Helping research on distributed systems with a *functional* package manager

Ten Years of Guix

Quentin GUILLOTEAU, Jonathan BLEUZEN, Millian POQUET, Olivier RICHARD

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

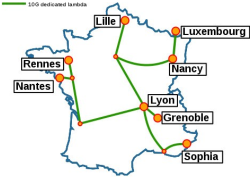
2022-09-16

Outline

The Grid'5000 testbed (https://www.grid5000.fr)

A large-scale testbed for distributed computing

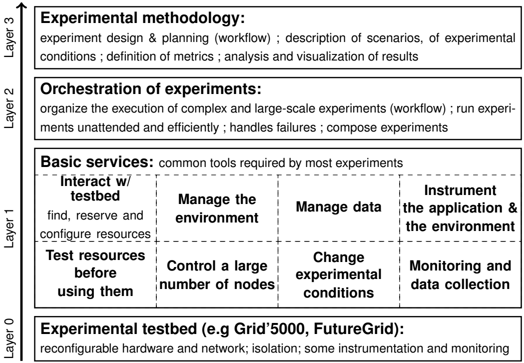
- 8 sites, 31 clusters, 828 nodes, 12328 cores
- Dedicated 10-Gbps backbone network
- 550 users and 120 publications per year



A powerfull place to experiment

- Used by CS researchers in HPC, Clouds, Big Data, Networking, AI
- To experiment in a fully controllable and observable environment
- Low-level access (bare-metal deployement, serial console,...)
- Similar problem space as Chameleon and Cloudlab (US)

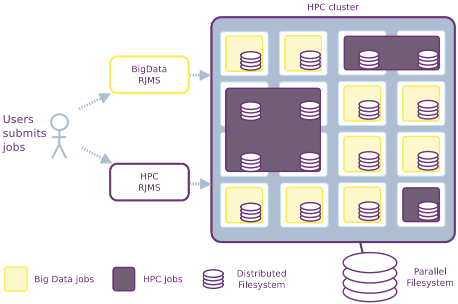
Experiment Layers



[1]

[1] Figure: Grid'5000 - Lucas Nussbaum

Example: Mixing HPC and BigData Workloads

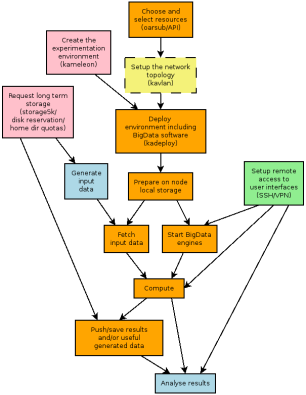


- **Simple Idea**: Idle HPC resources used for BigData workload
  - HPC jobs have priority
  - Resource and Job Management Systems (HPC/RJMS): Slurm / OAR
  - BigData Framework: Spark/Yarn, HDFS
  - Evaluating costs of starting/stopping tasks (Spark/Yarn) and data transferts (HDFS)

Mixing HPC and BigData Workloads: OAR + Spark/Yarn

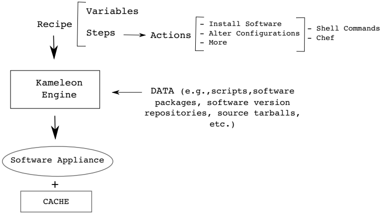
Experiment's Workflow and Some Issues

- 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



Kameleon:A tool to generate *software appliancies* (image)

- 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: recipe

- A *Yaml* File

```
global:
  workdir: /tmp/kameleon
  distrib: debian
  debian_version_name: etch
  distrib_repository:  http://archive.debian.org/debian-archive/debian/
  output_environment_file_system_type: ext3
  arch: i386
  network_hostname: "test"
  extra_packages: "mysql-server mysql-client mingetty "
  oar_repository: "deb http://oar-ftp.imag.fr/oar/2.2/debian/stable/ ./"
 steps:
  - bootstrap
  - system_config
  - mount_proc
  - software_install:
  - extra_packages
  - oar_2.2/oar_debian_install
  - oar_2.2/oar_system_config
  - oar_2.2/oar_config
  - autologin
  - kernel_install
  - umount_proc
  - build_appliance_kpartx:
    - create_raw_image
    - attach_kpartx_device
    - mkfs
    - mount_image
    - copy_system_tree
    - install_extlinux
    - umount_image
    - save_as_vdi
```

```
oar_config:
  - config_mysql:
      - exec_chroot: /etc/init.d/mysql start || service mysql start || true
      - exec_on_clean: chroot $$chroot bash -c "/etc/init.d/mysql stop || true"
  - mysql_db_init:
      - exec_appliance: cp $$stepdir/data/oar_mysql_db_init $$chroot/usr/lib/oar/
      - exec_chroot: oar_mysql_db_init
  - update_hostfile:
      - append_file:
           - /etc/hosts
           - |
              127.0.0.1 node1 node2
  - create_resources:
      - exec_chroot: oarnodesetting -a -h node1
```

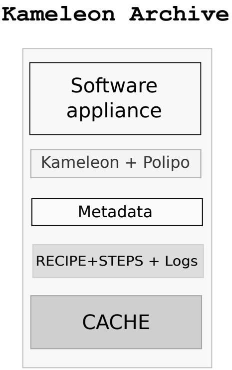
- Rustic approach: **execute imperatively shell commands**

Kameleon: Toward Reconstructability

Kameleon's Archive

The problem of reconstructing a given software appliance is reduced to keeping three main parts unchanged.

- Recipe, steps and all the metadata.
- DATA.
- Kameleon engine.



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

⟹ **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

$\hookrightarrow$ **Share the experimental environment and how to build/modify it**

How to share a Software Environment in HPC?

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

Nix and NixOS

The Nix Package Manager (similar to Guix)

- Functional Package Manager
- Nix Lang $\simeq$ json $+$ λ
- 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
└── nc5qbagm3wqfg2lv1gwj3r3bn88dpqr8-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-41-link -> /nix/store/k72d...-user-env
└── profile-42-link -> /nix/store/zfhd...-user-env
/nix/store
├── zfhd...-user-env
│   └── bin
│       └── batsim
├── 0kkz...-batsim-4.1.0
│   └── bin
│       └── batsim
└── 6k6f...-simgrid-3.31
    └── lib
        └── libsimgrid.so.3.31
```

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**

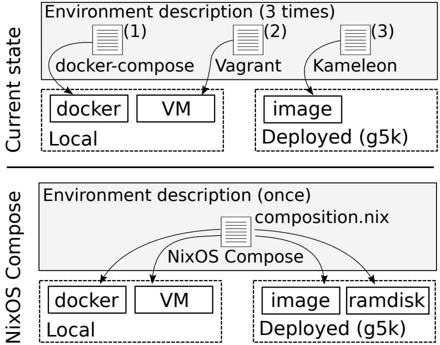
NixOS Compose - Introduction

Goal

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

The Tool

- Python + Nix ($\simeq$ 6000 l.o.c.)
- an extension of **Nixos-Test**
- **One Definition**
  $\hookrightarrow$ 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

```
{ pkgs , ... }:
let k3sToken = "df54383b5659b9280aa1e73e60ef78fc";
in {
  nodes = {
    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;
      };
    };
  };
}
```

Role

Packages

Ports

Services

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.

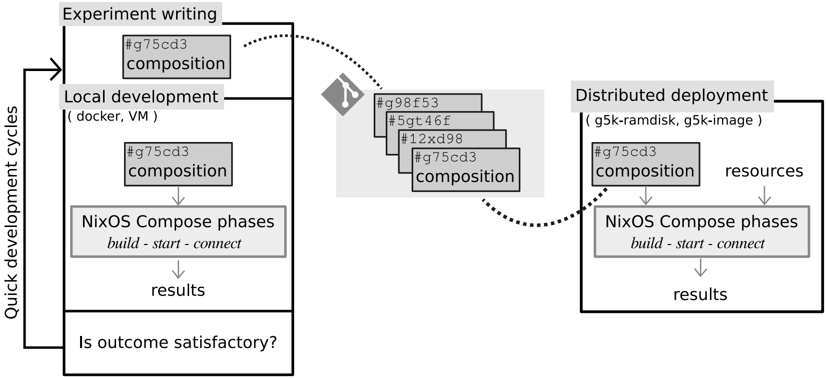
`g5k-ramdisk` - distributed and physical

`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



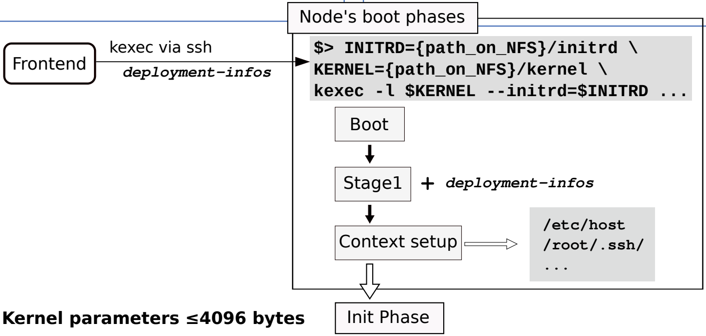
NixOS Compose - Technical Details (`g5k-ramdisk`)

Building

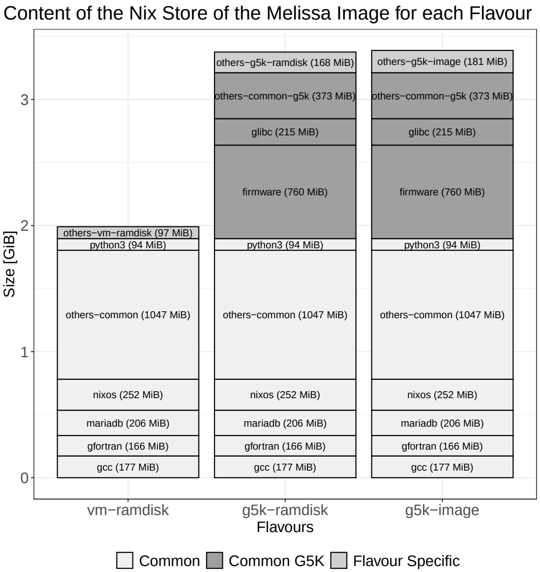
1. Eval. of the NixOS configuration (+firmware)
2. Generation of the kernel, image, initrd, store, one system profile per role

Deploying

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



NixOS Compose - Difference per Flavours



Example: Melissa

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

Common base for all flavours

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

Experimental Evaluation

Experimental Setup

- Grid'5000: dahu cluster
- 192 GiB of RAM
- Intel Xeon Gold 6130 (2 $\times$ 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**

$\hookrightarrow$ Will not evaluate the deployment times as we use third party tools.

Evaluation vs. Kameleon

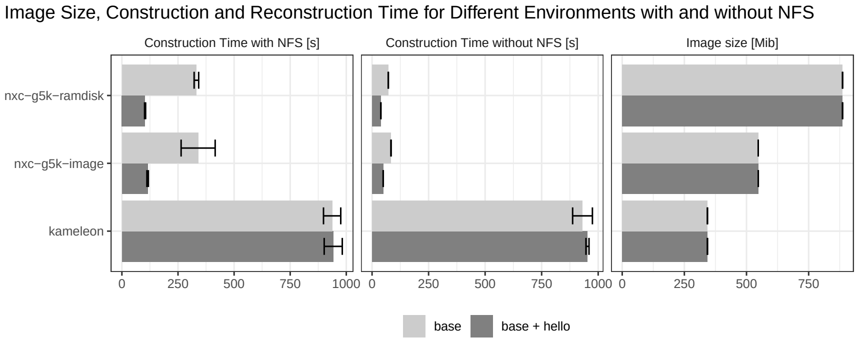
Experiment Goals

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



- 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

Evaluation vs. EnOSlib

Experiment Goals

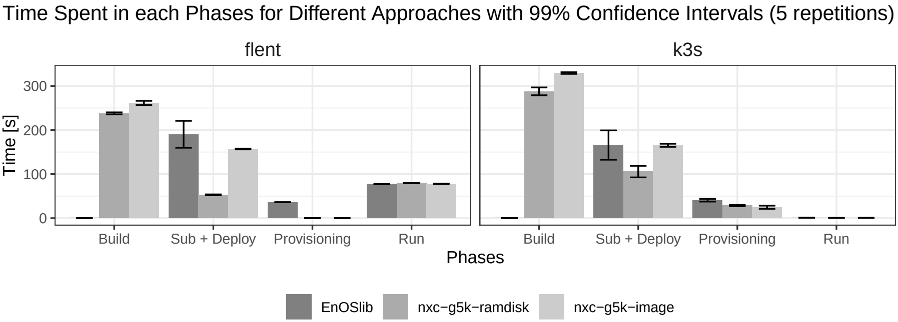
Eval. Deployment Cycles vs. EnOSlib **with Reproducibility considerations**

4 Phases: Build ⇝ Deploy ⇝ Provisioning ⇝ 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



- 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

Benefits, limitations, lessons

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

Benefits

- **Reproducibiliy (reconstructability) by design**
- **Powerful framework** to describe all part of distritued system
- Accurate image generation (put only what you want/need)
- More pleasant experiment development (time, debugging, tranposition)
- Focus on **essential complexity** / less **accidental complexity** [a]
- 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 custumization 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 (e.g. **store on NFS**, Chameleon ...)
- Integration w/ EnOSlib (experiment orchestration)

Questions ?

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