

Transposition d'environnements distribués reproductibles avec NixOS Compose COMPAS22

Quentin GUILLOTEAU, Jonathan BLEUZEN, Millian POQUET,
Olivier RICHARD

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

2022-07-07

Détails

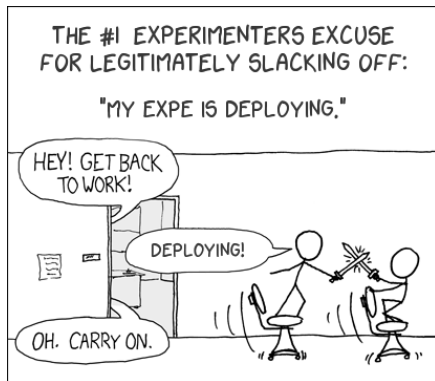
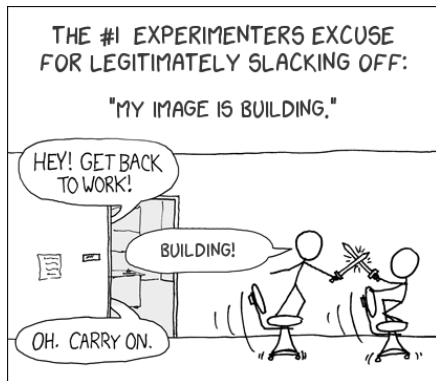
Version étendue acceptée à IEEE CLUSTER22 (!)

↪ Slides de cette version

Motivation

Setting up Distributed Environments for Distributed Experiments

↪ **Difficult**, **Time-consuming** and **Iterative** process



⇒ **Does not encourage good reproducibility practices**

The Reproducibility Problem

Different Levels of Reproducibility

- 1 **Repetition:** Run exact same experiment
- 2 **Replication:** Run experiment with different parameters
- 3 **Variation:** Run experiment with different environment

↪ **Share the experimental environment and how to build/modify it**

How to share a Software Environment in HPC?

- Containers? ↪ need Dockerfile to rebuild/modify. might not be repo (e.g., apt update, curl, commit)
- Modules? ↪ cluster dependent. how to modify?
- Spack? ↪ share through modules...

Nix and NixOS

The Nix Package Manager

- Functional Package Manager
- Packages are functions
 - Inputs = dependencies
 - Body of function = how to build
- No side-effect
- (\simeq) Solves Dependencies Hell
- Reproducible by design



The NixOS Linux Distribution

- Based on Nix
- Declarative approach
- Complete description of the system (kernel, services, pkgs)

How to store the packages?

Usual approach: **Merge them all**

- Conflicts
- PATH=/usr/bin

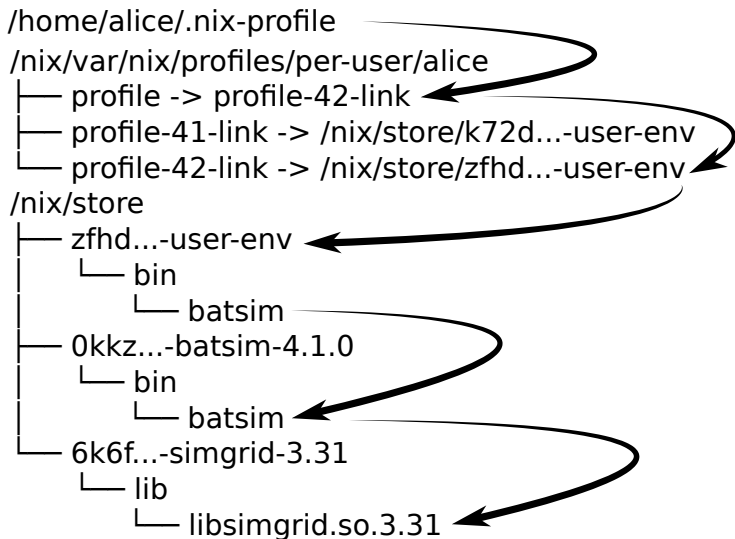
```
/usr
├── bin
│   └── myprogram
└── lib
    ├── libc.so
    └── libmylib.so
```

Nix approach: **Keep them separated**

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

```
/nix/store
├── y9zg6ryffgc5c9y67fcmfdkyyiivjzpj-glibc-2.27
│   └── lib
│       └── libc.so
└── nc5qbagm3wqfg2lv1gwj3r3bn88dpqr8-mypkg-0.1.0
    ├── bin
    │   └── myprogram
    ├── lib
    │   └── libmylib.so
```

Nix Profiles



1 Introduction & Concepts

2 NixOS Compose

3 Experimental Evaluation

4 Conclusion & Perspectives

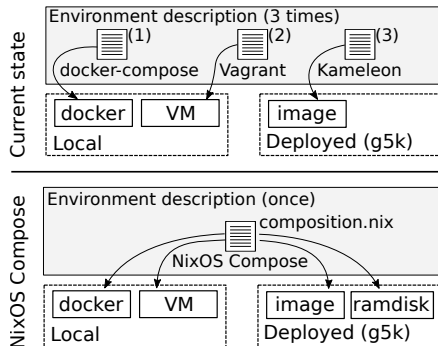
NixOS Compose - Introduction

Goal

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

The Tool

- Python + Nix ($\simeq 4000$ l.o.c.)
- 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

```

1 { pkgs, ... }:
2 let k3sToken = "df54383b5659b9280aa1e73e60ef78fc";
3 in {
4   nodes = {
5     server = { pkgs, ... }: {
6       environment.systemPackages = with pkgs; [
7         k3s gzip
8       ];
9       networking.firewall.allowedTCPPorts = [
10        6443
11      ];
12       services.k3s = {
13         enable = true;
14         role = "server";
15         package = pkgs.k3s;
16         extraFlags = "--agent-token ${k3sToken}";
17       };
18     };
19     agent = { pkgs, ... }: {
20       environment.systemPackages = with pkgs; [
21         k3s gzip
22       ];
23       services.k3s = {
24         enable = true;
25         role = "agent";
26         serverAddr = "https://server:6443";
27         token = k3sToken;
28       };
29     };
30   };
31 }

```

Annotations:

- Packages**: Points to lines 6-8 (environment.systemPackages).
- Ports**: Points to lines 9-10 (networking.firewall.allowedTCPPorts).
- Services**: Points to lines 12-16 (services.k3s).
- Role**: Points to line 14 (role = "server").

NixOS Compose - Flavours = Target Platform + Variant

`docker` - local and virtual

Generate a docker-compose configuration.

`vm-ramdisk` - local and virtual

In memory QEMU Virtual Machines.

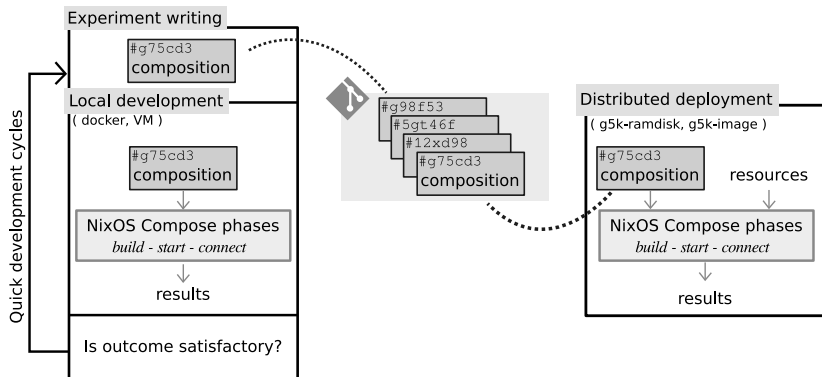
`g5k-ramdisk` - distributed and physical

initrds deployed in memory without reboot on G5K (via kexec).

`g5k-image` - distributed and physical

Full system tarball images on G5K via Kadeploy.

NixOS Compose - Workflow



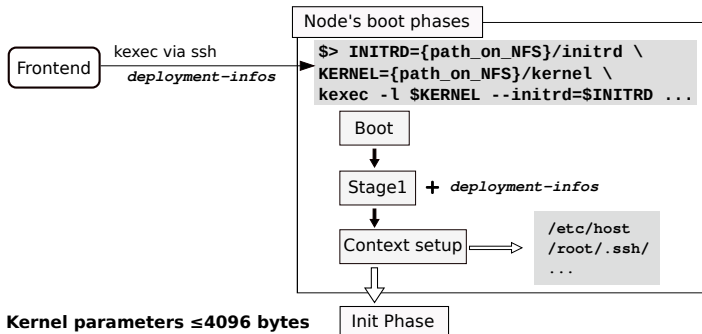
NixOS Compose - Technical Details (g5k-ramdisk)

Building

- 1 Eval. of the NixOS configuration (+firmware)
- 2 Generation of the kernel, image, initrd (in the store)

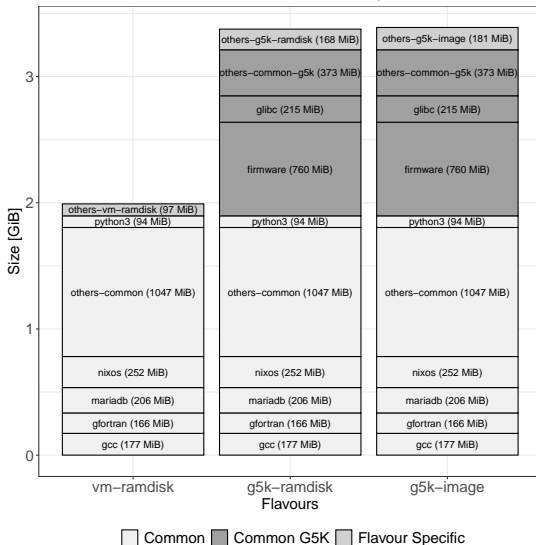
Deploying

- 1 Get the addresses of the nodes
- 2 Run kexec on the nodes
- 3 Setup the info for the nodes (hostname, ssh keys, role)



NixOS Compose - Difference per Flavours

Content of the Nix Store of the Melissa Image for each Flavour



Example: Melissa

- Distributed Runner for Data Assimilation
- Slurm, DB, ...
- Several roles

Common base for all flavours

↪ then variations based on platform/flavour (e.g., firmwares, boot loader)

1 Introduction & Concepts

2 NixOS Compose

3 Experimental Evaluation

4 Conclusion & Perspectives

Experimental Evaluation

Experimental Setup

- Grid'5000: dahu cluster
- Intel Xeon Gold 6130 (2×16 cores)
- 192 GiB of RAM
- 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

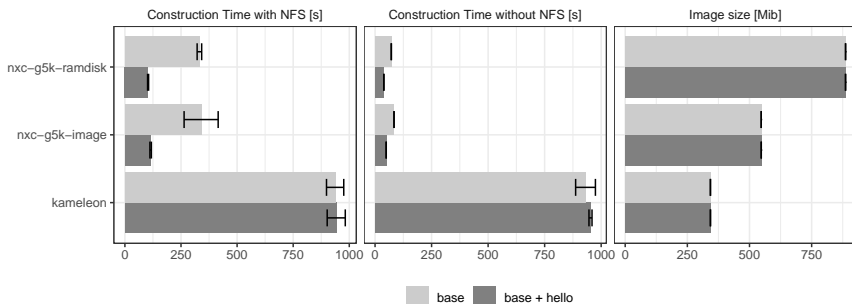
Eval. Images **Construction** and **Reconstruction** Times + Images **Sizes**

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** ($> 10\times$)
- NXC produces larger images than Kameleon
- NFS introduces a overhead due to many reads/writes of Nix

Evaluation vs. EnOSlib

Experiment Goals

Eval. Deployment Cycles vs. EnOSlib **with Reproducibility considerations**

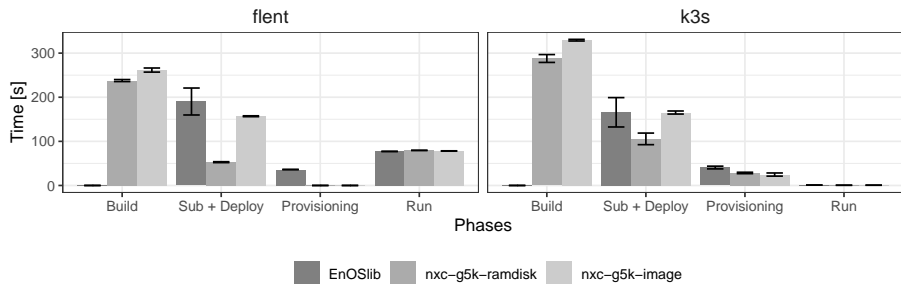
4 Phases: Build \rightsquigarrow Deploy \rightsquigarrow Provisioning \rightsquigarrow Run.

Protocol

- 1 Write an experiment with EnOSlib and NXC (+ Execo)
- 2 Build the image if needed (EnOSlib uses a prebuilt G5K image)
- 3 Deploy the image
- 4 Do the Provisioning phase (i.e., installing pkgs + config)
- 5 Run the actual experiment
- 6 Measure the time spent in each phase

Evaluation vs. EnOSlib - Results

Time Spent in each Phases for Different Approaches with 99% Confidence Intervals (5 repetitions)



- No building for EnOSlib (might need it if image no longer available)
- **Fast Deploy with g5k-ramdisk** (via kexec)
- Manage to **reduce provisioning phase** with NXC in the image

1 Introduction & Concepts

2 NixOS Compose

3 Experimental Evaluation

4 Conclusion & Perspectives

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 development cycles (containers, VM, ramdisk)
- NXC builds faster (but larger) images than Kameleon

Perspectives

- Target others platforms (e.g., Chameleon)
- Hybrid deployment (e.g., multiple VMs on G5K nodes)
- Integration with EnOSlib
- Documentation and Tutorials