# Charon: Declarative Provisioning and Deployment

Eelco Dolstra, Rob Vermaas, and Shea Levy
LogicBlox, Inc., Atlanta, GA, USA, {eelco.dolstra, rob.vermaas, shea.levy}@logicblox.com

***Abstract*—We introduce Charon, a tool for automated provisioning and deployment of networks of machines from declarative specifications. Building upon NixOS, a Linux distribution with a purely functional configuration management model, Charon specifications completely describe the desired configuration of sets of "logical" machines, including all software packages and services that need to be present on those machines, as well as their desired "physical" characteristics. Given such specifications, Charon will provision cloud resources (such as Amazon EC2 instances) as required, build and deploy packages, and activate services. We argue why declarativity and integrated provisioning and configuration management are important properties, and describe our experience with Charon.**

## I. Introduction

An essential part of the software life cycle is to deploy software to its target environment. In a cloud-based, infrastructure-as-a-service setting, this also involves provisioning the cloud resources that constitute the target environment. These actions should be automated: given a specification of the desired system configuration, a deployment tool should automatically create the required cloud resources and deploy the necessary software packages and configuration files.

In this paper we present Charon[1], a tool for automated provisioning and deployment of machines. Charon has several important characteristics:

- It is *declarative*: it provisions and deploys sets of machines from specifications that state the desired configuration of each machine. Charon then figures out the actions necessary to realise that configuration. Thus (with the exception of mutable state such as database contents) there is no difference between doing a new deployment or doing an upgrade of an existing deployment: the resulting machine configurations will be the same, allowing deployments to be upgraded or reproduced reliably. This is in contrast to cloud deployment tools with *imperative* configuration models, e.g. Juju [1], where users configure a deployment by executing a sequence of stateful commands (e.g. `juju deploy mysql`). In such systems, the configuration is the result of a set of (possibly untracked) actions by administrators, making it hard to reproduce a configuration. (See [2] for a discussion of the perils of statefulness.)
  Charon builds upon NixOS [2], a Linux distribution that is in turn based on Nix [3], a package manager that builds and stores packages in a *purely functional* manner. Concretely this means that multiple versions of packages can coexist on a system, that packages can be upgraded or rolled back atomically, that dependency specifications can be guaranteed to be complete, and so on. NixOS has a declarative, stateless approach to describing the desired configuration of a machine, making it an ideal basis for automated configuration management of sets of machines. It also has desirable properties such as (nearly) atomic upgrades and the ability to roll back to previous configurations.
  There are several prominent configuration management systems with declarative models, such as Cfengine [4], Puppet [5] and Chef [6]. However, the systems they manage still have underlying imperative configuration models, such as configuration files in `/etc` that are updated in place by deployment actions. Thus the result of a deployment may still depend on the previous configuration of the system.
- It performs provisioning. The integration of provisioning and configuration management is important, because provisioning affects configuration files: for instance, if we instantiate an Amazon EC2 machine as part of a larger deployment, it may be necessary to put the IP address or host name of that machine in a configuration file on another machine, and to ensure that any changes are propagated properly. Charon takes care of this automatically.
- It allows abstracting over the target environment: the same specification can be deployed to different cloud backends (e.g. VirtualBox for testing and EC2 or OpenStack for production). The configuration language is modular, so backend-specific parameters (say, the EC2 instance type) can be separated from the "logical" aspects of the deployment. The specifications also allow abstracting over network connectivity: Charon ensures that machines can talk to each other, e.g. by creating tunnels between machines in different EC2 regions.
  By contrast, Vagrant [7] provisions VirtualBox virtual machines to set up test environments which can then be configured by tools such as Chef; to deploy to (say) EC2, other tools are required. With Charon, the same toolchain supports both development and production use.
- It uses a single configuration formalism (Nix's purely functional language) for package management and system configuration management. This makes it very easy to add *ad hoc* packages to a deployment. It also allows powerful abstractions to be expressed. For instance, common boilerplate code between machine definitions can be abstracted away, and functions can be defined to generate part or all of a network specification from higher-level parameters. However, the functional approach is less suited to automatically finding optimal solutions to sets of constraints (e.g. to find a deployment that satisfies a feature model [8]), as done

[1]Available under the LGPLv3 license at https://github.com/NixOS/charon/.

by ConfSolve [9].

In the remainder of this paper, we give an overview of Charon, and discuss our experiences with it.

## II. Overview

### A. NixOS

Charon deploys NixOS machines, so we start with a brief overview of NixOS' configuration model. In NixOS, machines are configured by providing a file (typically /etc/nixos/configuration.nix) that specifies the desired configuration of the system. For instance, the following file specifies that we want a machine that runs the Apache web server:

```
{ services.httpd.enable = true;
  services.httpd.documentRoot = "/data";
}
```

Configuration changes are realised by running the command nixos-rebuild, which evaluates the system configuration, builds all dependencies, and finally starts, restarts or stops any new, changed or removed system services in the new configuration. For instance, if the previous configuration had services.httpd.enable = false, then running nixos-rebuild will cause Apache httpd to be built or downloaded (if it wasn't already present in the system), an httpd.conf configuration file to be generated, and finally httpd to be started.

NixOS builds on Nix, a *purely functional* package manager. NixOS uses Nix to build packages and other static system configuration artifacts such as configuration files in a reproducible way. The file configuration.nix is essentially a parameter to a Nix function that evaluates to a large dependency graph of packages, configuration files and boot scripts in the *Nix store*, together constituting the system. See [2] for details.

Nix stores these artifacts in the filesystem in locations such as /nix/store/wjbcr40b...-apache-httpd-2.2.23/, where wjbcr40b... is a cryptographic hash of the dependencies of the artifact (in this case, everything used to build Apache 2.2.23). This property makes upgrading a NixOS system transactional: for instance, if we update the Nix file that specifies the Apache package to build version 2.4.3 and then run nixos-rebuild, a new instance of Apache will be built in a *different* location in the filesystem, e.g. /nix/store/jscp2ym2...-apache-httpd-2.4.3/. The same applies the configuration files and service scripts that refer to those paths. Thus, the old system configuration is not overwritten, allowing efficient rollbacks and nearly atomic upgrades. For instance, if the system crashes during an upgrade, we either get the old configuration or the new one, but not something in between.

### B. Network Configurations

Charon extends NixOS' configuration model to *networks of machines* (where a "network" is a set of machines that are deployed and managed together). A Charon specification is essentially a set of NixOS machine configurations. For instance, the following specifies a set of two machines – a machine webserver that serves files stored in a directory on the machine fileserver mounted via the Network Filesystem (NFS):

```
{ webserver =
    { services.httpd.enable = true;
      services.httpd.documentRoot = "/data";
      fileSystems."/data" = # /data mountpoint
        { fsType = "nfs4";
          device = "fileserver:/"; };
    };
  fileserver =
    { services.nfs.server.enable = true;
      services.nfs.server.exports = "...";
    };
}
```

We can now deploy this configuration. In the simplest scenario, the logical machine configurations webserver and fileserver are deployed to pre-existing (e.g. physical) NixOS machines with corresponding host names. (This requires that each machine is reachable via SSH.) Charon tracks the state of deployments in a SQLite database. To create a new deployment, given a specification like the one above, we do:

```
$ charon create network.nix --name simple
```

This creates a new deployment named simple, defined by the file network.nix shown above. We can then perform the actual deployment:

```
$ charon deploy --name simple
```

This will build or download all the software dependencies of the new configuration, copy them to the two target machines, and then activate the new configuration by (re-)starting modified system services. For instance, on fileserver the NFS server will be started, while on webserver the /data filesystem will be mounted and Apache will be started.

Changing the configuration is a matter of changing the specification and rerunning charon deploy. Due to the non-destructive nature of the Nix package manager, it is possible to roll back efficiently to a previous configuration by running charon rollback.

### C. Provisioning Machines

Provisioning cloud machines is done in almost the same way; all that is needed is to tell Charon, in the network specification, that a machine should be instantiated as a virtual machine in a specific cloud environment. For instance, we can add the following attributes to the definition of webserver and fileserver in network.nix:

```
deployment.targetEnv = "ec2";
deployment.region = "eu-west-1";
deployment.instanceType = "m1.large";
```

Now when we run charon create and charon deploy, Charon will notice (by consulting its state) that it has not created EC2 instances corresponding to the logical machines webserver and fileserver yet. It will therefore fire up two basic NixOS instances, wait until they are up and reachable via SSH, and then build and deploy the specified configuration as described above. Subsequent redeployments will "reuse" the previously created instances; adding a new logical machine to the specification and rerunning charon deploy will cause the missing instance to be created; and removing a logical machine from the specification and rerunning charon deploy will cause the corresponding instance to be destroyed. Thus, Charon's general invariant is that after running charon deploy, *the actual configuration of the network is in sync with the specified configuration.*

Similarly, we can deploy the same specification as a set of VirtualBox VMs running on the user's system by specifying:

```
deployment.targetEnv = "virtualbox";
```

This is convenient for testing: the same "logical" specification can be deployed to different environments for testing or production use.

Charon specifications also abstract over the network connectivity between machines: it can be assumed that each machine in the network can reach every other machine through its host name. Charon ensures this by generating an appropriate /etc/hosts file on each machine. Thus, the NFS mount on webserver (the line device = "fileserver:/") will work correctly, even if webserver and fileserver are in (say) different EC2 regions. If necessary, machine configurations can refer to the IP addresses of other logical machines through the attribute nodes.*machine*.config.networking.privateIPv4.

This kind of abstraction allows interesting changes to the network setup without affecting the logical specification. For instance, by adding the single line

```
deployment.encryptedLinksTo = ["fileserver"];
```

to the specification of webserver, Charon then injects NixOS configuration fragments to set up an encrypted VPN tunnel between the two machines, and to make the hostname fileserver in /etc/hosts on webserver refer to fileserver's VPN endpoint address. Thus the NFS mount will subsequently run over an encrypted connection.

## *D. MindMup Example*

We now show a larger example of a Charon deployment: a specification to deploy *MindMup* (http://mindmup.com), an online open source application that allows users to create mindmaps and share these with other users. MindMup is a Ruby application designed to run on Amazon EC2 and use Amazon S3 for storage. To make the example more compelling, we deploy this application using a multi-machine network that implements the common deployment pattern of using multiple backend application servers with a reverse proxy in front for fault tolerance and load balancing.

The MindMup network consists of three machines: a reverse proxy server, and two application servers running the application code. Figure 1 shows a Charon specification of the logical aspects of of the MindMup deployment, i.e. the software packages and configuration required on each machine. The network specification defines the three machines proxy, backend1 and backend2.

The MindMup application code runs on the machines backend1 and backend2, defined at $\boxed{3}$, which have the same configuration. The mindmup function at $\boxed{1}$ includes the NixOS module for MindMup, and enables it to make sure the machine will start the application code. It also defines the siteURL option of the MindMup configuration, which is used to redirect to the application after saving a mindmap. As we are running with a reverse proxy, we want to redirect to the proxy, which we do by injecting the public IP address of the proxy machine $\boxed{2}$.

```
let
  mindmup = [1]
    { resources, nodes, ... }:
    { require = [ ./mindmup.nix ];
      services.mindmup.enable = true;
      services.mindmup.siteURL = "http://${ [2]
        nodes.proxy.config.publicIPv4}/";
    };
in {
  backend1 = mindmup; [3]
  backend2 = mindmup;
  proxy = [4]
    { services.httpd.enable = true;
      services.httpd.extraModules =
        [ "proxy_balancer" ]; [5]
      services.httpd.extraConfig = ''
        <Proxy balancer://cluster>
          BalancerMember http://backend1:8080
          BalancerMember http://backend2:8080
        </Proxy>
        ProxyPass / balancer://cluster/
      '';
    };
}
```

Fig. 1. MindMup logical network specification

The proxy machine at $\boxed{4}$ uses the httpd NixOS module, which allows us to configure an Apache HTTP server. At $\boxed{5}$ we state that the proxy_balancer Apache module should be loaded. This allows Apache to act as a reverse proxy, configured by specifying a BalancerMember for each backend.

Now we need to define the physical specification that states how to instantiate the needed infrastructure. In addition to virtual machines, Charon can provision other types of cloud resources. In this case, we need the following:

- An Amazon S3 bucket to store the mindmaps.
- An Amazon EC2 keypair to securely connect to the instances.
- An Amazon IAM role, a security policy to restrict access to the S3 bucket.

Figure 2 shows the (somewhat abbreviated) physical network specification of the MindMup deployment. All machines and resources are provisioned in the us-east-1 EC2 region. The specifications at $\boxed{13}$ and $\boxed{6}$ define the machines that we want to instantiate on Amazon EC2 with its desired properties such as *type* and *region*.

In addition, the non-machine resources are specified. The IAM role definition at $\boxed{12}$ ensures that different deployments of the same application do not interfere with each other. The role is defined to allow all S3 operations on the S3 bucket

```
let
  region = "us-east-1";
  accessKeyId = "mindmup";
  mindmup = [6]
    { deployment.targetEnv = "ec2";
      deployment.ec2.region = region;
      deployment.ec2.keyPair = [7]
        resources.ec2KeyPairs.mindmup-kp.name;
      deployment.ec2.instanceProfile = [8]
        resources.iamRoles.mindmup-role.name;
      services.mindmup.s3BucketName = [9]
        resources.s3Buckets.mindmup-s3.name;
    };
in {
  resources.s3Buckets.mindmup-bucket = [10]
    { inherit region accessKeyId; };
  resources.ec2KeyPairs.mindmup-kp = [11]
    { inherit region accessKeyId; };
  resources.iamRoles.mindmup-role = [12]
    { inherit accessKeyId;
      policy = ...;
    };
  backend1 = mindmup;
  backend2 = mindmup;
  proxy = [13]
    { deployment.targetEnv = "ec2";
      deployment.ec2.region = region; ...
    };
}
```

Fig. 2. MindMup physical network specification

created as part of this deployment. Amazon IAM uses so-called *Amazon Resource Names* (ARN) to refer to Amazon provisioned resources. The role is applied at $\boxed{8}$ to the backend machines only, as the reverse proxy does not need to have access to the S3 bucket. The EC2 keypair mindmup-kp is defined at $\boxed{11}$ and is applied to all machines in the network (e.g. at $\boxed{7}$). The S3 bucket mindmup-s3, used to store the mindmaps, is defined at $\boxed{10}$. The application is configured to use this bucket at $\boxed{9}$ using a configuration option defined in the MindMup NixOS module.

## III. Experience

LogicBlox develops a declarative cloud platform for the development of a new class of enterprise applications that combine transactions with analytics. We are using Charon at LogicBlox both in testing and production use, deploying to Amazon EC2 as well as to physical clusters. As our applications have very specific software dependencies for the different type of nodes, and typically involve large data sets or computational requirements that require distribution over a large number of machines, it is not an option to manually install machines or to have a single EC2 disk image (AMI) to handle all deployments.

Charon makes it easy to express this variability in our deployments and ensures that they are automatic and reproducible. The latter is especially important for disaster recovery. It also ensures that it is easy to fire up test deployments from our continuous build system. Since these are identical to production deployments, the gap between development and operational use is reduced (preventing the common phenomenon of code being "thrown over the wall" to operations).

Charon also offers us the ease of enabling elasticity in deployments in a declarative way, where the number of machines is scaled up or down depending on requirements (e.g. to shard the database across a variable number of machines). For instance, we can generalise the MindMup network to support a variable number of backends: rather than hardcoding backend1 and backend2, we just write

```
let makeBackend = i: { ... };
in map makeBackend (range 0 n)
```

to generate a list of $n$ backend machines. Similarly, the configuration of the reverse proxy in Figure 1 can map over this list to emit BalancerMember lines for each backend machine. Changing $n$ and running charon deploy will then cause machines to be created or destroyed, and the proxy configuration to be updated, as necessary.

We have used Charon to deploy clusters of up to 52 machines. Charon was primarily designed to improve the manageability of many deployments with different configurations, rather than a few large deployments of thousands of identical machines. More work will be needed to support the latter.

Another scenario where Charon has shown its strength is in allowing to easily define and run benchmarks in a reproducible way on different configurations, such as varying kernel versions, application versions, RAID setup and EC2 instance sizes. The NixOS module system and its declarativity allow for easy composition of such variants.

## IV. Conclusion

In this paper we have presented Charon, a tool for provisioning and deployment of networks of machines from declarative specifications. It has several important properties, such as reproducibility, integration of provisioning and deployment, and abstraction over cloud backends. In future work, we intend to improve management of mutable state, e.g., to allow migration of machines between cloud backends or regions.

## References

[1] Canonical, "Juju – devops distilled," https://juju.ubuntu.com/, 2013.

[2] E. Dolstra and A. Löh, "NixOS: A purely functional Linux distribution," in *13th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP 2008)*. ACM, Sep. 2008.

[3] E. Dolstra, E. Visser, and M. de Jonge, "Imposing a memory management discipline on software deployment," in *26th Intl. Conf. on Software Engineering (ICSE 2004)*. IEEE Computer Society, May 2004, pp. 583–592.

[4] M. Burgess, "Cfengine: a site configuration engine," *Computing Systems*, vol. 8, no. 3, 1995.

[5] Puppet Labs, "Puppet Labs: IT automation software for system administrators," https://puppetlabs.com/, 2013.

[6] Opscode, "Chef," http://www.opscode.com/chef/, 2013.

[7] M. Hashimoto, "Vagrant," http://www.vagrantup.com/, 2013.

[8] C. Quinton, R. Rouvoy, and L. Duchien, "Leveraging feature models to configure virtual appliances," in *Proceedings of the 2nd Intl. Workshop on Cloud Computing Platforms*. ACM, 2012, pp. 2:1–2:6.

[9] J. A. Hewson, P. Anderson, and A. D. Gordon, "A declarative approach to automated configuration," in *Proceedings of the 26th Intl. Conf. on Large Installation System Administration*. USENIX, 2012, pp. 51–66.