# Service Configuration Management

Eelco Dolstra, Martin Bravenboer, and Eelco Visser

Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
{eelco, martin, visser}@cs.uu.nl

**Abstract.** The deployment of services — sets of running programs that provide some useful facility on a system or network — is typically implemented through a manual, time-consuming and error-prone process. For instance, system administrators must deploy the necessary software components, edit configuration files, start or stop processes, and so on. This is often done in an *ad hoc* style with no reproducibility, violating proper configuration management practices. In this paper we show that build management, software deployment and service deployment can be integrated into a single formalism. We do this in the context of the Nix software deployment system, and show that its advantages — co-existence of versions and variants, atomic upgrades and rollbacks, and component closure — extend naturally to service deployment. The approach also elegantly extends to distributed services. In addition, we show that the Nix expression language can simplify the implementation of crosscutting variation points in services.

## 1 Introduction

The deployment of software services — sets of running programs that provide some useful facility on a system or network — is very often a time-consuming and error-prone activity for developers and system administrators. In order to produce a working service, one must typically install a large set of components, put them in the right locations, write configuration files, create state directories or initialise databases, make sure that all sorts of processes are started in the right order on the right machines, and so on. These activities are quite often performed manually, or scripted in an *ad hoc* way.

A software service typically consists of a set of processes running on one or more machines that cooperate to provide a useful facility to an end-user or to some other software system. An example might be a bug tracking service, implemented through a web server running certain servlets, and a back-end database storing persistent data. A service generally consists of a set of software components, static data files, dynamic state (such as databases and log files), and configuration files that tie all these together.

A particularly troubling aspect of common service deployment practice is the lack of good *configuration management*. For instance, the software environment of a machine may be under the control of a package manager, and the configuration files and static data of the service under the control of a version management system. That is, these two parts of the service are both under CM control. However, the *combination* of the two *isn't*: we don't have a way to express that (say) the configuration of a web server consists of a composition of a specific instance of Apache, specific versions of configuration files and data files, and so on.

This means that we lack important features of good CM practice. There is no *identification*: we do not have a way to name what the running configuration on a system is. We may have a way to identify the configurations of the code and data bits (e.g., through package management and version management tools), but we have no handle on the composition. Likewise, there is no *derivation management*: a software service is ideally an automatically constructed derivate of code and data artifacts, meaning that we can always automatically rebuild it, e.g., to move it to another machine. However, if administrators ever manually edit a file of a running service, we lose this property.

In practice, we see that important service deployment operations are quite hard. Moving a service to another machine can be time-consuming if we have to figure out exactly what software components to install to establish the proper environment for the service. Having multiple instances of a service (e.g., a test and production instance) running side-by-side on the same machine is difficult since we must arrange for the instances not to overwrite each other, which often entails manually copying files and tweaking paths in configuration files. Performing a roll-back of a configuration after an upgrade might be very difficult, in particular if software components were replaced.

In this paper we argue that we can overcome these problems by integrating build management, software deployment, and service deployment into a single formalism. We do this in the context of the *Nix deployment system* [7, 6], which we developed to overcome a number of problems in the field of software deployment, such as the difficulty in reliably identifying component dependencies and in deploying versions of components side-by-side. We show in this paper that the Nix framework is well suited for service deployment, simply by treating the static, non-code parts of service configurations as components. By doing so, we can build, deploy and manage services with the same techniques that we previously applied to component deployment.

The central contribution of this paper is that we posit that deployment and service configuration can be integrated by viewing service configurations as components. This allows all of Nix's advantages to apply to the service configuration realm. Specifically, this yields the following contributions:

- Services become *closures*. Since Nix helps to prevent undeclared dependencies, we can be reasonably certain that the *Nix expression* that describes a service has no external dependencies, i.e., is self-contained. Thus, we can unambiguously reproduce such a service. For instance, we can trivially pick it up and move it to another machine, and it will still work.
- Different instances of a service can automatically co-exist on the same machine. This includes not just variation in feature space (e.g., test and production instances), but also variation in time (e.g., older versions of the service). This allows us to efficiently and reliably roll back to previous configurations.
- Since deployment of code and non-code parts of a service are now integrated in a single formalism and tool, deployment effort and complexity is substantially reduced.
- Nix's functional expression language allows variability in configurations to be expressed succinctly. A major annoyance in writing configuration files for services is a *lack of abstraction*: crosscutting configuration choices (e.g., a port number) often end up in many places in many different configuration files and scripts. By gener-

ating these from feature specifications specified in Nix expressions, we reduce the deployment effort and the opportunity for errors due to inconsistencies.
- We show how services can be composed from smaller sub-services, rather than having a single monolithic description of the service as a whole.
- Multi-machine, multi-platform configurations can be described succinctly because Nix expressions can specify for what platform each component is to be built, and on what machine it is to run. Thus we obtain a centralised view of a distributed service.

*Outline* This paper is structured as follows. In Section 2 we motivate our work, present a simple, 'monolithic' example of service deployment through Nix, and introduce the underlying concepts and techniques. Section 3 shows how Nix services can be made compositional. In Section 4 we show how crosscutting configuration aspects can be implemented. We demonstrate how distributed deployment can be elegantly expressed in Nix in Section 5. We discuss our experiences with Nix service deployment in Section 6, and related work in Section 7.

## 2 Overview

### 2.1 Service components

From the perspective of a user, a service is a collection of data in some format with a coherent set of operations (use cases) on those data. Typical examples are a Subversion version management service in which the data are repositories and the operations are version management actions such as committing, adding, and renaming files; a Wiki service in which the data are web pages and operations are viewing, editing, and adding pages; and a Jira issue-tracking service in which the data consists of entries in an issue data-base and the operations are issue management actions such as opening, updating, and closing issues. In these examples, the service has persistent data that is stored on the server and that can be modified by the operations of the server. However, a service can also be *stateless* and just provide some computation on data provided by the client at each request, e.g., a translation service that translates a document uploaded by the client.

From the perspective of a system administrator, a service consists of *data directories* containing the persistent state, *software components* that implement the operations on the data, and a *configuration* of those components binding the software to the data and to the local environment. Typically, the configuration includes definitions of paths to data directories and software components, and things such as the URLs and ports at which the service is provided. Furthermore, the configuration provides *meta-operations* for initialising, starting, and stopping the service. **Fig. 1** illustrates this with a diagram sketching the composition of a version management service based on Apache and Subversion components. The *control* component is a script that implements the meta-operations, while the file `httpd.conf` defines the configuration. The software components underlying a service are generally not self-contained, but rather composed from a (large) number of auxiliary components. For example, **Fig. 1** shows that Subversion and Apache depend on a number of other software components such as OpenSSL and Berkeley DB.

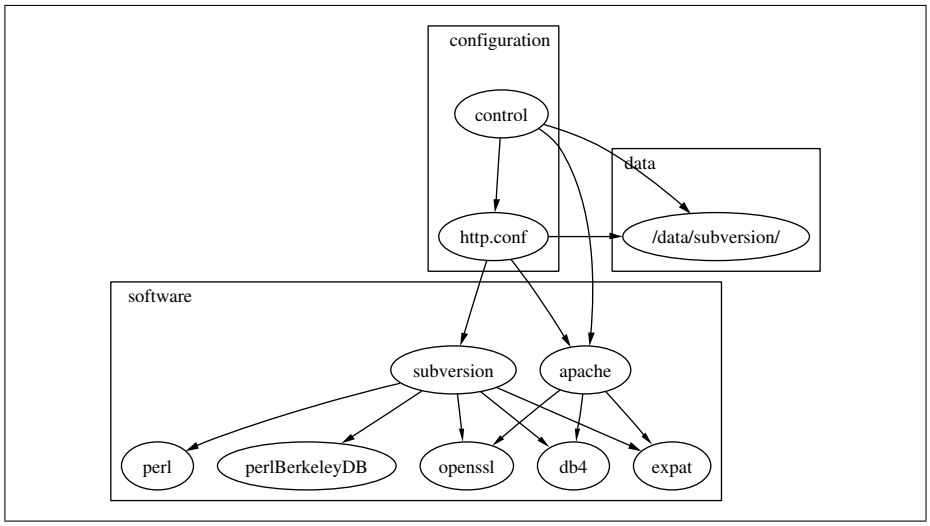



**Fig. 1.** Dependencies between code, configuration, and data components in a subversion service.

### 2.2 Service configuration and deployment

Setting up a service requires installing the software components and all of their dependencies, ensuring that the versions of the installed components are compatible with each other. The data directories should be initialised to the required format and possibly filled with an initial data set. Finally, a configuration file should be created to point to the software and data components. For example, **Fig. 2** shows an excerpt of an Apache `httpd.conf` configuration file for a Subversion service. The file uses absolute pathnames to software components such as a WebDAV module for Subversion, and data directories such as the place where repositories are stored.

Installation of software components using traditional package management systems is fraught with problems. Package managers do not enforce the completeness of dependency declarations of a component. The fact that a component works in a test environment, does not guarantee that it will work on a client site, since the test environment may provide components that are not explicitly declared as dependencies. As a consequence, component compositions are not reproducible. The installation of components in a common directory makes it hard to install multiple versions of a component or to roll back to a previous configuration if it turns out that upgrading produces a faulty configuration.

These problems are compounded in the case of service management. Typically, management of configuration files is not coupled to management of software components. Configuration files are maintained in some specific directory in the file system and changing a configuration is a destructive operation. Even if the configuration files are under version management, there is no coupling to the versions of the software components that they configure. Thus, even if it is possible to do a roll back of the configuration files to a previous time, the installation of the software components may

```
ServerRoot "/var/httpd"
ServerAdmin eelco@cs.uu.nl
ServerName svn.cs.uu.nl:8080
DocumentRoot "/var/httpd/root"
LoadModule dav_svn_module /usr/lib/modules/mod_dav_svn.so
<Location /repos>
    AuthType Basic
    AuthDBMUserFile /data/subversion/db/svn-users
    ...
    DAV svn
    SVNParentPath /data/subversion/repos
</Location>
```

**Fig. 2.** Service configuration. Excerpts from a httpd.conf file showing hosting of Subversion by an Apache server.

have changed in the meantime and become out of synch with the old configuration files. Finally, having a global configuration makes it hard to have multiple versions of a service running side by side, for instance a production version and a version for testing an upgrade.

### 2.3 Capturing component compositions with Nix expressions

The Nix software deployment system was developed to deal with the problems of correctness and variability in software deployment. Here we show that Nix can be applied to the deployment of services as well, treating configuration and software components uniformly. We first show the high-level definition of service components and their composition using Nix expressions and then discuss the underlying implementation techniques.

The Nix expression language is a simple functional language that is used to define the derivation of software component from their dependencies. (A more detailed exposition of the language is given in [6].) The basic values of the language are strings, *paths* such as ./builder.sh, and *attribute sets* of the form $\{f_1{=}e_1;\ldots f_n{=}e_n;\}$, binding the value of expressions $e_i$ to fields $f_i$. The form $\{f_1,\ldots,f_n\}$: $e$ defines a function with body $e$ that takes as argument an attribute set with fields named $f_1$, ..., $f_n$. A function call $e_1\,e_2$ calls the function $e_1$ with arguments specified in the attribute set $e_2$.

For example, **Fig. 3** shows the definition of a function for building a Subversion component given its build-time dependencies. The keyword inherit in attribute sets causes values to be inherited from the surrounding scope. The dependency stdenv comprises a collection of standard utilities for building software components, including a C compiler and library. The function fetchurl downloads a file given its URL and MD5 hash. Likewise, the other arguments represent components such as OpenSSL. Thus, the body of the function is an attribute set with everything needed for building Subversion. The build script builder.sh (**Fig. 4**) defines how to build the component given its inputs, in this case, using a typical command sequence for Unix components. The locations of the build inputs in the file system are provided to the script using environment variables.

```
{ stdenv, fetchurl, openssl, httpd, db4, expat }:
stdenv.mkDerivation {
  name = "subversion-1.2.0";
  builder = ./builder.sh;
  src = fetchurl {
   url = http://.../subversion-1.2.0.tar.bz2;
   md5 = "f25c0c884201f411e99a6cb6c25529ff";
  };
  inherit openssl httpd expat db4;
}
```

**Fig. 3.** subversion/default.nix: Nix expression defining a function that derives a Subversion component from its sources and dependencies.

```
tar xvfj $src
cd subversion-*
./configure --prefix=$out --with-ssl --with-libs=$openssl ...
make
make install
```

**Fig. 4.** subversion/builder.sh: Build script for the Subversion component.

The special environment variable out indicates the location where the result of the build action should be installed. An instance of the Subversion component can be created by calling the function with concrete values for its dependencies. For example, assuming that pkgs/i686-linux.nix defines a specific instance of the Subversion function, the invocation

```
nix-env -f pkgs/i686-linux.nix -i subversion
```

of the command nix-env builds this instance and makes it available in the environment of the user. The result is a component composition that can be uniquely identified and reproduced, as will be discussed below.

A service can be constructed just like a software component by composing the required software, configuration, and control components. For example, **Fig. 5** defines a function for producing a Subversion service, when given Apache (httpd) and Subversion components. The build script of the service creates the Apache configuration file httpd.conf and the control script bin/control from templates by filling in the paths to the appropriate software components. That is, the template httpd.conf.in contains placeholders such as

```
LoadModule dav_svn_module @subversion@/modules/mod_dav_svn.so
```

rather than absolute paths. The funcion in **Fig. 5** can be instantiated to create a concrete instance of the service. For example, **Fig. 6** defines the composition of a concrete Subversion service using a ./httpd.conf file defining the particulars of the service. Just like installing a software component composition, service composition can be reproducibly installed using the nix-env command:

```
{ stdenv, apacheHttpd, subversion }:
stdenv.mkDerivation {
  name    = "svn-service";
  builder = ./builder.sh; # Build script.
  control = ./control.in; # Control script template.
  conf    = ./httpd.conf.in; # Apache configuration template.
  inherit apacheHttpd subversion;
}
```

**Fig. 5.** services/svn.nix: Function for creating an Apache-based Subversion service.

```
pkgs = import ../pkgs/system/all-packages.nix;
subversion = import ../subversion/ {
  # Get dependencies from all-packages.nix.
  inherit (pkgs) stdenv fetchurl openssl httpd ...;
};
webServer = import ../services/svn.nix {
  inherit (pkgs) stdenv apacheHttpd;
};
```

**Fig. 6.** configuration/svn.nix: Composition of a Subversion service.

```
nix-env -p /nix/profiles/svn -f configuration/svn.nix -i
```

The service installation is made available through a *profile*, e.g., /nix/profiles/svn, which is a symbolic link to the composition just created. The execution of the service, initialisation, activation, and deactivation, can be controlled using the control script with commands such as

```
/nix/profiles/svn/bin/control start
```

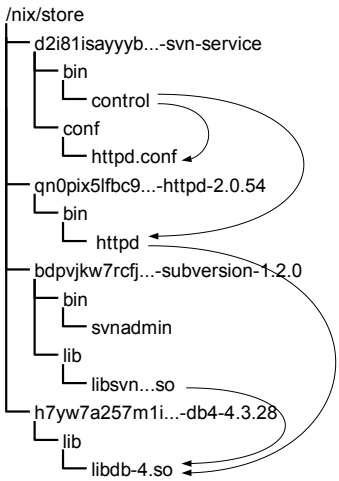
### 2.4 Maintenance

A service evolves over time. New versions of components become available with new functionality or security fixes; the configuration needs to be adapted to reflect changing requirements or changes in the environment; the machine that hosts the service needs to be rebooted; or the machine is retired and the service needs to be migrated to another machine. Such changes can be expressed by adapting the Nix expressions appropriately and rebuilding the service. The upgrade-server script in **Fig. 7** implements a common sequence of actions to upgrade a service: build the new service, stop the old service, and start the new one. (In Section 6.2 we sketch how we can prevent the downtime between stopping and starting the old and new configurations.) Since changes are non-destructive, it is possible (through a similar command sequence) to roll back to a previous installation if the new one is not satisfactory. By keeping the Nix expressions defining a service under version management, the complete history of all aspects of the service is managed, and any version can be reproduced at any time.

```
# Recall the old server
oldServer=$(readlink -f $profiles/$serverName || true)

# Build and install the new server.
nix-env -p $profiles/$serverName -f "$nixExpr" -i

# Stop the old server.
if test -n "$oldServer"; then
    $oldServer/bin/control stop || true
fi

# Start the new server.
$profiles/$serverName/bin/control start
```

**Fig. 7.** upgrade-server script: Upgrading coupled to activation and deactivation.

### 2.5 Implementation




Nix supports side-by-side deployment of configurations and roll-backs by storing components in isolation from each other in a *component store* [7]. An example for some of the components built from the webServer value in **Fig. 6** is shown on the right. The arrows indicate references (by file name) between components. Each call to mkDerivation results in the construction of a component in the Nix store, residing in a path name containing a *cryptographic hash* of all inputs (dependencies) of the derivation. That is, when we build a derivation described in a Nix expression, Nix recursively builds its inputs, computes a path for the component by hashing the inputs, then runs its builder to produce the component. Due to the hashing scheme, changing any input in the Nix expression and rebuilding causes the new component to end up in a different path in the store. Hence, different components never overwrite each other. The advantage of hashes is that they prevent undeclared build-time dependencies, and enable detection of runtime dependencies by scanning for hashes in the files of components.

By storing components in isolation, Nix enables side by side installation of services. That is, there can be multiple configuration files instantiating the same software components for different services. For example, two Subversion servers with different authentication regimes, or webservers for different domains can be hosted on the same machine. Furthermore, since a collection of Nix expressions exactly describes all aspects of a service, it is easy to reproduce service installations on multiple machines.

We support roll-backs through profiles, which as mentioned above are symbolic links to the current instance of a service configuration. When we build a new configura-

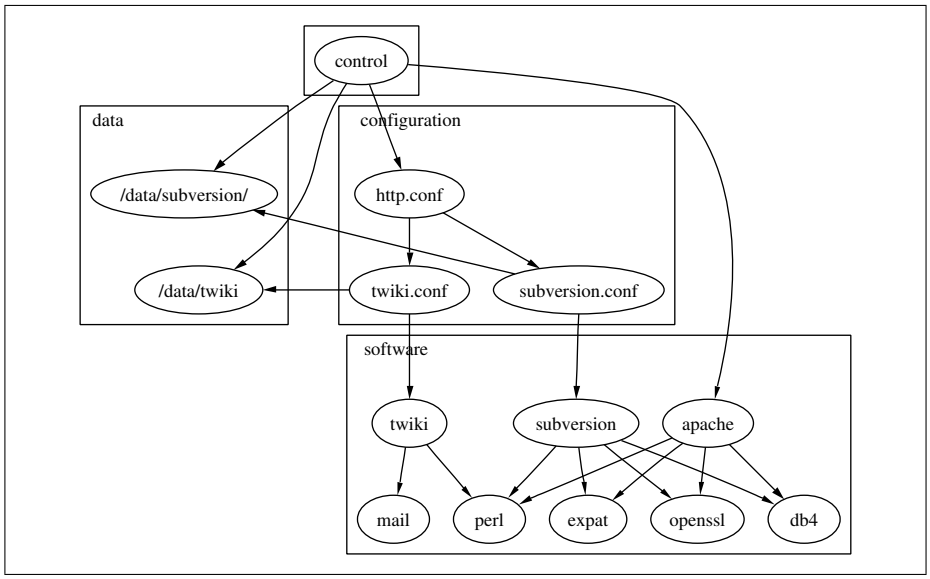



**Fig. 8.** Architecture of an Apache-based composition of the Subversion and TWiki services.

tion, we make the symlink point to the store path of the new configuration component. To perform a roll-back, we simply switch the symlink back to the store path of the previous configuration.

## 3 Composition of Services

In the previous section we have shown a sketch of an Apache-based Subversion service deployed using Nix. The major advantage is that such a service is a closure: all code and configuration elements of the service are described in the Nix expression, and can thus be reproduced arbitrarily. Clearly, we can deploy other services along similar lines. For instance, we might want to deploy an Apache-based TWiki service by providing Nix expressions to build the TWiki components and to compose them with Apache (which entails writing a parameterised `httpd.conf` that enables the TWiki CGI scripts in Apache). However, suppose we now want a *single* Apache server that provides both the Subversion and TWiki services. How can we easily *compose* such services?

We solve this by factoring out the service-specific parts of `httpd.conf` into separate components called *subservices*. That is, we create configuration components that contain `httpd.conf` fragments such as `twiki.conf` and `subversion.conf`. The top-level `httpd.conf` merely contains global server configuration such as host names or port numbers, and includes the service-specific fragments. A sketch of this composition is shown in **Fig. 8**.

**Fig. 9** shows a concrete elaboration. Here, the functions imported from `../subversion-service` and `../jira-service` build the Subversion and JIRA configuration fragments and

```
subversionService = import ../subversion-service {
  httpPort = 80; # see Section 4.
  reposDir = "/data/subversion"; ...
};
jiraService = import ../jira-service {
  twikisDir = "/data/twiki"; ...
};
webServer = import ../apache-httpd {
  inherit (pkgs) stdenv apacheHttpd;
  hostName = "svn.cs.uu.nl";
  httpPort = 80;
  subServices = [subversionService jiraService];
};
```

**Fig. 9.** Nix expression for the service in **Fig. 8**.

store them under a subdirectory types/apache-httpd/conf/ in their prefixes. The function imported from ../apache-httpd then builds a top-level httpd.conf that includes those fragments. That is, there is a *contract* between the Apache service builder and the subservices that allows them to be composed.

The subServices argument of the function in ../apache-httpd specifies the list of service components that are to be composed. By modifying this list and running upgrade-server, we can easily enable or disable subservices.

## 4 Variability and crosscutting configuration choices

A major factor in the difficulty of deploying services is that many configuration choices are *crosscutting*: the realisation of such choices affects multiple configuration files or multiple points in the same file. For instance, the Apache server in **Fig. 9** requires a TCP *port number* on which the server listen for requests, so we pass that information to the top-level webServer component. However, the user management interface of the Subversion service also needs the port number to produce URLs pointing to itself. Hence, we see that the port number is specified in two different places. In fact, Apache's configuration itself already needs the port in several places, e.g.,

```
ServerName example.org:8080
Listen 8080
<VirtualHost _default_:8080>
```

are some configuration statements that can occur concurrently in a typical httpd.conf. This leads to obvious dangers: if we update one, we can easily forget to update the other.

A related problem is that we often want to build configurations in several variants. For instance, we might want to build a server in test and production variants, with the former listening on a different port. We could make the appropriate edits to the Nix expression every time we build either a test or production variant, or maintain two copies of the Nix expression, but both are inconvenient and unsafe.

```
{productionServer}:
let {
  port = if productionServer then 80 else 8080;
  webServer = import ./apache-httpd {
    inherit (pkgs) stdenv apacheHttpd;
    hostName = "svn.cs.uu.nl";
    httpPort = port;
    subServices = [subversionService jiraService];
  };
  subversionService = import ./subversion {
    httpPort = port;
    reposDir = "/data/subversion"; ...
  };
  jiraService = import ./jira {
    twikisDir = "/data/twiki"; ...
  };
}
```

**Fig. 10.** Dealing with crosscutting configuration choices

Using the Nix expression language we have the abstraction facilities to easily support possibly crosscutting variation points. **Fig. 10** shows a refinement of the Apache composition in **Fig. 9**. This Nix expression is now a *function* accepting a single boolean argument productionServer that determines whether the instance is a test or production configuration. This argument drives the value selected for the port number, which is propagated to the two components that require this information. Thus, this crosscutting parameter is specified in only one place (though implemented in two). This is a major advantage over most configuration file formats, which typically lack variables or other means of abstraction. For example, Enterprise JavaBeans deployment descriptors frequently become unwieldy due to crosscutting variation points impacting many descriptors.

It is important to note that due to the cryptographic hashing scheme (Section 2.5), building the server with different values for productionServer (or manually editing in the Nix expression any aspect such as the port attribute) yields different hashes and thus different paths. Therefore, multiple configurations automatically can exist side by side on the same machine.

## 5 Distributed deployment

Complex services are often composed of several subservices *running on different machines*. For instance, consider a simple scenario of a *JIRA bug tracking system*. This service consists of two separately running subservices, possibly on different machines: a Jetty servlet container, and a PostgreSQL database server. These communicate with each other through TCP connections.

Such configurations are often labourious to deploy and maintain, since we now have two machines to administer and configure. This means that administrators have to login

to multiple machines, make sure that services are started and running, and so on. That is, without sufficient automation, the deployment effort rises linearly.

In this section we show how one can implement distributed services by writing a single Nix expression that describes each subservice and the machine on which it is to run. A special *service runner component* will then take care of distributing the closure of each subservice to the appropriate machines, and remotely running their control scripts.

An interesting complication is that the various machines may be of different machine types, or may be running different operating systems. For instance, the Jetty container might be running on a Linux machine, and the PostgreSQL database on a FreeBSD machine. Nix has the ability to do distributed builds. Nix derivations must specify the *platform type* on which the derivation is to be performed:

```
derivation {
  name = "foo";
  builder = ./builder.sh;
  system = "i686-linux"; ... }
```

(Usually, the system argument is omitted because it is inherited from the standard environment (stdenv).) When we build such a derivation on a machine that is not of the appriate type (e.g., i686-linux), Nix can automatically forward the build to another machine of the appriate type, if one is known. This is done by recursively building the build inputs, copying the closures of the build inputs to the remote machine, invoking the builder on the remote machine, and finally copying the build result back to the local machine. Thus, the user needs not be aware of the distributed build: there is no apparent difference between a remote and local build.

**Fig. 11** shows the Nix expression for the JIRA example. We have two generic services, namely PostgreSQL and Jetty. There is one concrete subservice, namely the JIRA servlet. This component is plugged into both generic services as a subservice, though it provides a different interface to each (i.e., implementing different contracts). To PostgreSQL, it provides an initialisation script that creates the database and tables. To Jetty, it provides a WAR servlet that can be loaded at a certain URI path.

The PostgreSQL service is built for FreeBSD, while all other components are built for Linux (all on x86 hardware). This is accomplished by passing input packages to the builders either for FreeBSD or for Linux (e.g., inherit (pkgsFreeBSD) stdenv ...), which include the standard environment and therefore specify the system on which to build.

The two generic servers are combined into a single logical service by building a *service runner component*, which is a simple wrapper component that at build time takes a list of services, and generates a control script that starts or stops each in sequence. It also takes care of distribution by deploying the closure of each service to the machine identified by its host attribute, e.g., itchy.labs.cs.uu.nl for the Jetty service. For instance, when we run run the service runner's start action, it copies each service, then executes each service's start action remotely.

An interesting point is that the Nix expression nicely deals with a crosscutting aspect of the configuration: the host names of the machines on which the services are to run. These are crosscutting because the services need to know each other's host names. In order for the JIRA servlet to access the database, the JIRA servlet needs to know the

```
# Build a Postgres server on FreeBSD.
postgresService = import ./postgresql {
  inherit (pkgsFreeBSD) stdenv postgresql;
  host = "losser.labs.cs.uu.nl"; # Machine to run on.
  dataDir = "/var/postgres/jira-data";

  subServices = [jiraService];
  allowedHosts = [jettyService.host]; # Access control.
};

# Build a Jetty container on Linux.
jettyService = import ./jetty {
  inherit (pkgsLinux) stdenv jetty j2re;
  host = "itchy.labs.cs.uu.nl"; # Machine to run on.

  # Include the JIRA servlet at URL path.
  subServices = [ { path = "/jira"; war = jiraService; } ];
};

# Build a JIRA service.
jiraService = import ./jira/server-pkgs/jira/jira-war.nix {
  inherit (pkgsLinux) stdenv fetchurl ant postgresql_jdbc;
  databaseHost = postgresService.host;  # Database to use.
};

# Compose the two services.
serviceRunner = import ./runner {
  inherit (pkgsLinux) stdenv substituter;
  services = [postgresService jettyService];
};
```

**Fig. 11.** 2-machine distributed service

host name of the database server. Conversely, the database server must allow access to the machine running the Jetty container. Here, host names are specified only once, and are propagated using expressions such as allowedHosts = [jettyService.host].

## 6 Discussion

### 6.1 Experience

We have used the Nix-based approach described in this paper to deploy a number of services, some in production use and some in education or development[1]. The production services are:

[1] The sources of the services described in this paper are available at http://svn.cs.uu.nl/repos/trace/services/trunk. Nix itself can be found at [1].

– An Apache-based Subversion server (svn.cs.uu.nl) with various extensions, such as a user and repository management system. This is essentially the service described in Section 3.
– An Apache-based TWiki server (http://www.cs.uu.nl/wiki/Center), also using the composable infrastructure of Section 3. Thus, it is easy to create an Apache server providing both the Subversion and TWiki services.
– A Jetty-based JIRA bug tracking system with a PostgreSQL database backend, as described in Section 5.

Also, Nix services were used in a Software Engineering course to allow teams of students working on the implementation of a Jetty-based web service (a Wiki) to easily build and run the service.

In all these cases, we have found that the greatest advantage of Nix service deployment is the ease with which configurations can be reproduced: if a developer wants to create a running instance of a service on his own machine, it is just a matter of a checkout of the Nix expressions and associated files, and a call to upgrade-server. Without Nix, setting up the requisite software environment for the service would be much more work: installing software, tweaking configuration files to the local machine, creating state locations, and so on. The Nix services described above are essentially "plug-and-play". Also, developer machines can be quite heterogenous. For instance, since Nix closures are self-contained, there are no dependencies on the particulars of various Linux distributions that might be used by the developers.

The ability to easily set up a test configuration is invaluable, as it makes it fairly trivial to experiment with new configurations. The ability to reliably perform a rollback, even in the face of software upgrades, is an important safety net if testing has failed to show a problem with the new configuration.

### 6.2 Online upgrading

As described in Section 2.4, to upgrade a service from an old to a new configuration requires that we first run the stop action of the old service, and then the start action on the new service. However, this means that there is a time window in which the service is not available.

In the most general case, this is unavoidable. For instance, if we upgrade the Apache httpd server, then we *must* restart, since neither the C language nor Apache itself has any provisions for dynamic updating (replacing the code of an executing process). Dynamic updating is only very rarely available, so in such cases we have no choice but to restart. However, if the new configuration only differs from the old one in that the Apache configuration file or some other data file has changed, we can typically upgrade on the fly. Usually this this is accomplished by modifying the files in question (e.g., editing httpd.conf), and then notifying the running service that it is to reload its configuration files (e.g., by sending it a HUP signal in Unix).

The problem is that this fits poorly in the Nix model. This is because Nix components, *such as service configuration files*, are pure — they cannot be modified after they have been built. Also, we would like to use a service's reload feature *only* if that is all we have to do. For instance, if code components were also changed, we want to

restart instead of reload. In other words, we wish to use configuration file reloading as an automatic optimisation of restarting only if it is safe to do so.

We can solve the first problem by adding a level of indirection to configuration files. Rather than having services load their configuration files directly from their location in the Nix store, e.g., /nix/store/4mm5dzlnhs20.../httpd.conf, we load them through *temporary symbolic links* that point to the actual locations, e.g., /tmp/instance-943/httpd.conf. We also keep a mapping from running services to the temporary links they are using. When a service is first started, we create this symlink. When we upgrade, upgrade-service simply change the symlink to point to the new configuration file (which is an atomic action on Unix), and signal the running server process to reload the configuration file.

We can address the second problem — performing a reload only when that is sufficient — by having upgrade-server compute the differences between the dependency graphs and contents of the old and new configurations, and only reload only when only "reloadable" parts have changed. This is a conservative approach. For instance, if two Apache configurations only differ in that their top-level store paths differ and contain differing httpd.conf files, then a reload is safe. If, on the other hand, any dependencies of the top-level store path changed, we consider a reload unsafe and restart instead.

## 7 Related work

We are not the first to realize that the deployment of software should be treated as part of software configuration. In particular the SWERL group at the University of Colorado at Boulder has argued for the application of software configuration management to software deployment [11], developed a framework for characterizing software deployment processes and tools [3], and the experimental system Software Dock integrating software deployment processes [10]. However, it appears that our work is unique in unifying the deployment of software and configuration components.

Nix is both a high-level build manager for components, and a deployment tool. As such it subsumes some of the tasks of both build managers (e.g., Make [8]), and deployment tools (e.g., RPM [9]).

Make is sometimes used to build various configuration files. However, Make doesn't allow side-by-side deployment, as running make overwrites the previously built configurations. Thus, rollbacks are not possible. Also, the Make language is rather primitive. In particular, since the only abstraction mechanisms are global variables and patterns, it is hard to instantiate a subservice multiple times. Build tools such as Odin [4] and Maak [5] have more functional specification languages.

Cfengine is a popular tool system administration tool [2]. A declarative description of sets of machines and the functionality they should provide is given, along with imperative actions that can realise a configuration, e.g., by rewriting configuration files in /etc. The principal downside of such a model is that it is *destructive*: it realises a configuration by overwriting the current one, which therefore disappears. Also, it is hard to predict what the result of a Cfengine run will be, since actions are typically specified as edit actions on system files, i.e., the initial state is not always specified. This is in contrast to the fully generational approach advocated here, i.e., Nix builder generate

configurations fully indepently from any previous configurations. Finally, since actions are specified with respect to fixed configuration file locations (e.g., /etc/sendmail.mc), it is not easy for multiple configurations to co-exist. In the present work, fixed paths are only used for truly mutable state such as databases and log files.

## 8 Conclusion

In this paper we have presented a method for service deployment based on the Nix deployment system. We have argued that its software deployment properties carry over nicely into the domain of service deployment. Thanks to Nix's cryptographic hashing property, we gain the ability to reliably recreate configurations, to have multiple instances of configurations exist side by side, and to roll back to old configurations. Nix's functional expression language gives us a way to deal with variability and crosscutting variation points in an efficient way, and to componentise services. Finally, the approach extends to distributed deployment.

## References


1. Nix deployment system. http://www.cs.uu.nl/wiki/Trace/Nix, 2005.
2. M. Burgess. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3), 1995.
3. A. Carzaniga et al. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, University of Colorado, April 1998.
4. G. M. Clemm. *The Odin System — An Object Manager for Extensible Software Environments.* PhD thesis, University of Colorado at Boulder, February 1986.
5. E. Dolstra. Integrating software construction and software deployment. In B. Westfechtel, editor, *11th International Workshop on Software Configuration Management (SCM-11)*, volume 2649 of *Lecture Notes in Computer Science*, pages 102–117, Portland, Oregon, USA, May 2003. Springer-Verlag.
6. E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In L. Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, November 2004. USENIX.
7. E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.
8. S. I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–65, 1979.
9. E. Foster-Johnson. *Red Hat RPM Guide*. John Wiley and Sons, 2003.
10. R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide area network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, Maryland, USA, May 1997.
11. A. van der Hoek, R. S. Hall, A. Carzaniga, D. Heimbigner, and A. L. Wolf. Software deployment: Extending configuration management support into the field. *Crosstalk, The Journal of Defense Software Engineering*, 11(2), February 1998.