NixOps for Proxmox

Ryan Lahfa (ryan at lahfa dot xyz)

17 October 2020

Plan

Plan

Before everything, the talk will be uploaded on https://github.com/RaitoBezarius/nixops-proxmox [1]

- What is Proxmox?
- What is NixOps and what are backends in NixOps?
- A Proxmox backend for NixOps: `nixops_proxmox`
- Challenges encountered
- Future work
- Demonstration: deploying your sr.ht on multiple virtual machines on Proxmox[2]

[1] With the code of this piece of software!

[2] The demonstration will be available separately and it is not part of the main talk for length reasons

What is Proxmox?

Overview

Proxmox is a Linux distribution tailored to host virtual machines and containers on the top of a bare-metal machine, using KVM/QEMU. It comes with a nice web UI and CLI/API, handles **a part of** the networking and storage for you. It supports highly available (multi-cluster) or hyper-converged (Ceph) design.

It can be effectively used as a replacement for managing a (very) small to large [3] inventories.

[3] I have no exact numbers on who uses Proxmox beyond 100 physical machines and 1000 VMs

What is NixOps?

Overview

NixOps is a Python application which aims to provide deployment primitives based on Nix expressions, external APIs, external resources, and NixOS. It is designed to be modular.

For example, NixOps has support for AWS (virtual private network, elastic IP, EC2, ...) or Hetzner (machines). Those plugins which NixOps can use are called 'backends' — we will see more about this in the next slide.

Backends

As said, there are multiple backends in NixOps currently:

- AWS
- GCP
- Azure (broken in the newest versions)
- Hetzner
- libvirtd (wrapper around QEMU/KVM)
- VirtualBox
- None (it is a trivial backend which consist in supposing that the target is a NixOS machine and ignoring the meta-details, you still need to provide a baseline configuration for networking, disks, etc.)

What does a backend?

A backend implements a part of a NixOps interface and bring a definition (think a Nix expression) into reality. For example, if you add a disk in your definition, the backend will notice this change and use the API to create a disk, then attach it whether you specified to attach it.

```
{
  machine =
    { imports = [ ./ec2-info.nix ];
      deployment.targetEnv = "ec2";
      deployment.ec2.region = "us-east-1";
      deployment.ec2.instanceType = "t2.medium";
    };
}
```


NixOps + Proxmox = <3

Self-hosted mini-cloud

As NixOps has no real backend for private infrastructure[4], I grown tired of installing NixOS manually myself all the time and decided to write a NixOps backend in 48 hours[5].

As a result, the initial proof of concept offered me a way to trivially deploy NixOS nodes on my Proxmox cluster:

- private actual mini-cloud ;
- more or less declarative (see the 'future work' part) ;
- extensions are possible using NixOps extra plugins: handle storage backend as resources, handle disks as resources and even more.

[4] None does not solve the problem of provisioning at the start your virtual machine.

[5] Hence, the alpha quality!

How it works: part 1

It will perform those tasks (in order):

- Create the selected disks with their size and storage backend (ZFS, NFS, etc.) ;
- Create a virtual machine in Proxmox with the selected parameters (network interfaces, RAM, vCores, see `nixops_proxmox/nix/proxmox.nix` for details) ;
- Attach the custom NixOS ISO (if it's the first startup) ;
- Start it and wait until QEMU Agent is up ;
- If it's in live CD, provision a temporary SSH key through the QEMU agent (which enable arbitrary read/write/exec without any real network stack) ;
- It will wait for an (reachable) IP address ;

How it works: part 2

- If it's in live CD, it will wait for SSH, check if the partitioning script has changed and offer to repartition or not, then proceed to partitioning, then reboot if necessary ;
- If it's in live CD and it has been partitioned after a reboot, it will write the initial configuration.nix after a `nixos-generate-config` and some changes and mounting all the partitions, then install NixOS, reboot ;
- If it was not marked as installed yet, it will wait for the post-installation phase, reinstall a host key in our known hosts through QEMU Agent, then wait for SSH and mark it as installed ;
- Then, it gives the hand to NixOps to do its magic.

Challenge 1: IPv4 vs IPv6

NixOps has no real native support for IPv6 as far as I know, moreover, it hardcodes stuff like this: https://github.com/NixOS/nixops/blob/d5baedfa79c32327e23d34813a7e75f5979f7560/nixops/deployment.py#L659

So I just decided to implement IPv6 support in my plugin (which is not really good, but shows that it is possible/usable and NixOps is indeed modular).

Challenge 1: Extract from `nixops_proxmox`

```
...
ip_addresses = list(chain.from_iterable(
  map(lambda i: ip_address(i['ip-address']), if_['ip-addres
  for name, if_ in net_ifs.items()
  if if_['ip-addresses'] and name != "lo"))
private_ips = {str(ip) for ip in ip_addresses
  if ip.is_private and not ip.is_link_local}
public_ips = {str(ip) for ip in ip_addresses
  if not ip.is_private}
ip_v6 = {str(ip) for ip in ip_addresses
  if isinstance(ip, IPv6Address)}
ip_v4 = {str(ip) for ip in ip_addresses
  if isinstance(ip, IPv4Address)}
...
```


Challenge 1: How to get actual IPs

```
...
self.private_ipv4 = first_reachable_or_none(
  private_ips & ip_v4)
self.public_ipv4 = first_reachable_or_none(
  public_ips & ip_v4)
self.private_ipv6 = first_reachable_or_none(
  private_ips & ip_v6)
self.public_ipv6 = first_reachable_or_none(
  public_ips & ip_v6)
...
```

Challenge 2: How to select the right endpoint ?

One issue with having private/public IPv4/IPv6 is to know which one you can reach and which one you cannot:

- Imagine you're in a café, no IPv6, you want to deploy or SSH into one of your machine, fortunately, you're on the VPN so you can reach the private IP, the backend has to guess it and select the appropriate IP ;
- Imagine you're in a café, with IPv6, you're actually also on the VPN, the backend should determine the fastest endpoint, as the IPv6 could be also a tunnel in reality, and the VPN might be faster[6] ;

[6] This is not a solved problem in my backend, but I'm open to comments regarding those ideas.

Challenge 3: (Not Very Declarative) Partitioning

If you use Proxmox and not AWS/GCP/Azure/yourself, you will have to effectively encode your partitioning script, which could be for the worst or the better, a bit annoying and error-prone.

I looked for ways to simplify this, using Nixpart (which I will mention later) and other tooling but found out no reasonable and acceptable tool so I fall back on classical Bash and let user decide how he wants to do his thing.

Actually, this is not that bad, you just need to read the manual page until you understand you forget to set the right type for the EFI partition. :-)

Future work

Current state: buggy but not that much

The resulting plugin is usable, I use it, but is alpha-quality and has a lot of issue:

- the partitioning is bothersome ;
- the state machine between “rescue mode” and running/stopped is messy ;
- the backend requires a custom NixOS image which enables the QEMU Agent (and sshd) ;
- no tags support ;
- no backups ;
- no automatic node selection (you have to explicitly choose the node on which you deploy)

Extending the work: Routing

Proxmox is not a router, for example, it's quite difficult to add new VLAN, etc. It depends highly on your infrastructure behind your Proxmox nodes.

In my infrastructure, I use VyOS, which has an HTTP API and could be transformed into a NixOps resource, bringing the routing table to my Nix expressions ! This is an experimentation that I'd very much like to perform.

Extending the work: Declarative partitioning

Declarative partitioning in the Nix community is not a solved problem as far as I know, some efforts have been made in the direction with https://github.com/NixOS/nixpart.

Also, it is a required step to enable **easy** LUKS automatic decryption, which is the next point.

Extending the work: Automatic full disk decryption

Indeed, without declarative partitioning[7], it's not super meaningful to reproduce full disk encryption and manage its lock/unlock lifecycle.

Thus, I want to experiment with fully encrypted guests with automatic decryption, using https://www.recompile.se/mandos [8].

It enables automatic full disk encryption by receiving PGP-encrypted passphrases by local **trusted** peers for which only you have the key, and can be deployed in a fully-meshed fashion to ensure reliability across a datacenter, without removing the manual passphrase entry feature[9].

[7] And also (proper) secrets management à la Vault by HashiCorp.
[8] Which I'm trying to package in NixOS, but didn't have time these months...
[9] a.k.a. escape hatch.

Conclusion : what can be learnt, what can be used

One of my personal objectives is to see more offerings of private flexible infrastructures à la AWS, without having to use these infrastructures. I think that Nix(OS) is an important key towards achieving this goal.

NixOps is well-known to be the odd piece of software of the Nix community[10], but I believe that it's definitely salvageable and can become a versatile piece of software.

If most of the future work would be implemented, it would enable IPv6-only, NixOS-only, fully declarative micro or mini datacenter, which would be definitely exciting for a first step to become independent from public clouds.

[10] I can definitely agree with some pain points.

Thank you for watching, I will be available for your questions and don't hesitate to watch the demo if you find this interesting!