

# Declarative Testing and Deployment of Distributed Systems

Sander van der Burg  
Delft University of Technology  
The Netherlands  
s.vanderburg@tudelft.nl

Eelco Dolstra  
Delft University of Technology  
The Netherlands  
e.dolstra@tudelft.nl

## ABSTRACT

System administrators and developers who deploy distributed systems have to deal with a deployment process that is largely manual and hard to reproduce. This paper describes how networks of computer systems can be reproducibly and automatically deployed from declarative specifications. Reproducibility also ensures that users can easily instantiate a test environment, before deploying the specification to the production environment. Furthermore, from the same specifications we can instantiate virtual networks of virtual machines for both interactive and automated testing. This makes it easy to write automated regression tests that require external machines, need special privileges, or depend on the network topology. We instantiate machines from the specifications using NixOS, a Linux distribution built from a purely functional specification. We have applied our approach to a number of representative problems, including automatic regression testing of a Linux distribution and deployment of a continuous integration environment.

## 1. INTRODUCTION

In this paper we show how we can deploy networks of computer systems automatically on the basis of declarative specifications of such networks. Figure 1 shows an example of what we mean by this: it is a specification, in the formalism described in the remainder of this paper, of a small network that implements a Trac software project management service [7]. The network consists of an NFS server that provides storage to the other machines, a PostgreSQL database server, and an Apache webserver that provides the Trac web application. It is an executable specification: we can automatically build operating system instances that implement the specified functionality. Making deployment declarative in this manner has several major applications. Given such specifications, we can do a number of things automatically:

1. *Build and deploy* operating system instances that implement the specified functionality.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE 2010

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

```
network = {
  storage =
    { config, pkgs, webserver, ... }:
    {
      services.nfsd.enable = true;
      services.nfsd.exports =
        "/repos ${webserver.networking.hostName}(rw)";
    };
  postgresql =
    { config, pkgs, webserver, ... }:
    {
      services.postgresql.enable = true;
      services.postgresql.authentication =
        "host ... ${webserver.networking.ipAddress} trust";
    };
  webserver =
    { config, pkgs, storage, ... }:
    {
      fileSystems = [ {
        mountPoint = "/repos";
        device = "${storage.networking.hostName}:/repos";
      } ];
      services.nfs.enable = true;
      services.httpd.enable = true;
      services.httpd.extraSubservices = [
        { serviceType = "trac";
          projects = [
            { name = "Project Foobar";
              db = "postgresql://" +
                "${postgresql.networking.hostName}/foobar";
              svnRepo = "/repos/foobar";
            } ];
          } ];
    };
};
```

Figure 1: Network specification for a Trac service

2. *Build virtual machines* running those same operating system instances, and connect them in a virtual network. This allows a user to build and interactively test a test instance of a network before deploying it to the production environment.
3. Use those same virtual machines to *run an automated test suite* to perform integration or system testing.

This work straddles the fields of software testing and software deployment (or system configuration management), and is motivated by practical problems in each of these. In software testing, we are faced with the problem that it is hard to write automated system or integration tests for certain kinds of software. For instance, it is easy to write an auto-

mated regression test suite for a compiler – simply supply a set of input programs, compile them, run them, and check the output against the expected result. Such tests can easily be run from (say) a Makefile, and from a continuous build system. On the other hand, it is much harder to automate tests that have environmental dependencies (e.g., the webserver in Figure 1 that needs a database server on another machine), that need special privileges (e.g., we may want to run Apache as the `root` user so that we can test the code path that switches to the `wwwrun` user), or that depend on the network topology (e.g., whether a Bittorrent client can successfully connect through a NAT-enabled router). As a result, such tests tend to be done on an *ad hoc*, interactive basis. For example, many major Unix system components (e.g., the Linux kernel or the Apache web server) do not have automated regression test suites.

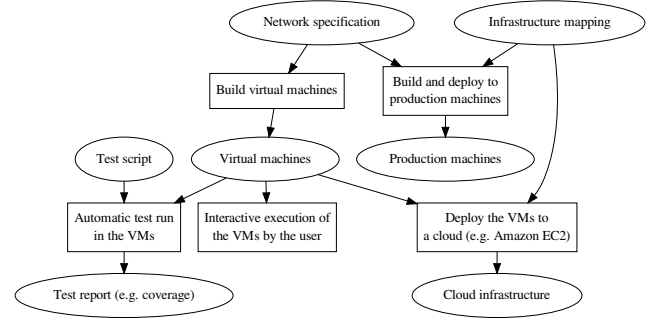
In software deployment, system administrators and developers who deploy distributed systems have to deal with a process that is still largely semi-manual, error-prone, and above all, hard to reproduce. For instance, a typical web application consisting of a database and a web server might require us to install PostgreSQL on one machine, Apache on another, and tweak configuration files and scripts until the desired functionality is achieved. The lack of reproducibility in particular means that it takes a lot of effort to set up a test environment to test configuration changes (such as software upgrades) before promoting them to production use, and we may not know for sure if the test environment actually matches the production environment.

In this work, we therefore have a number of goals:

- To make an automated system test as easy to write and run as a unit test, e.g., it should be easy to include in a `make check`. This opens a whole class of regression tests to developers.
- To make distributed deployment repeatable and reliable.
- To make it straightforward to test a network deployment by instantiating a clone of the production environment.

We achieve these goals by expanding on our previous work on *NixOS*, a Linux-based operating system distribution [5], which in turn builds on the purely functional package manager *Nix* [6]. In *NixOS*, the entire operating system – system packages such as the kernel, server packages such as Apache, end user packages such as Firefox, configuration files in `/etc`, boot scripts, and so on – is built from source from a specification in what is in essence a purely functional “Makefile”. For instance, the `webserver` and `storage` definitions in Figure 1 are so-called *NixOS* system configuration modules, each specifying the configuration of a *NixOS* machine. (We give an overview of the relevant concepts in *Nix* and *NixOS* in Section 2.) The way in which the underlying *Nix* package manager stores the *NixOS* artifacts in the filesystem ensures that upgrading a system is “safe” in that it does not depend on the previous state of the system and that users can easily roll back to previous configurations. The fact that *Nix* builds from a purely functional specification means that configurations can easily be reproduced.

The latter aspect forms the basis for this paper. In a normal *NixOS* system, the system configuration specification is used to build and activate a configuration on the local



**Figure 2: Use cases of a declarative network specification**

machine. In Section 3, we show how these machine specifications can be extended to networks of machines. By applying different “top-level” functions to such specifications, we can do a number of things. First, in Section 4, we use the model to perform distributed deployment by building each machine configuration locally, then copying the result to each machine, given an infrastructure mapping that tells us which target machine to use for each configuration (e.g., the `webserver` configuration is to be deployed to the machine `www.example.org`). Second, in Section 5, we use the same model to build virtual machines, as well as scripts that automatically start the virtual machines for a network and apply a test suite to them. Other deployment functions are possible: for instance, we could deploy the distributed system by generating virtual machine images and uploading them to a cloud infrastructure. Figure 2 summarises the various use cases of a declarative specification of a network.

We have applied our approach to a number of real-world scenarios, described throughout the paper, including the deployment of a multi-machine Trac configuration management server, deployment of a 9-machine production network for a continuous build system, automated coverage analysis of a Linux distribution, and automatic testing of a Quake server and clients. We discuss various aspects of our work in Section 6, and related work in Section 7.

## 2. BACKGROUND: NIX AND NIXOS

In our approach, distributed networks are specified in the *Nix expression language*, and built using the purely functional package manager *Nix*. Furthermore, the operating system instances that we build from the specifications are instances of *NixOS*, a Linux distribution based on *Nix*. In this section we give a brief overview of *Nix* and *NixOS*.

### *Nix*.

For the purposes of this paper, *Nix* [6, 4] (<http://nixos.org/>) can be seen as a purely functional “Make”. That is, like *Make* [8] and many other build tools, it performs build actions on the basis of a declarative specification of a graph of actions and their dependencies, but unlike *Make*, the specification is given in a lazy, purely functional language – the *Nix expression language*. This allows much more powerful abstractions to be expressed. Moreover, *Nix* stores the results of build actions in a way such that they cannot interfere with each other, e.g. that the results of multiple invocations

of a function do not overwrite each other. Namely, the output of a build step – or *derivation* – is stored under a unique path such as

```
/nix/store/q325djkc1ivlfyzan22197dc62gbq04z-firefox-3.5.2
```

where `q325djkc1ivl...` is a cryptographic hash of the inputs of the derivation, such as sources, compilers, libraries and build scripts.

The fundamental operation in the Nix expression language is the builtin function `derivation`, which takes as argument a set of name/value pairs, or *attributes*:

```
derivation {
  name = "foo";
  builder = "${bash}/bin/sh";
  args = [ "-c" "echo Hello $who > $out" ];
  who = "world";
}
```

A derivation describes the invocation of a command (usually a shell script) that must produce output under a path in the Nix store. The derivation is built by executing a program, whose path and command-line arguments are specified in the attributes `builder` and `args`, respectively. The other attributes are passed to the builder as environment variables. Attribute values can be (lists of) strings or other derivations. The latter denote the dependencies of the current derivation. When building a derivation, its dependencies are built first. The path of each dependency’s output in the Nix store is placed in the corresponding environment variable. Strings can also contain references to other derivations, enclosed in `{...}`. These are replaced by the derivation’s output path in the Nix store.

For instance, if we evaluate the derivation above, first the derivation denoted by the variable `bash` (not shown here) is built, resulting in a store path like `/nix/store/49ndfiqrlc9b...-bash-4.0-p17`. Then the present derivation is built, with the environment variable `who` set to `world`. Nix passes the intended location of the output in the Nix store, computed by hashing the input attributes, through the environment variable `out`. Thus, the derivation above will write the string `Hello world` to a path such as `/nix/store/6dsdb0j20n3b...-foo`.

A derivation can build anything, as long as it is pure, i.e. depends only on its explicitly defined inputs, and produces output under the path denoted by the environment variable `out`. Nix is primarily intended as a deployment tool – a *package manager*. Thus derivations are typically large steps that build entire packages. Figure 3 shows an example of a Nix expression to build the Apache web server. The language construct `rec { ... }` defines a set of variable bindings that can refer to each other, e.g. `httpd` refers to `apr` (the derivation that builds the Apache runtime package).

The derivation `httpd` shows the use of function abstractions to capture common build patterns: it calls the function `stdenv.mkDerivation`, which performs a build of a standard Unix-style package (namely, unpack the source, run an Autoconf configure script, run `make` to build, and finally `make install` to install the package under `$out`). Functions are defined using the syntax `arg: body`. Functions can also pattern-match on attribute sets: a function `{arg1, ..., argn}`: `body` must be called with an attribute set containing the named attributes. Ellipses can be used in the argument list to denote that additional attributes are to be ignored. For instance, the `webserver` function in Figure 1 must be called with a set containing at least attributes `name` `config`, `pkgs`

```
rec {
  httpd = stdenv.mkDerivation {
    name = "apache-httpd-2.2.13";
    src = fetchurl {
      url = mirror://apache/httpd/httpd-2.2.13.tar.bz2;
      md5 = "8d8d904e7342125825ec70f03c5745ef";
    };
    buildInputs = [perl apr aprutil pcre openssl];
    configureFlags = "--enable-mods-shared=all ...";
  };

  apr = stdenv.mkDerivation {
    name = "apr-1.3.8";
    ...
  };

  stdenv.mkDerivation = args: derivation {
    ...
    builder = ...
    ,,
    PATH=${gcc}/bin:${coreutils}/bin:...
    tar xf ${args.src}
    ./configure --prefix=$out ${args.configureFlags}
    make
    make install
    ,,
  };
  ...
}
```

Figure 3: Nix expression to build Apache

and storage.

We can build Apache from the command line as follows:

```
$ nix-build pkgs.nix -A httpd
```

(where `pkgs.nix` denotes the expression in Figure 3). Nix will recursively begin to build the dependencies of Apache, such as `perl`, `apr` and `gcc`. This is a *source deployment model*, but as an optimisation, Nix will automatically download pre-built store paths from repositories on the Internet if they are available. The result of building Apache on the Nix store is seen in Figure 4. Nix automatically tracks runtime dependencies between packages by scanning for store path hashes. For instance, the `httpd` binary will have a reference to `libapr-1.so.0.3.8` compiled in at build time. Thus, the *closure* of a package under the references relation gives us all the store paths that must be copied to another machine to successfully run it on that machine. This approach prevents missing dependencies, a common problem with other package managers.

As a package manager, Nix has several advantages over conventional tools such as RPM [9]. The fact that packages never overwrite each other means that there is no “DLL hell”: the installation, upgrade or deinstallation of one package can never break others. It is trivial to roll back to older versions. Different users can have different “views” of the set of installed applications. By copying the closure of a package, we can reliably deploy it to another machine.

There is a large distribution of Nix expressions, the *Nix Packages collection*, that contains almost 2000 packages, and supports a variety of operating systems.

## NixOS.

Nix has been used to build a Linux distribution, NixOS [5]. NixOS uses Nix to build the entire system from a specifi-

```

/nix/store
├── snws5xld6iyx...-apache-httpd-2.2.13
│   ├── bin
│   │   ├── httpd
│   │   └── apachectl
│   └── ...
├── rl384gzsay47...-apr-1.3.8
│   ├── lib
│   │   └── libapr-1.so.0.3.8
│   └── ...
├── nqapqr5cyk4k...-glibc-2.9
│   ├── lib
│   │   ├── ld-linux.so.2
│   │   └── libc.so.6
│   └── ...
└── ...

```

Figure 4: Partial closure of Apache in the Nix store

cation in the Nix expression language – not just software packages. *All* static parts of the system – packages, the kernel, boot scripts, scripts to manage system services, configuration data, and so on – are built by Nix derivations. In fact, there is a single top-level derivation, that, when built, causes all static parts of the system to be built as dependencies. The advantage of such a purely functional approach to system configuration management is that upgrading the system is safe (since the old configuration in the Nix store is not overwritten) and reliable (since due to purity it does not rely on the previous state of the system), we can always roll back to previous configurations, and we can deterministically rebuild a configuration.

Of course, it’s not enough to merely *build* a system. The most important thing that the top-level derivation builds is the *activation script*, which takes care of the “imperative” aspects of switching to a new configuration, such as stopping and starting system services. The activation script also updates the boot loader of the operating system to ensure that when the system is restarted, the new configuration is booted. Thus, if we make a change to the Nix expressions that constitute NixOS, we effectuate the change by building the top-level derivation and running the activation script:

```

$ nix-build /etc/nixos/nixos \
  -A config.build.system.toplevel
$ ./result/bin/switch-to-configuration

```

(Users actually run a high-level command that wraps these actions.) The value `config.build.system.toplevel` evaluates to the top-level derivation, which builds the activation script and the rest of the system through its dependencies.

The Nix expressions that constitute NixOS are organised into *modules*. The modules together define a nested attribute set, the system configuration `config`. Each module contributes values to this set and can use values defined by other modules. The basic structure of a NixOS module is:

```

{ config, pkgs, ... }:

{ ... configuration values ... }

```

Thus, a module is a function that accepts at least two arguments: `config`, which contains the full system configuration, and `pkgs`, which contains the Nix Packages collection for convenience. For instance, the value `pkgs.httpd` is the derivation that builds the Apache web server. The system configuration `config` is computed by calling every NixOS module and

merging the attribute sets of configuration values returned by each. The result of the merge is passed back as the `config` function argument to each module. Thus the input to NixOS modules depends on their output. This kind of circularity works thanks to the laziness of the language.

For instance, the value of the `webserver` variable in Figure 1 is a NixOS module. It defines a number of attributes, such as `services.httpd.enable` and `services.httpd.extraSubservices`. These are used by another module – the Apache webserver module – to determine whether to generate an “Upstart” job file to start Apache, and to include another module that contributes configuration values to build the Trac web service. Most configuration values are system options relevant to end users, but others are “computed” values, such as the derivation `build.system.toplevel`, that are defined using other configuration values.

NixOS currently consists of around 125 modules, each implementing some part of the system (e.g. building the boot scripts, the Apache configuration, or the X11 GUI environment). These are added automatically to the end-user configuration module (such as the `webserver` configuration in Figure 1).

### 3. SPECIFYING DISTRIBUTED SYSTEMS, DECLARATIVELY

In NixOS, we have a declarative specification of how to build an entire operating system from a set of Nix expressions. We can now extend this to networks of machines. At the basic level this is quite easy: a network is simply an attribute set of NixOS configuration modules that specify the desired functionality for each machine. For instance, the Trac network in Figure 1 contains an attribute set `network` that specifies a network of three machines: the contents of the `storage`, `postgresql` and `webserver` attributes are each NixOS configuration modules. The attribute names are symbolic names for the machines in the network. The attribute set `network` in itself does not do anything – it just specifies a set of machines. In the next sections we will see that by applying different functions and tools to `network`, we can deploy it to existing machines, instantiate virtual machines for testing, and so on.

There is a slight complication: the configurations of machines should be able to refer to the configurations of other machines. For instance, the database server needs to know the IP address of the webserver to generate PostgreSQL’s access control list, which specifies the machines that are allowed to connect to the database (in `services.postgresql.authentication`). Likewise, the web server needs to specify the hostname of the PostgreSQL server to be able to connect to it. We do not want to hard-code such information in the network specification, since it then becomes harder to deploy a test environment or build a virtual network. Instead, as we shall see, we either specify such information separately (for real deployment) or generate it automatically (for VM generation).

Therefore, we extend the interface of NixOS modules to accept the system configurations of other machines in the network. For instance, the configuration modules for the `storage`, `postgresql` and `webserver` machines in Figure 1 are called with arguments `storage`, `postgresql`, `webserver`, containing the the computed (merged) configurations for those machines, respectively. (Due to the ellipses in the list of func-

tion arguments, modules can ignore arguments that they are not interested in; e.g., the NFS server does not need to know anything about the PostgreSQL database.) For instance, in the call to `storage`, `webserver.services.httpd.enable` will evaluate to `true`, while `webserver.networking.hostName` will contain the host name of the webserver. The latter is not explicitly specified by the configuration module `webserver`; it is defined in another module and depends on the kind of deployment that we are doing.

Note that machines can be mutually recursive in their configurations. This is important: for instance, as in the case of the Trac network, a webserver may need the hostname of a database server, while the database server may need to add the IP address of the webserver to its access control list. Again, due to laziness, this kind of circularity works fine.

Many extensions to network specifications are imaginable. For instance, we can allow the model to express that a configuration is to be instantiated a variable number of times. For instance, a load-balancing web application network may consist of a front-end *reverse proxy*, and a number of (almost) identical back-end webserver. The proxy configuration module will then receive an argument `webserver` that contains a list of all the machines in that class. This allows us to map over this list to generate Apache's proxy configuration:

```
services.httpd.extraConfig = ''
  <Proxy balancer://cluster>
    Allow from all
    ${concatMapStrings (machine:
      "BalancerMember http://${
        machine.networking.hostName}" ) }
  </Proxy>
'';
```

Another extension is to be able to specify an abstract topology, e.g., that the back-end webserver and proxy must be on their own LAN with private (unroutable) IP addresses, with the proxy also connected (through a separate network interface) to the Internet.

## 4. DEPLOYMENT

The first application of declarative specifications of networks is to *deploy* them to real machines. This requires that the target machines run NixOS and an SSH daemon to support automated remote logins. Deployment in our approach has a centralised model: a specific machine (the distribution machine) builds the configurations for each machine and copies them to the target machines.

Because network models such as that in Figure 1 abstract over the concrete identity and location of machines in the network, it is necessary to supply a second model, the *infrastructure model*, that tells the deployment tool to what actual machines the configurations are to be deployed. (In trivial cases where the hostnames of the targets match the attribute names in the network model, the infrastructure model can be omitted.) Moreover, the infrastructure model also specifies the architecture of each machine (e.g. `i686-linux` or `x86_64-linux` for 32-bit or 64-bit Linux on Intel-based systems).

Figure 5 shows an example of an infrastructure model for the Trac network. The structure is very similar to a network expression and contains additional properties for every machine in the network: the hostname and the `system` attribute, which describes the architecture of the target ma-

```
{
  storage = {
    hostName = "storage.example.org";
    system = "x86_64-linux";
  };
  postgresql = {
    hostName = "db.example.org";
    system = "x86_64-linux";
  };
  webserver = {
    hostName = "www.example.org";
    system = "i686-linux";
  };
}
```

Figure 5: Infrastructure model

chine. Given the network and infrastructure models, the user can build and deploy the specification with a single command:

```
$ nixos-deploy-network -n network.nix \
  -i infrastructure.nix
```

This will cause all the packages and other derivations needed by the configurations of the three machines in the network to be built or downloaded to the user's local machine. Next, the configurations are copied from the local Nix store to the Nix stores of the target machines, and the NixOS activation script is run to switch over to the new configuration on each machine. When repeating this process, e.g. in order to upgrade a package, only those parts of the derivation dependency graph that differ are rebuilt and redeployed.

### Implementation.

Deployment works as follows, given `network` and `infrastructure` attribute sets. First, for each machine in `network`, we compute the full system configuration by combining the machine's specification with the standard NixOS modules, plus a module that contains the corresponding information from the infrastructure model, e.g.

```
{ config, pkgs, ... }:

{ networking.hostName = "storage.example.org"; }
```

for the `storage` machine. The result of the evaluation of the full system configuration for each machine is passed as additional function arguments back into the function that specifies the machine's configuration. E.g., the result of the evaluation of the configuration for `webserver` is passed as an argument to `postgresql`, and vice versa. Again, this works due to the laziness of the language. The `system` property is used to build packages on the right architecture. (Nix supports transparent multi-platform building; if a derivation must be built for another architecture than the user's local machine, it can automatically perform the derivation on another machine of the right architecture, if properly configured.)

From each machine's full configuration, we then evaluate the `system.build.toplevel` attribute. This builds the complete operating system needed for that machine on the local machine.

After building the configurations, we copy the closure of each built configuration from the local Nix store to the Nix



store on the corresponding target machine. This is an imperative action, so it cannot be done in a Nix expression. Nix provides a command, `nix-copy-closure`, which copies the system configuration and all its dependencies in the Nix store to target machines in the network. This process is efficient, because it only transfers the Nix store paths in the closure that are missing in the Nix store on the target machines.

Finally, we run each configuration's activation script, i.e. the path `${system.build.toplevel}/bin/switch-to-configuration`, on the corresponding target machine (via `ssh`). If desired, we also update certain configuration files (such as the Grub boot loader) to ensure that this configuration starts when the machine is restarted.

## Evaluation.

We have used `nixos-deploy-network` in production use to upgrade the *Hydra build farm*, a continuous integration system based on Nix (<http://hydra.nixos.org/>). This network consists of nine NixOS machines: a front-end webserver running Apache, Subversion, PostgreSQL, a Jetty servlet container running the JIRA issue tracking system; the Hydra build scheduler and back-end webserver; two 32-bit build machines; and five 64-bit build machines (from different manufacturers). We wrote network and infrastructure models to deploy this network automatically<sup>1</sup>. Evaluating the system configurations for the complete network took 45.1 seconds on an 8-core Intel Xeon E5430 machine with 16 GiB of RAM.

We built the network configuration on a distribution machine with an initially empty Nix store. (This is a worst-case assumption: usually, the user's Nix store already contains many packages that the network configuration needs.) It then took 541 seconds to build or download the 1368 derivations in the build graph; 323 derivations were downloaded from the Nixpkgs distribution server (382 MiB), the others were built from source. The total size of the closures to be copied to the target machines was 7.6 GiB (without compression). Reconfiguration is of course much cheaper. For example, when we changed the model to use Linux 2.6.28 instead of 2.6.29, it took 23.8 seconds to build the new configuration (because the new kernel is available on the Nixpkgs distribution server), and 1.2 GiB of data had to be copied to the targets (again without compression).

## Testing the network.

If we want to *test* changes to the network specification before deploying them to the production environment, we can do that by deploying the network first to a test environment. That is, we simply specify a different infrastructure model that lists the test machines instead of the production machines. However, this requires us to have a bunch of spare machines available for the test environment. In the next section, we will see that it is much more convenient to instantiate a virtual network of virtual machines from the specification.

## 5. TESTING USING VIRTUAL MACHINES

The next major application of declarative specifications of networks is to build virtual networks of virtual machines automatically. This can be used to interactively test a clone of

<sup>1</sup>The Nix expressions can be found at <https://svn.nixos.org/repos/nix/configurations/trunk/tud>.

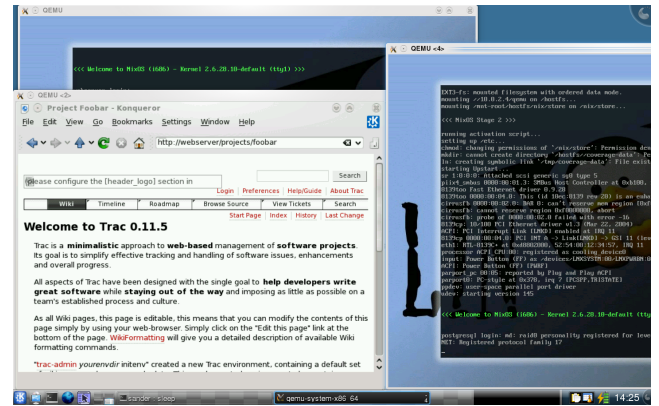


Figure 6: Interactive testing of a virtual Trac network

a production network on the local machine of a developer or system administrator, or to run automated regression tests.

The basic idea is that there is a function `buildVirtualNetwork` that we can apply to a *network*, which returns a derivation to builds a shell script that starts VMs running the specified configurations. That is, if we extend the Trac Nix expression in Figure 1 with

```
vms = buildVirtualNetwork { inherit network; };
```

then we can perform the following commands:

```
$ nix-build trac.nix -A vms
$ ./result/bin/run-vms
```

The latter command then starts a number of virtual machines on the user's desktop, each booting a NixOS instance with the specified functionality. The VMs are connected together in a virtual network (with IP addresses in the private space `192.168.1.n`). Certain TCP ports on the host machine are connected to ports on each VM. For instance, port 8082 is connected to the Trac webserver. Figure 6 shows a screenshot of the running VMs, along with a Firefox browser on the host that accesses the webserver.

## Building and running virtual networks.

Virtual machines are built by a NixOS module `qemu-vm.nix` that defines a configuration value `system.build.vm`, which is a derivation to build a shell script that starts the NixOS system built by `system.build.toplevel` in a virtual machine. We use KVM (<http://www.linux-kvm.org/>), a modified version of the open source QEMU processor emulator that uses the hardware virtualisation features of modern CPUs to run VMs at near-native speed. An important feature of KVM over most other VM implementations is that it can easily be started and controlled from the command line. This includes the fully automated starting, running and stopping of a VM in a derivation. Furthermore, QEMU provides special support for booting Linux-based operating systems: it can directly boot from a kernel and initial ramdisk image on the host filesystem, rather than requiring a full hard disk image with that kernel installed. (The initial ramdisk in Linux is a small filesystem image responsible for mounting the real root filesystem.) For instance, the `system.build.vm` derivation generates essentially this script:

```

${pkgs.qemu_kvm}/bin/qemu-system-x86_64 -smb / \
-kernel ${config.boot.kernelPackages.kernel} \
-initrd ${config.system.build.initialRamdisk} \
-append "init=${config.system.build.bootStage2}
systemConfig=${config.system.build.toplevel}"

```

QEMU provides virtualised network connectivity to VMs. We configure each VM with two network interfaces. The first, `eth0`, allows the VMs to talk to the host and to the Internet (if desired). The guest has IP address 10.0.2.15, QEMU's virtual gateway to the host is 10.0.2.2, and there is an implementation of Windows' file sharing protocol SMB/-CIFS on 10.0.2.4 that provides access to the host filesystem (the `-smb /` option above). QEMU provides Network Address Translation (NAT) on outgoing packets to allow the guest to access the host's network (including the Internet). This feature is implemented entirely in user space: it requires no root privileges.

The second interface, `eth1`, allows the VMs to talk to each other. `buildVirtualNetworks` assigns each machine a private 192.168.1.*n* address in sequential order. QEMU propagates any packet sent on this interface to all other VMs in the same virtual network. The machines are assigned hostnames equal to the corresponding attribute name in the model, so the hostname of the machine built from the `postgresql` configuration will be `postgresql`. Thus, for testing, the user does not need to specify an infrastructure model.

The `system.build.vm` derivation does not build a virtual hard disk image for the VM. Rather, the initial ramdisk of the VM mounts the Nix store of the host through SMB/-CIFS. This is a crucial feature: the closure of a system is hundreds of megabytes in size at the least, so to build such an image every time we reconfigure the VMs would be very wasteful in time and space. Thus, building VMs for the Trac example takes almost exactly as long as building it for deployment to actual machines.

The VM start script does create an empty `ext3` root filesystem for each guest at startup, to hold mutable state such as the contents of `/var` or the system account file `/etc/passwd`. Thanks to sparse allocation of blocks in the virtual disk image, image creation takes only a few seconds. NixOS' activation script is self-initialising, so at boot time it initialises all state needed to run the system. For interactive use, the filesystem is preserved across restarts of the VM, saved in the image file `./hostname.qcow2`.

### Running automated tests.

An important application of our work is to automatically build and run a virtual network, and then execute a test suite on the virtual machines. NixOS system configurations allow complex environments, such as multiple machines with different services (including GUIs), to be expressed concisely, and built using a single command. It is also worth noting that in our approach, no root privileges are needed to either build or run the virtual network. This makes it feasible to include such tests in the standard test suite of a software package, or to perform it in a continuous build system.

Figure 7 shows a (small) automated test for Quake 3 Arena, a multi-player 3D computer game. It specifies a network of two machines: `server`, which automatically starts a Quake server daemon, and `client`, which runs an X11 graphical user interface, but otherwise does nothing. The value `test` evaluates to a derivation that executes the VMs in a virtual network and runs the given test suite, specified as a

```

network =
{ server =
  { config, pkgs, ... }:
  { jobs = pkgs.lib.singleton
    { name = "quake3-server";
      startOn = "startup";
      exec =
        "${pkgs.quake3demo}/bin/quake3"
        + " +set dedicated 1 +set g_gametype 0"
        + " +map q3dm7 +addbot grunt 2> /tmp/log";
    };
  };

  client =
  { config, pkgs, ... }:
  { services.xserver.enable = true;
    environment.systemPackages = [ pkgs.quake3demo ];
  };
};

vms = buildVirtualNetwork { inherit nodes; };

test = runTests vms
'',
  startAll;
  $server→waitForJob("quake3-server");
  $client→waitForJob("xserver");
  $client→execute(
    "quake3 +set name Foo +connect server &");
  sleep 40;
  $server→mustSucceed(
    "grep -q 'Foo.*entered the game' /tmp/log");
  $client→screenshot("${ENV{out}}/screen.png");
'',

```

Figure 7: Specification of a Quake client/server regression test

Perl script. The test script calls a simple test driver that can perform various actions such as: execute a command on a machine; make a machine unreachable to simulate a network outage; stop, start and crash machines; wait until a machine is listening on a port; and so on.

The test script in the example first starts both machines and waits until they are ready. It then executes a command on the client to start a graphical Quake client and connect to the server. The client then does nothing (except possibly getting blown up by the bots spawned by the server). After a while, we verify on the server that the client did indeed connect. The derivation will fail to build if this is not the case. Finally, we make a screenshot of the client to allow visual inspection of the end state, if desired.

(GUI testing is a notoriously difficult subject [11]. The point here is not to make a contribution to GUI testing techniques per se, but to show that we can easily set up the infrastructure needed for such tests. In the test script, we can run any automated GUI testing tool we want.)

The test driver executes commands on a VM by connecting to TCP port 514 on the guest, on which a root shell is listening. (The remotely accessible root shell is provided by a NixOS module added to the machine configuration by `buildVirtualNetwork`. It doesn't exist in normal use.) We patched QEMU to allow TCP ports on the guest to be connected to *Unix domain sockets* [17] in the host filesystem rather than TCP ports on the host. This is important for security: we do not want anybody other than the test driver

connecting to the port. The use of a Unix domain socket rather than a port also means that any number of `runTests` derivations can execute in parallel, without fear of clashing port assignments. These features are important for continuous build environments, where any number of builds may execute concurrently.

### *Distributed coverage analysis.*

A declarative specification of a network and an associated test suite makes it easy to perform a *distributed code coverage analysis*. Again, we make no contributions to the technique of coverage analysis per se; we improve its deployability. First, the abstraction facilities of the Nix expression language make it easy to specify that parts of the dependency graph of a large system are to be compiled with coverage instrumentation (or any other form of build-time instrumentation one might want to apply). Second, by collecting coverage data from every machine in a test run of a virtual network, we get more complete coverage information. For instance, if we add a client machine to the Trac network and run a test that performs a Subversion checkout from the server, different paths in the Subversion code will be exercised on the client than on the server.

We can add coverage instrumentation to a package by setting the configuration value `nixpkgs.config.packageOverrides`. This value is a function that takes the original contents of the Nix Packages collection as an argument, and returns a set of replacement packages:

```
nixpkgs.config.packageOverrides = pkgs: {
  subversion = pkgs.subversion.override {
    stdenv = pkgs.addCoverageInstrumentation pkgs.stdenv;
  };
};
```

The original Subversion package, `pkgs.subversion`, contains a function, `override`, that allows the original dependencies of the package to be overridden. In this case, we pass a modified version of the standard build environment (`stdenv`) that automatically adds the flag `--coverage` to every invocation of the GNU C Compiler. This causes GCC to instrument object code to collect coverage data and write it to disk. Most C or C++-based packages can be instrumented in this way, including the Linux kernel.

The function `runTests` automatically collects the coverage data from each machine in the virtual network at the conclusion of the test script, and writes it to `$out`. Another function, `makeReport`, then combines the coverage data from each virtual machine and uses the `lcov` tool [13] to make a set of HTML pages showing a coverage report and each source file decorated with the line coverage. For example, we have run the Trac test script with coverage instrumentation on Apache, Subversion, Apr, Apr-util and the Linux kernel. Figure 8 shows a small part of the distributed coverage analysis report resulting from a Trac test suite run. The line and function coverage statistics combine the coverage from each of the four machines in the network.

One application of distributed coverage analysis is to determine code coverage of large systems, such as entire Linux distributions, on system-level tests (rather than unit tests at the level of individual packages). This is useful for system integrators, such as Linux distributors, as it reveals the extent to which test suites exercise system features. For instance, the full version of the coverage report in Figure 8 readily shows which kernel and Apache modules are exe-

cuted by the tests, often at a very specific level: e.g., the `ext2` filesystem does not get executed at all, while the `ext3` filesystem *is* used, except for its extended attributes feature.

### *Continuous builds.*

The ability to build and execute a test with complex dependencies is very valuable for continuous integration. A continuous integration tool (e.g. CruiseControl) continuously checks out the latest source code of a project, builds it, runs associated tests, and produces a report [10]. A problem with the management of such tools is to ensure that all the dependencies of the build and the test are available on the continuous build system (e.g., a database server to test a web application). In the worst case, the administrator of the continuous build machines must install such dependencies manually. By contrast, the single command

```
$ nix-build trac.nix -A report
```

causes Nix to build or download everything needed to produce the Trac coverage report: the Linux kernel, QEMU, the C compiler, the C library, Apache, PostgreSQL, the coverage analysis tools, and so on. In total this is 954 derivations, the vast majority of which can be reused between builds.

This automation makes it easy to stick such tests in a continuous build system. In fact, there is a Nix-based continuous build system, *Hydra*, that continuously checks out Nix expressions describing build tasks from a revision control systems, builds them, and makes the output available through a web interface. Developers do not have to do anything on the Hydra build farm to set up the environment for a test, as each test completely defines how to build its own dependencies. The outputs of the Trac and Quake tests in Hydra can be found at <http://hydra.nixos.org/job/nixos/trunk/tests.{trac,quake3}/latest>, respectively. The Nix expressions for these and other tests are at <https://svn.nixos.org/repos/nix/nixos/trunk/tests>.

## 6. DISCUSSION

### *Generality.*

The network specifications described in this paper build upon NixOS: they build NixOS operating system instances. This obviously limits the generality of our current implementation, in particular regarding the deployment in heterogeneous networks. In this sense, it shows an “ideal” situation, in which entire networks of machines can be built from a purely functional specification. It is certainly possible to back away from the ideal and build only *parts* of systems. For instance, to support deploying the Trac example to a Red Hat Linux or Windows Server target machine, we could build and deploy only the Apache or PostgreSQL services, and leave the native operating system untouched. (The Nix package manager itself is portable across a variety of operating systems.)

It is worth noting that the generation of virtual machines works on any Linux host machine (and probably other operating systems supported by QEMU). Thus, the interactive or automated tests in Section 5 can very well be run on (say) an Ubuntu Linux system. This is particularly important for automated regression test suites, where in many test cases we likely do not care particularly about the specific brand of guest Linux distribution.







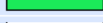

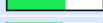


<a href="#">httpd-2.2.13/os/unix</a>		36.6 %	64 / 175	75.0 %	12 / 16
<a href="#">httpd-2.2.13/server</a>		48.0 %	3601 / 7508	60.1 %	351 / 584
<a href="#">httpd-2.2.13/server/mpm/prefork</a>		47.1 %	220 / 467	60.9 %	14 / 23
<a href="#">linux-2.6.28.10/arch/x86/include/asm</a>		49.7 %	446 / 897	6.2 %	2 / 32
<a href="#">linux-2.6.28.10/arch/x86/include/asm/mach-default</a>		100.0 %	5 / 5	-	0 / 0
<a href="#">linux-2.6.28.10/arch/x86/include/asm/xen</a>		0.0 %	0 / 80	-	0 / 0
<a href="#">linux-2.6.28.10/arch/x86/lib</a>		62.3 %	119 / 191	62.8 %	27 / 43
<a href="#">linux-2.6.28.10/arch/x86/mach-default</a>		59.4 %	19 / 32	87.5 %	7 / 8
<a href="#">linux-2.6.28.10/arch/x86/mm</a>		42.5 %	852 / 2006	51.3 %	80 / 156

Figure 8: Part of the distributed code coverage analysis report for the Trac network

### Declarative model.

To what extent do we need the properties of Nix and NixOS, in particular the fact that an entire deployable operating system environment is built from source from a specification in a single formalism, and the purely functional nature of the Nix store? There are many tools to automate deployment of machines. For instance, Red Hat’s Kickstart tool installs RPM-based Linux systems from a textual specification and can be used to create virtual machines automatically, with a single command-line invocation [14].

However, there are many limitations to such tools. 1) Having a single formalism that describes the construction of an entire network from sources makes hard things easy, such as building part of the system with coverage analysis. In a tool such as Kickstart, the binary software packages are a given; we cannot modify the build processes of those packages. 2) Systems such as Kickstart do not offer the same reliability guarantees for upgrades. Unless the previous root filesystem is wiped, the old configuration might interfere with the intended new configuration. Thus, we might find that a deployment that succeeded in a test environment fails in the production environment, because of differences in the prior state of the systems. 3) For testing in virtual machines, it is important that VMs can be built efficiently. With Nix, this is efficient because the VM can use the Nix store of the host. With other package managers, that’s not an option because the host filesystem may not contain the (versions of) packages that a VM needs. One would also need to be root to install packages in the host filesystem; this makes any such approach undesirable for automated test suites in a software package. 4) For automatic testing or deployment, one needs a formalism to describe the desired configurations. In NixOS this is already given: it is what users use to describe regular system configurations. In conventional Unix systems, the configuration is a result of many “unmanaged” modifications to system configuration files (e.g. in /etc). Thus, given an existing Unix system, it is hard to distill the “logical” configuration of a system (i.e., specification in terms of high-level requirements) from the multitude of configuration files.

### Granularity.

The network specifications in this paper have a machine-level granularity: they declare a set of logical machines, each of which are mapped onto either a real machine or a virtual machine. We are working on an more fine-grained approach, *Disnix* [19], where the model specifies smaller components, such as web services. These are then mapped onto concrete machines using quality-of-service attributes specified in the component and infrastructure models.

### Atomicity.

A desirably property of a deployment system is to have *atomic upgrades*. That is, during the upgrade, there should be no point in time during which the system is in an inconsistent state. Nix has this property, contrary to most package management systems, as package are never overwritten. NixOS system upgrades are almost atomic: building or copying a configuration is atomic, but the activation step is not. This is a problem for single machines, but even more so for distributed systems. For instance, during an upgrade of the Trac network, a new version of the webserver may find itself talking to an old version of the database, or vice versa.

In [19] we sketched a solution to this problem. At the start of the activation step, we should *block* new requests to services (such as HTTP or SQL requests) and wait for outstanding requests to finish. Then, we run the activation scripts on all machines in the network to realise the new configuration. Finally, we unblock services and allow waiting requests to proceed. In this way, no concurrent requests will ever observe an inconsistent configuration state – they will either go to the old or the new configurations of the servers. Unfortunately, blocking requests cannot be done generically: it requires protocol-specific proxies to wrap services (e.g., for HTTP or PostgreSQL).

### Mutable state.

We do not address the problem of mutable state (such as the contents of databases) on machines in the network. It may be that upgrades require state to be upgraded as well, e.g., a database upgrade or web application schema change might require a dump/load cycle. These can be accommodated in NixOS through the activation script. However, more research into a structural treatment of mutable state is needed.

## 7. RELATED WORK

Most work on deployment of distributed systems takes place in the context of system administration research. Cf-engine [2] maintains systems on the basis of a declarative specification of actions to be performed on each (class of) machine. Stork [3] is a package management system used to deploy virtual machines in the PlanetLab testbed. These and most other deployment tools have *convergent* models [18], meaning that due to statefulness, the actual configuration of a system after an upgrade may not match the intended configuration. By contrast, NixOS’ purely functional model ensures *congruent* behaviour: apart from mutable state, the system configuration always matches the specification.

Virtualisation does not make deployment easier; apart from simpler hardware management, it makes it harder,

since without proper deployment tools, it simply leads to more machines to be managed [15].

The `buildVirtualNetwork` function in Section 5 currently assumes a simple network topology, where all machines are on the same virtual network. MLN [1], a tool for managing large networks of VMs, has a declarative language to specify arbitrary topologies. It does not manage the contents of VMs beyond a templating mechanism.

Our VM testing approach currently is only appropriate for relatively small virtual networks. This is usually sufficient for regression testing of typical bugs, since they can generally be reproduced in a small configuration. It is not appropriate for scalability testing or network experiments involving thousands of nodes, since all VMs are executed in the same derivation and therefore on the same host. However, depending on the level of virtualisation required for a test, it is possible to use virtualisation techniques that scale to hundreds of nodes on a single machine [12].

There is a growing body of research on testing of distributed systems; see [16, Section 5.4] for an overview. However, the deployment and management of test environments appears a somewhat neglected issue. An exception is Weevil [20], a tool for the deployment and execution of experiments in testbeds such as PlanetLab. We are not aware of tools to support the synthesis of VMs in automatic regression tests as part of the build processes of software packages.

## 8. CONCLUSION

In this paper, we have shown an extension of our work on purely functional software deployment to distributed systems. The great advantage of this approach is that a single specification can support a number of deployment scenarios: deployment to a production environment; deployment to a test environment; instantiating a virtual network of virtual machines on a user's machine; and using the virtual network to run automatic tests and analyses. The latter application is particularly important, as it allows developers to write integration tests for their software that would otherwise require a great deal of manual configuration, and would likely not be done at all.

## 9. REFERENCES

- [1] K. Begnum. Managing large networks of virtual machines. In *LISA'06: Proc. of the 20th Conference on Large Installation System Administration Conference*, pages 205–214, Berkeley, CA, USA, 2006. USENIX.
- [2] M. Burgess. Cfengine: a site configuration engine. *Computing Systems*, 8(3), 1995.
- [3] J. Cappos, S. Baker, J. Plichta, D. Nyugen, J. Hardies, M. Borgard, J. Johnston, and J. H. Hartman. Stork: package management for distributed VM environments. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–16, Berkeley, CA, USA, 2007. USENIX.
- [4] E. Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Faculty of Science, Utrecht University, The Netherlands, 2006.
- [5] E. Dolstra and A. Löb. NixOS: A purely functional Linux distribution. In *ICFP 2008: 13th ACM SIGPLAN Intl. Conf. on Functional Programming*. ACM, Sept. 2008.
- [6] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proc. 26th Intl. Conf. on Software Engineering (ICSE 2004)*, pages 583–592. IEEE Computer Society, May 2004.
- [7] Edgewall Software. Trac – integrated SCM & project management. <http://trac.edgewall.org/>, 2009.
- [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 & Sons, 2003.
- [10] M. Fowler and M. Foemmel. Continuous integration. <http://www.martinfowler.com/articles/continuousIntegration.html>. Accessed 11 August 2005.
- [11] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *ICSE '09: 31st Intl. Conf. on Software Engineering*, pages 408–418, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [12] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the Emulab network testbed. In *2008 USENIX Annual Technical Conference*, pages 113–128, Berkeley, CA, USA, 2008. USENIX.
- [13] P. Larson, N. Hinds, R. Ravindran, and H. Franke. Improving the Linux Test Project with kernel code coverage analysis. In *Proceedings of the 2003 Ottawa Linux Symposium*, July 2003.
- [14] Red Hat, Inc. *Red Hat Enterprise Linux 5 Virtualization Guide*. Red Hat, Inc., fourth edition, 2009.
- [15] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: Using semantic information to combat virtual machine image sprawl. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 111–120. ACM, 2008.
- [16] M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. *IEEE Transactions on Software Engineering*, 34(4):452–470, 2008.
- [17] W. R. Stevens and S. A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley, second edition, June 2005.
- [18] S. Traugott and L. Brown. Why order matters: Turing equivalence in automated systems administration. In *Proceedings of the 16th Systems Administration Conference (LISA '02)*, pages 99–120. USENIX, Nov. 2002.
- [19] S. van der Burg, E. Dolstra, and M. de Jonge. Atomic upgrading of distributed systems. In T. Dumitras, D. Dig, and I. Neamtiu, editors, *First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp)*, pages 1–5. ACM, Oct. 2008.
- [20] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Automating experimentation on distributed testbeds. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 164–173. ACM, Nov. 2005.