-automatic deployment process of a particular existing distributed system (SDS2) into a fully automatic deployment process

for single systems and distributed systems by using *Nix* and *Disnix*. There were some challenges we have to overcome to make this possible however. The entire SDS2 platform and all its dependencies had to be modelled in Nix and Disnix and some modifications and additional features had to be developed in SDS2. We identified some of these design constraints in this thesis.

The SDS2 platform uses many open source components. We have contributed many new Nix expressions for open source components that have become part of the Nixpkgs collection and NixOS that everyone can use. Some components that we have contributed are: fetchsvnssh, Apache Tomcat, Jetty, erlang, ejabberd, Smack, Google Web Toolkit, MySQL JDBC driver, various GWT libraries and NixOS services to run Apache Tomcat and ejabberd.

Finally, the Disnix toolset and framework is Open Source Software (OSS) just like all the other components related to Nix such as NixOS and Nixpkgs. The deployment tools are free for use, improvement and studying. Disnix will be released soon on the NixOS website, which can be reached at `http://www.nixos.org`.

10.2 Conclusions

10.2.1 Vision on software deployment

In our vision on software deployment in Section 1.4, software deployment should be a simple process and not a complex one. Therefore we want a *fully automatic* software deployment process instead of a *semi-automatic* process. In order to make that process fully automatic we need to capture a configuration of a system in a model.

Disnix satisfies the deployment vision as we have demonstrated with the SDS2 example case. The models that Disnix supports: the *services*, *infrastructure* and *distribution* models enable capturing properties of all services and machines of the SDS2 platform. The configuration described in these models can be activated automatically. By comparing the amount of steps of the semi-automatic deployment process and the fully automatic deployment process it is obvious that the software deployment process of SDS2 is improved.

10.2.2 Software deployment process of SDS2

It is obvious that the deployment process of SDS2 is improved by using Nix and Disnix. The number of tasks to performed by a user in order to get a running SDS2 system is significantly less by using Nix than the semi-automatic deployment process, which is a reduction of 51 tasks to 9 tasks that should be performed the user.

With Disnix the amount of tasks is even less than by using the standard *Nix*, which is a reduction of 9 manual tasks to 3 manual tasks *and* it also applies to an entire network of computers instead of a single machine. The difference can be noticed by comparing the number of steps in Section 4.1 which describes the semi-automatic process, Section 4.6 which describes the deployment of SDS2 on single systems by using Nix and Section 8.6 which gives an example of how to deploy an SDS2 configuration by using Disnix.

Also the amount of time to execute the deployment is significantly less, but no exact measurements are made however, since SDS2 is still an experimental system and still in development. Also at the time that Disnix was developed SDS2 had very poor scalability support which we improved a bit later, thus we did not test SDS2 in large, realistic environments. Some modifications we made are explained in Chapter 8. Some details of *Disnix* can be improved and the scope of Disnix can be broadened in the future. This will be discussed Section 10.4.

The software deployment process of SDS2 by using Disnix is *efficient*; It only builds components when it is needed and during an upgrade of a distributed system, it will only replace the services that are different in the new configuration compared to the existing configuration. The software deployment process is also *reliable*; It reuses all concepts of Nix such as atomic commits for local deployment and it also uses a distributed transaction method to make upgrades atomic, which has some disadvantages however because blocking access can be expensive.

10.2.3 Mapping of concepts

Nix	⇔	***Disnix***
Single system	⇔	Distributed system
Packages	⇔	Services
Dependencies	⇔	Intra-dependencies, Inter-dependencies
Store derivation	⇔	Distribution export file
Realisation	⇔	Distribution
Nix profile	⇔	Distributed profile
Nix environment	⇔	Deployment state
Installing, upgrading, uninstalling components in a profile	⇔	Transition from current deployment state to new deployment state
Symlink flipping commit	⇔	Two-phase commit

Figure 10.1: Nix / Disnix analogies

Disnix also reuses the Nix primitives of software deployment in order to execute distributed software deployment operations. Figure 10.1 shows the mapping of concepts of Nix and Disnix analogies.

10.2.4 Design constraints

A mandatory design constraint that components should meet in order to be automatically deployable with the Nix deployment system is that the build process should be scripted

and non-interactive. We also identified some design constraints on components that should be deployed with the *Disnix deployment system*. Design constraints we identified in this project for distributed systems are:

- *Local deployment*. A system should be deployable on single systems by using the Nix deployment system. This is a mandatory constraint, since Disnix reuses the Nix primitives of software deployment.
- *Parametrization*. The services should be parametrized so that the information about the distribution of services in a network can be passed to services at deployment time, e.g. the HelloWorldService could have an inter-dependency on HelloService and thus the HelloWorldService must know how to reach HelloService.
- *Less strict inter-dependencies*. Inter-dependencies are run-time dependencies on services on machines in the network and are usually connected by a network link. While it is possible to specify *inter-dependency closures* and initialize them in the right order, it is difficult (even impossible) to guarantee that they are *activated* in the right order. Also, it is always possible in a network that a specific network link fails. Therefore the system that is being deployed should deal with failing/interrupting network links or the platform infrastructure should provide robustness. A system should not completely fail in case of a network problem.
- *Dynamic binding*. Services should use a dynamic binding mechanism for instance by using a lookup service or be designed in such a way that it is able to find other services dynamically, e.g. a Distributed Hash Table [48]. This prevents that services should be (partially) rebuilt if the configuration of a distributed system changes, which could be an *expensive* operation.
- *Prevent blocking*. In order to guarantee atomicity in the transition phase between an old and a new configuration access to system can be *blocked* but this is an *expensive* operation. In order to prevent blocking the system should be designed in such a way that during the transition phase from an existing configuration to a new configuration a service of the new configuration should not be talking to a service of the old configuration and the other way around. The *Disnix deployment system* can assist the system being deployed by using extension mechanisms, but is not able to solve this problem in a generic manner.

10.2.5 Hospital environments

In my opinion Disnix is a useful tool for use in hospital environments. By using this deployment system we can model configurations of distributed systems and enable new configurations automatically, correctly, faster and more efficiently. This results in less errors, a faster deployment process and lower costs. Also by using separation of concerns in the models (separate service, infrastructure and distribution models) we can reproduce a configuration of a system in an environment of choice. For instance a system in development can be tested in a test environment. In order to activate the configuration in a production environment one

can substitute the infrastructure model of the test environment with the infrastructure model of a production environment.

If the software deployment process in hospital environments becomes less expensive and more accessible then it might be possible to upgrade software system more often instead of only once or twice a year so that customers have access to new features sooner.

10.3 Discussion/Reflection

In this project we have developed the *Disnix* extension but its scope is still limited; it is focused on the deployment of service oriented architectures and it currently only integrates with technologies SDS2 uses. In this case it should be deployed on an Apache Tomcat server.

Also the *Disnix* system does not support all features (yet) that we would like to have. For instance mapping services to machines is a static process; we have to specify this mapping explicitly in a distribution model which is not very flexible. Also the infrastructure in which the services are deployed is not heterogeneous. It requires that the coordinator machine is of the same type as the target machines in the network.

We also noticed that providing a deployment system is not enough in order to automatically deploy a distributed system. There are design constraints that have to be met in order to manage the deployment process properly. For instance *blocking access* to the entire system during the transition phase from an old to a new configuration is a way to achieve atomicity but is an *expensive* operation. In order to do this more efficiently, the system has to be modified to provide atomicity without blocking the entire system.

Although we developed a tool called disnix-tcp-proxy to help manage stateful TCP connections during the transition phase, we did not implement other tools for different types of bindings. We also did not investigate how to achieve atomicity without blocking by adapting SDS2, but this is also not very meaningful for SDS2 in its current state. Since the web services of SDS2 provide abstractions over one *single* database component the system will not execute destructive operations during the transition phase. The only issue is that certain features are temporary not available during this transition phase.

10.4 Future work

The Disnix Deployment System is currently designed for the deployment of the SDS2 platform. The scope of the Disnix Deployment System could be broadened in the future. Also some useful new features can be introduced. We will discuss this future work in this section.

10.4.1 Dynamic distribution

Currently the distribution of services to machines in the network is a *static* process. The mapping should be specified explicitly by the user in the distribution model. A better approach is to do this distribution *dynamically*. In this case a distribution model is generated

from the services and infrastructure model based on a model which contains quality of services attributes.

The Disnix toolset currently contains a very simple generator called disnix-soap-dist-roundrobin which generates a distribution model from a services and infrastructure model by using the Round robin scheduling method. It is a simple distribution generator that does not use a quality of service model.

A possible more sophisticated distribution generator is a weight-based generator. In this case we deploy a web service called DisnixSimpleQualityExampleService on each node in the network which returns a weight that represents the capacity of a system. We can assign a weight to each service as well. By capturing all the weights of the nodes in the network we can distribute each service in such a way that the sum of the weights is balanced and fits in the capacity of each system. By mapping this weight to a real system property, for instance the amount of RAM, we already have a very simple quality of service support.

Another approach which is discussed in [28] is to reduce network links in order to improve reliability. They argue that the reliability of a distributed system is inversely proportional to the number of network links, since a failing/interrupting network is the major factor of a failing distributed system. A pure local deployment scenario is the ideal situation according to this paper, but not always possible. The paper proposes a graph-based deployment planning approach which can be transformed to an instance of the Multiway Cut Problem. Since that problem is an NP-complete problem, it is not doable to solve large instances of this problem. By using an approximation algorithm for this problem we can find a solution close to the optimal solution in order to find the most reliable deployment scenario. The paper does not provide any implementation. It is also possible to create a generator that uses the approach discussed in the paper to generate a distribution model. By using this method we can implement a distribution generator optimized for reliability.

In [37] the authors address some challenges of a software deployment architecture and how to provide quality of service. They also illustrate a model with several layers of quality attributes, which is basically a multi dimensional mode, and they illustrate that it is not always possible to find the optimal solution, which is a phenomenon known as *Pareto Optimal*. It could be possible to use a heuristic algorithm that finds an acceptable solution for this which we can use to generate a distribution model.

10.4.2 Support for other protocols and other types of systems

The Disnix implementation currently provides a SOAP and a D-Bus interface. The SOAP interface also requires the user to deploy an Apache Axis2 archive on an Apache Axis2 container.

There are different kinds of SOAP implementations as well as different types of distributed systems that use different communication protocols such as RMI, CORBA, DCE and so on. It is also nice to have support for different types of protocols and systems. Disnix is designed with support for multiple protocols in mind. In order to support a different RPC protocol a developer can create a new layer on top of the *core Disnix* component.

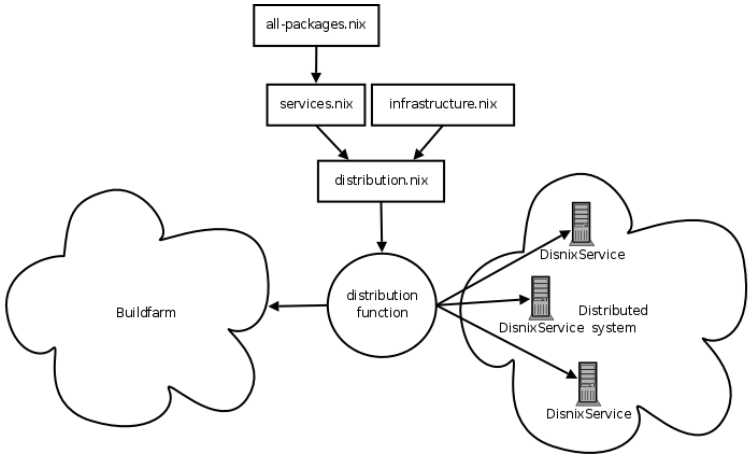


Figure 10.2: Disnix with symmetric build and deliver support

10.4.3 Symmetric build and deliver support

The Disnix toolset provides a formal interface to machines in the network that should run the system that is being deployed. Building of components is done on either the coordinator machine or on the target machines in the network. Sometimes you do not want the builds to be performed on any of those machines but elsewhere, e.g. a buildfarm [18]. The Nix package manager supports delegating builds to other machines by an external process which is provided by setting the NIX_BUILD_HOOK environment variable [15]. Disnix provides no interface yet to use this mechanism nor does it support any process to delegate these builds elsewhere by using an RPC interface.

A useful application for Disnix is to support build and deliver symmetrically as illustrated in Figure 10.2.

10.4.4 Support for heterogeneous networks

The network in which the SDS2 platform components were tested were all x86 based systems running NixOS, which is a Linux based system. Components of a distributed system could also run on multiple systems having different architectures and running different operating systems.

In order to support heterogeneous networks we need to extend the infrastructure model to capture properties such as the architecture and operating system. We also need the symmetric build and deliver support discussed in the previous section, since the coordinator machine could be a different platform than all other machines in the network and thus unable to build packages for that machine. In order to overcome that limitation the coordinator has to delegate the build to another machine that is capable of building a package for that specific target machine. We also need a decent Windows port of Nix, since support in Nix and Nixpkgs for Windows is currently very limited.

10.4.5 Implement a check phase

Although Disnix provides capturing *inter-dependencies* in a model, it cannot guarantee that a modeled system will work after it is deployed since those dependencies are *runtime dependencies*. The effect of inter-dependencies is only visible at runtime and not at build or deployment time. We can provide better guarantees of a working system by implementing a check phase which, for instance, verifies an inter-dependency by checking if the client interface to a web service conforms to a specific WSDL file or by running other kinds of tests which we should identify in the future.

10.4.6 Authentication support

The Disnix toolset currently provides no authentication support, except for the disnix-soap-client that use a username and password mechanism configured on the webserver where the Disnix SOAP interface is hosted.

Authentication is difficult to support. In addition to that we need to authenticate to an individual machine in a network we should also be able to authenticate to machines in a different domain by various authentication mechanisms such as public/private key methods and so on.

10.4.7 Hybrid transition method

In this thesis we have implemented two transition methods that update an existing configuration of a system to a new configuration, each method having its own advantages and disadvantages.

The first method is the *transition by distribution* variant discussed in Section 7.4. In this variant we build all packages on the coordinator machine and finally the packages are transfered and activated on the target machines in the network. An advantage of this method is that all the packages are stored on one location and in case of a reconfiguration we do not have to build all packages that we have built before. The disadvantage of this approach is that we depend on the resources of the coordinator machine to build all the packages. For instance if we have 50 machines each with a different service, we have to build all the 50 services on the coordinator machine which takes some time. In this case it could be faster to compile all the packages on the 50 target machines in parallel.

The second method is the *transition by compilation* variant discussed in Section 7.5. In this variant we build all packages on the target machines and finally the packages are activated on the target machines in the network. An advantage of this method is that we can build multiple packages in parallel. A disadvantage of this method is that packages that are shared between machines in a network will be built multiple times which could result in longer deployment times. Another disadvantage is that customers in a commercial environment do not prefer compilation of source code on their machines.

We can also combine both methods in order to have the advantages of both transition methods. This transition method transfers packages that are already in the Nix store on the coordinator machine to the target machines in the network. If a build of a package is missing in the Nix store on the coordinator machine the coordinator will build the package

on the specified target machine on the network. Finally the closure of the build is transferred back from the target machine to the coordinator machine and is stored in the Nix store of the coordinator machine. The advantage of this transition method is that we still have all packages stored in a central place on the coordinator machine, while it is also possible to build multiple packages in parallel on the target machines.

This method has also a disadvantage, just like the transition by compilation method, that it is not always possible to build software on target machines in the network. In hospital environments, this method is probably not attractive for use in most cases, however there are other use cases where this solution is acceptable since not everyone has a dedicated buildfarm available for building software.

10.4.8 User application front-ends

It is also useful to have end user applications for Disnix, such as tools to model a network infrastructure. Using higher level tools is easier for end users than writing Nix expressions themselves. A program transformation toolset such as Stratego/XT [51, 8] can be used to generate Nix expressions from domain specific models.

10.4.9 Design constraints

We identified some design constraints in Section 10.2.4 based on our experience with adapting the SDS2 platform. In the future we have to identify more of these constraints. Also we have to investigate methods to achieve atomicity during the transition phase from an old to a new configuration of a system without blocking the entire system, which is an expensive operation.

10.4.10 An architecture for Pull Deployment of Services

The Disnix framework and toolset is a basis for the Pull Deployment of Services (PDS) project, which is a research project in collaboration with Delft University of Technology and Philips Healthcare and funded by NWO/Jacquard [50]. The goal of this project is to build an architecture for Pull Deployment of Services by investigating new techniques and developing tools and frameworks that can be applied in hospital environments to offer services in a flexible manner. Services should be deployed and activated *on demand*, i.e. *pull deployment of services*.

Pull deployment requires considering the properties and state of an application, the network, and the current requirements and capabilities of the user. The network topology and configuration is dynamic, since there are machines added and removed to the network and some may break. Such events require a redistribution of service components which is done automatically.

Figure 10.3 illustrates the architecture of various types of a specific service. A service can be built in several ways, such as a thin client where only the graphical user interface is deployed on the target machine of the user and all computations are done on a server, and a fat "thick" client where all the computations are done on the machine of the user. Fig-

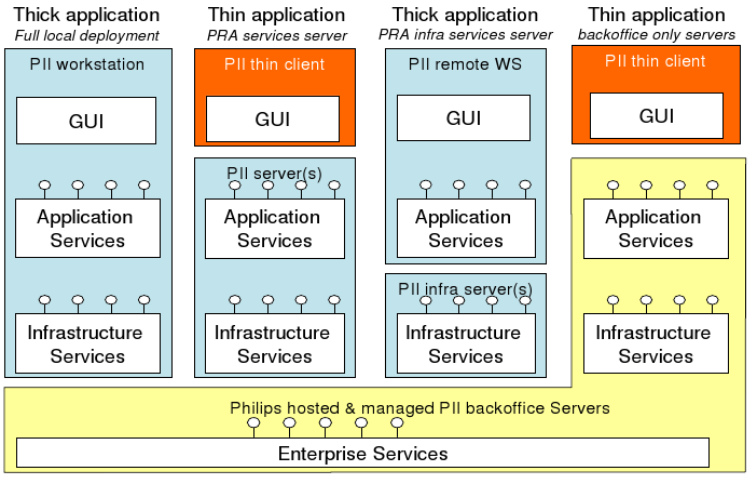


Figure 10.3: Pull Deployment of Services

ure 10.3 also illustrates two variants between a thin and fat client, where some components implementing a service are located on a client and others on servers.

We need to offer multiple variants of services, since the infrastructure of a hospital is heterogeneous. In a hospital we have a wide range of devices ranging from medical equipment to workstations. For instance a small device such as a PDA needs a thin client of a specific service, since it lacks the system resources to do all computations by itself, while a workstation could use a fat client of a service because it has plenty of system resources available. In this case we also reduce the amount of used system resources on the server.

The Disnix toolset and framework fits in this architecture to make the deployment of services in a distributed environment possible. In the future more concepts will be implemented in Disnix and the scope of Disnix will be broadened which we mentioned in the earlier sections of this chapter. Except for the deployment part which Nix and Disnix provides, other issues should be investigated as well.

For instance in order to offer services on demand and construct them in such a way that we can offer the same service as a thin, fat or something "in between" system, we have to investigate how we can build applications in such a way. For instance a thin client has an *inter-dependency* binding with the server component, since it uses a specific RPC protocol with network communication, while a fat client has an *intra-dependency* with the same component since it uses a binding mechanism such as library that is linked to the component. Also the binding for the fat client application is realized at build time whereas the binding for the thin client application is realized at runtime.

Another challenge is that services should fit in the software deployment system (Disnix) and should meet the design constraints we identified earlier in this thesis and possible more design constraints that we have yet to identify. For instance blocking an entire hospital environment during the upgrade phase in not an option so we have to find ways to avoid blocking. We preferably want to develop services in this architecture model driven as well.

Bibliography

[1] Carzaniga A., Fuggetta A., Hall R. S., Van Der Hoek A., Heimbigner D., and Wolf A. L. A characterization framework for software deployment technologies. In *Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado*, April 1998.

[2] Rainer Anzböck, Schahram Dustdar, and Harald Gall. Software configuration, distribution, and deployment of web-services. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 649–656, New York, NY, USA, 2002. ACM.

[3] Apache Software Foundation. *Apache Ant User Manual*. Also at `http://ant.apache.org/manual/index.html`.

[4] Apache Software Foundation. Apache Axis2 installation guide, 2008. Also at `http://ws.apache.org/axis2/1_4/installationguide.html`.

[5] Apache Software Foundation. Apache Tomcat 6.0 - documentation index, 2008. Also at `http://tomcat.apache.org/tomcat-6.0-doc/index.html`.

[6] Apache Software Foundation. Handling binary data with axis2 (mtom/swa), 2008. Also at `http://ws.apache.org/axis2/1_4/mtom-guide.html`.

[7] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. Ph.D., MIT, 1983. Also as MIT LCS Tech. Report 303.

[8] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[9] Eddy Caron, Pushpinder Kaur Chouhan, and Holly Dail. Godiet: A deployment tool for distributed middleware on grid 5000. Technical Report RR-5886, Laboratoire de l'Informatique du Parallélisme (LIP), April 2006. Also available as INRIA Research Report 5886.

[10] Ing-Yi Chen and Chao-Chi Huang. An soa-based software development management system. In *WI '06: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 617–620, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. 2008. Also at `http://svnbook.red-bean.com/en/1.5/index.html`.

[12] Merijn de Jonge, Tijmen Hennink, and Luuk Kuiper. *Installation & Deployment of SDS2*, January 2008.

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

[14] Wouter den Breejen. Managing state in a purely functional deployment model. Master's thesis, Faculty of Science, Utrecht University, The Netherlands, March 2008.

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

[16] Eelco Dolstra, Martin Bravenboer, and Eelco Visser. Service configuration management. In *SCM '05: Proceedings of the 12th international workshop on Software configuration management*, pages 83–98, New York, NY, USA, 2005. ACM.

[17] Eelco Dolstra and Andres Löh. NixOS: A purely functional Linux distribution. In *ICFP 2008: 13th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, September 2008. To appear.

[18] Eelco Dolstra and Eelco Visser. The nix build farm: A declarative approach to continuous integration. In Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008), jul 2008.

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

[20] Tudor Dumitras, Jiaqi Tan, Zhengheng Gho, and Priya Narasimhan. No more hotdependencies: toward dependency-agnostic online upgrades in distributed systems. In *HotDep'07: Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, page 14, Berkeley, CA, USA, 2007. USENIX Association.

[21] Eclipse Foundation. Eclipse.org home, 2008. Also at `http://www.eclipse.org`.

[22] Eelco Dolstra et al. Nix packages collection. Also at `http://nixos.org/nixpkgs.html`.

[23] Ejabberd community. *ejabberd Book*, 2008. Also at http://www.ejabberd.im/book.

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

[25] Gerard Beekmans et al. Linux from scratch. Also at http://www.linuxfromscratch.org.

[26] Goole Inc. Product overview - goole web toolkit, 2008. Also at http://code.google.com/webtoolkit/overview.html.

[27] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the software dock. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 174–183, New York, NY, USA, 1999. ACM.

[28] Abbas Heydarnoori and Farhad Mavaddat. Reliable deployment of component-based applications into distributed environments. In *ITNG '06: Proceedings of the Third International Conference on Information Technology: New Generations*, pages 52–57, Washington, DC, USA, 2006. IEEE Computer Society.

[29] Petr Hnetynka. Making deployment of distributed component-based software unified. In *Austrian Computer Society*, pages 157–161, 2004.

[30] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[31] Jive Software. Ignite realtime: Smack api, 2008. Also at http://www.igniterealtime.org/projects/smack/index.jsp.

[32] Matthew Johnson. *D-Bus programming in Java 1.5*. February 2008. Also at http://dbus.freedesktop.org/doc/dbus-java/dbus-java/.

[33] Abdelmadjid Ketfi and Noureddine Belkhatir. Model-driven framework for dynamic deployment and reconfiguration of component-based software systems. In *MIS '05: Proceedings of the 2005 symposia on Metainformatics*, page 8, New York, NY, USA, 2005. ACM.

[34] Yellowdog Linux. *Yum - Trac*. 2008. Also at http://yum.baseurl.org/.

[35] R. Willems M. de Jonge, W. van der Linden. E-services for hospital equipment, September 2008.

[36] David MacKenzie. *Autoconf, Automake, and Libtool: Autoconf, Automake, and Libtool*. February 2006. Also at http://sources.redhat.com/autobook/autobook/autobook.html.

[37] Nenad Medvidovic and Sam Malek. Software deployment architecture and quality-of-service in pervasive environments. In *ESSPE '07: International workshop on Engineering of software services for pervasive environments*, pages 47–51, New York, NY, USA, 2007. ACM.

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

[39] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *VLDB J.*, 16(3):389–415, 2007.

[40] Havoc Pennington, David Wheeler, John Palmieri, and Colin Walters. D-bus tutorial. Also at `http://dbus.freedesktop.org/doc/dbus-tutorial.html`.

[41] OpenBSD project. Openssh. Also at `http://www.openssh.org`.

[42] Raman Ramsin and Richard F. Paige. Process-centered review of object oriented software development methodologies. *ACM Comput. Surv.*, 40(1):1–89, 2008.

[43] P. Saint-Andre. *RFC 3920: Extensible Messaging and Presence Protocol (XMPP): Core*, October 2004. Also at `http://tools.ietf.org/html/rfc3920`.

[44] Scott James Remnant. upstart - event-based init daemon, 2008. Also at `http://upstart.ubuntu.com`.

[45] Gustavo Noronha Silva. *APT HOWTO*. 2004. Also at `http://www.debian.org/doc/manuals/apt-howto`.

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

[47] Raphaël Slinckx. Dbus activation tutorial. Also at `http://raphael.slinckx.net/blog/documents/dbus-tutorial`.

[48] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[49] Sun Microsystems. Jdk 6 java database connectivity (jdbc)-related apis and developer guides. Also at `http://java.sun.com/javase/6/docs/technotes/guides/jdbc`.

[50] Eelco Visser, Merijn de Jonge, and Eelco Dolstra. Pull deployment of services, May 2008. Also at `www.jacquard.nl/8/assets/File/Vierde%20ronde/PDS-Jacquard-kickoff.pdf`.

[51] Eelco Visser and Martin Bravenboer et al. Stratego program transformation language. Also at `http://www.strategoxt.org`.