Déploiements reproductibles dans le temps avec GNU Guix

Ludovic Courtès

Déploiements reproductibles dans le temps avec GNU Guix

Ludovic Courtès

mars 2021

Résumé

Pour la recherche scientifique comme pour d'autres domaines, on a souvent besoin de reproduire un environnement logiciel à l'identique non seulement sur différentes machines mais aussi à différents instants dans le temps. Docker et les machines virtuelles, qui sont souvent la solution choisie pour répondre à ce besoin, ont des limitations qu'il est facile de ne pas voir. Cet article illustre l'utilisation de GNU Guix pour des déploiements reproductibles, au bit près, dans l'espace et dans le temps.

Introduction

Comment relancer du code dans les conditions d'origine ? C'est une question qui se pose souvent dans le milieu de la recherche : pour vérifier des résultats scientifiques, il faut déjà être capable de relancer le code qui y a mené — on parle de *recherche reproductible*, ce qui est dans le fond un pléonasme. La question se pose aussi dans l'ingénierie logicielle : on a parfois besoin de reproduire le même environnement logiciel, même ancien, sur différentes machines, pour obtenir le même comportement ou chercher l'origine d'un défaut, ou simplement pour partager le même environnement de travail entre collègues.

GNU Guix est un outil de déploiement logiciel et un distribution GNU/Linux complète. Il y a beaucoup à en dire mais un de ses objectifs peut se résumer ainsi : combiner la transparence des gestionnaires de paquets comme `apt` avec la simplicité de déploiement à l'identique d'outils comme Docker. Mais puisque Docker est souvent le premier choix pour cet exercice, jetons y d'abord un œil.

Docker & co : une solution ?

Analyser l'état des logiciels déployés sur un système est difficile, encore plus si on a utilisé plusieurs outils : `apt`, `pip`, etc. Les outils comme Docker permettent de contourner la difficulté en « figeant » une image de l'état du système. Pour y parvenir, on écrit dans un `Dockerfile` la séquence de commandes permettant *a priori* d'atteindre l'état souhaité.

Le résultat est une *image* qui contient l'environnement logiciel souhaité et qu'on peut partager. La personne qui reçoit l'image est sûre de reproduire l'environnement logiciel à l'identique puisqu'elle dispose de tous les octets qui le composent.

Elle n'a en revanche *que* ces octets puisque la recette décrite par le `Dockerfile` n'est pas reproductible : l'effet des commandes exécutées dépend de l'état des dépôts des gestionnaires de paquets et autres services tiers utilisés par la recette. Impossible, donc, de vérifier qu'un `Dockerfile` est bien celui qui a mené à l'image qu'on utilise.

Quand bien même le `Dockerfile` décrirait un processus reproductible, il favorise un empilement de couches sans donner une vision globale dans la composition des paquets. Enfin, cette approche a d'autres inconvénients comme l'utilisation inefficace du stockage et la difficulté de s'assurer que chaque image contient les mises à jour de sécurité critiques.

Démarrer avec Guix

Que permet Guix ? Dans GLMF 194 de juin 2016, je donnais un aperçu de Guix qui reste d'actualité : Guix a énormément évolué en quatre ans (à peu près 60 000 changements !) mais les fondements restent les mêmes.

Guix peut s'utiliser comme une distribution à part entière, Guix System, ou alors comme un outil de déploiement par dessus une distribution existante et donnant accès à plus de 15 000 logiciels libres. Il a une interface de type « gestionnaire de paquets » : `guix install inkscape` fait ce que vous imaginez. Il a aussi une interface de « gestion d'environnement », à la VirtualEnv pour les personnes habituées à Python, mais accessible à tous les logiciels sans distinction de langage de programmation. Par exemple, pour un environnement Python avec NumPy, on peut lancer :

```
$ guix environment --ad-hoc python python-numpy -- \
       python3 -c "import numpy; print(numpy.version.version)"
1.17.3
```

La commande `guix environment` a créé un environnement contenant seulement Python et NumPy, a positionné `PATH` et `PYTHONPATH` pour faire référence à cet environnement, et y a lancé `python3`. De cette manière, l'environnement d'exécution de `python3` est parfaitement contrôlé.

Avec ça on a déjà de quoi reproduire un environnement logiciel. Mais en quoi est-ce différent de `apt` ou `yum` ?

Approche « fonctionnelle » et reproductibilité bit-à-bit

Faisons un aparté pour rappeler ce qui distingue Guix des outils de gestion de paquets « classiques », c'est son approche « fonctionnelle » héritée de Nix [1] (on peut dire que Nix et Guix sont deux outils de déploiement fonctionnels, de la même manière que `apt` et `yum` sont deux gestionnaires de paquets « impératifs », avec des similarités et des différences majeures). En résumé, un paquet dans Guix est définit par son graphe de dépendances *complet* et la manière dont il est construit — « complet » ici signifie que, à part le noyau du système d'exploitation, un paquet ne peut pas avoir d'autres dépendances que celles décrites par son graphe. La construction du paquet est donc vue comme une fonction pure comme on en voit en maths ou dans la programmation fonctionnelle avec des langages comme OCaml, Haskell ou Scheme : on lui passe des

1. `https://nixos.org/nix/`

arguments tels que du code source, un compilateur, des bibliothèques et elle retourne un paquet compilé.

Avec cette information, Guix a tout ce qu'il faut pour pouvoir (re)construire ses paquets. Le démon de compilation, `guix-daemon`, s'arrange pour que la construction ait lieu dans un environnement isolé du reste du système, un conteneur où seules les dépendances qui apparaissent dans la définition du paquet sont accessibles. Cela maximise les chances que la construction du paquet soit *reproductible bit à bit* : que je le construise sur ma machine ou sur la tienne, maintenant ou dans six mois, le résultat sera (sauf rares exceptions) le même au bit près. C'est à cette condition seulement qu'on peut vérifier que le binaire qu'on fait tourner correspond *vraiment* au code source qu'on a sous les yeux.

Le projet Guix est d'ailleurs impliqué dans l'action Reproducible Builds [2] aux côtés de nombreuses autres distributions dont Debian et NixOS.

La construction de paquets reproductible bit-à-bit et la possibilité de vérification qui en découle, c'est bien, mais ça n'a de sens que si il y a effectivement construction à partir du code source au départ. Guix va donc plus loin en éliminant les cas où les distributions partent habituellement d'un binaire pré-compilé, comme par exemple pour le compilateur du compilateur. C'est aujourd'hui la première distribution à être amorcée entièrement depuis du code source. Mais bon, ce sera le sujet d'un autre article ! [3]

Déclarer un environnement

On a vu que Guix permet de créer des environnements persistants à la volée avec `guix environment` ou de manière persistante avec `guix install`. Pour faciliter le partage, on peut aussi déclarer un environnement dans un fichier qu'on appelle un *manifeste* et qui ressemble à ça :

```
(specifications->manifest
 (list "python" "python-numpy" "gcc-toolchain@10"))
```

Il s'agit en fait d'un morceau de code Scheme, le langage fonctionnel de la famille Lisp dans lequel est écrit Guix et avec lequel on peut le configurer et l'étendre. Ce code liste les trois paquets qu'on a spécifiés, cherche leur définition et renvoie un objet de type `manifest`. Ce fichier, appelons le `manifeste.scm`, on peut le passer à la commande `guix package` pour qu'elle l'instancie, c'est-à-dire qu'elle installe précisément ces trois paquets et rien d'autre :

```
$ guix package -m manifeste.scm
Les paquets suivants seront installés :
   gcc-toolchain 10.2.0
   python        3.8.2
   python-numpy  1.17.3

[. . . ]
$ guix package -p /tmp/test --list-installed
gcc-toolchain   10.2.0  . . .
```

2. `https://reproducible-builds.org/`

3. En attendant, vous pouvez jeter un œil à `https://bootstrappable.org` et au blog de Guix pour en savoir plus.

```
python-numpy    1.17.3  . . .
python  3.8.2   . . .
```

On peut aussi le passer à `guix environment` pour lancer un *shell* contenant ces trois paquets (ici on rajoute `--container` pour que ce *shell* soit en plus isolé dans un conteneur) :

```
$ guix environment -m manifeste.scm --container
[env]$ python3 --version
Python 3.8.2
[env]$ ls /bin
sh: ls: command not found
[env]$ echo /bin/*
/bin/sh
```

On voit dans cet exemple qu'il n'y a vraiment rien d'autre dans cet environnement que les trois paquets spécifiés dans `manifeste.scm`, si ce n'est `/bin/sh`.

Ce manifeste on peut donc le mettre en gestion de version et le partager avec d'autres personnes qui pourront reproduire le même environnement. Mission accomplie ?

Pas tout à fait. Si on regarde de plus près, la version de Python que fournit Guix aujourd'hui, la 3.8.2, ce ne sera plus la même dans six mois. Autrement dit, si je passe `manifeste.scm` à un Guix post-COVID (fin 2021 ?), il va bien créer un environnement contenant mes trois paquets, mais ce seront des versions différentes et des dépendances différentes. Il me manque donc une information si je veux pouvoir recréer cet environnement plus tard.

Canaux et choix d'une révision

L'équivalent de `apt update`, c'est `guix pull`, qui met à jour Guix et les paquets qu'il fournit. Si je veux reproduire mon environnement du jour dans six mois, il faut que, en plus de mon manifeste, je connaisse la version précise de Guix que j'utilise. C'est cette information que nous donne `guix describe` :

```
$ guix describe
Génération 170  26 Dec 2020 16:45:39    (actuelle)
  guix 4969b51
    URL du dépôt : https://git.savannah.gnu.org/git/guix.git
    branche : master
    commit : 4969b51d175497bfcc354c91803e9d70542b7113
```

Cette commande m'affiche la révision (*commit*) précise de Guix que je suis en train d'utiliser. Si je veux obtenir cette même révision sur une autre machine, ou dans six mois, je peux la demander explicitement à `guix pull` :

```
$ guix pull --commit=4969b51d17
```

À l'issue de cette opération, `guix describe` m'affichera la même révision que ci-dessus. Je pourrai passer `manifeste.scm` à `guix environment` ou `guix install` et avoir la garantie d'obtenir le même environnement logiciel, bit à bit, que précédemment.

On peut même faire plus simple et stocker la sortie de `guix describe` dans un format consommable par `guix pull` :

```
$ guix describe -f channels > canaux.scm
```

Ce fichier, je peux lui aussi le mettre en gestion de version et le partager, pour plus tard le passer à `guix pull` :

```
$ guix pull --channels=canaux.scm
```

Je peux comme ça figer la révision de Guix qui me convient, pour pouvoir la redéployer sur d'autres machines ou plus tard.

En résumé, je peux décrire complètement mon environnement logiciel pour pouvoir le reproduire bit-à-bit à n'importe quel moment avec ces deux fichiers :

1. `manifeste.scm`, qui liste les paquets que je veux dans mon environnement ;
2. `canaux.scm` qui définit la révision de Guix à utiliser.

Au fait, pourquoi parle-t-on de « canaux » ? Un canal, dans le jargon Guix, c'est en gros un dépôt Git qui contient des définitions de paquet. On utilise toujours au moins un canal, le canal `guix` que nous montre la commande `guix describe` ci-dessus : c'est celui qui contient Guix même, ses commandes et ses paquets. Mais on peut aussi, dans le fichier `canaux.scm` ou dans `~/.config/guix/channels.scm`, lister des canaux supplémentaires fournissant d'autres paquets. Pour chacun d'eux, on peut demander une révision spécifique comme on l'a fait ici, ou la dernière révision d'une branche.

Comme on le voit, Guix se focalise sur le chemin qui mène du source au binaire. Le gros avantage, par rapport à un `Dockerfile` ou par rapport à une image préconstruite, c'est la reproductibilité et la transparence. Je sais exactement d'où proviennent les binaires que je lance, je peux choisir de les compiler moi même, je peux ausculter les versions, options de compilation, etc. de chaque paquet. Et puis, comme Guix dispose de toute cette information, je peux aussi construire des variantes de ces paquets, par exemple en utilisant une option de « transformation de paquets » en ligne de commande.

C'est aussi fondamentalement différent de l'archivage de binaires pré-compilés sur `snapshot.debian.org` ou sur `anaconda.org` (pour le gestionnaire de paquets CONDA) : on peut toujours recompiler le paquet, et donc vérifier que le binaire correspond au source.

Voyager dans le temps

Avec `guix pull --commit` et `guix pull --channels` on peut donc, tel McFly au volant de sa DeLorean dans *Retour vers le futur*, voyager dans le temps. Un inconvénient est que `guix pull` modifie ma commande `guix`, et même si je peux toujours revenir à la révision que j'utilisais avant avec `guix pull --roll-back`, ça peut s'avérer peu pratique. C'est pour cette raison qu'on a créé `guix time-machine`, la commande deux en un : elle voyage jusqu'aux canaux demandés et de là, elle exécute la commande souhaitée. Si on reprend mes deux fichiers précédents, on peut donc déployer mon environnement directement avec :

```
$ guix time-machine --channels=canaux.scm -- \
       environment -m manifeste.scm
```

Les deux tirets sont suivis par la commande à exécuter, celle qu'on aurait normalement lancée après `guix pull`. J'aurais aussi pu utiliser la commande `package` :

```
$ guix time-machine --channels=canaux.scm -- \
       package -m manifeste.scm
```

J'aurais aussi pu passer un commit et utiliser `guix install` (pratique pour installer rapidement une vieille version) :

```
$ guix time-machine --commit=4969b51d17 -- install gimp
```

Ici on part du principe qu'on connaît déjà la révision qu'on souhaite utiliser parce qu'on l'utilisait avant et qu'on l'a sauvegardée avec `guix describe`. Parfois, on ne connaît pas la révision à l'avance mais on souhaiterait trouver une révision qui contient une version précise d'un paquet. Le Guix Data Service [4], qui stocke les données sur les paquets fournis par chaque révision de Guix, permet de répondre à ce genre de question. Un stage Outreachy actuellement en cours porte également sur le développement d'une commande pour naviguer l'historique des versions de paquets. À suivre !

Et le code source ?

Il y a quand même une hypothèse implicite qui est faite ici : que le code source des paquets sera toujours disponible quand on voyagera, plus tard, dans le passé, et que Guix essayera de recompiler des paquets. Parce qu'en effet, une définition de paquet commence par spécifier l'URL du code source et son condensé (*hash*) :

```
(define-public hello
  (package
    (name "hello")
    (version "2.10")
    (source (origin
              (method url-fetch)
              (uri (string-append "mirror://gnu/hello/hello-"
                                  version ".tar.gz"))
              (sha256
               (base32
                "0ssi1wpaf7plaswqqjwigppsg5f. . . "))))
    ;; . . .
    ))
```

Ou de manière similaire pour du code issu d'un dépôt de gestion de version :

```
(source (origin
          (method git-fetch)
          (uri (git-reference
                 (url "https://example.org/hello.git")
                 (commit "v2.10")))
          (sha256
           (base32
            "1c8l7829b5yx8wdc0mrhzjfwb6h9. . . "))))
```

4. https://data.guix.gnu.org

Quand on demande un paquet, Guix va automatiquement chercher des binaires pré-compilés comme « substituts » à une compilation locale, si ceux-ci sont disponibles sur les serveurs choisis. Quand ces substituts ne sont pas disponibles, ou quand on a demandé explicitement à ne pas y recourir, Guix construit les paquets depuis le code source. Que se passe-t-il si le code source a disparu ?

Il y a plusieurs niveaux de protection. D'abord, les URL `mirror://` lui permettent de parcourir tous les miroirs du site en question (gnu.org dans l'exemple ci-dessus) au cas où l'un d'eux ne disposerait plus du code. Ensuite, le serveur officiel du projet, `ci.guix.gnu.org`, préserve des copies du source (archives tar, *checkouts* Git, etc.) pendant un certain temps ; elles sont automatiquement téléchargées si disponibles, mais le projet ne peut pas garantir qu'elles resteront éternellement.

C'est pour cette raison que le lancement de Software Heritage [5] en 2016 a été accueilli par le projet comme un soulagement ! Software Heritage (SWH) a pour mission rien de moins que d'archiver tout le code source public disponible. Du point de vue de Guix, c'était le chaînon manquant en amont pour la reproductibilité d'environnements logiciels. SWH archive ainsi depuis plus de quatre ans tout l'historique des dépôts de gestion de version publics et tout le contenu des archives de code. Le site `archive.softwareheritage.org` fournit une interface HTTP/JSON pour y accéder.

Avec l'aide de l'équipe SWH, nous avons d'abord intégré dans Guix la possibilité de récupérer le code sur SWH quand les autres méthodes ont échoué [6]. Comme le paquet Guix fournit un condensé SHA256 du code source attendu, on est sûr de récupérer le « bon » code.

Il restait une inconnue : cela ne fonctionne qu'à condition que SWH archive bien l'ensemble du code auquel Guix fait référence. Nous avons d'abord étendu l'outil de vérification de paquet `guix lint` pour qu'il demande automatiquement à SWH l'archivage du code des paquets donnés.

L'entreprise Tweag, impliquée dans le développement de Nix, a ensuite travaillé avec SWH ; le résultat est que NixOS et Guix publient dorénavant un fichier JSON listant les URL du code source de leurs paquets et SWH le récupère automatiquement à intervalles réguliers pour lancer l'archivage de tout ce code.

Nous travaillons encore sur un point pour que l'intégration soit parfaite. SWH archive le *contenu* des archives de type tar, pas les archives elles-mêmes. L'archive SWH est « adressée par le contenu », c'est-à-dire qu'on peut obtenir un code en donnant le condensé de son contenu. Or, une définition de paquet Guix donne généralement le condensé de l'archive elle-même (le fichier tar.gz) alors que SWH ne retient que le condensé du contenu de l'archive (le répertoire produit par extraction de l'archive).

Il nous faut donc créer un pont entre les archives auxquelles Guix fait référence et leur contenu, pour que Guix puisse récupérer le contenu de ces archives sur SWH. Une solution a été proposée par une personne de l'équipe Guix et pourrait bien être testée rapidement [7], ce qui réglerait le problème.

Préhistoire

Un aspect du voyage dans le temps que je n'ai pas mentionné, c'est la question de la préhistoire : Guix date de 2012, le mécanisme qui lui permet de reproduire une

5. `https://www.softwareheritage.org/`
6. `https://guix.gnu.org/blog/2019/connecting-reproducible-deployment-to-a-long-term-source-code-archive/`
7. `https://issues.guix.gnu.org/42162`

version passée ou future n'est stable que depuis la version 0.15.0 (juillet 2018) et on ne peut donc pas faire de `time-machine` au-delà de cette limite.

La revue en ligne ReScience a justement organisé récemment un défi intéressant, le *Ten Years Reproducibility Challenge*, dont le but est de reproduire les résultats parus dans un article scientifique qu'on a soi-même écrit il y a dix ans ou plus [8]. Pour y participer, j'avais moi-même besoin d'une pile logicielle datant de 2006 et que Guix n'a donc jamais fournie. C'est comme ça qu'avec d'autres nous avons créé Guix-Past, un canal fournissant exclusivement des logiciels *vintage* [9]. On y trouve des vieilles versions de Python et de logiciels associés, de Perl, de Boost, des « Autotools » GNU, de diverses bibliothèques et de logiciels scientifiques.

En rajoutant ce canal à votre `~/.config/guix/channels.scm`, vous pouvez démarrer un Python 2.4 ancré dans la pile GNU/Linux de 2021 ! Encore une fois, c'est réalisé sans trucage : Guix vous garantit que vous pouvez recompiler ces vieux logiciels ici et maintenant.

Interopérabilité

J'ai eu beau critiquer l'approche de Docker, il faut reconnaître que la possibilité d'échanger des images binaires prêtes à l'emploi est bien pratique — tout du moins dans un souci d'interopérabilité avec les personnes n'utilisant pas (pas encore ?) Guix, ou pour de l'archivage à long terme.

C'est dans cette optique qu'a été développé `guix pack`, qui permet de créer un « lot applicatif » : une image binaire contenant des paquets et tout ce dont ils dépendent à l'exécution. Par exemple, je peux créer une archive tar.gz contenant Python, NumPy et toutes leurs dépendances avec cette commande :

```
$ guix pack -RR -S /bin=bin python python-numpy
. . .
/gnu/store/. . . -tarball-pack.tar.gz
```

Le fichier `/gnu/store/. . . -tarball-pack.tar.gz` que retourne la commande contient tous les fichiers nécessaires à l'exécution de Python et NumPy. Compte tenu de l'option `-S /bin=bin`, l'archive contient également un lien symbolique `/bin` pointant vers le sous-répertoire `bin` de ces deux paquets. L'option `-RR` spécifie que les exécutables contenus dans l'archive doivent être *relogeables*, c'est-à-dire exécutables depuis n'importe quel répertoire. Je peux donc aller sur n'importe quel machine faisant tourner le noyau Linux, extraire l'archive, et démarrer Python :

```
$ tar xf tarball-pack.tar.gz
$ ./bin/python
```

Magique ! Sous le capot, le binaire `python` créée automatiquement un *user namespace* dans lequel il rend le répertoire `/gnu/store` visible (c'est l'endroit où sont normalement stockés tous les résultats de compilation, donc les binaires eux-mêmes) ; si les *user namespaces* ne sont pas pris en charge sur la machine, il utilise le logiciel PRoot qui est embarqué pour arriver au même résultat, au prix d'une baisse de performances.

L'option `-f` permet de choisir un autre format, comme le format d'image Docker :

8. https://rescience.github.io/ten-years/
9. https://gitlab.inria.fr/guix-hpc/guix-past

```
$ guix pack -f docker -S /bin=bin --save-provenance \
           -m manifeste.scm
. . .
/gnu/store/. . . -docker-pack.tar.gz
```

Ici on créée une image Docker contenant les paquets listés dans mon fichier `manifeste.scm`, le même que celui discuté plus haut (et c'est probablement plus simple que d'écrire un `Dockerfile`). Je peux passer le fichier que me retourne la commande à `docker load` puis lancer le code en utilisant le nom que `guix pack` a généré pour l'image :

```
$ docker load < docker-pack.tar.gz
$ docker run -ti python-python-numpy-gcc-toolchain \
         /bin/python
```

En rajoutant `--save-provenance`, les informations sur les canaux que j'utilise — les mêmes informations que celles renvoyées par `guix describe` — sont stockées dans l'image. De cette manière, quelqu'un qui reçoit l'image a également toute l'information nécessaire pour la reproduire avec Guix.

Conclusion

Et voilà ! On a vu l'essentiel de ce que permet GNU Guix en termes de reproductibilité d'environnement logiciels dans l'espace et dans le temps. Je me suis concentré sur les paquets mais Guix sait aussi gérer des systèmes complets et donc toutes ces bonnes propriétés sont valables au niveau du système ; on se le garde sous le coude pour un prochain article.

L'approche de Guix est radicale : être capable de recompiler une pile logicielle à l'identique, au bit près, à différents moments dans le temps, avec une traçabilité parfaite. C'est aux antipodes de l'approche qui consiste à générer une image Docker ou de machine virtuelle une fois pour toutes. C'est une approche plus ambitieuse, donc plus difficile à mettre en œuvre, mais je suis convaincu que les bénéfices en termes de transparence, et donc de sécurité et de liberté des usagers, en valent la chandelle.

Pour aller plus loin vous pouvez jeter un œil au manuel de Guix et à sa traduction en français [10], l'essayer vous-même, et venir discuter avec nous sur les listes de diffusion ou le canal IRC du projet. La dernière version en date est la 1.2.0, sortie en novembre 2020. Nous serions ravi·es d'accepter vos contributions pour la prochaine version, ou peut-être même dans un des canaux tiers comme Guix-Past !

Mise à disposition de cet article

10. https://guix.gnu.org/manual/fr/html_node/
11. https://creativecommons.org/licenses/by-nc-nd/4.0/deed.fr