Disnix User’s Guide

Draft (Version 0.6.1)
Contents

1 Introduction 1
  1.1 Features .............................................................. 1
  1.1.1 Declarative distributed systems modeling ......................... 1
  1.1.2 Complete dependencies ......................................... 2
  1.1.3 Atomic upgrades and rollbacks .................................. 2
  1.1.4 Garbage collection ................................................ 2
  1.1.5 Portability ....................................................... 2
  1.1.6 Extensibility ..................................................... 2
  1.1.7 Data migration and backups ...................................... 3
  1.2 License ............................................................. 3

2 Overview ........................................................... 4

3 Installation ....................................................... 6
  3.1 Installation on NixOS ................................................ 6
  3.1.1 Configuration of the coordinator machine ....................... 6
  3.1.2 Configuration of a target machine ................................ 6
  3.2 Manual installation ................................................ 7
    3.2.1 Installing Dysnomia from source .............................. 7
    3.2.2 Installing Disnix from source .................................. 7
      3.2.2.1 Prerequisites ............................................... 7
      3.2.2.2 Compiling Disnix ........................................... 8
    3.2.3 Installing Disnix with Nix ...................................... 8
    3.2.4 Configuring the D-Bus service .................................. 8
    3.2.5 Starting the Disnix service on startup ........................ 8
      3.2.5.1 Composing an init.d script ............................... 9
      3.2.5.2 Composing a Windows system service .................... 10
    3.2.6 Setting the log directory permissions ........................ 10
    3.2.7 Configuring the SSH protocol wrapper ........................ 10
    3.2.8 Configuring Dysnomia containers (optional) ................. 11
  3.3 Configuring user accounts ........................................ 12
  3.4 Additional SSH settings .......................................... 12
4 A basic usage scenario

4.1 Background

4.2 Architecture

4.3 Writing Disnix deployment models

4.3.1 A basic Nix example

4.3.2 Writing a Disnix expression for a service

4.3.3 Intra-dependency composition

4.3.4 Services model

4.3.5 Infrastructure model

4.3.6 Distribution model

4.4 Usage

4.4.1 Deploying a system from scratch

4.4.2 Upgrading a system

4.4.3 Roll back to a previous configuration

4.4.4 Deploying a system and building the services on the target machines

4.4.5 Collecting garbage

5 Managing state

5.1 Annotating services in the services model

5.2 Usage

5.2.1 Deploying a system and migrating its state

5.2.2 Snapshotting and restoring the state of a system

5.2.3 Cleaning snapshots on the target machines

5.3 Notes on state deployment

6 Deploying target-specific services

6.1 Why are services target-agnostic by default in Disnix?

6.2 Deploying target-agnostic services

6.3 Deploying target-specific services

6.3.1 Manually specifying target-specific services

6.3.2 Generating target-specific services

7 Architecture

7.1 Communication flow

7.2 Composition of disnix-env

7.3 Low-level usage examples

7.3.1 Building a system on the coordinator machine

7.3.2 Building services on target machines

7.3.3 Distributing services to target machines

7.3.4 Deactivating obsolete services and activating services on target machines
8 Using Disnix as a remote package deployer

8.1 Specifying packages as services ................................................. 40
8.2 Composing packages locally ...................................................... 41
8.3 Writing a minimal services model .............................................. 41
8.4 Configuring the remote machine’s search paths ............................. 42
8.5 Deploying packages ................................................................. 42
8.6 Deploying any package from the Nixpkgs repository ..................... 43
8.7 Multi-user package management ................................................. 44

9 Dysnomia modules

9.1 Structure ................................................................................. 45
9.2 Dysnomia module for the mysql-database type ......................... 45
9.3 Implementing a custom activation interface ............................... 47

10 Advanced options

10.1 Configuring a custom connection protocol ................................ 49
10.2 Managing multiple distributed system configurations ................ 49
10.3 Enabling state deployment by default ...................................... 50
10.4 Multi-container deployment ..................................................... 50

11 Extensions

11.1 DisnixWebService ................................................................... 53
11.2 DisnixOS ................................................................................. 53
11.3 Dynamic Disnix ......................................................................... 53

A Command Reference

A.1 Main commands ................................................................. 54
A.1.1 disnix-collect-garbage ..................................................... 54
A.1.2 disnix-clean-snapshots .................................................... 55
A.1.3 disnix-env ................................................................. 55
A.2 Utilities ............................................................... 57
A.2.1 disnix-capture-infra ....................................................... 57
A.2.2 disnix-activate .............................................................. 58
A.2.3 disnix-build ................................................................. 59
A.2.4 disnix-client ................................................................. 59
A.2.5 disnix-copy-closure ...................................................... 61
A.2.6 disnix-copy-snapshots .................................................. 62
A.2.7 disnix-delete-state ....................................................... 63
A.2.8 disnix-distribute ........................................................... 64
A.2.9 disnix-gendist-roundrobin ............................................... 64
A.2.10  disnix-instantiate ................................................................. 65
A.2.11  disnix-lock ................................................................. 66
A.2.12  disnix-manifest ............................................................... 67
A.2.13  disnix-query ................................................................. 68
A.2.14  disnix-restore ................................................................. 68
A.2.15  disnix-service ................................................................. 69
A.2.16  disnix-set ................................................................. 70
A.2.17  disnix-snapshot ............................................................... 70
A.2.18  disnix-ssh-client ............................................................. 71
A.2.19  disnix-visualize ............................................................... 74
List of Figures

2.1 An overview of Disnix .......................................................... 4
3.1 Starting the Disnix Windows system service ............................. 11
3.2 A Dysnomia container configuration file for a MySQL server ....... 11
3.3 A Dysnomia container configuration file for a MySQL server ...... 12
4.1 Architecture of the StaffTracker ........................................... 13
6.1 Deployment architecture containing a single target-agnostic reverse proxy .................................................. 30
6.2 Deployment architecture containing multiple target-agnostic reverse proxies ................................................. 31
6.3 Deployment architecture containing multiple target-specific reverse proxies .................................................. 32
7.1 Communication flow of the deployment operations .................. 36
7.2 Architecture of the StaffTracker ........................................... 37
Chapter 1

Introduction

Disnix is a distributed service deployment toolset which main purpose is to deploy service-oriented systems (i.e. systems that can be decomposed into "distributable units") into networks of machines having various characteristics (such as operating systems) and is built on top of Nix; a package manager which has some unique features compared to conventional package managers to make deployment safe, reliable, and reproducible.

Most of Nix’s unique features stem from the way packages are stored. In Nix, packages are stored in isolation in a special purpose directory called the **Nix store**. Every package has a special file name such as:

```
/nix/store/h3ybhij06f3lhjx0p9axfcbyg9z9bljj-firefox-3.6.8
```

The first part `h3ybhij06f3lhjx0p9axfcbyg9z9bljj` is a cryptographic hash derived from all the inputs (e.g. libraries, compilers, build scripts) used to build a package. If a user decides to build the component with, say, a different compiler, a different hash will be computed and the package has a different filename. Hashing makes it safe to store multiple versions and variants next to each other, because no package shares the same name.

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

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

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

Because of these features, Nix is a very good basis to build a distributed service deployment tool to make the deployment process of service-oriented systems efficient, reliable and atomic. However, Nix has a few limitations -- it only handles **intra-dependencies**, which are either run-time or build-time dependencies residing on the same machine. In order to deploy a distributed service-oriented system into a network of machines, additional features are required, which are provided by Disnix.

Most importantly, Disnix supports models to describe machines in the network, and manages **inter-dependencies** -- the run-time dependencies between components residing on different machines. Besides building and transferring services, Disnix also takes care of all remaining deployment activities, such as activation and deactivation steps of the services of which the system consists.

1.1 Features

1.1.1 Declarative distributed systems modeling

Like the standard Nix deployment system, Disnix uses the Nix expression language, which is used to write specifications for the deployment of distributed systems.
Disnix requires three kinds of models, each capturing a specific deployment concern. The services model is used for specifying the components of a distributed system and its inter-dependencies. The infrastructure model is used for specifying the network of machines and their relevant properties. The distribution model is used to map services to machines in the network.

1.1.2 Complete dependencies

The Nix package manager ensures that package dependency specifications are complete on a single system, i.e. intra-dependencies. Components of a distributed service-oriented system may have dependencies on components running on different machines in a network, i.e. inter-dependencies.

Disnix allows someone to specify inter-dependencies of distributed system components, which can be composed into a complete distributed service-oriented system. If a service is deployed, Disnix ensures that its inter-dependencies are deployed first so that we never have a failing system due to missing dependencies.

Moreover, Disnix uses inter-dependency specifications for the installation or upgrade process of a distributed system to ensure that every service is activated or deactivated in the right order.

1.1.3 Atomic upgrades and rollbacks

Like the Nix package manager, which support atomic upgrades, Disnix extends this concept to service-oriented systems by mapping the concepts of the two-phase commit protocol onto Nix deployment operations to upgrade a distributed system (almost) atomically. Since the Nix package manager always stores components next to each other in a Nix store and never overwrites existing files, upgrading a distributed system is very safe and we can almost always perform a rollback.

The only impure step while upgrading, is the deactivation of obsolete services and activation of newly installed services, a phase in which users may observe that the system is changing. To make this process truly atomic, Disnix uses an extension mechanism that can be used to temporary queue/block incoming connections until the transition is finished. We have developed a simple prototype example with stateful TCP connections to demonstrate this.

1.1.4 Garbage collection

Like the Nix package manager, Disnix provides a distributed garbage collector, which safely removes all obsolete packages from the machines in the network.

1.1.5 Portability

Disnix is, like Nix, supported on several platforms including most Unix flavours such as Linux, FreeBSD, OpenBSD and Mac OS X. Disnix is also supported on Windows using Cygwin.

Apart from the portability of Disnix itself, Disnix also allows a user to deploy a service oriented system into a heterogeneous network (i.e. a network consisting of various types of machines, running various operating systems). Disnix reuses Nix’s delegation mechanism to build a component for an alien target platform. Optionally, it can also delegate builds to the target machines in the network.

1.1.6 Extensibility

Since service-oriented systems can be deployed in heterogeneous networks consisting of various platforms and using various communication protocols, and their components can have basically any form, not all deployment operations can be executed in a generic manner.

The architecture of the Disnix toolset is very modular. Disnix uses a plugin system called Dysnomia to integrate customly developed modules to execute various non-generic deployment activities, and a plugin system that provides remote access through various RPC protocols.

Currently, Disnix includes a SSH wrapper which can be used to access remote machines through a SSH connection. A separate extension that uses SOAP + MTOM is also available. A custom extension can be developed in a straight forward manner.
1.1.7 Data migration and backups

Disnix can also optionally take and restore snapshots of the state of deployed services for backup and migration purposes, e.g. when a service is moved from one machine to another.

Since the snapshot and restore operations differ among component types, Dysnomia’s plugin system is consulted to execute the required snapshot and restore operations for a given service type.

1.2 License

Disnix is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. Disnix is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.
Chapter 2

Overview

The figure above illustrates Disnix’s deployment concepts in a nutshell. In the center of the figure the disnix-env command line tool is shown, which takes care of performing a complete deployment process of a distributed system.

This tool requires three models as input parameters shown on the left side of the figure:

- The services model describes of which distributable components (called services) the system consists, their inter-dependencies and their types. The service model has a reference to a file named all-packages.nix, which captures all the intra-dependency compositions of every service, such as libraries, configuration files and compilers.

- The infrastructure model describes all the machines in the network and their relevant properties and capabilities.

- The distribution model maps services defined in the services model to machines defined in the infrastructure model.

On the right side of the figure, the network is shown in which the system has to be deployed. Every machine in the network requires a DisnixService instance to be installed, so that remote deployment steps can be performed from the coordinator machine.
The machine on which `disnix-env` is executed is called the *coordinator* machine. The machines in the network are called *targets*.

By writing instances of the specifications shown above and running the following command on the coordinator machine:

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

All the services that are defined in the distribution model (including all their intra-dependencies) are built from source code, then transferred to the machines in the network and finally activated in the right order.

By adapting one of the models and by running `disnix-env` again, an upgrade is performed. In case of an upgrade, only services that have changed are built from source and transferred to the target machines. Moreover, the services that are no longer in the new configuration are deactivated and the new services that were not in the old configuration are activated (also in the right order without breaking any inter-dependencies), making this phase efficient and reliable.

Since the coordinator machine may be of a different type (e.g. CPU or operating system) as one of the machines in the network, it may not be possible to compile a service on the coordinator machine for a given target platform. In such cases, Disnix can also build a service on a target machine in the network or use Nix to delegate a build to a machine capable of building it.

Optionally, Disnix can also move the state of a service from machine to another. For example, when deploying databases, Disnix only ensures that they are created with a schema and initial dataset. However, when the structure and contents of a database evolves, their changes are not migrated automatically. As a solution, Disnix has a mechanism that snapshots, moves and restores state of a service if it has been explicitly enabled for that particular service.
Chapter 3

Installation

This chapter explains how Disnix can be installed. Disnix can be used on many kinds of software distributions, such as Linux, Mac OS X and Windows (through Cygwin). In NixOS (the Nix-based Linux distribution) the deployment of Disnix is fully automated -- it only requires a user to enable it in the NixOS configuration. In other software distributions, Disnix must be installed manually.

Moreover, Disnix must be installed on both the coordinator machine responsible for carrying out the deployment process, and each available target machine in the network that must run the Disnix service exposing deployment operations remotely.

3.1 Installation on NixOS

Deploying NixOS-based configurations is quite straightforward. A coordinator machine simply requires the presence of the Dysnomia and Disnix utilities in the system environment or a Nix user profile. Each target machine requires the Disnix service to be deployed and a number of container services.

3.1.1 Configuration of the coordinator machine

We can add the required Disnix utilities to the system environment by adding the following line to `/etc/nixos/configuration.nix`:

```nix
environment.systemPackages = [ pkgs.dysnomia pkgs.disnix ];
```

and rebuilding the NixOS configuration:

```
$ nixos-rebuild switch
```

3.1.2 Configuration of a target machine

A NixOS-based target machine requires a number of additional aspects to be configured.

First, we must enable the core Disnix service that exposes deployment operations through D-Bus in the NixOS configuration file (`/etc/nixos/configuration.nix`):

```nix
services.disnix.enable = true;
```

The deployment operations must be exposed remotely. Disnix supports various kinds of protocol wrappers to accomplish this. To use SSH (which is the default), we must enable the OpenSSH service in the NixOS configuration:

```nix
services.openssh.enable = true;
```
Besides SSH it also possible to use other kinds of communication protocols. Chapter 10 has more information on this.

Disnix executes a number of non-generic deployment activities that are carried out by Dysnomia. Simply enabling a container service in a NixOS configuration suffices to instruct the Dysnomia configure script to install the corresponding plugins. For example, we can enable MySQL and Apache Tomcat in a NixOS configuration:

```makefile
services.mysql.enable = true;
services.tomcat.enable = true;
```

With these two options enabled, Disnix will consult Dysnomia that has been automatically configured by the NixOS module system to deploy MySQL databases to the MySQL DBMS and Java web applications to the Apache Tomcat servlet container.

### 3.2 Manual installation

If it is desired to install Disnix manually, we must first install the Disnix and Dysnomia packages on every machine in the network. On each target machine, we must perform a number of additional steps to get the Disnix service to work properly.

#### 3.2.1 Installing Dysnomia from source

The `dysnomia` package is GNU Autotools based, which must be installed by executing the following commands:

```
$ ./configure options...
$ make
$ make install
```

The installation path can be specified by passing the `--prefix=prefix` to `configure`. The default installation directory is `/usr/local`. You can change this to any location you like. You must have write permissions to the `prefix` path.

The configure script tries to detect which Dysnomia modules can be used on the system. For example, if the configure script is able to detect the `mysql` command on the host system, then the `mysql-database` module is configured and installed. As result, a MySQL database can be deployed on this system (this requires you to somehow install MySQL first, e.g. with the distribution’s package manager or with Nix).

It may be possible that not every capability can be detected automatically. Moreover, some modules may need some manual configuration. Invoke the command:

```
$ ./configure --help
```

for more information about Dysnomia’s configuration parameters. Dysnomia’s `README.md` file can also be consulted for more general information.

#### 3.2.2 Installing Disnix from source

##### 3.2.2.1 Prerequisites

In order to build Disnix from source code, a number of dependencies are required. Disnix uses XML for representing the lower level data formats and requires `libxml2` and `libxslt` to parse and transform them. It uses `glib` as a utility library and means to communicate through D-Bus.

Since Disnix is built on top of the Nix package manager it also requires Nix to be installed on the same machine. Consult the Nix homepage for more details on how to install it.
3.2.2 Compiling Disnix

After unpacking or checking out the Disnix sources, it can be compiled by executing the following commands:

```
$ ./configure options...
$ make
$ make install
```

When building from the Git repository, these should be preceded by the command:

```
$ ./bootstrap
```

The installation path can be specified by passing the `--prefix=prefix` to `configure`. The default installation directory is `/usr/local`. You can change this to any location you like. You must have write permissions to the `prefix` path.

3.2.3 Installing Disnix with Nix

A more convenient way to install Disnix is using the Nix package manager. The Nixpkgs collection has Disnix (as well as some of its extensions) packaged. It can be deployed by running the following command-line instruction:

```
$ nix-env -i disnix
```

3.2.4 Configuring the D-Bus service

The Disnix service is a D-Bus service operating on the system bus. As a dependency, it requires D-Bus to be running on the same system, which is typically part of the standard installation in nearly any modern Linux distribution.

On different kinds of software distributions, such as FreeBSD or Cygwin, the D-Bus package must be installed manually. Consult your host system’s package manager for more information.

Besides installing D-Bus, it is required that the Disnix service configuration file (which has been installed in `/usr/local/etc/dbus-1/system.d`) resides in the right location so that it is allowed to register itself on the system bus with the right access permissions. In most Linux distributions the configuration files of the system services reside in `/etc/dbus-1/system.d`.

The following command copies the configuration file into the right location:

```
$ cp /usr/local/etc/dbus-1/system.d/disnix.conf /etc/dbus-1/system.d
```

If a single user Nix installation has been performed (which is typically the case on non-NixOS systems), you may want to change the root user permission into the user that owns Nix. For example, we may want to change this line:

```
<policy user="root">
```

into:

```
<policy user="sander">
```

to grant the user: `sander`, that owns a single user Nix installation, the permissions to own the Disnix service as well.

3.2.5 Starting the Disnix service on startup

Another important installation concern is to start the Disnix D-Bus service automatically on startup. The following sub sections describe how to accomplish this for several kinds of service managers.
3.2.5.1 Composing an init.d script

Most conventional Linux distributions support starting and stopping system services by composing an init script that typically resides in `/etc/init.d/disnix`. An init.d script for the Disnix service could look like this:

```bash
#!/bin/sh
# Start/stop the disnix-service.
#
### BEGIN INIT INFO
# Provides: disnix-service
# Default-Start: 2 3 4 5
# Default-Stop:
# Short-Description: Disnix Service
# Description: Exposes deployment operations to remote machines that are
carried out by Nix and Dysnomia
### END INIT INFO

PATH=/home/sander/.nix-profile/bin:/bin:/usr/bin:/sbin:/usr/sbin
DESC="disnix service"
NAME=disnix-service
DAEMONUSER=sander
DAEMON=/home/$DAEMONUSER/.nix-profile/bin/disnix-service
PIDFILE=/var/run/disnix-service.pid
SCRIPTNAME=/etc/init.d/"$NAME"

test -f $DAEMON || exit 0

./lib/lsb/init-functions
case "$1" in
  start) log_daemon_msg "Starting disnix service" "$NAME"
     start-stop-daemon --start --quiet --pidfile $PIDFILE --user $DAEMONUSER --name "$NAME" --background --chuid $DAEMONUSER --exec $DAEMON $EXTRA_OPTS
     log_end_msg $? ;;
  stop) log_daemon_msg "Stopping disnix service" "$NAME"
     start-stop-daemon --stop --quiet --user $DAEMONUSER --name $NAME --retry 5
     RETVAL=$?
     [ $RETVAL -eq 0 ] && [ -e "$PIDFILE" ] && rm -f $PIDFILE
     log_end_msg $RETVAL ;;
  restart) log_daemon_msg "Restarting disnix service" "$NAME"
     $0 stop
     $0 start ;;
  status) start-stop-daemon --status --pidfile $PIDFILE --name $NAME &> exit 0 || exit $?
     ;;
  *) log_action_msg "Usage: /etc/init.d/disnix-service {start|stop|status|restart}" exit 2 ;;
esac
exit 0
```

Refer to your distribution’s init.d script style to see how services are configured and launched. For convenience, a Debian-compatible init.d script named disnix-service.initd has been placed in the `$PREFIX/share/doc` folder of this package.

An important aspect to keep in mind is that both Nix and Dysnomia should be in the `PATH` of the init.d script so that the service can execute all required deployment activities. Another important thing is that the Disnix service should be started after the D-Bus system service and stopped in exactly the opposite order.
If a single user Nix installation has been performed, then the `DAEMONUSER` environment variable should correspond to the name of the user that is allowed to use it. The name should correspond to `root` in case of a multi-user Nix installation.

### 3.2.5.2 Composing a Windows system service

Windows/Cygwin does not use boot scripts for starting and stopping system services. Instead, it provides the `cygrunsrv` command, to run Cygwin programs as Windows system services.

Since the core Disnix daemon is a D-Bus service, we need to run the D-Bus system daemon, which can be configured by executing the following command:

```bash
$ cygrunsrv -I dbus -p /usr/bin/dbus-daemon.exe -a '--system --nofork'
```

The Disnix service can be configured as follows:

```bash
$ cygrunsrv -I disnix -p /usr/local/bin/disnix-service.exe \
-e 'PATH=/bin:/usr/bin:/usr/local/bin' \
-y dbus -u sander
```

The `-u` parameter specifies under which the Disnix service runs. If a single user Nix installation has been performed, this username should be substituted by the actual username under which Disnix has been installed. In multi-user Nix installations, the `-u` parameter should be omitted.

The user under which the Disnix service runs should have service logon permissions. To check which permissions a user has, run:

```bash
$ editrights -u sander -l
```

It should list `SeServiceLogonRight`. If this is not the case, this permission can be granted by running:

```bash
$ editrights -u sander -a SeServiceLogonRight
```

### 3.2.6 Setting the log directory permissions

The Disnix service writes log entries for each operation that it executes. If a single user Nix installation has been performed, you probably want to grant the user that owns the installation the rights to own the log directory as well:

```bash
$ mkdir -p /var/log/disnix
$ chown sander:users /var/log/disnix
```

### 3.2.7 Configuring the SSH protocol wrapper

Disnix also needs to be remotely connectible. In order to connect through SSH, you must install an SSH server, such as OpenSSH. Consult your system distribution’s package manager for more information.

On Cygwin, we can configure SSH by running the following command-line instruction:

```bash
$ ssh-host-config
```

After configuring the services, you probably need to activate them for the first time, which can be done by the Windows service manager (Control Panel -> System and Security -> Administrative Tools -> Services). You need to pick the Disnix service and select the start option. If you want to use the SSH server, you need to pick and start the `CYGWIN sshd` service as well. A screenshot of this is shown in Figure 3.1.
3.2.8 Configuring Dysnomia containers (optional)

Although not required, it is also possible to configure Dysnomia containers on every target machine in the network so that the state of mutable components can be managed locally.

For example, we can declare a MySQL DBMS server as a Dysnomia container by writing the following Dysnomia container configuration file (`/etc/dysnomia/containers/mysql-database`):

```
mysqlUsername=root
mysqlPassword=secret
mysqlPort=3306
type=mysql-database
```

The configuration shown in Figure 3.2 states that we have a MySQL server that binds to TCP port 3306 and requires some authentication credentials to connect to it.

We can use the above configuration in combination with a mutable component configuration that defines a database:
create table author
( AUTHOR_ID INTEGER NOT NULL,
 FirstName VARCHAR(255) NOT NULL,
 LastName VARCHAR(255) NOT NULL,
 PRIMARY KEY(AUTHOR_ID)
);

create table books
( ISBN VARCHAR(255) NOT NULL,
 Title VARCHAR(255) NOT NULL,
 AUTHOR_ID VARCHAR(255) NOT NULL,
 PRIMARY KEY(ISBN),
 FOREIGN KEY(AUTHOR_ID) references author(AUTHOR_ID) on update cascade on delete cascade
);

Figure 3.3: A Dysnomia container configuration file for a MySQL server

With the following command-line instruction, we can deploy the database schema defined in Figure 3.3 to the MySQL server:

```
$ dysnomia --operation activate --component ./schema.sql --container mysql-database
```

When Dysnomia containers have been preconfigured on the target machines in the network, Disnix can automatically capture their configurations and generate a Disnix infrastructure model from them through the `dysnomix-capture-infra(1)` command.

Without preconfigured Dysnomia containers, it is still possible to deploy any service using hand written infrastructure models, but this is typically tedious, time consuming and error prone.

More information on how to configure Dysnomia containers can be found in the Dysnomia README.md file.

### 3.3 Configuring user accounts

By default, only the super user and users who are members of the disnix group may access operations of the core Disnix service. In order to access the Disnix operations remotely, either an account with the right permissions is required or the protocol wrapper should perform the authentication to the core Disnix service.

The SSH wrapper, for instance, uses the credentials of the calling user on the coordinator by default. Therefore, every target machine requires the user to be defined in `/etc/passwd` and the user should be member of the disnix group.

In NixOS, the disnix user group is automatically added. For other systems this must be done by the system administrator. On most systems this user group can be added by typing:

```
$ groupadd disnix
```

A particular user can be made member of the disnix group by the following command-line instruction (`someuser` must be replaced by a desired username):

```
$ usermod -a -G disnix someuser
```

### 3.4 Additional SSH settings

If an SSH connection is used, disnix-env may ask you to provide user credentials for each operation. This is not a bug, but an implication of using SSH. In order to make this process non-interactive, you must either generate an SSH keypair through ssh-keygen or use ssh-agent to remember the authentication settings.
Chapter 4

A basic usage scenario

In this chapter, we show a basic Disnix usage scenario. The purpose of this chapter is to demonstrate how to write simple models capturing various deployment aspects of a service-oriented system, and how to use Disnix to automatically deploy such a configuration to a network of machines.

We use a particular flavour of the StaffTracker system as an example, which is a toy system consisting of MySQL databases, Java web services, and Java web applications. Other types of service-oriented systems, with different types of components, can be deployed by Disnix as well, such as PHP web applications and UNIX processes. There are other StaffTracker variants available implementing the same concepts using different kinds of technologies.

Some examples (including the StaffTracker) can be found on my Github page: https://github.com/svanderburg?tab=repositories.

4.1 Background

The example used in this section is a system which is used to manage staff of a fictional university. This system stores various kinds of records about staff members, such as their names, room numbers and IP addresses.

The system can determine a zipcode from the staff member’s room number. From the zipcode it can determine the address of the building. From the IP address of a staff member, the system can determine the current location of the staff member. All the data repositories are stored in separate databases. Each data repository can be accessed through a web service.

4.2 Architecture

Figure 4.1 shows the architecture of the StaffTracker system. The architecture consists of three layers: a data layer, service layer and presentation layer. The database layer contains MySQL databases, storing data records such as zipcodes and staff
members. The web service layer contains web service components, which expose create, read, update and delete operations for each data set (the GeolocationService uses GeoIP to track a location). In the presentation layer, the StaffTracker web application front-end is shown, that can be used by end users to manage staff of a university.

All the components shown in the picture are distributable deployment units (or services). For instance, the GeolocationService can be deployed to a different machine in the network as the StaffTracker web application front-end. The arrows denote inter-dependency relationships. Inter-dependency correspond to network links between services. In this particular example, they are SOAP/HTTP or plain TCP connections.

4.3 Writing Disnix deployment models

The Nix package manager builds components from specifications called Nix expressions, written in the Nix expression language. Disnix also uses the Nix expression language to capture deployment aspects and defines service builds in a quite similar way. In this section, we first show how an ordinary Nix expression is written. Then we explain how this concept is extended to service-oriented systems.

4.3.1 A basic Nix example

Example 4.1 Nix expression for the GNU Hello package

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

stdenv.mkDerivation {
  name = "hello-2.1.1";
  src = fetchurl {
    md5 = "70c9ccf9fac07f762c24f2df229078ad";
  }
  buildInputs = [ perl ];
  meta = {
    description = "GNU Hello, a classic computer science tool";
    homepage = http://www.gnu.org/software/hello/;
  }
}
```

Example 4.1 shows a Nix expression for the GNU Hello package, a trivial example package containing the hello command showing the famous "Hello World" quote.

1. A Nix expression defines a function in which every argument represents a dependency to build a package. This particular example takes three arguments:
   - stdenv is a package representing a standard environment containing a set of basic UNIX build utilities, such as cat, ls and gcc.
   - The fetchurl argument refers to a function that is used to download a source tarball from a particular URL.
   - The perl argument corresponds to the Perl interpreter.
2. In the body of the function, we invoke stdenv.mkDerivation which is used to compose an isolated environment in which a build is performed. In this example, we have not specified the build steps that must be executed to build the package. If no build steps are given, then the builder assumes that this package is a GNU Autotools based package and basically executes the following instructions: ./configure;make;make install.
3. Every package requires a name, which becomes part of the filename of the resulting package filename in the Nix store. It also gives a user the option to look it up after it has been installed.
This attribute specifies the location of source code that we want to compile. In our example, it is bound to the result of the `fetchurl` function invocation, which downloads the GNU Hello source tarball from the GNU FTP site. The MD5 hash is used to verify whether the source tarball matches the expected version.

This attribute is used to specify which packages must be used in the build environment. In our example, `perl` has been provided as a build input. The builder automatically sets the `PATH` and `PERL5LIB` environment variables, so that Perl can be found by build process that is executed in the builder environment. Not providing a dependency also makes it (nearly) impossible for a build process to find it.

We can also specify some meta data for a package, so that we have a description and other useful information, such as the homepage and a license. These properties are not used while building the package.

Although the expression in Example 4.1 defines how to build a package from source code and its dependencies, we cannot use this expression to build the package directly because we do not know which version/variant of the dependencies we want to use, such as the version of Perl. Therefore, we have to compose the package in a separate expression, in which we call the function shown earlier with its required parameters.

**Example 4.2 all-packages.nix: Partial composition expression**

```nix
rec {  
  stdenv = ...;
  fetchurl = ...;
  perl = import ../pkgs/perl {
    inherit stdenv fetchurl;
  };
  hello = import ../pkgs/hello {
    inherit stdenv fetchurl perl;
  };
  ...
}
```

Example 4.2 shows a partial composition expression, in which the GNU Hello build function is called with its required function arguments. Moreover, all the other packages where GNU Hello depends on are also composed in this expression.

This expression is a mutually recursive attribute set in which attributes can refer to each other. Without recursion, we cannot pass attributes in this set as function parameters.

Here, the expression defined in Example 4.1 is imported and called with the its required function parameters. The corresponding dependencies are composed in the same composition expression shown in Example 4.2.

We can also use the function parameters to make different kinds of compositions of the same package. For example, by replacing `inherit perl;` (which is syntactic sugar for `perl =perl;`) with `perl =perl54;` we can build GNU Hello using a different version of Perl.

All the dependencies of the GNU Hello package, such as the Perl interpreter are composed in this expression as well.

There is often a lot of boilerplate code that must be written to compose packages. However, in most cases, the function arguments of each package correspond to attributes with the same name. The `callPackage {}` function can be used to automatically compose the package by providing the compositions with the same names as the function arguments. Using this function considerably reduces the amount of code that must be written.
Example 4.3 all-packages.nix: Simplified partial composition expression

```nix
rec {
  stdenv = ...;
  fetchurl = ...;
  perl = callPackage ../.pkgs/perl { }
  hello = callPackage ../.pkgs/hello { }
  ...
}
```

Example 4.3 shows a simplified equivalent of the previous composition expression using callPackage {}.

### 4.3.2 Writing a Disnix expression for a service

Similar to writing a Nix expressions for arbitrary packages, every service deployed by Disnix also requires a expression describing how to build it from source code and its dependencies. A Disnix expression is nearly identical to an ordinary Nix expression. Its only difference is that it also takes inter-dependencies into account while configuring or building a component.

Example 4.4 Disnix expression for the ZipcodeService

```nix
let
  contextXML = ''
  <Context>
    <Resource name="jdbc/ZipcodeDB" auth="Container" type="javax.sql.DataSource"
      maxActivate="100" maxIdle="30" maxWait="10000"
      username="${zipcodes.target.container.mysqlUsername}" password="${zipcodes.target.container.mysqlPassword}"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://${zipcodes.target.properties.hostname}:${toString (zipcodes.target.container.mysqlPort)}/${zipcodes.name}?autoReconnect=true" />
  </Context>
'';

stdenv.mkDerivation {
  name = "ZipcodeService";
  src = ../../services/webservices/ZipcodeService;
  buildInputs = [ apacheAnt ];
  AXIS2_LIB = "${axis2}/lib";
  AXIS2_WEBAPP = "${axis2}/webapps/axis2";
  buildPhase = "ant generate.war";
  installPhase = ''
    ensureDir $out/conf/Catalina
    cat > $out/conf/Catalina/ZipcodeService.xml <<EOF
    ${contextXML}
    EOF
    ensureDir $out/webapps
    cp *.war $out/webapps
  '';
}
```

Example 4.4 shows a Disnix expression for a particular web service component of the StaffTracker system, called ZipcodeService, which provides access to records in the zipcode database. A Disnix expression is a nested function (having two function headers instead of one).
This is the outer function header, which specifies all the local dependencies, or intra-dependencies. Intra-dependencies are all build-time and run-time dependencies located on the same system:

- `stdenv` provides an environment containing basic UNIX utilities and build environment.
- The `apacheAnt` argument refers to Apache Ant that is required to compile the project.
- The `axis2` argument refers to the Apache Axis2 library that is used to implement web services.

This is the inner function header, which defines all the inter-dependencies. The `zipcodes` argument refers to a database in which all the zipcodes are stored. The database may be running on a different machine as the `ZipcodeService`. The service establishes a TCP connection to the database.

Each inter-dependency parameter is an attribute set, which properties are taken from the services model, shown in Section 4.3.4.

In addition, each inter-dependency parameter provides an attribute named `targets` referring to list of target machines to which the inter-dependency has been deployed. This mapping is defined in the distribution model shown in Section 4.3.5. In the majority of cases, an inter-dependency only maps to one machine. The attribute `target` refers to the first list element for convenience.

Each target attribute set defines two properties. The `properties` attribute refers to general machine properties and correspond to the properties attribute defined in the infrastructure model. The `container` attribute refers to the set of container properties to which the services has been deployed.

This string composes a so-called context XML file. This is a configuration file used by Apache Tomcat to configure web application specific settings. One of these configuration settings are database settings.

To compose this string, we use the inter-dependency parameter `zipcodes` to generate a JDBC connection string that can be used to connect to the remote database. We use the machine-specific `hostname` attribute to determine the hostname to remotely connect and the `container` specific properties to fetch MySQL’s port number and authentication credentials.

Like ordinary Nix expressions, we must call the derivation function to build a component from source code (that stores its build result in the Nix store). It also requires you to define similar build attributes, such as `name` that is used to identify a package.

Here, the context XML file defined earlier is written into a text file and bundled with the generated web application so that the configuration settings can be found by the Apache Tomcat web service. The Dysnomia module for Apache Tomcat takes care of the activation process.

Example 4.4 shows you how to build and configure an Apache Axis2 web service. Other types of services may have different kinds configuration steps, but can also be configured and built by Disnix, like ordinary Nix packages. What Disnix basically provides is the locations of the inter-dependencies and the properties of the services and machines.

### 4.3.3 Intra-dependency composition

Like ordinary Nix expressions, we cannot use the expression in Example 4.4 directly to build the service. We need to compose it by calling the function with its required arguments. With Disnix, we need to compose a service twice. First, we have to compose this expression locally, by calling it with the required intra-dependency arguments. Later, we have to compose it using the inter-dependency arguments.

**Example 4.5 Intra-dependency composition for the StaffTracker**

```nix
{system, pkgs}:
  rec {
    ### Databases
    rooms = import ../pkgs/databases/rooms {
      inherit (pkgs) stdenv;
    };
    staff = import ../pkgs/databases/staff {
```
Example 4.5 shows a Nix expression in which services are composed locally by calling the expressions with its required intra-dependency arguments.

1. Like the pkgs/top-level/all-packages.nix file in Nixpkgs, this composition expression is a mutually recursive attribute set in which attributes can refer to each other.

2. Here, the expression from Example 4.4 is imported and called with the right intra-dependency arguments. The dependencies are either defined in the same model or in Nixpkgs.

We can also simplify the intra-dependency composition by composing our custom callPackage {} function:

Example 4.6 Simplified intra-dependency composition for the StaffTracker

```nix
{system, pkgs}:

let
  callPackage = pkgs.lib.callPackageWith (pkgs // self); ❹

  self = {
    ### Databases
      rooms = callPackage ../pkgs/databases/rooms { }; ➋
      staff = callPackage ../pkgs/databases/staff { }; ➋
      zipcodes = callPackage ../pkgs/databases/zipcodes { }; ➋
    ### Web services + Clients
      ZipcodeService = callPackage ../pkgs/webservices/ZipcodeService { }; ➋
      ZipcodeServiceClient = callPackage ../pkgs/webservices/ZipcodeServiceClient { }; ➋
    ...
  }
```
In the simplified expression, we compose a custom \texttt{callPackage}() function that first consults \texttt{self} containing our custom intra-dependency compositions and then any package provided by Nixpkgs.

### 4.3.4 Services model

Apart from specifying how to build packages and what its local (intra) dependencies are, we need to provide additional settings to allow it to be deployed into a network of machines. The distributed deployment aspects of the components of a service-oriented system are captured in the services model.

Example 4.7 Services model for the StaffTracker

```nix
{system, pkgs, distribution, invDistribution}: 1
let customPkgs = import ../top-level/all-packages.nix { inherit system; }; 2
in rec { 3

### Databases
zipcodes = {
  name = "zipcodes";
  pkg = customPkgs.zipcodes;
  dependsOn = {};
  type = "mysql-database";
};
...

### Web services
ZipcodeService = {
  name = "ZipcodeService";
  pkg = customPkgs.ZipcodeService;
  dependsOn = {
    inherit zipcodes;
  };
  type = "tomcat-webapplication";
};
...

### Web applications
StaffTracker = {
  name = "StaffTracker";
  pkg = customPkgs.StaffTracker;
  dependsOn = {
    inherit GeolocationService RoomService StaffService ZipcodeService;
  };
  type = "tomcat-webapplication";
};
}
```

Example 4.7 represents the services model of the StaffTracker system. This model is a function returning an attribute set in which each attribute corresponds to a distributable component in the architecture diagram, shown earlier in Figure 4.1.

1 A services expression is a function taking four arguments:
• The `system` parameter takes an identifier of a system architecture that we want to build a service for. An example of an identifier is: `i686-linux` representing a 32-bit x86 Linux machine.

• The `pkgs` parameter refers to the Nixpkgs collection for the corresponding system identifier. Nixpkgs contains a large collection of free and open source packages (including some proprietary ones).

• The `distribution` parameter refers to the distribution model, in which services are mapped onto machines in the network (this model is shown in Section 4.3.6).

• The `invDistribution` parameter refers to an inverse distribution model. In Chapter 6 we show how this model can be used to generate target-specific services.

Here, the expression from Example 4.5 is imported so that we can use the intra-dependency compositions of all packages implementing the services.

Like the intra-dependency composition expression in the previous example, the services expression is also a mutually recursive attribute set in which attributes can refer to each other. This is required to pass the services as inter-dependency arguments to each service.

Every attribute represents a service (i.e. a distributable component). Services correspond to the components shown in the architecture in Figure 4.1.

Every service has a canonical name, so that it is known to which one is referred from the inter-dependency arguments. This name must match the attribute name.

For every service, we must know how to build it and what their intra- and inter-dependencies are. This attribute refers to the intra-dependency closure of the service, which is composed in Example 4.6.

We also need to know for each service what the inter-dependencies actually are. Inter-dependencies correspond to the arrows shown in Figure 4.1.

In this particular example, we specify that the attribute `zipcodes` (a service also defined in the services model) is an inter-dependency of the ZipcodeService.

Similar to intra-dependency parameters, you can also create different inter-dependency compositions. For instance by specifying: `zipcodes = mycustomzipcodes;` instead of `inherit zipcodes;` (syntactic sugar for `zipcodes = zipcodes;`), you can configure the ZipcodeService to use a different database.

Finally, we must know how to activate and deactivate a service. Since services can represent nearly anything, such as a database or web application, we cannot perform this step generically. This attribute specifies the type of the service, which is used by Disnix to consult the Dysnomia plugin on the target machine performing the corresponding non-generic deployment steps.

Various types of services are supported by Dysnomia, such as: `tomcat-webapplication` which will activate a web application on Apache Tomcat, `mysql-database` which will import a MySQL database schema on first startup and `process` which activates a generic UNIX process. Moreover, also a `wrapper` type exists, allowing a developer to include his own activation script inside a service.

The `package` type can be used to indicate that the service is an ordinary Nix package that should be deployed remotely. Services of this type will be installed, but no additional deployment steps will be performed, such activation or deactivation. Additional steps besides installing are not necessary for packages. See Chapter 8 for more information.

### 4.3.5 Infrastructure model

Besides the services of which a system consists, we need to know what machines are available and what their relevant properties and capabilities are. These attributes are captured in the infrastructure model.
Example 4.8 Infrastructure model

```json
{
  test1 = {
    properties = {
      hostname = "test1.example.org";
    },
    containers = {
      tomcat-webapplication = {
        tomcatPort = 8080;
      },
    },
    system = "i686-linux";
  },
  test2 = {
    properties = {
      hostname = "test2.example.org";
    },
    containers = {
      tomcat-webapplication = {
        tomcatPort = 8080;
      },
      mysql-database = {
        mysqlPort = 3306;
        mysqlUsername = "root";
        mysqlPassword = "admin";
      },
    },
    system = "x86_64-linux";
    numOfCores = 1;
    targetProperty = "hostname";
    clientInterface = "disnix-ssh-client";
  }
}
```

Example 4.8 shows an infrastructure model describing two machines and their properties.

The infrastructure model contains the following basic properties:

1. The infrastructure model is an attribute set in which each attribute represents a machine in the network. This attribute refers to the properties of a machine called test2.
2. Each target can define a set of properties describing arbitrary characteristics of a machine.
3. In order to perform deployment steps remotely, we need to know how to connect to the machine. By default, the `hostname` attribute is used for this. Disnix can be configured to use a different property as well through a command-line parameter or environment variable.
4. Besides general target properties, each service has a specific type and will be deployed to a container that can host them. The `containers` attribute set defines properties of each container. These container-specific properties are used both for building the service, and for activation and deactivation.
5. The `mysqlPort` specifies on which port the MySQL can be reached. The MySQL properties in this model are used by the MySQL Dysnomia module for activating or deactivating the database.
The infrastructure model also allows you to specify a number of advanced system properties. These properties are optional and typically reasonable default values are provided for them. They only need to be specified in special circumstances:

1. A machine in the network may have a different system architecture as the coordinator machine. The `system` attribute can be used to specify the system architecture of a target machine.
   In some cases, Nix may not be able to build a service for a target machine directly, because it is incapable to run a compiler for that platform or it has no dedicated build machine with the right architecture to delegate the build to. In such cases, you can build the service on a target machine in the network.
   By omitting the `system` attribute, Disnix builds the service for the same architecture as the coordinator machine.
   This property is particularly useful when deploying to a heterogeneous network consisting of machines running various kinds of operating systems having various kinds of system architectures. You do not need a dedicated cluster of machines to build for these architecture, but you can reuse the existing deployment infrastructure.

2. In the activation phase, Disnix tries to concurrently activate as many services as possible. The `numOfCores` attribute can be used to indicate the amount of CPU cores available on the target machine. If, for example, this value is set to 2, Disnix tries to activate two services concurrently on this machine when possible.
   If this attribute is omitted, it will default to 1.

3. This is a reserved property defining which attribute in the `properties` set specifies how to connect to this machine. If this attribute is not specified then the value provided by the `--target-property` parameter, or `DISNIX_TARGET_PROPERTY` environment variable is used, which defaults to `hostname`. See Chapter 10 for more information.

4. This reserved property specifies which executable needs to be invoked to connect to the target machine. If this attribute is not specified then the value provided by the `--interface` parameter, or the `DISNIX_CLIENT_INTERFACE` environment variable is used, which defaults to `disnix-ssh-client`. See Chapter 10 for more information.

Besides specifying the available machines and their properties, you as a developer or system administrator has to responsibility that the machines in the network match the given configuration. There are some tools which can assist you while doing this job.

For example, when having Dysnomia containers preconfigured on the target machines in the network, we can capture their configurations and generate an infrastructure model from them using the `disnix-capture-infra(1)` command. This command is driven by a simple bootstrap infrastructure model only containing connectivity attributes as shown in Example 4.9.

### Example 4.9 Basic infrastructure model only containing connectivity properties

```nix
{  
  test1.properties.hostname = "test1.example.org";
  test2 = {
    properties.hostname = "test2.example.org";
    targetProperty = "hostname";
    clientInterface = "disnix-ssh-client";
  };
}
```

Running the following command will automatically capture the configurations of the machines and redirects it to the standard output:

```
$ disnix-capture-infra infrastructure-basic.nix
{  
  test1 = {
    properties = {
      "hostname" = "test1.example.org";
    };
  };
  containers = {
    "tomcat-webapplication" = {
      "tomcatPort" = "8080";
    };
  };
```
Alternatively, if only NixOS machines are used, an infrastructure model can be generated from a network of NixOS configurations. See the DisnixOS extension, described in Chapter 11 for more information.

### 4.3.6 Distribution model

Disnix also needs to know to which machine each service must be deployed. A distribution model is used to specify these mappings.

**Example 4.10** Distribution model for the StaffTracker

```json
{infrastructure}:
{
  zipcodes = [ infrastructure.test2 ];
  ZipcodeService = [ infrastructure.test1 ];
  StaffTracker = [ infrastructure.test1 infrastructure.test2 ];
  ...
}
```

Example 4.10 shows a distribution model for a particular deployment scenario of the StaffTracker system.

1. This attribute assignment states that the zipcodes service should be deployed to machine test2. The machine and its properties can be accessed in a service build through the target property of an inter-dependency argument in a Disnix expression, such as Example 4.4.

2. This attribute assignment states that the ZipcodeService service should be deployed to machine test1.

3. This attribute assignment states that the StaffTracker service should be deployed to both test1 and test2. Specifying multiple machines is useful for deploying redundant services, that can be used as a fallback or reached by load balancer. The list of machines can be accessed in a service build by referencing the targets property of an inter-dependency argument in a Disnix expression, such as Example 4.4.

As may be observed in the distribution model, each service is mapped to a target machine. Each service is supposed to be deployed to a container (predeployed to a machine) that is capable of hosting the service. By default, Disnix uses an automapping strategy to determine to which container a service gets deployed. In principle, the container has the same name as the type to which the service belongs. In most cases, automapping suffices -- you typically only run one instance of a MySQL DBMS or Apache Tomcat service on one machine.
In some unconventional cases, it may be desired to run multiple instances of the same container on one machine, e.g. two MySQL DBMSes or two Apache Tomcat services. In such cases, more control in the distribution model needed. Disnix also supports an alternative (and more verbose) distribution model notation in which the container mapping can be specified. See Chapter 10 for more information.

4.4 Usage

We can perform a number of deployment activities with the deployment models shown earlier.

4.4.1 Deploying a system from scratch

By running the following command-line instruction, the StaffTracker system can be deployed:

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

The above instruction execute all the required steps to get the system running -- the services and their dependencies are built from source code on the coordinator machine, then the services and their intra-dependencies are transferred to the target machines in the network. Finally, the services are activated.

4.4.2 Upgrading a system

Upgrading a system can be done by changing the Disnix models and running `disnix-env` again. For example, we can open the distribution model and change the line:

```
ZipcodeService = [ infrastructure.test1 ];
```

into:

```
ZipcodeService = [ infrastructure.test2 ];
```

to move the `ZipcodeService` from machine `test1` to `test2`. By running the following command:

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

Disnix upgrades the environment and only performs the steps that are necessary -- it moves the build of the `ZipcodeService` from the coordinator machine to `test2` and redeploy the web application front-end to connect to the new location of the `ZipcodeService`. All the other parts of the system will remain untouched.

4.4.3 Roll back to a previous configuration

As with the Nix package manager, Disnix retains older configurations by default unless they are explicitly garbage collected. If we regret doing the last upgrade, we can roll back to its previously deployed configuration, by running:

```
$ disnix-env --rollback
```

Besides the last configuration, we can also roll back to any previous generation that has not been garbage collected yet. The following command switches the deployment to a specific generation number:

```
$ disnix-env --switch-to-generation 1
```
4.4.4 Deploying a system and building the services on the target machines

It may be possible that the coordinator machine is not able build services for every machine in the network (e.g. if a target is a i686-cygwin and the coordinator is a i686-linux machine). In such cases, it is also possible to perform builds on machines in the network and then copy the build results back to the coordinator for further use:

$ disnix-env --build-on-targets -s services.nix -i infrastructure.nix -d distribution.nix

Before the actual configuration gets deployed, Disnix delegates the builds of all the services to target machines (instead of doing the builds itself) and retrieves the results from the target machines.

4.4.5 Collecting garbage

As with the standard Nix package manager, Disnix also offers an option to safely remove packages and their intra-dependencies that are no longer in use. The following command removes garbage from every machine defined in the infrastructure model:

$ disnix-collect-garbage infrastructure.nix

By default, Disnix will not remove all its previous deployment states. In order to remove previous deployment states, including the corresponding garbage, the -d option can be used:

$ disnix-collect-garbage -d infrastructure.nix
Chapter 5

Managing state

As described in the previous chapter, Disnix automatically deploys service-oriented systems into heterogeneous networks of machines running various kinds of operating systems. However, there is one major unaddressed concern when using Disnix to deploy a service-oriented system. Like the Nix the package manager -- that serves as the basis of Disnix --, Disnix’s deployment approach is stateless.

The absence of state management has a number of implications. For example, when deploying a database, it gets created on first startup, often with a schema and initial data set. However, the structure and contents of a database typically evolves over time. When updating a deployment configuration that (for example) moves a database from one machine to another, the changes that have been made since its initial deployment are not migrated.

In a large network of machines it can be quite costly to manage state manually. Disnix also provides experimental state management facilities to cope with this. However, they are disabled by default. To manage state for a specific subset of services, they must be annotated as such in the services model.

5.1 Annotating services in the services model

Example 5.1 Annotated services model for the StaffTracker

```nix
let customPkgs = import ../top-level/all-packages.nix { inherit system; }; in rec {### Databases zipcodes = { name = "zipcodes"; pkg = customPkgs.zipcodes; dependsOn = {}; type = "mysql-database"; deployState = true; 1 };
...
}
```

Example 5.1 shows a partial services model that is based on the services model of the StaffTracker example, described earlier in Example 4.7:

1 This boolean value indicates that Disnix should also do state deployment for this service, causing it to move data if the database if moved from one machine to another. Moreover, when executing the snapshot or restore commands, a dump of the database is made, transferred and restored.
There is an important caveat when deploying multiple redundant instances of the same services. If state deployment has been enabled, Disnix assumes that redundant instances of the services all have the same state (that are synchronized somehow). If this is not the case, then you must give each service a unique identity.

Besides annotating individual services, it also possible to globally enable state deployment for all services without annotating them. See Chapter 10 for more information.

5.2 Usage

With state management enabled, we can do various kinds of additional deployment tasks.

5.2.1 Deploying a system and migrating its state

Running `disnix-env` with an annotated services model causes it to migrate data after deploying the services. For example, when changing the target machine of the `zipcodes` database in the distribution model from:

```nix
zipcodes = [ infrastructure.test2 ];
```

into:

```nix
zipcodes = [ infrastructure.test1 ];
```

and by running the following command:

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

Disnix executes the data migration phase after the configuration has been successfully activated. In this phase, Disnix snapshots the state of the annotated services on the target machines, transfers the snapshots to the new targets (through the coordinator machine), and finally restores their state.

When state management has been enabled, Disnix attempts to restore the state of a service whenever it has been added to the services model. Similarly, when removing a service from the service model, and undeploying it, its state will be captured first and transferred to the coordinator machine.

5.2.2 Snapshotting and restoring the state of a system

In addition to data migration, Disnix can also be used as a primitive backup tool. Running the following command:

```sh
$ disnix-snapshot
```

Captures the state of all annotated services in the configuration that have been previously deployed and transfers their snapshots to the coordinator machine’s snapshot store.

Likewise, the snapshots can be restored as follows:

```sh
$ disnix-restore
```

By default, the above command only restores the state of the services that are in the last configuration, but not in the configuration before. However, it may also be desirable to force the state of all annotated services in the current configuration to be restored. This can be done as follows:

```sh
$ disnix-restore --no-upgrade
```
5.2.3 Cleaning snapshots on the target machines

The snapshots that are taken on the target machines are not deleted automatically. Disnix can clean the snapshot stores of all the machines in a network:

```
$ disnix-clean-snapshots --keep 3 infrastructure.nix
```

The above command deletes all but the last three snapshot generations from all machines defined in the infrastructure model.

Keep in mind that Disnix always transfers the snapshots to the coordinator machine, so there is typically little reason (besides improving the efficiency of the snapshot operations) to keep them on the targets. All snapshots on the target machines can be wiped as follows:

```
$ disnix-clean-snapshots --keep 0 infrastructure.nix
```

5.3 Notes on state deployment

Although Disnix supports state deployment, it has been disabled by default. The reason why this is the default policy is that its mechanisms may neither be the preferred nor the optimal way of doing it.

Disnix uses Dysnomia’s plugin system that invokes external tools taking care of snapshotting and restoring the state of the corresponding components. The good part of this approach is that it is general and works for all kinds of mutable components. Moreover, the tools that are consulted by the plugin system typically dump state in a portable and consistent format, which is also useful for a variety of reasons.

However, some of the drawbacks of this approach is that the process of snapshotting and restoring portable dumps is typically very slow, especially for large data sets. Moreover, the snapshot tools also write dumps entirely to the filesystem which is not always desired.

Alternative approaches of doing state deployment are the following:

- **Filesystem level snapshots.** This is an approach that works typically much faster. However, some drawbacks of working on filesystem level is that physical state may be inconsistent (because of incomplete write operations), and non-portable. Also, for some types of containers, it is difficult to manage chunks of data, such as individual databases. Filesystem level state management is unsupported by Disnix and must be done by other means.

- **Replication engines.** Replication engines of DBMSes can typically move data from one machine to another much faster and more efficiently. Disnix does not take care of configuring a DBMS’ replication engine automatically. Instead, while deploying a machine’s configuration this feature must be explicitly enabled by the deployer.
Chapter 6

Deploying target-specific services

As explained in the previous chapters, services in a Disnix context have many forms and Disnix uses plugin system to execute deployment steps that cannot be execute generically. Moreover, Disnix can also optionally manage state albeit the provided facilities have some expensive drawbacks.

Besides the earlier described properties, services have another important characteristic from a deployment perspective. By default, services are target-agnostic, which means that they always have the same form regardless to what machine they are deployed in the network. In most cases this is considered a good thing.

However, there are also situations in which we want to deploy services that are built and configured specifically for a target machine.

6.1 Why are services target-agnostic by default in Disnix?

This property actually stems from the way "ordinary packages" are built with the Nix package manager which is used as a basis for Disnix.

Nix package builds are influenced by its declared inputs only, such as the source code, build scripts and other kinds of dependencies, e.g. a compiler and libraries. Nix has means to ensure that undeclared dependencies cannot influence a build and that dependencies never collide with each other.

Moreover, since it does not matter where a package has been built, we can, for example, also download a package built from identical inputs from a remote location, instead of building it ourselves improving the efficiency of deployment processes.

In Disnix, Nix’s concept of building packages has been extended to services in a distributed setting. The major difference between a package and a service is that services take an additional class of dependencies into account. Besides the intra-dependencies that Nix manages, services may also have inter-dependencies on services that may be deployed to remote machines in a network. Disnix can be used to configure services in such a way that a service knows how to reach them and that the system is activated and deactivated in the right order.

As a consequence, it does not take a machine’s properties into account when deploying it to a target machine in the network unless a machine’s properties are explicitly provided as dependencies of a service. In many cases, this is a considered a good thing. One of the things we could do is changing the location of the StaffTracker web application front-end service (the example shown in Chapter 4), by changing the following line in the distribution model:

```
StaffTracker = [ infrastructure.test2 ];
```

to:

```
StaffTracker = [ infrastructure.test1 ];
```

Performing the redeployment procedure is actually quite efficient. Since the intra-dependencies and inter-dependencies of the StaffTracker service have not changed, we do not have to rebuild and reconfigure the StaffTracker service. We can simply take
the existing build result from the coordinator machine (that has been previously distributed to machine test1) and distribute it to test2. Also, because the build result is the same, we have better guarantees that if the service worked on machine test1, it should work on machine test2 as well.

(As a sidenote: there is actually a situation in which a service will get rebuilt when moving it from one machine to another while its intra-dependencies and inter-dependencies have not changed. As shown in Example 4.8 Disnix also supports heterogeneous service deployment meaning that we can run target machines having different CPU architectures and operating systems. For example, if test2 were a Linux machine and test1 a Mac OS X machine, Disnix attempts to rebuild it for the new platform. However, if all machines have the CPU architecture and operating system this will not happen).

6.2 Deploying target-agnostic services

Target-agnostic services are generally considered good because they improve reproducibility and efficiency when moving a service from machine to another. However, in some situations you may need to configure a service for a target machine specifically.

An example of a deployment scenario in which we need to deploy target-specific services, is when we want to deploy a collection of Node.js web applications and an nginx reverse proxy in which each web application should be reached by its own unique DNS domain name (e.g. http://webapp1.local, http://webapp2.local etc.).

This particular scenario is implemented as an example package, known as the Disnix virtual hosts example.

We could model the nginx reverse proxy and each web application as (target-agnostic) distributable services, and deploy them in a network with Disnix as follows:

![Figure 6.1: Deployment architecture containing a single target-agnostic reverse proxy](image)

We can declare the web applications to be inter-dependencies of the nginx service and generate its configuration accordingly.

Although this approach works, the downside is that in the above deployment architecture, the test1 machine has to handle all the network traffic including the requests that should be propagated to the web applications deployed to test2 making the system not very scalable, because only one machine is responsible for handling all the network load.

We can also deploy two redundant instances of the nginx service by specifying the following attribute in the distribution model:

```text
nginx = [ infrastructure.test1 infrastructure.test2 ];
```

The above modification yields the following deployment architecture:
The above deployment architecture is more scalable -- now requests meant for any of the web applications deployed to machine test1 can be handled by the nginx server deployed to test1 and the nginx server deployed to test2 can handle all the requests meant for the web applications deployed to test2.

Unfortunately, there is also an undesired side effect. As all the nginx services have the same form regardless to which machines they have been deployed, they have inter-dependencies on all web applications in the entire network including the ones that are not running on the same machine.

This property makes upgrading the system very inefficient. For example, if we update the webapp3 service (deployed to machine test2), the nginx configurations on all the other machines must be updated as well causing all nginx services on all machines to be upgraded, because they also have an inter-dependency on the upgraded web application.

In a 2 machine scenario with 4 web applications, this inefficiency may still be acceptable, but in a big environment with tens of web applications and tens of machines, we most likely suffer from many (hundreds of) unnecessary redeployment activities bringing the system down for a unnecessary long time.

### 6.3 Deploying target-specific services

A more efficient deployment architecture would be the following:
We deploy two target-specific nginx services that only have inter-dependencies on the web applications deployed to the same machine. In this scenario, upgrading webapp3 does not affect the configurations of any of the services deployed to the test1 machine.

### 6.3.1 Manually specifying target-specific services

A dumb way to specify target-specific services is defining for each service and target machine pair, a service in the service model:

**Example 6.1** A services model with statically composed target-specific services

```plaintext
{pkgs, system, distribution, invDistribution}:

let
    customPkgs = ...
  in
  rec {
    ...
    nginx-wrapper-test1 = rec {
        name = "nginx-wrapper-test1";
        pkg = customPkgs.nginx-wrapper;
        dependsOn = {
            inherit webapp1 webapp2;
        };
        type = "wrapper";
    };
    nginx-wrapper-test2 = rec {
        name = "nginx-wrapper-test2";
        pkg = customPkgs.nginx-wrapper;
        dependsOn = {
            inherit webapp3 webapp4;
        };
        type = "wrapper";
    };
  }
```
And then distributing them to the appropriate target machines in the Disnix distribution model:

**Example 6.2** A distribution model with target-specific services mapped statically

```nix
{infrastructure}:
{
...  
nginx-wrapper-test1 = [ infrastructure.test1 ];
nginx-wrapper-test2 = [ infrastructure.test2 ];
}
```

Manually specifying target-specific services is quite tedious and labourious especially if you have tens of services and tens of machines. We have to specify `machines x components services` resulting in hundreds of target-specific service configurations.

Furthermore, there is a bit of repetition. Both the distribution model and the service models reflect mappings from services to target machines.

### 6.3.2 Generating target-specific services

A better approach would be to generate target-specific services. An example of such an approach is to specify the mappings of these services in the distribution model first:

**Example 6.3** A distribution model with target-specific services generated dynamically

```nix
let
  inherit (builtins) listToAttrs attrNames getAttr;
in
{
  webapp1 = [ infrastructure.test1 ];
  webapp2 = [ infrastructure.test1 ];
  webapp3 = [ infrastructure.test2 ];
  webapp4 = [ infrastructure.test2 ];
} //
# To each target, distribute a reverse proxy
listToAttrs (map (targetName: {
  name = "nginx-wrapper-${targetName}";
  value = [ (getAttr targetName infrastructure) ];
}) (attrNames infrastructure))
```

In Example 6.3, we statically map all the target-agnostic web application services, and for each target machine in the infrastructure model we generate a mapping of the target-specific nginx service to its target machine.

We can generate the target-specific nginx service configurations in the services model as follows:

**Example 6.4** A services model with dynamically generated target-specific services

```nix
{system, pkgs, distribution, invDistribution}:
let
  customPkgs = import ../top-level/all-packages.nix {
    inherit pkgs system;
  };
in
{
```
To generate the nginx services, we iterate over a so-called inverse distribution model mapping targets to services that has been computed from the distribution model (mapping services to one or more machines in the network).

The inverse distribution model is basically just the infrastructure model in which each target attribute set has been augmented with a services attribute containing the properties of the services that have been deployed to it. The services attribute refers to an attribute set in which each key is the name of the service and each value the service configuration properties defined in the services model:

**Example 6.5** A partial inverse distribution model

```json
{
    test1 = {
        services = {
            nginx-wrapper-test1 = {
                name = "nginx-wrapper-test1";
                pkg = customPkgs.nginx-wrapper;
                dependsOn = {
                    inherit webapp1 webapp2;
                },
                type = "wrapper";
            }
        },
        webapp1 = ...,
        webapp2 = ...,
    },
    properties = {
        hostname = "test1";
    }
}

    test2 = {
        services = {
            nginx-wrapper-test2 = {
                name = "nginx-wrapper-test2";
            }
        }
    }
```
For example, if we refer to `invDistribution.test1.services` we get all the configurations of the services that are deployed to machine `test1`. If we remove the reference to the nginx reverse proxy, we can pass this entire attribute set as inter-dependencies to configure the reverse proxy on machine `test1`. (The reason why we remove the reverse proxy as a dependency is because it is meaningless to let it refer to itself. Furthermore, this would also cause infinite recursion).

With this approach we can also easily scale up the environment. By simply adding more machines in the infrastructure model and additional web application service mappings in the distribution model, the service configurations in the service model get adjusted automatically not requiring us to think about specifying inter-dependencies at all.
Chapter 7

Architecture

Disnix is not a self-contained toolset and has a modular and extensible architecture. This chapter describes the architecture of Disnix and shows how certain low-level activities can be performed.

7.1 Communication flow

Figure 7.1: Communication flow of the deployment operations

Figure 7.1 illustrates the communication flow of the disnix deployment utilities, such as disnix-env. When executing a deployment step that needs to be executed remotely, a connection client process is consulted that connects to a protocol wrapper on the remote machine. By default, Disnix uses disnix-ssh-client that connects through SSH. Other kinds of communication protocols are supported through extensions, such as SOAP/HTTP. See Chapter 10 for more information.

The protocol wrapper connects to the core Disnix service, a D-Bus service exposing the deployment operations that Disnix needs to execute remotely. The core Disnix service invokes two external tools to carry out certain deployment activities. Nix is used for building and distributing packages. Dysnomia is used for activation, deactivation, locking, snapshotting, and restoring.
7.2 Composition of disnix-env

Figure 7.2: Architecture of the StaffTracker
As shown in Figure 7.2, disnix-env, the command-line tool that performs all activities to make a service-oriented system available for use, is composed of several lower-level command-line tools each executing a specific deployment activity.

In the default workflow, the following command-line utilities are invoked:

- **disnix-manifest** produces a low-level manifest file from the three Disnix models, describing which profiles to distribute, which services to activate and their dependencies, and which snapshots to transfer. This manifest file is used by most of the other low-level command-line tools.

  The manifest file contains Nix store paths to refer to profiles and services. As a side effect of computing these paths, all services will be built from source code, except the ones that have been built previously.

- **disnix-distribute**. Distributes the intra-dependency closures of the Nix profiles in the manifest file, that contains all services that should be deployed to a particular machine.

- **disnix-lock**. Locks or unlocks the Disnix service instances on the target machines, so that other deployment processes cannot interfere. Additionally, it uses Dysnomia to notify all the services so that they can optionally take precautions while they are being upgraded.

- **disnix-activate**. Deactivates obsolete services from the previous configuration and activates new services in the new configuration. Moreover, it uses the inter-dependency parameters from the models to execute these steps in the right order.

- **disnix-set**. Sets the profiles on the target machines to the ones that have been transferred with **disnix-distribute** so that the services are considered used and will not be garbage collected. On the coordinator machine, it sets the new coordinator profile referring to the manifest of the last executed deployment.

Two additional tools are used when the building on targets option has been enabled:

- **disnix-instantiate**. Creates a distributed derivation file from the three Disnix models that maps Nix store derivation files (low level build specifications that Nix uses to build a package) to target machines in the network.

- **disnix-build**. Distributes the store derivation closures from the distributed derivation file to the target machines in the network, builds the corresponding packages on the target machines and fetches the closures of the build results.

When state deployment has been enabled, another three tools are invoked:

- **disnix-snapshot**. Snapshots the state of all services that were in the previous configuration, but not in the current configuration and transfers them to the coordinator machine.

- **disnix-restore**. Transfers and restores the snapshots of all services that have been added in the new configuration.

- **disnix-delete-state**. Deletes the physical state of the services that have become obsolete in the new configuration.

### 7.3 Low-level usage examples

As described earlier, we can also invoke the low-level utilities that **disnix-env** consults, to execute a certain deployment activity individually. In this section, we provide a few examples.

#### 7.3.1 Building a system on the coordinator machine

In order to build all the services from source code, the following command can be used:

```bash
$ disnix-manifest -s services.nix -i infrastructure.nix -d distribution.nix
```
This command produces a manifest file, which is basically a more concrete version of the distribution model. This file contains references to the actual Nix store paths of all the build results. As a side effect, all the services that are specified in the distribution model are built from source code. The manifest is also a Nix package residing in the Nix store. For convenience, this tool creates a symlink called result pointing to it.

For instance, by querying the runtime dependencies of the generated manifest file, all the services including their runtime dependencies can be retrieved:

```bash
$ nix-store -qR ./result
```

### 7.3.2 Building services on target machines

You can also perform all the builds on the target machines and then retrieve back the results. The following command generates a distributed derivation file, which is basically a similar file as a manifest, except that it maps Nix store derivation files (low-level specifications that Nix uses to build a component) to target machines.

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

Like the manifest file, the distributed derivation file is also stored in the Nix store and a result symlink is stored in the current directory pointing to it.

By querying the runtime dependencies of a distributed derivation file, all the store derivations files of the services, including their build-time dependencies can be retrieved:

```bash
$ nix-store -qR ./result
```

The distributed derivation file can then be used to perform the builds:

```bash
$ disnix-build ./result
```

This command distributes the store derivation file of each service and its dependencies to the machines in the network, then it builds them on each machine and finally copies the build results back into the Nix store of the coordinator machine.

### 7.3.3 Distributing services to target machines

After all services have been built by invoking disnix-manifest, then the services including their runtime dependencies can be distributed to machines in the network by calling:

```bash
$ disnix-distribute ./result
```

### 7.3.4 Deactivating obsolete services and activating services on target machines

After services have been distributed by invoking disnix-distribute, the obsolete services can be deactivated and the new services activated by running:

```bash
$ disnix-activate ./result
```
Chapter 8

Using Disnix as a remote package deployer

As described in the previous chapters, Disnix’s primary purpose is deploying systems that can be decomposed into services to networks of machines. However, a service deployment process is basically a superset of an "ordinary" package deployment process. This chapter describes how we can do remote package deployment by instructing Disnix to only use a relevant subset of features.

8.1 Specifying packages as services

In Nixpkgs, it is a common habit to write each package specification as a function in which the parameters denote intra-dependencies. In Chapter 4 we have shown that Disnix services follow the same convention and extend this approach with nested functions in which the inner function takes inter-dependencies into account.

For services that have no inter-dependencies, a Disnix expression is identical to an ordinary package expression. This means that, for example, an expression for a package such as the Midnight Commander shown in Example 8.1 is also a valid Disnix service with no inter-dependencies:

Example 8.1 An example package expression and service expression with no inter-dependencies

```nix
{ stdenv, fetchurl, pkgconfig, glib, gpm, file, e2fsprogs
, libX11, libICE, perl, zip, unzip, gettext, slang
}:

stdenv.mkDerivation {
  name = "mc-4.8.12";

  src = fetchurl {
    sha256 = "15lkwcis0labshq9k8c2fqdwv8az2c87qpdqwp5p3ls8gb1gqm0h";
  };

  buildInputs = [ pkgconfig perl glib gpm slang zip unzip file gettext
                libX11 libICE e2fsprogs ];

  meta = {
    description = "File Manager and User Shell for the GNU Project";
    homepage = http://www.midnight-commander.org;
    license = "GPLv2+";
    maintainers = [ stdenv.lib.maintainers.sander ];
  };
}
```
8.2 Composing packages locally

Package and service expressions are functions that do not specify the versions or variants of the dependencies that should be used. To allow services to be deployed, we must compose them by providing the desired versions or variants of the dependencies as function parameters.

As shown in Chapter 4 we have to compose a Disnix service twice -- first its intra-dependencies in a composition expression and later its inter-dependencies in the services model.

Example 8.2 Composing packages locally

```nix
{pkgs, system}:

let
    callPackage = pkgs.lib.callPackageWith (pkgs // self);

    self = {
        pkgconfig = callPackage ./pkgs/pkgconfig { }
    }
    gpm = callPackage ./pkgs/gpm { }
    mc = callPackage ./pkgs/mc { }
;
in
    self
```

Example 8.2 composes the Midnight Commander package by providing its intra-dependencies as function parameters. The third attribute (mc) invokes a function named: callPackage {} that imports the previous package expression and automatically provides the parameters having the same names as the function parameters. The callPackage {} function first consults the self attribute set (that composes some of Midnight Commander’s dependencies as well, such as gpm and pkgconfig) and then any package from the Nixpkgs repository.

8.3 Writing a minimal services model

Previously, we have shown how to build packages from source code and its dependencies, and how to compose packages locally. For the deployment of services, more information is needed. For example, we need to compose their inter-dependencies so that services know how to reach them.

Furthermore, Disnix’s end objective is to get a running service-oriented system and carries out extra deployment activities for services to accomplish this, such as activation and deactivation. The latter two steps are executed by a Dysnomia plugin that is determined by annotating a service with a type attribute.

For package deployment, specifying these extra attributes and executing these remaining activities are in principle not required. Nonetheless, we still need to provide a minimal services model so that Disnix knows which units can be deployed.

Example 8.3 Exposing a package as a service

```nix
{pkgs, system, distribution, invDistribution}:

let
    customPkgs = import ./custom-packages.nix {
        inherit pkgs system;
    }
;

{mc = {
    name = "mc";
    pkg = customPkgs.mc;
    type = "package";
}
```
In Example 8.3 we import our intra-dependency composition expression and we use the `pkg` sub attribute to refer to the intra-dependency composition of the Midnight Commander. We annotate the Midnight Commander service with a `package` type to instruct Disnix that no additional deployment steps need to be performed beyond the installation of the package, such activation or deactivation.

Since the above pattern is common to all packages, we can also automatically generate services for any package in the composition expression, as shown in Example 8.4:

**Example 8.4 Exposing all locally composed packages as services**

```nix
{ pkgs, system, distribution, invDistribution }:

let
  customPkgs = import ./custom-packages.nix {
    inherit pkgs system;
  };

in
pkgs.lib.mapAttrs (name: pkg: {
  inherit name pkg;
  type = "package";
}) customPkgs
```

8.4 Configuring the remote machine’s search paths

To allow users on the remote machines to conveniently access their packages, we must add Disnix’ Nix profile to the PATH of a user on the remote machines:

```bash
$ export PATH=/nix/var/nix/profiles/disnix/default/bin:$PATH
```

When using NixOS, this variable can be extended by adding the following line to `/etc/nixos/configuration.nix`:

```nix
environment.variables.PATH = [ "/nix/var/nix/profiles/disnix/default/bin" ];
```

8.5 Deploying packages

By providing an infrastructure model and distribution model, we can use Disnix to deploy packages to remote machines.

**Example 8.5 A basic infrastructure model for package deployment**

```nix
{ 
  test1.properties.hostname = "test1";
  test2 = {
    properties.hostname = "test2";
    system = "x86_64-darwin";
  };
}
```

Example 8.5 describes two machines that have hostname `test1` and `test2`. Furthermore, machine `test2` has a specific system architecture: `x86_64-darwin` that corresponds to a 64-bit Intel-based Mac OS X.
Example 8.6 A basic distribution model for package deployment

```
{infrastructure}:
{
    gpm = [ infrastructure.test1 ];
    pkgconfig = [ infrastructure.test2 ];
    mc = [ infrastructure.test1 infrastructure.test2 ];
}
```

In Example 8.6, we distribute package gpm to machine test1, pkgconfig to machine test2 and mc to both machines.

When running the following command-line instruction:

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

Disnix executes all activities to get the packages in the distribution model deployed to the machines, such as building them from source code (including its dependencies), and distributing their dependency closures to the target machines. Because machine test2 may have a different system architecture as the coordinator machine, Disnix can use Nix’s delegation mechanism to forward a build to a machine that is capable of doing it.

Alternatively, packages can also be built on the target machines through Disnix:

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

After the deployment above command-line instructions have succeeded, we should be able to start the Midnight Commander on any of the target machines, by running:

```
$ mc
```

8.6 Deploying any package from the Nixpkgs repository

Besides deploying a custom set of packages, it is also possible to use Disnix to remotely deploy any package in the Nixpkgs repository, but doing so is a bit tricky. The main challenge lies in the fact that the Nix packages set is a nested set of attributes, whereas Disnix expects services to be addressed in one attribute set only.

Fortunately, the Nix expression language and Disnix models are flexible enough to implement a solution.

Example 8.7 A distribution model referring to packages in Nixpkgs

```
{infrastructure}:
{
    mc = [ infrastructure.test1 ];
    qic = [ infrastructure.test1 ];
    wget = [ infrastructure.test1 ];
    "xlibs.libX11" = [ infrastructure.test1 ];
}
```

Example 8.7 shows a distribution model mapping a number of packages from the Nix packages repository to machines in the network. Note that we use a dot notation: xlibs.libX11 as an attribute name to refer to libX11 that can only be referenced as a sub attribute in Nixpkgs.

We can write a services model that uses the attribute names in the distribution model to refer to the corresponding package in Nixpkgs:
Example 8.8 A services model referring to packages in Nixpkgs

```nix
{pkgs, system, distribution, invDistribution}:

pkgs.lib.mapAttrs (name: targets: let
    attrPath = pkgs.lib.splitString "." name;
  in
    { inherit name;
      pkg = pkgs.lib.attrByPath attrPath
      (throw "package: $name cannot be referenced in the package set")
      pkgs;
      type = "package";
    }
) distribution
```

With Example 8.8 we can deploy any Nix package to any remote machine with Disnix.

8.7 Multi-user package management

Besides supporting single user installations, Nix also supports *multi-user installations* in which every user has its own private Nix profile with its own set of packages. With Disnix we can also manage multiple profiles. For example, by adding the `--profile` parameter, we can deploy another Nix profile that, for example, contains a set of packages for the user: `sander`:

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

The user: `sander` can access its own set of packages by setting the `PATH` environment variable to:

```
$ export PATH=/nix/var/nix/profiles/disnix/sander:$PATH
```
Chapter 9

Dysnomia modules

As explained in Chapter 4, a type is assigned to each service in the services model. In Chapter 7 we have shown the architecture of the Disnix service (running on every target machine) that consults Dysnomia to execute these activities. Types are used to determine how to non-generic deployment activities, such as activating or deactivating a service in a container. Each type is mapped to a Dysnomia module that performs the actual deployment steps.

In this chapter, we explain how Dysnomia modules work.

9.1 Structure

Basically, every Dysnomia module implements a process, which takes two command-line parameters. The first parameter is one of the following:

- **activate.** Invoked to activate a service in a container
- **deactivate.** Invoked to deactivate a service in a container
- **lock.** Invoked for each service before the activation phase starts
- **unlock.** Invoked for each service after the activation phase
- **snapshot.** Invoked to capture the state of the service
- **restore.** Invoked to restore the state of the service
- **collect-garbage.** Invoked to delete the state of the service

The second parameter is a Nix path referring to a service. Moreover, all the container properties in the infrastructure model to which the service is deployed are passed as environment variables so that all relevant deployment properties are known.

9.2 Dysnomia module for the mysql-database type

**Example 9.1 MySQL database Dysnomia module**

```bash
#!/bin/bash
set -e
set -o pipefail
shopt -s nullglob

# Autoconf settings
export prefix=@prefix@
```
# Import utility functions
source @datadir@/@PACKAGE@/util

determineComponentName $2
dcheckStateDir
determineTypeIdentifier $0
composeSnapshotsPath
composeGarbagePath
composeGenerationsPath

case "$1" in
  activate)
    # Initialize the given schema if the database does not exists
    if [ "$(echo "show databases" | @mysql@ --user=$mysqlUsername --password=←$mysqlPassword -N | grep -x $componentName)" = "" ]
      then
        ( echo "create database $componentName;"
echo "use $componentName;"
        if [ -d $2/mysql-databases ]
          then
cat $2/mysql-databases/*.sql
          fi
        ) | @mysql@ --user=$mysqlUsername --password=$mysqlPassword -N
      fi
    unmarkStateAsGarbage
  ;;
deactivate)
    markStateAsGarbage
  ;;
snapshot)
    tmpdir=$(mktemp -d)
cd $tmpdir
    @mysqldump@ --single-transaction --quick --user=$mysqlUsername --password=←$mysqlPassword $componentName | head -n-1 | xz > dump.sql.xz
    hash=$(cat dump.sql.xz | sha256sum)
    hash=${hash:0:64}
    if [ -d $snapshotsPath/$hash ]
      then
        rm -Rf $tmpdir
      else
        mkdir -p $snapshotsPath/$hash
        mv dump.sql.xz $snapshotsPath/$hash
    fi
    rmdir $tmpdir
    createGenerationSymlink $snapshotsPath/$hash
  ;;
restore)
  determineLastSnapshot
  if [ "$lastSnapshot" != "" ]
    then
      ( echo "use $componentName;"
xzcat $snapshotsPath/$lastSnapshot/dump.sql.xz
      ) | @mysql@ --user=$mysqlUsername --password=$mysqlPassword -N
    fi
  ;;
collect-garbage)
    if [ -f $garbagePath ]
then
    echo "drop database $componentName;" | @mysql@ --user=$mysqlUsername --password←=
    $mysqlPassword -N
    unmarkStateAsGarbage
fi
;

Example 9.1 shows the Dysnomia module used for the mysql-database type:

1. This part performs the activation step of a MySQL service. In this step, we first check whether the database already exists. If the database is not there yet, we create a MySQL database having the same name as the Nix component and finally we import the attached MySQL dump (defining the schema) into the database. Moreover, it also marks the database as being used, so that it will not be garbage collected.

2. The deactivation step consists of marking the database as garbage, so that it will be removed by the garbage collector.

3. Dumps the state of the MySQL database in a single transaction into a dump file and places it in the Dysnomia snapshot folder using the output hash as a naming convention. If a dump with an identical hash exists, then the snapshot is discarded, because there is no reason to store the same output twice. After the snapshot has been taken, the generation symlink (that indicates what the last snapshot is) is updated.

4. Determines what the latest snapshot of the database is and restores the corresponding snapshot. If no snapshot exists, it does nothing.

5. Checks whether the database has been marked as garbage and drops it if this is the case. Otherwise, it does not nothing.

9.3 Implementing a custom activation interface

Although the dysnomia package contains plugins for commonly found services, there may be special cases where the activation and deactivation procedure have to be executed directly by the services. For example, because there is no activation type available or the service requires a specialized activation procedure.

For these purposes, the wrapper type can be used. Basically, the wrapper module invokes the bin/wrapper executable in the Nix package with the first parameter passed to the activation script (which is activate, deactivate etc.). Then this process performs all the steps to activate the component and so on.

Example 9.2 Disnix TCP proxy wrapper script

```bash
#!/bin/bash -e
export PATH=@PREFIX@/bin:$PATH

case "$1" in
  activate)
    nohup disnix-tcp-proxy @srcPort@ @destHostname@ @destPort@ /tmp/disnix-tcp-proxy-←=
      @srcPort@.lock > /var/log/$(basename @PREFIX@).log & pid=$!
    echo $pid > /var/run/$(basename @PREFIX@).pid
  ;;
  deactivate)
    kill $(cat /var/run/@PREFIX@.pid)
    rm -f /var/run/$(basename @PREFIX@).pid
    touch /tmp/disnix-tcp-proxy-@srcPort@.lock
  ;;
  lock)
    if [ -f /tmp/disnix-tcp-proxy-@srcPort@.lock ]
    then
      exit 1
    else
      touch /tmp/disnix-tcp-proxy-@srcPort@.lock
```
Example 9.2 shows an example of a wrapper script used for the TCP proxy example available from my Github page. As you may notice, the structure is very similar to an activation script, since it also contains the activate and deactivate steps. Moreover, this wrapper script also implements a lock and unlock step, which notify the service that an upgrade is starting (a phase in which services are deactivated and activated and temporarily makes certain parts of the system unavailable for use) and that the upgrade phase is finished.
Chapter 10

Advanced options

10.1 Configuring a custom connection protocol

As mentioned before, the Disnix service consists of a core service and a protocol wrapper. By default, an SSH wrapper is used, but other types of wrappers can be used as well, such as SOAP, provided by the external DisnixWebService package.

The coordinator machine invokes an external process which performs communication with the Disnix service. By default disnix-ssh-client is consulted. A different client can be used by either setting the DISNIX_CLIENT_INTERFACE environment variable with the path to the executable or by using the --interface command-line option, for commands such as disnix-env.

For example, by specifying:

```bash
$ export DISNIX_CLIENT_INTERFACE=disnix-soap-client
```

The disnix-soap-client command is used to communicate with a remote Disnix service.

Apart from configuring the coordinator machine, each target machine must also run the connection wrapper so that it can use the given protocol. Refer to the documentation of the extension for specific instructions.

Another wrapper is disnix-client that connects directly to the D-Bus system bus to invoke Disnix service operations. This wrapper is useful for debugging purposes, but you cannot use this client for remote connections.

In some cases, also the target property must be configured. By default, Disnix uses the hostname property in the infrastructure model, to determine how to connect to the remote Disnix service in order to perform remote deployment steps. This property is not suitable for every protocol. A web service interface interace, for example, requires an URL.

The connection attribute can be changed by either setting the DISNIX_TARGET_PROPERTY environment variable with the attribute name that contains the address of the remote Disnix service or by using the --target-property command-line option.

For example, by specifying:

```bash
$ export DISNIX_TARGET_PROPERTY=sshTarget
```

The sshTarget attribute defined in the infrastructure model is used to determine the address of the Disnix service.

It is also possible to define a target property and client interface for each individual machine to support multi connection protocol deployment. See Example 4.8 for more information.

10.2 Managing multiple distributed system configurations

By default, Disnix assumes that the models that you are currently using represent one particular distributed environment. You can also use multiple profiles, which allow you to maintain multiple distributed system environments from one coordinator machine.
By using the --profile option for the disnix-env and disnix-activate commands, you can specify which profile you want to use, so that they do not interfere with each other.

The following instructions will install a particular distributed environment:

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

By running the following command with three models of another distributed environment:

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

Disnix will upgrade the previous the default environment to match the models defined in the other environment, which is not desirable. However, by using the --profile option Disnix deploys the new distributed system without looking to the default system’s deployment state and maintains two separate configurations next to each other:

```
$ disnix-env --profile other -s my-other-services.nix -i my-other-infrastructure.nix -d my-\→
other-distribution.nix
```

Besides using the --profile option, you can also use an environment variable:

```
$ export DISNIX_PROFILE=other
```

### 10.3 Enabling state deployment by default

If it desired to let Disnix manage state, you must annotate the corresponding services in the service model. However, it is also possible to override Disnix’s default behaviour to enable state management for all services by default.

Global state deployment can be enabled by providing the --deploy-state command-line option to commands such as disnix-env or by setting the following environment variable:

```
$ export DISNIX_DEPLOY_STATE=1
```

### 10.4 Multi-container deployment

As described in Chapter 4, when mapping a service to a target machine in the distribution model, Disnix automatically maps the service to the appropriate container, by referring to a container with the same name as the type the service belongs to.

In some unconventional scenarios, it may be desired to run multiple instances of the same container on one machine. In such cases, automapping no longer works and a different notation is required. In Disnix, the following mapping in the distribution model (mapping a service to a list of target machines in the infrastructure model):

```
ZipcodeService = [ infrastructure.test2 ];
```

Is equivalent to the mapping in the following notation:

```
ZipcodeService = {
    targets = [ { target = infrastructure.test2; } ];
};
```

In the above notation, we define an attribute set in which the targets property refers to a list of attribute sets defining all possible attributes of a mapping. This alternative notation is more verbose and allows more mapping properties to be specified.
Example 10.1 Infrastructure model with multiple container instances

```json
{  
  test1 = {  
    properties = {  
      hostname = "test1.example.org";  
    };  
    containers = {  
      tomcat-first = {  
        tomcatPort = 8080;  
      };  
      tomcat-second = {  
        tomcatPort = 8081;  
      };  
    };  
  };  
  test2 = {  
    properties = {  
      hostname = "test2.example.org";  
    };  
    containers = {  
      mysql-first = {  
        mysqlPort = 3306;  
        mysqlUsername = "root";  
        mysqlPassword = "admin";  
      };  
      mysql-second = {  
        mysqlPort = 3307;  
        mysqlUsername = "root";  
        mysqlPassword = "secret";  
      };  
    };  
  };  
}
```

Example 10.1 is based on the StaffTracker infrastructure model shown in Example 4.8. In this modified infrastructure model, the test1 machine hosts two Apache Tomcat servers (one listening on TCP port 8080 and the other on TCP port 8081) and the test2 machine hosts two MySQL DBMSes (one listening on TCP port 3306 and the other on TCP port 3307). Because we have two instances of each container and their names do not correspond to the services types, automapping no longer works.

Example 10.2 Distribution model for the StaffTracker mapping to multiple containers

```json
{  
  {infrastructure}:  
  {  
    zipcodes = {  
      targets = [ { target = infrastructure.test2; container = "mysql-first"; } ];  
    };  
    ZipcodeService = {  
      targets = [ { target = infrastructure.test1; container = "tomcat-second"; } ];  
    };  
    StaffTracker = {  
      targets = [  
        { target = infrastructure.test1; container = "tomcat-first"; }  
        { target = infrastructure.test1; container = "tomcat-second"; }  
      ];  
    }  
  }  
```
Example 10.2 shows a distribution model using the alternative notation to directly control the container mappings:

1. This mapping states that the zipcodes database should be deployed to the first MySQL instance container (mysql-first) on machine test2.

2. This mapping states that the ZipcodeService application should be deployed to the second Apache Tomcat container (tomcat-second) on machine test1.

3. It also possible to map a service to multiple containers on multiple machines. This line states that the StaffTracker service should be deployed to both the first and second Apache Tomcat container on machine test1.
Chapter 11

Extensions

Although Disnix makes the deployment of a distributed system more convenient and offers various kinds of features to make this process reliable and efficient, some extensions have been developed to make deployment processes more convenient.

11.1 DisnixWebService

In some cases, other communication kinds of communication protocols are needed to connect to remote machines besides SSH. The DisnixWebService is a package that implements a SOAP interface for deployment operations and a client interface named disnix-soap-client to connect to it.

11.2 DisnixOS

Although Disnix manages the distributable components (services) of which a distributed system is composed and the inter-dependencies, Disnix does not manage the underlying system configurations of the actual machines to which services are deployed. The DisnixOS package combines the Disnix tooling for service deployment and NixOS tooling to deploy the underlying infrastructure.

Furthermore, it integrates with the NixOS test driver that can be used to automatically create virtual machine instances from the network configuration for testing.

11.3 Dynamic Disnix

Disnix implements a static deployment model -- to make it work it requires a infrastructure model and distribution model to be manually specified every time something in the configuration changes.

Dynamic Disnix is an extended toolset providing a discovery service to capture machine configurations in a network, as well as a distribution generation framework that can be used to generate a distribution model from technical and non-functional properties of services and machines.

These tools can be used to develop a framework enabling self adaptive (re)deployment of service-oriented systems.
Appendix A

Command Reference

A.1 Main commands

A.1.1 disnix-collect-garbage

disnix-collect-garbage — Delete garbage from a network of machines

Synopsis

disnix-collect-garbage [OPTION] infrastructure_nix

DESCRIPTION

The command `disnix-collect-garbage` collects all garbage from all the machines defined in an infrastructure Nix expression and optionally removes all the older profiles.

OPTIONS

-d, --delete-old Removes all the old Nix profile generations

--interface=INTERFACE Path to executable that communicates with a Disnix interface. Defaults to `disnix-ssh-client`

--target-property=PROP The target property of an infrastructure model, that specifies how to connect to the remote Disnix interface. (Defaults to: hostname)

-h, --help Shows the usage of this command to the user

-v, --version Shows the version of this command to the user

ENVIRONMENT

DISNIX_CLIENT_INTERFACE Sets the client interface (which defaults to `disnix-ssh-client`)

DISNIX_TARGET_PROPERTY Specifies which property in the infrastructure Nix expression specifies how to connect to the remote interface (defaults to: hostname)
A.1.2 disnix-clean-snapshots

disnix-clean-snapshots — Delete older snapshots stored in a network machines

Synopsis

disnix-clean-snapshots [OPTION] infrastructure_nix

DESCRIPTION

The command `disnix-clean-snapshots` removes all older snapshot generations stored on the machines in the network.

OPTIONS

--interface=INTERFACE Path to executable that communicates with a Disnix interface. Defaults to `disnix-ssh-client`
--target-property=PROP The target property of an infrastructure model, that specifies how to connect to the remote Disnix
--keep=NUM Amount of snapshot generations to keep. Defaults to: 1
-h, --help Shows the usage of this command to the user
-v, --version Shows the version of this command to the user

ENVIRONMENT

DISNIX_CLIENT_INTERFACE Sets the client interface (which defaults to `disnix-ssh-client’)
DISNIX_TARGET_PROPERTY Specifies which property in the infrastructure Nix expression specifies how to connect to the remote interface (defaults to: hostname)

A.1.3 disnix-env

disnix-env — Installs or updates the environment of a distributed system

Synopsis

disnix-env -s services_nix -i infrastructure_nix -d distribution_nix [OPTION]
disnix-env --rollback [OPTION]
disnix-env --switch-generation NUM [OPTION]
DESCRIPTION

The command `disnix-env` is used to install, upgrade, or roll back a service-oriented system in a given environment.

This command requires three Nix expressions as input parameters -- a services model capturing the components of a distributed system and its inter-dependencies, an infrastructure model capturing the machines in the network and its properties and a distribution model which maps services to machines.

By invoking this command, first all the services that are defined in the distribution model are built from source code including all its dependencies. If all the services are successfully built, the closures of the services are transferred to the target machines in the network. Finally, the services are activated by traversing the inter-dependency graph of all the services.

In case of a failure, a rollback is performed to bring the system back in its previous configuration.

If there is already a distributed system configuration deployed, an upgrade is performed. In this phase only the changed parts of the system are deactivated and activated. In this process we also deal with the inter-dependencies so that no services fails due to a missing inter-dependency.

Since the target machines could be of a different type or architecture as the coordinator machine, we may not be able to build a specific service for the given target machine. In such cases, `disnix-env` also provides the option to build the services on the target machines and to keep the build results for future use.

If state deployment has been enabled for a service and that particular service has been moved from one machine to another, then a snapshot of the state is taken, transferred to the new machine, and finally restored.

OPTIONS

-s, --services=services_nix Services Nix expression which describes all components of the distributed system

-i, --infrastructure=infrastructure_nix Infrastructure Nix expression which captures properties of machines in the network

-d, --distribution=distribution_nix Distribution Nix expression which maps services to machines in the network

--switch-to-generation=NUM Switches to a specific profile generation

--rollback Switches back to the previously deployed configuration

--target-property=PROP The target property of an infrastructure model, that specifies how to

--deploy-state Indicates whether to globally deploy state (disabled by default)

-p, --profile=PROFILE Name of the profile that is used for this system. Defaults to: default

-m, --max-concurrent-transfers=NUM Maximum amount of concurrent closure transfers. Defauls to: 2

--build-on-targets Build the services on the target machines in the network instead of managing the build by the coordinator

--coordinator-profile-path=PATH Path where to store the coordinator profile generations

--no-upgrade Do not perform an upgrade, but activate all services of the new configuration

--no-lock Do not attempt to acquire and release any locks

--no-coordinator-profile Specifies that the coordinator profile should not be updated

--no-target-profiles Specifies that the target profiles should not be updated

--no-migration Do not migrate the state of services from one machine to another, even if they have been annotated as such

--no-delete-state Do not remove the state of deactivated services

--show-trace Shows a trace of the output

-h, --help Shows the usage of this command

-v, --version Shows the version of this command
ENVIRONMENT

**DISNIX_CLIENT_INTERFACE** Sets the client interface (defaults to: disnix-ssh-client)

**DISNIX_TARGET_PROPERTY** Sets the target property of an infrastructure model, that specifies how to connect to the remote Disnix interface. (Defaults to: hostname)

**DISNIX_PROFILE** Sets the name of the profile that stores the manifest on the coordinator machine and the deployed services per machine on each target (Defaults to: default).

**DISNIX_DEPLOY_STATE** If set to 1 it also deploys the state of all components. (defaults to: 0)

**DISNIX_NO_DELETE_STATE** If set to 1 it does not delete the obsolete state after upgrading. (defaults to: 0)

**DYSNOMIA_STATEDIR** Specifies where the snapshots must be stored on the coordinator machine (defaults to: /var/state/dysnomia)

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2 Utilities

A.2.1 disnix-capture-infra

disinx-capture-infra — Captures the container configurations of machines and generates an infrastructure expression from it

Synopsis

disinx-capture-infra [OPTION] infrastructure_nix

DESCRIPTION

The command `disnix-capture-infra` captures the container configurations of all machines defined in the infrastructure model and generates an infrastructure expression from it.

OPTIONS

`--interface=INTERFACE` Path to executable that communicates with a Disnix interface. Defaults to `disnix-ssh-client`

`--target-property=PROP` The target property of an infrastructure model, that specifies how to connect to the remote Disnix interface. (Defaults to: hostname)

`-h, --help` Shows the usage of this command to the user

`-v, --version` Shows the version of this command to the user

ENVIRONMENT

**DISNIX_CLIENT_INTERFACE** Sets the client interface (which defaults to `disnix-ssh-client`)

**DISNIX_TARGET_PROPERTY** Specifies which property in the infrastructure Nix expression specifies how to connect to the remote interface (defaults to: hostname)
A.2.2 disnix-activate

disinix-activate — Activate a configuration described in a manifest

Synopsis

disinix-activate [OPTION] MANIFEST

DESCRIPTION

The command `disnix-activate' will activate all the services in the given manifest file on the target machines in the right order, by traversing the inter-dependency graph of the services.

If there is already a configuration deployed, then this command will perform an upgrade.

First it deactivates all obsolete services which are not present in the new configuration, finally it will activate all the new services in the new configuration. During this phase it takes inter-dependencies into account, so that no service will fail due to a broken inter-dependency closure.

In case of a failure, a rollback is performed and all the newly activated services are deactivated and all deactivated services are activated again.

Most users don’t need to use this command directly. The `disnix-env' command will automatically invoke this command to activate the new configuration.

OPTIONS

-p, --profile=PROFILE Name of the profile in which the services are registered. Defaults to: default

--coordinator-profile-path=PATH Path to the manifest of the previous configuration. By default this tool will use the manifest stored in the disnix coordinator profile instead of the specified one, which is usually sufficient in most cases.

-o, --old-manifest=MANIFEST Nix profile path where the manifest should be stored, so that Disnix knows the current configuration of a distributed system. By default it is stored in the profile directory of the user. Most users do not want to use this option directly, but it is used by e.g. the virtualization extension to store virtual machine profile in a separate directory.

--no-upgrade By enabling this option Disnix does not store the deployment state for further use, such as upgrading

--no-rollback Do not roll back if an error occurs while deactivating and activating services

--dry-run Prints the activation and deactivation steps that will be performed but does not actually execute them

-h, --help Shows the usage of this command to the user

-v, --version Shows the version of this command to the user

ENVIRONMENT

DISNIX_PROFILE Sets the name of the profile that stores the manifest on the coordinator machine and the deployed services per machine on each target (Defaults to: default)
A.2.3 disnix-build

disinix-build — Build store derivations on target machines in a network

Synopsis

disinix-build [OPTION] DISTRIBUTED_DERIVATION

Description

The command `disnix-build' builds derivations on the given target machines specified in a distributed derivation XML file. When the building process is complete, the results are transferred back to the coordinator machine, so that they are kept for further use and do not have to be rebuilt again in case of a configuration change.

In most cases this command should not be called directly. The command `disnix-env' automatically uses this command if the --build-on-targets is specified.

Options

- `-m`, `--max-concurrent-transfers=NUM` Maximum amount of concurrent closure transfers. Defaults to: 2
- `-h`, `--help` Shows the usage of this command to the user
- `-v`, `--version` Shows the version of this command to the user

Environment

`DISNIX_TARGET_PROPERTY` Specifies which property in the infrastructure Nix expression specifies how to connect to the remote interface (defaults to: hostname)

Copyright

Copyright (C) 2008-2016 Sander van der Burg

A.2.4 disnix-client

disinix-client — Build store derivations on target machines in a network

Synopsis

disinix-client [OPTION] operation

Description

The command `disnix-client' provides access to a `disnix-service' instance running on the same machine by connecting to the D-Bus system or session bus.

In most cases this tool is only needed for debugging purposes, since it only uses the D-Bus protocol and cannot connect to a remote machine. A more useful client for use in production environments is: `disnix-ssh-client`. 
OPTIONS

Operations:

--import Imports a given closure into the Nix store of the target machine
--export Exports the closure of a given Nix store path of the target machine into a file
--print-invalid Prints all the paths that are not valid in the Nix store of the target machine
--realise Realises the given store derivation on the target machine
--set Creates a Disnix profile only containing the given derivation on the target machine
--query-installed Queries all the installed services on the given target machine
--query-requisites Queries all the requisites (intra-dependencies) of the given services on the target machine
--collect-garbage Collects garbage on the given target machine
--activate Activates the given service on the target machine
--deactivate Deactivates the given service on the target machine
--lock Acquires a lock on a Disnix profile of the target machine
--unlock Release the lock on a Disnix profile of the target machine
--snapshot Snapshots the logical state of a component on the given target machine
--restore Restores the logical state of a component on the given target machine
--delete-state Deletes the state of a component on the given machine
--query-all-snapshots Queries all available snapshots of a component on the given target machine
--query-latest-snapshot Queries the latest snapshot of a component on the given target machine
--print-missing-snapshots Prints the paths of all snapshots not present on the given target machine
--import-snapshots Imports the specified snapshots into the remote snapshot store
--export-snapshots Exports the specified snapshot to the local snapshot store
--resolve-snapshots Converts the relative paths to the snapshots to absolute paths
--clean-snapshots Removes older snapshots from the snapshot store
--capture-config Captures the configuration of the machine from the Dysnomia container properties in a Nix expression
--help Shows the usage of this command to the user
--version Shows the version of this command to the user

General options:

--target=TARGET Specifies the target to connect to. This property is ignored by this client because it only supports loopback connections.
--session-bus Connects to the session bus instead of the system bus. This is useful for testing/debugging purposes
Import/Export/Import snapshots/Export snapshots options:

`--localfile` Specifies that the given paths are stored locally and must be transferred to the remote machine if needed

`--remotefile` Specifies that the given paths are stored remotely and must be transferred from the remote machine if needed

Set/Query installed/Lock/Unlock options:

`-p, --profile=PROFILE` Name of the Disnix profile. Defaults to: default

Collect garbage options:

`-d, --delete-old` Indicates whether all older generations of Nix profiles must be removed as well

Activation/Deactivation/Snapshot/Restore/Delete state options:

`--type=TYPE` Specifies the activation module that should be used, such as echo or process.

`--arguments=ARGUMENTS` Specifies the arguments passed to the Dysnomia module, which is a string with key=value pairs

`--container=CONTAINER` Name of the container in which the component is managed. If omitted it will default to the same value as the type.

Query all snapshots/Query latest snapshot options:

`-C, --container=CONTAINER` Name of the container in which the component is managed

`-c, --component=COMPONENT` Name of the component hosted in a container

Clean snapshots options:

`--keep=NUM` Amount of snapshot generations to keep. Defaults to: 1

ENVIRONMENT

DISNIX_PROFILE Sets the name of the profile that stores the manifest on the coordinator machine and the deployed services per machine on each target (Defaults to: default).

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2.5 disnix-copy-closure

disnix-copy-closure — Copy a closure from or to a remote machine through a Disnix interface
 Synopsis

disnix-copy-closure [OPTION] --to --target TARGET path
disnix-copy-closure [OPTION] --from --target TARGET path

 DESCRIPTION

The command `disnix-copy-closure' copies a Nix store component and all its intra-dependencies to or from a given target machine through a Disnix interface. This process is very efficient, because it scans for all intra-dependencies and only copies the missing parts.

This command is very similar to the `nix-copy-closure' command, except that it uses a Disnix interface for transport (which optionally uses SSH or a custom protocol) instead of using SSH directly.

 OPTIONS

--to Copy closure to the given target
--from Copy closure from the given target
-t, --target=TARGET Address of the Disnix service running on the remote machine
--interface=INTERFACE Path to executable that communicates with a Disnix interface. Defaults to: disnix-ssh-client
-h, --help Shows the usage of this command to the user
-v, --version Shows the version of this command to the user

 ENVIRONMENT

DISNIX_CLIENT_INTERFACE Sets the client interface (which defaults to: disnix-ssh-client)

 COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

 A.2.6 disnix-copy-snapshots

disnix-copy-snapshots — Copy a set of snapshots from or to a remote machine through a Disnix interface

 Synopsis

disnix-copy-snapshots [OPTION] --from --target TARGET -c CONTAINER -C COMPONENT
disnix-copy-snapshots [OPTION] --to --target TARGET -c CONTAINER -C COMPONENT

 DESCRIPTION

The command `disnix-copy-snapshots’ transfers the logical state (typically represented as snapshots in a consistent and portable format) of a component residing in a container from and to a remote machine through a Disnix interface.
OPTIONS

--to Copy snapshots to the given target
--from Copy snapshots from the given target
-t, --target=TARGET Address of the Disnix service running on the remote machine
-c, --container=CONTAINER Name of the container in which the mutable component is deployed
-C, --component=COMPONENT Name of the mutable component to take snapshots from
--all Transfer all snapshot generations instead of the latest only
--interface=INTERFACE Path to executable that communicates with a Disnix interface. Defaults to: disnix-ssh-client
-h, --help Shows the usage of this command to the user
-v, --version Shows the version of this command to the user

ENVIRONMENT

DISNIX_CLIENT_INTERFACE Sets the client interface (defaults to: disnix-ssh-client)
DYSNOMIA_STATEDIR Specifies where the snapshots must be stored on the coordinator machine (defaults to: /var/state/dysnomia)

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2.7 disnix-delete-state

disnix-delete-state — Deletes state of components that have become obsolete

Synopsis

disnix-delete-state [OPTION] [MANIFEST]

DESCRIPTION

The command `disnix-delete-state’ removes the state of all the components in a given deployment manifest that have been marked as garbage. If no manifest file is given, it uses the manifest of last deployed configuration.

Most users don’t need to use this command directly. The `disnix-env’ command will automatically invoke this command after the new configuration has been deployed.

OPTIONS

-p, --profile=PROFILE Name of the profile in which the services are registered. Defaults to: default
--coordinator-profile-path=PATH Path to the manifest of the previous configuration. By default this tool will use the manifest stored in the disnix coordinator profile instead of the specified one, which is usually sufficient in most cases.
-h, --help Shows the usage of this command to the user
-v, --version Shows the version of this command to the user
A.2.8 disnix-distribute

disinx-distribute — Distributes intra-dependency closures of services to target machines

Synopsis

disinx-distribute [OPTION] MANIFEST

DESCRIPTION

The command `disnix-distribute' copies all the intra-dependency closures of services in a manifest file to the target machines in the network. This process is very efficient, since it scans for all intra-dependencies and only copies the missing parts.

Most users don’t need to use this command directly. The `disnix-env' command will automatically invoke this command to distribute the services if necessary.

OPTIONS

-\texttt{-m, --max-concurrent-transfers=NUM} Maximum amount of concurrent closure transfers. Defaults to: 2
-\texttt{-h, --help} Shows the usage of this command to the user
-\texttt{-v, --version} Shows the version of this command to the user

A.2.9 disnix-gendist-roundrobin

disinx-gendist-roundrobin — Generate a distribution expression from a service and infrastructure expression

Synopsis

disinx-gendist-roundrobin -s \texttt{services\_nix} -i \texttt{infrastructure\_nix} [OPTION]

DESCRIPTION

The command `disnix-gendist-roundrobin' generates a distribution expression from a given services and infrastructure expression. It uses the round robin scheduling method to distribute every service in the services model over each machine in the infrastructure in equal proportions and circular order.
OPTIONS

-s, --services=services_nix  Services Nix expression which describes all components of the distributed system
-i, --infrastructure=infrastructure_nix  Infrastructure Nix expression which captures properties of machines in the network
--no-out-link  Do not create a 'result' symlink
--show-trace  Shows a trace of the output
-h, --help  Shows the usage of this command to the user
-v, --version  Shows the version of this command to the user

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2.10 disnix-instantiate

disinix-instantiate — Instantiate a distributed derivation from Disnix expressions

Synopsis

disinix-instantiate -s services_nix -i infrastructure_nix -d distribution_nix [OPTION]

DESCRIPTION

The command `disnix-instantiate' generates a distributed derivation file from a service, infrastructure and distribution Nix expression, which can be used to build the services on the target machines from source code by using the `disnix-build' command. Most users and developers don’t need to use this command directly. The command `disnix-env' performs instantiation of a distributed derivation automatically. It is mostly used for debugging purposes or to perform certain tasks manually.

OPTIONS

-s, --services=services_nix  Services Nix expression which describes all components of the distributed system
-i, --infrastructure=infrastructure_nix  Infrastructure Nix expression which captures properties of machines in the network
-d, --distribution=distribution_nix  Distribution Nix expression which maps services to machines in the network
--target-property=PROP  The target property of an infrastructure model, that specifies how to connect to the remote Disnix interface. (Defaults to: hostname)
--interface=INTERFACE  Path to executable that communicates with a Disnix interface. Defaults to: disnix-ssh-client
--no-out-link  Do not create a 'result' symlink
--show-trace  Shows a trace of the output
-h, --help  Shows the usage of this command to the user
-v, --version  Shows the version of this command to the user
ENVIRONMENT

**DISNIX_CLIENT_INTERFACE** Sets the client interface (defaults to: disnix-ssh-client)

**DISNIX_TARGET_PROPERTY** Sets the target property of an infrastructure model, that specifies how to connect to the remote Disnix interface. (Defaults to: hostname)

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2.11 **disnix-lock**

*disnix-lock* — Notifies services to lock or unlock themselves

**Synopsis**

```

disinix-lock [--unlock] [MANIFEST]
```

**DESCRIPTION**

Notifies all services on the machines that the transition phase starts or ends, so that they can temporarily lock or unlock themselves (or take other precautions to make the transition to go smooth)

If no manifest is specified, the manifest of the last deployed configuration will be used

Most users don’t need to use this command directly. The `disnix-env’ command will automatically invoke this command before upgrading the configuration.

**OPTIONS**

- **-u, --unlock** Executes an unlock operation instead of a lock
- **-p, --profile=PROFILE** Name of the profile in which the services are registered. Defaults to: default
- **--coordinator-profile-path=PATH** Path to the manifest of the previous configuration. By default this tool will use the manifest stored in the disnix coordinator profile instead of the specified one, which is usually sufficient in most cases.
- **-h, --help** Shows the usage of this command to the user
- **-v, --version** Shows the version of this command to the user

ENVIRONMENT

**DISNIX_PROFILE** Sets the name of the profile that stores the manifest on the coordinator machine and the deployed services per machine on each target (Defaults to: default)

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg
A.2.12 disnix-manifest

disnix-manifest — Generate a manifest file from Disnix expressions

Synopsis

disnix-manifest -s services_nix -i infrastructure_nix -d distribution_nix [OPTION]

DESCRIPTION

The command `disnix-manifest' generates a manifest file from a service, infrastructure and distribution Nix expression, which can be used for the distribution of services to machines in the network and for the activation of services on target machines in the right order.

Since the manifest file contains Nix store paths of every service, a side effect of running this command is that all the services that have to be activated are automatically built from source and stored in the Nix store of the coordinator machine.

Most users and developers don’t need to use this command directly. The command `disnix-env’ performs generation of a manifest automatically. It is mostly used for debugging purposes or to perform certain tasks manually.

OPTIONS

-\texttt{-s, \textbf{--services=services\_nix}} Services Nix expression which describes all components of the distributed system

-\texttt{-i, \textbf{--infrastructure=infrastructure\_nix}} Infrastructure Nix expression which captures properties of machines in the network

-\texttt{-d, \textbf{--distribution=distribution\_nix}} Distribution Nix expression which maps services to machines in the network

-\texttt{--target\_property=PROP} The target property of an infrastructure model, that specifies how to connect to the remote Disnix interface. (Defaults to hostname)

-\texttt{--interface=INTERFACE} Path to executable that communicates with a Disnix interface. Defaults to: disnix-ssh-client

-\texttt{--deploy\_state} Indicates whether to globally deploy state (disabled by default)

-\texttt{--no\_out\_link} Do not create a ‘result’ symlink

-\texttt{--show\_trace} Shows a trace of the output

-\texttt{-h, \textbf{--help}} Shows the usage of this command to the user

-\texttt{-v, \textbf{--version}} Shows the version of this command to the user

ENVIRONMENT

\textbf{DISNIX\_CLIENT\_INTERFACE} Sets the client interface (defaults to: disnix-ssh-client)

\textbf{DISNIX\_TARGET\_PROPERTY} Sets the target property of an infrastructure model, that specifies how to connect to the remote Disnix interface. (Defaults to: hostname)

\textbf{DISNIX\_DEPLOY\_STATE} If set to 1 it also deploys the state of all components. (defaults to: 0)

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg
A.2.13 disnix-query

disnix-query — Query the installed services from machines

Synopsis

disnix-query [OPTION] infrastructure_nix

DESCRIPTION

The command `disnix-query' collects and displays all the installed services from the machines defined in a given infrastructure model.

OPTIONS

-p, --profile=PROFILE Name of the profile in which the services are registered. Defaults to: default
--interface=INTERFACE Path to executable that communicates with a Disnix interface. Defaults to `disnix-ssh-client'
--target-property=PROP The target property of an infrastructure model, that specifies how to connect to the remote Disnix interface. (Defaults to hostname)
-h, --help Shows the usage of this command to the user
-v, --version Shows the version of this command to the user

ENVIRONMENT

DISNIX_CLIENT_INTERFACE Sets the client interface (which defaults to `disnix-ssh-client')
DISNIX_TARGET_PROPERTY Specifies which property in the infrastructure Nix expression specifies how to connect to the remote interface (defaults to: hostname)
DISNIX_PROFILE Sets the name of the profile that stores the manifest on the coordinator machine and the deployed services per machine on each target (Defaults to: default).

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2.14 disnix-restore

disnix-restore — Restores the state of components

Synopsis

disnix-restore [OPTION][MANIFEST]
DESCRIPTION

Restores the state of components deployed in a network of machines. By default, this command only restores the state of components that are recently activated, i.e. the ones that are defined in the current deployment manifest, but not the previous one.

A full restore of the state of the entire system can be done by providing the `--no-upgrade` parameter.

If no manifest has been provided, the last deployed one is used.

OPTIONS

- `--old-manifest=MANIFEST` Nix profile path where the manifest should be stored, so that Disnix knows the current configuration of a distributed system. By default it is stored in the profile directory of the user.
- `--no-upgrade` Indicates that no upgrade should be performed and the state of all components should be restored.
- `--transfer-only` Transfers the snapshot from the target machines, but does not actually restore them
- `--all` Transfers all snapshot generations of the target machines, not the latest
- `--profile=PROFILE` Name of the profile in which the services are registered. Defaults to: default
- `--coordinator-profile-path=PATH` Path to the manifest of the previous configuration. By default this tool will use the manifest stored in the disnix coordinator profile instead of the specified one, which is usually sufficient in most cases.
- `--max-concurrent-transfers=NUM` Maximum amount of concurrent closure transfers. Defaults to: 2
- `--help` Shows the usage of this command to the user

ENVIRONMENT

DISNIX_PROFILE Sets the name of the profile that stores the manifest on the coordinator machine and the deployed services per machine on each target (Defaults to: default)

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2.15 disnix-service

disnix-service — Build store derivations on target machines in a network

Synopsis

disnix-service [OPTION]

DESCRIPTION

The `disnix-service` tool is a daemon running on either the D-Bus system or session bus, which provides remote access to various deployment operations, such as importing, exporting, activating and deactivating services.

The daemon is not very useful on its own, since it requires a wrapper that exposes the methods to remote users. The simplest wrapper that can be used is just running a SSH server and by using the `disnix-ssh-client` from the client machines.

Other wrappers can also be used, which are basically thin layers that map the RPC protocol calls to D-Bus calls. A web service layer and client, for instance, is also available from the Disnix webpage which allows a user to use SOAP for executing deployment operations instead of SSH.
OPTIONS

--session-bus  Register the Disnix service on the session bus instead of the system bus (useful for testing)
--log-dir  Specify the directory in which the logfiles are stored (defaults to: /var/log/disnix)
-h, --help  Shows the usage of this command to the user
-v, --version  Shows the version of this command to the user

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2.16 disnix-set

disinix-set — Updates the coordinator and target Nix profiles

Synopsis

disinix-set [OPTION] MANIFEST

DESCRIPTION

The command `disnix-set' updates the coordinator profile referring to the last deployed manifest and the Disnix profiles on the target machines referring to the set of installed services. Updating the profiles prevents the configuration from being garbage collected.

This command should almost never be called directly. The command `disnix-env' invokes this command to update the profiles automatically.

OPTIONS

-p, --profile=PROFILE  Name of the profile in which the services are registered. Defaults to: default
--coordinator-profile-path=PATH  Path to the manifest of the previous configuration. By default this tool will use the manifest stored in the disnix coordinator profile instead of the specified one, which is usually sufficient in most cases.
--no-coordinator-profile  Specifies that the coordinator profile should not be updated
--no-target-profiles  Specifies that the target profiles should not be updated
-h, --help  Shows the usage of this command to the user
-v, --version  Shows the version of this command to the user

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2.17 disnix-snapshot

disinix-snapshot — Snapshots the state of components
Synopsis

disnix-snapshot [OPTION] [MANIFEST]

DESCRIPTION

Snapshots the state of components deployed in a network of machines.

By default, this command only snapshots the state of components that are recently deactivated, i.e. the ones that are not defined in the current deployment manifest, but in the previous one.

A full snapshot of the state of the entire system can be done by providing the --no-upgrade parameter.

If no manifest has been provided, the last deployed one is used. Moreover, it also snapshots the entire system if no manifest is provided. As a result, simply running `disnix-snapshot` makes a backup of the state of the entire system.

OPTIONS

-o, --old-manifest=MANIFEST Nix profile path where the manifest should be stored, so that Disnix knows the current configuration of a distributed system. By default it is stored in the profile directory of the user.

--no-upgrade Indicates that no upgrade should be performed and the state of all components should be restored.

--transfer-only Transfers the snapshot from the target machines, but does not actually restore them

--all Transfers all snapshot generations of the target machines, not the latest

-p, --profile=PROFILE Name of the profile in which the services are registered. Defaults to: default

--coordinator-profile-path=PATH Path to the manifest of the previous configuration. By default this tool will use the manifest stored in the disnix coordinator profile instead of the specified one, which is usually sufficient in most cases.

-m, --max-concurrent-transfers=NUM Maximum amount of concurrent closure transfers. Defaults to: 2

-h, --help Shows the usage of this command to the user

ENVIRONMENT

DISNIX_PROFILE Sets the name of the profile that stores the manifest on the coordinator machine and the deployed services per machine on each target (Defaults to: default)

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg

A.2.18 disnix-ssh-client

disnix-ssh-client — Provides access to the disnix-service through a SSH interface

Synopsis

disnix-ssh-client --target hostname[:port] operation[OPTION] [paths]
DESCRIPTION

The command `disnix-ssh-client' provides remote access to a `disnix-service' instance running on a machine in the network by using a SSH connection. This allows the user to perform remote deployment operations on a target machine through SSH.

Because this command uses `ssh' to connect to the target machine, for every operation the user has to provide a password. If this bothers you, have a look at `ssh-agent'.

In most cases this command is not used directly, but is used by specifying the --interface option for a Disnix command-line utility (such as `disnix-env') or by setting the `DISNIX_CLIENT_INTERFACE' environment variable. By using one of those properties, the Disnix tools will use the given interface instead of the standard `disnix-client' which only provides loopback access.

OPTIONS

Operations:

--import Imports a given closure into the Nix store of the target machine. Optionally, transfers the closure from this machine to the target machine

--export Exports the closure of a given Nix store path of the target machine into a file, and optionally downloads it

--print-invalid Prints all the paths that are not valid in the Nix store of the target machine

--realise Realises the given store derivation on the target machine

--set Creates a Disnix profile only containing the given derivation on the target machine

--query-installed Queries all the installed services on the given target machine

--query-requisites Queries all the requisites (intra-dependencies) of the given services on the target machine

--collect-garbage Collects garbage on the given target machine

--activate Activates the given service on the target machine

--deactivate Deactivates the given service on the target machine

--lock Acquires a lock on a Disnix profile of the target machine

--unlock Release the lock on a Disnix profile of the target machine

--snapshot Snapshots the logical state of a component on the given target machine

--restore Restores the logical state of a component on the given target machine

--delete-state Deletes the state of a component on the given machine

--query-all-snapshots Queries all available snapshots of a component on the given target machine

--query-latest-snapshot Queries the latest snapshot of a component on the given target machine

--print-missing-snapshots Prints the paths of all snapshots not present on the given target machine

--import-snapshots Imports the specified snapshots into the remote snapshot store

--export-snapshots Exports the specified snapshot to the local snapshot store

--resolve-snapshots Converts the relative paths to the snapshots to absolute paths

--clean-snapshots Removes older snapshots from the snapshot store

--capture-config Captures the configuration of the machine from the Dysnomia container properties in a Nix expression

--help Shows the usage of this command to the user

--version Shows the version of this command to the user
General options:

-t, --target=TARGET  Specifies the hostname and optional port number of the SSH server used to connect to the target machine

Import/Export/Import snapshots/Export snapshots options:

--localfile  Specifies that the given paths are stored locally and must be transferred to the remote machine if needed
--remotefile  Specifies that the given paths are stored remotely and must transferred from the remote machine if needed

Set/Query installed/Lock/Unlock options:

-p, --profile=PROFILE  Name of the Disnix profile. Defaults to: default

Collect garbage options:

-d, --delete-old  Indicates whether all older generations of Nix profiles must be removed as well

Activation/Deactivation/Snapshot/Restore/Delete state options:

--type=TYPE  Specifies the activation module that should be used, such as echo or process.
--arguments=ARGUMENTS  Specifies the arguments passed to the Dysnomia module, which is a string with key=value pairs
--container=CONTAINER  Name of the container in which the component is managed. If omitted it will default to the same value as the type.

Query all snapshots/Query latest snapshot options:

-C, --container=CONTAINER  Name of the container in which the component is managed
-c, --component=COMPONENT  Name of the component hosted in a container

Clean snapshots options:

--keep=NUM  Amount of snapshot generations to keep. Defaults to: 1

ENVIRONMENT

SSH_USER  Username that should be used to connect to remote machines
SSH_OPTS  Additional properties which are passed to the ssh command
DISNIX PROFILE  Sets the name of the profile that stores the manifest on the coordinator machine and the deployed services per machine on each target (Defaults to: default)
DYSNOMIA_STATE_DIR  Specifies where the snapshots must be stored on the coordinator machine (defaults to: /var/state/dysnomia)
A.2.19 disnix-visualize

disnix-visualize — Generate a visualization graph of a manifest

Synopsis

disnix-visualize [OPTION] [MANIFEST]

DESCRIPTION

The command `disnix-visualize` generates a graph showing services (as nodes), inter-dependencies (as arrows) and target machines (as clusters) from a manifest file generated by `disnix-manifest`. If no manifest file is given, it uses the manifest of the last deployed configuration.

The graph is exported as dot format, which can be transformed in a raster image format by using the `dot` command.

OPTIONS

- `-p`, `--profile=PROFILE` Name of the profile in which the services are registered. Defaults to: default
- `--coordinator-profile-path=PATH` Path to the manifest of the previous configuration. By default this tool will use the manifest stored in the disnix coordinator profile instead of the specified one, which is usually sufficient in most cases.
- `--no-containers` Do not visualize the containers.
- `-h`, `--help` Shows the usage of this command to the user
- `-v`, `--version` Shows the version of this command to the user

COPYRIGHT

Copyright (C) 2008-2016 Sander van der Burg