Simplify development of software stacks for experimentation with NixOS-Compose

GDR RSD: Journées non thématiques 2023

Quentin GUILLOTEAU, Jonathan BLEUZEN, Millian POQUET, Olivier RICHARD

Université Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

2023-01-27
Outline

1. Context & Motivation
2. NixOS Compose
3. Experimental Evaluation
4. Benefits, Limitations and Lessons
5. Conclusion & Perspectives
Experiment’s Workflow and Some Issues

- Real experiment’s workflow can be complex and tricky to develop and tune
- **Reproducibility** objective must be considered at the beginning
  - At mid and long terms: lot of time saved
- HPC and BigData stacks:
  - Complex pieces of software, lot of parameters
- Input Workloads
  - Too few HPC and BigData traces
  - Lot of hypothesis
How to build customized Grid’5000 image(s) ?

- **Recipe** (high level) how the software appliance is going to be built. Meta-data in form of global variable and steps (mid and low-level)
- **Data** which is used as an input of all the build steps described in the recipe. It takes the form of prebuilt software packages, tarballs, configuration files, control version repositories and so on.
- **Kameleon engine**, which parses the recipe and carry out the process of building.
Kameleon approach: issues

**Pro**

- Overall it does the job
- All Linux distributions can be supported (Debian, Ubuntu, Centos)
- Comparable tool: Packer from Harsicorp

**Limitations**

- Development of recipe is tedious and error prone
- Build time can be/is huge $>10$ min
- During experiment’s development some tests could be done on VMs or Containers
- Not adapted for frequent changes
The Problem

Setting up Distributed Environments for Distributed Experiments → Difficult, Time-consuming and Iterative process

A moving target

I’m done with experiments.

Well, there’s one more. But then I’ll be done.

The last three years of your Ph.D.

Does not encourage good reproducibility practices
Nix and NixOS

The Nix Package Manager

- Functional Package Manager
- Nix Lang $\simeq$ json + $\lambda$
- Nixpkgs (Nix expression of packages, OS...)
- Reproducible by design

The NixOS Linux Distribution

- Based on Nix
- Declarative approach
- Complete description of the system (kernel, services, pkgs, config)
How to store the packages?

**Usual approach:** **Merge them all**

- Conflicts
- PATH=/usr/bin

```
/usr
├── bin
│   └── myprogram
└── lib
    ├── libc.so
    └── libmylib.so
```

**Store approach:** **Keep them separated**

+ Pkg variation
+ Isolated
+ Well def. PATH
+ Use RPATH
+ Read-only

```
/nix/store
├── y9zg6ryffgc5c9y67fcmfdkyyiivjzpj-glibc-2.27
│   └── lib
│       └── libc.so
└── nc5qbagm3wqfg2lv1gwj3n3bn88dpqr8-mypkg-0.1.0
    └── bin
        └── myprogram
    └── lib
        └── libmylib.so
```
Nix Profiles 1/2

- User Profile

/home/alice/.nix-profile
/nix/var/nix/profiles/per-user/alice
  ├── profile -> profile-42-link
  │   └── profile-42-link -> /nix/store/zfhd...-user-env
  │       └── bin
  │           └── batsim
  └── 6k6f...-simgrid-3.31
     └── lib
         └── libsimgrid.so.3.31

Olivier RICHARD | UGA,LIG | 27/1/2023
Nix Profiles 2/2

System Profile for NixOS

- Define the kernel, Init script, initrd ...
- Fstab (file systems table)...
- Services (via Systemd)
- Immutable (part) configurations in /etc
1. Context & Motivation

2. NixOS Compose

3. Experimental Evaluation

4. Benefits, Limitations and Lessons

5. Conclusion & Perspectives
NixOS Compose - Introduction

Goal

Use Nix(OS) to reduce friction for the development of reproducible distributed environments

The Tool

- Python + Nix (≈ 6000 l.o.c.)
- an extension of Nixos-Test
- One Definition
  → Multiple Platforms
- Build and Deploy
- Reproducible by design
NixOS Compose - Terminology

Transposition
Capacity to deploy a uniquely defined environment on several platforms of different natures (flavours, see later).

Role
Type of configuration associated with the mission of a node. Example: One Server and several Clients.

Composition
Nix expression describing the NixOS configuration of every role in the environment.
NixOS Compose - Composition Example: K3S

```nix
{ pkgs, ... }: let k3sToken = "df54383b5659b9280aa1e73e60ef78fc";
in { roles = { server = { pkgs, ... }: {
    environment.systemPackages = with pkgs; [ k3s gzip ];
    networking.firewall.allowedTCPPorts = [ 6443 ];
    services.k3s = {
        enable = true;
        role = "server";
        package = pkgs.k3s;
        extraFlags = "--agent-token ${k3sToken}";
    };
};
agent = { pkgs, ... }: {
    environment.systemPackages = with pkgs; [ k3s gzip ];
    services.k3s = {
        enable = true;
        role = "agent";
        serverAddr = "https://server:6443";
        token = k3sToken;
    };
};
};
}```
### NixOS Compose - Flavours = Target Platform + Variant

<table>
<thead>
<tr>
<th>Flavour</th>
<th>Target Platform</th>
<th>Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>docker</strong></td>
<td>local and virtual</td>
<td>Generate a docker-compose configuration.</td>
</tr>
<tr>
<td><strong>vm</strong></td>
<td>local and virtual</td>
<td>QEMU Virtual Machines.</td>
</tr>
<tr>
<td><strong>g5k-ramdisk</strong></td>
<td>distributed and physical</td>
<td>initrds deployed in memory without reboot on G5K (via kexec).</td>
</tr>
<tr>
<td><strong>g5k-nfs-store</strong></td>
<td>distributed and physical</td>
<td>Same as g5k-ramdisk but store accessed by nfs.</td>
</tr>
<tr>
<td><strong>g5k-image</strong></td>
<td>distributed and physical</td>
<td>Full system tarball images on G5K via Kadeploy.</td>
</tr>
</tbody>
</table>
NixOS Compose - Workflow

Experiment writing

Local development
( docker, VM )

NixOS Compose phases
build - start - connect

Is outcome satisfactory?

Distributed deployment
( g5k-ramdisk, g5k-image )

NixOS Compose phases
build - start - connect

results

resources
NixOS Compose - Technical Details (g5k-ramdisk)

Building

1. Evaluation of the NixOS configuration (+firmware)
2. Generation of the kernel, image, initrd, store, one system profile per role

Deploying

1. Generate deployment info (contextualization data)
2. Run kexec on the nodes
3. Setup the info for the nodes (hostname, ssh keys, role)

Node's boot phases

```
$> INITRD={path_on_NFS}/initrd
KERNEL={path_on_NFS}/kernel
kexec -l $KERNEL --initrd=$INITRD ...
```

Kernel parameters ≤4096 bytes
1 Context & Motivation

2 NixOS Compose

3 Experimental Evaluation

4 Benefits, Limitations and Lessons

5 Conclusion & Perspectives
Experimental Evaluation

Experimental Setup

- Grid’5000: dahu cluster
- 192 GiB of RAM
- Intel Xeon Gold 6130 (2 × 16 cores)
- 240 GB SSD SATA Samsung

Goal of Experiments

- Evaluate the (re)construction times of images vs. Kameleon
- Evaluate the size of the images generated vs. Kameleon
- Evaluate the deployment cycle vs. EnOSlib

Will not evaluate the deployment times as we use third party tools.
Evaluation vs. Kameleon

Experiment Goals
Eval. Images **Construction** and **Reconstruction** Times + Images **Sizes**

Protocol
1. Empty the nix store (no cache for Kameleon)
2. Create a base recipe with NXC and Kameleon
3. Build and measure the building time and the size of the image
4. Add the hello package to the recipe (base + hello)
5. Build the base + hello image and measure time and size
**Evaluation vs. Kameleon - Results**

Image Size, Construction and Reconstruction Time for Different Environments with and without NFS

- **NXC faster to build and even faster to rebuild** (> 10x)
- **NXC produces larger images than Kameleon** (modules, firmware)
- **NFS introduces a overhead due to many reads/writes of Nix**
1 Context & Motivation

2 NixOS Compose

3 Experimental Evaluation

4 Benefits, Limitations and Lessons

5 Conclusion & Perspectives
Benefits, limitations, lessons

Use *FPM* (here Nix) to build/deploy distributed system for research purpose

**Benefits**

- Reproducibility (reconstructability) by design
- **Powerful framework** to describe all part of distributed system
- Accurate image generation (put only what you want/need)
- More pleasant experiment development (time, debugging, tranposition)
- Focus on essential complexity / less accidental complexity
- Modification, variation, extension ... in more simpler way
- Simple to use by new comers (students)

---

*a* "No Silver Bullet—Essence and Accident in Software Engineering” F. Brook 86
Benefits, limitations, lessons

Limitations and issues

- Radical approach Nix/NixOS (exclude other Linux distributions)
- Switch **declarative and functional paradigm**
- *Advanced* Nix: **steep learning curve** (internships are short !)
- **Nix ecosystem** is very **huge** (80K packages, constant evolutions, experimental features, lot of peripheral projects)
Benefits, limitations, lessons

Lessons (for Nixos-Compose)

- As usual: The Devil is in the details (corner cases, robustness at scale...)
- Importance of user experience/interface (UX/UI)
  - Workflow fluidity (CLI / features)
  - Simple customization must be simple to set up (source, parameter setting...)
- Packaging non trivial tool/service is not a beginner task (need good sysadmin skills)
- We need feedback for external (early) users
Conclusion & Perspectives

Reminder

Objective: Reduce the friction for dvp of reproducible distributed envs

Approach: used Nix(OS) to build NXC: a tool for transposing envs defs

Takeaway

- Fast (more fluid) development cycles (containers, VM, ramdisk)
- FPM (Nix/Guix) very pleasant/suitable to manage complex setup

Perspectives

- Stable Release
- Target others platforms, Chameleon ...)
- Integration w/ EnOSlib (experiment orchestration)
- More examples/compositions
Questions?

- Nixos-compose: https://gitlab.inria.fr/nixos-compose/nixos-compose
- Technical Paper: Cluster'22
  https://hal.archives-ouvertes.fr/hal-03723771/
- Tutorial https://nixos-compose.gitlabpages.inria.fr/tuto-nxc/
- Supported by the European Regale Project