Gestion de paquets sûre et flexible avec GNU Guix

Ludovic Courtès

April 5, 2016

Mots-clés : Gestion de paquets, distribution, programmation fonctionnelle, GNU

1 Résumé

Avril 2016. Linux Weekly News dénombre[1] plus de 500 distributions GNU/Linux et y voit une « célébration de la diversité ». Avec ça, le problème de la distribution de logiciels a été largement exploré, et probablement largement résolu, se dit on. Et pourtant ! Cet article traite d'un énième gestionnaire de paquet, GNU Guix[2], qui fournit mises à jour transactionnels, retours en arrières, et est extensible et personnalisable à souhait.

2 Intro (non numérotée en vrai)

Les distributions « classiques » ont certaines limitations, comme l'impossibilité de revenir en arrière après une mise à jour ou la difficulté de reproduire un environnement logiciel exact. La profusion de gestionnaires de paquets annexes ajoute à la confusion, et l'utilisation de Docker contourne ces difficultés sans les corriger. GNU Guix met en œuvre une gestion des paquets *fonctionnelle* qui entend résoudre certains de ces problèmes.

3 Le problème

3.1 Mises à jour dangereuses

Avec une distribution GNU/Linux « classique », une mise à jour est toujours quitte ou double : il ne vaut mieux pas qu'une coupure électrique intervienne

[1] `https://lwn.net/Distributions/`
[2] `https://gnu.org/software/guix/`

pendant la mise à jour (on risque de se retrouver avec un système inutilisable), ou qu'un des nouveaux paquets ne fonctionne pas (difficile de revenir en arrière).

3.2 Impossible de connaître/reproduire l'état d'un système

Une fois que l'on a une machine avec une distribution classique qui fonctionne bien, on aimerait pouvoir *reproduire* son état, c'est-à-dire l'ensemble des paquets installés et la configuration associée, sur une autre machine par exemple. C'est chose difficile car on n'est jamais sûr de pouvoir réinstaller les mêmes paquets aux mêmes versions, et parce qu'une partie de la configuration du système échappe complètement au gestionnaire de paquets. Le fait qu'il soit devenu courant de combiner beaucoup de gestionnaires de paquets (Bower, Cabal, CPAN, npm, pip, etc.) rend le contrôle de l'environnement logiciel plus difficile encore.

3.3 Docker : une vraie solution ?

Docker permet de contourner la difficulté en permettant de « figer » une image de l'état du système. Pour y parvenir, on va stocker dans un `Dockerfile` la séquence de commandes permettant *a priori* d'atteindre l'état souhaité.

Cette approche risque de ne pas être reproductible, puisque l'effet des commandes dépend de l'état des dépôts de code sources des gestionnaires de paquets utilisés dans l'image Docker. Elle favorise un empilement de couches sans donner une vision globale dans la composition des paquets. Enfin, elle 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.

4 La gestion de paquets « fonctionnelle »

Aux alentours de 2004, Eelco Dolstra a commencé son travail de thèse sur Nix[3], un gestionnaire de paquets *fonctionnel*. Ici « fonctionnel » fait référence non pas au fait qu'il fonctionne (bien qu'il fonctionne), mais au paradigme de gestion de paquets, qui s'inspire de la programmation fonctionnelle telle que mise en œuvre par des langages comme OCaml, Haskell ou Scheme.

L'idée est de voir chaque paquet comme une valeur immuable, résultat de l'application d'une fonction de compilation à un ensemble d'arguments. Par

[3] `https://nixos.org/nix/`

exemple, le binaire du logiciel GIMP est vu comme le résultat d'appliquer une fonction qui lance `./configure && make && make install` à un ensemble d'entrées : le source de GIMP, GCC, la bibliothèque standard du C (libc), la bibliothèque GTK+, etc. À son tour, GTK+ est le résultat de cette fonction appliquée à d'autres arguments. De cette façon on exprime un graphe de dépendances comme celui de la figure 1.

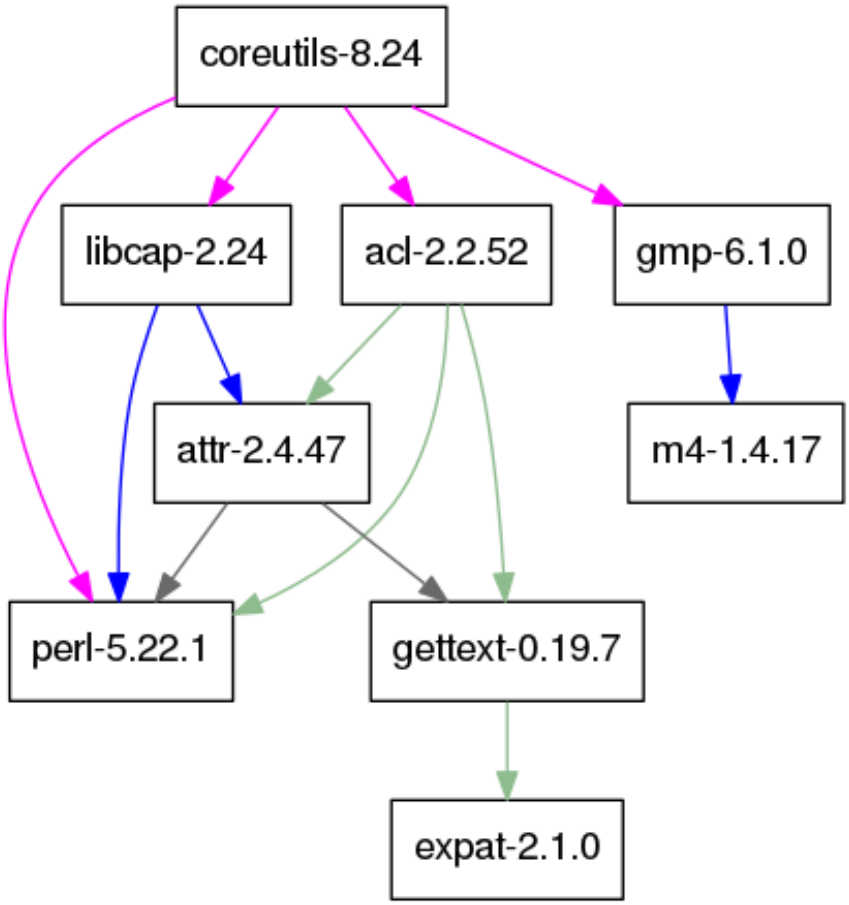

Figure 1: Graphe de dépendances à la compilation des outils de base GNU (Coreutils) produit par la commande `guix graph coreutils`.

GNU Guix a été créé en 2013 pour reprendre les fondements de Nix, mais en fournissant une interface de programmation unifiée et embarquée dans le langage Scheme, un langage de programmation fonctionnelle

générique, mis en œuvre par GNU Guile[4]. De cette façon, Guix a accès à tout Guile—compilateur, dévermineur, bibliothèques, environnement de développement, etc.—et tout Guix est accessible en Scheme.

Autrement dit, Guix est une bibliothèque Scheme comme une autre, et la distribution tout entière est une bibliothèque où chaque paquet est un objet Scheme. Le but recherché, et atteint, est que l'on puisse simplement écrire des fonctions qui manipulent des paquets, écrire des interfaces utilisateurs et autres applications qui se servent de Guix. Guix fournit ainsi un langage d'empaquetage universel (s'applique à des paquets C mais aussi PyPI, RubyGems, ELPA, etc.), plusieurs interfaces utilisateurs (ligne de commande, Emacs, et Web), un outil de gestion d'environnements de développement (`guix environment`, comme Virtualenv, rvm, etc. mais pour tout), un outil de vérification des paquets (`guix lint`), un outil de mise à jour des recettes de paquets (`guix refresh`), et d'autres.

Mais l'enjeu de cette approche est, plus généralement, de fournir un système transparent et bidouillable que les usagers puissent s'approprier plus facilement, pour exercer la liberté n°1 que le logiciel libre leur donne[5].

5 Installation

Il est possible d'installer Guix comme un gestionnaire de paquets supplémentaire (un de plus !) sur votre distribution GNU/Linux. Il cohabitera pacifiquement avec la distribution et sans interférence : Guix installe ses affaires dans `/gnu/store` comme nous le verrons plus bas, et vous pouvez à tout moment ajouter ou d'enlever les paquets installés avec Guix de `PATH` et autres variables.

L'installation peut se faire de plusieurs façons comme détaillé dans le manuel de Guix[6], le plus simple étant d'utiliser les binaires précompilés fournis par le projet.

Le démon `guix-daemon` prend en charge la compilation des paquets et/ou le téléchargement de binaires précompilés provenant de sources autorisées. Pour que la compilation de paquets puisse être vue comme une fonction « pure » et soit reproductible, ce démon s'assure que la compilation est effectuée dans un environnement isolé, un *conteneur*.

[4] `https://gnu.org/software/guile/`

[5] `https://www.gnu.org/philosophy/free-sw.fr.html`

[6] `https://gnu.org/software/guix/download/`

6 En avant !

Une fois `guix-daemon` démarré, on peut déjà lancer une construction de paquet :

```
$ guix build hello
Le fichier suivant sera téléchargé:
   /gnu/store/zby49aqfbd9w9br4l52mvb3y6f9vfv22-hello-2.10

Found valid signature for /gnu/store/zby49aqfbd9w9br4l52mvb3y6f9vfv22-hello-2.10
From https://mirror.hydra.gnu.org/nar/...-hello-2.10
Downloading zby49a...-hello-2.10 (170KiB installed)...
 https://mirror.hydra.gnu.org/nar/...-hello-2.10 737KiB/s 00:00 | 49KiB transferred
/gnu/store/zby49aqfbd9w9br4l52mvb3y6f9vfv22-hello-2.10
$ /gnu/store/zby49aqfbd9w9br4l52mvb3y6f9vfv22-hello-2.10/bin/hello
Bonjour, le monde !
```

Ce qu'on voit ici, c'est qu'à une compilation du paquet GNU Hello, `guix build` a *substitué* un binaire pré-compilé téléchargé directement depuis `mirror.hydra.gnu.org`. Le résultat est ce long nom de répertoire en `/gnu/store` qui contient effectivement la commande `hello`.

Tout ce que produit Guix arrive dans le répertoire `/gnu/store`, qu'on appelle *l'entrepôt* (le *store* en anglais). Par exemple, `hello` a notamment une dépendance à l'exécution sur la bibliothèque standard du langage C (libc), qui elle est aussi dans l'entrepôt, comme le montre cette commande qui liste les dépendances à l'exécution :

```
$ guix gc --references /gnu/store/zby49aqfbd9w9br4l52mvb3y6f9vfv22-hello-2.10
/gnu/store/8m00x5x8ykmar27s9248cmhnkdb2n54a-glibc-2.22
/gnu/store/v39bh3ln3ncnzhyw0kd12d46kww9747v-gcc-4.9.3-lib
/gnu/store/zby49aqfbd9w9br4l52mvb3y6f9vfv22-hello-2.10
```

Cette longue chaîne en base32 dans les chemins ci-dessus est en fait le condensé SHA256 de *toutes* les dépendances utilisées à la compilation pour produire ce résultat. Dans le cas de Hello, les dépendances à la compilation sont : le source de Hello, le script de compilation, mais aussi la libc, le compilateur, Bash, coreutils, sed, grep, awk, etc. Vraiment toutes les dépendances ! De cette manière, on a vraiment une correspondance directe entre le source, y compris les outils de compilation, et le binaire produit — c'est une formalisation de la notion de *Corresponding Source* telle que décrite dans la GNU GPL.

Cette correspondance source/binaire est cruciale. Elle signifie que les usagers n'ont pas à faire confiance aveuglément à un fournisseur de binaire : les usagers peuvent à tout moment compiler localement et vérifier qu'ils obtiennent le même résultat, à l'octet près, que le fournisseur de binaires. C'est exactement ce que vérifie la commande `guix challenge`.

7 Les profils

Évidemment, on n'a pas vraiment envie de taper ces chemins à la main. La commande `guix package` permet à chaque usager (pas besoin d'être `root`) de maintenir des *profils* où sont installés des paquets. Il suffit de rajouter un profil dans `PATH` et ses paquets deviennent disponibles. Par exemple, pour installer Emacs et Vim (on ne sait jamais) dans son profil par défaut, `~/.guix-profile`, on fait simplement :

```
$ guix package --install emacs vim
Les paquets suivants seront installés:
   vim  7.4     /gnu/store/...-vim-7.4
   emacs        24.5    /gnu/store/...-emacs-24.5

Les dérivations suivantes seront compilées:
   /gnu/store/...-profile.drv
   /gnu/store/...-gtk-icon-themes.drv
   /gnu/store/...-ca-certificate-bundle.drv
   /gnu/store/...-info-dir.drv
Le fichier suivant sera téléchargé:
   /gnu/store/...-vim-7.4

[...]

2 paquets dans le profil
Il pourrait être nécessaire de définir les variables d'environnement suivantes:
   export PATH="/home/alice/.guix-profile/bin"
   export INFOPATH="/home/alice/.guix-profile/share/info"
$ guix package --list-installed
emacs   24.5    out     /gnu/store/...-emacs-24.5
vim     7.4     out     /gnu/store/...-vim-7.4
```

Guix est attentionné et nous indique même les variables d'environnement à définir pour pouvoir utiliser les paquets installés. Pour que ces vari-

ables soient automatiquement définies, on peut rajouter cette ligne dans `~/.bash_profile` :

```
GUIX_PROFILE="$HOME/.guix-profile" . "$GUIX_PROFILE/etc/profile"
```

Il faut noter que cette opération est *transactionnelle* : on peut taper Ctrl-C à tout moment, et soit à la fois Emacs et Vim seront installés, soit aucun ne le sera. C'est aussi le cas pour des transactions plus complexes :

```
$ guix package -r vim -i nano
Le paquet suivant sera supprimé:
   vim  7.4     /gnu/store/...-vim-7.4

Le paquet suivant sera installé:
   nano 2.5.3   /gnu/store/...-nano-2.5.3

[...]

2 paquets dans le profil
```

Chaque transaction donne lieu a une nouvelle *génération* du profil :

```
$ guix package --list-generations
Génération 1    Mar 30 2016 14:29:05
  emacs 24.5    out     /gnu/store/...-emacs-24.5
  vim   7.4     out     /gnu/store/...-vim-7.4

Génération 2    Mar 30 2016 14:39:02    (actuel)
  emacs 24.5    out     /gnu/store/...-emacs-24.5
  nano  2.5.3   out     /gnu/store/...-nano-2.5.3
```

On peut à tout moment basculer vers une autre génération, la précédente par exemple :

```
$ guix package --roll-back
switched from generation 2 to 1
```

Ce mécanisme vaut aussi pour les mises à jour (avec `--upgrade`)… et c'est très rassurant !

8 Maîtriser ses environnements logiciels

Un profil n'est rien d'autre qu'une forêt de liens symboliques :

```
$ readlink -f ~/.guix-profile
/gnu/store/...-profile
$ readlink ~/.guix-profile/bin/emacs
/gnu/store/...-emacs-24.5/bin/emacs
```

On peut donc en créer autant qu'on veut, et Guix saura nous dire quelles sont les variables d'environment qui vont bien :

```
$ guix package -p ~/dev-python -i python@2.7 python2-numpy
Les paquets suivants seront installés:
   python2-numpy        1.10.4  /gnu/store/...-python2-numpy-1.10.4
   python       2.7.10  /gnu/store/...-python-2.7.10

2 paquets dans le profil
Il pourrait être nécessaire de définir les variables d'environnement suivantes:
   export PATH="/home/ludo/dev-python/bin"
   export PYTHONPATH="/home/ludo/dev-python/lib/python2.7/site-packages"
$ guix package -p ~/dev-python --search-paths
export PATH="/home/ludo/dev-python/bin"
export PYTHONPATH="/home/ludo/dev-python/lib/python2.7/site-packages"
$ eval `guix package -p ~/dev-python --search-paths`
$ python -c "import numpy; print(numpy.version.version)"
1.10.4
```

L'outil `guix environment` permet de créer des environnements de développement temporaires, à la volée. Par exemple, pour un environnement Python 2.x avec Numpy comme ci-dessus, on pourrait simplement faire :

```
$ guix environment --ad-hoc python@2 python2-numpy -- \
    python -c "import numpy; print(numpy.version.version)"
1.10.4
```

L'outil peut aussi nous mettre dans l'environnement de développement d'un logiciel spécifique. Par exemple, quelqu'un voulant bidouiller GIMP peut récupérer le source puis se mettre dans un environnement d'où on pourra recompiler la bête :

```
$ tar xf ‘guix build --source gimp‘
$ cd gimp-2.8.14
$ guix environment gimp --container
[env]# echo $PATH
/gnu/store/...-profile/bin:/gnu/store/...-profile/sbin
[env]# echo $C_INCLUDE_PATH
/gnu/store/...-profile/include
[env]# echo $PKG_CONFIG_PATH
/gnu/store/...-profile/lib/pkgconfig
[env]# ./configure && make
```

Ici `guix environment` a démarré un Bash dans lequel tous les paquets nécessaires pour compiler GIMP sont disponibles, et où toutes les variables requises sont définies. Plus simple que de le faire à la main, et sans interférence sur le reste du système !

L'option `--container` est facultative ; elle permet de créer l'environnement dans un conteneur isolé du reste du système, où seuls une partie de `/gnu/store` et le répertoire courant sont visibles, grâce à l'utilisation des *user namespaces* du noyau. Cela garantit d'avoir un environnement « propre » et isolé.

9 Bidouiller la distrib'

Guix est conçu pour faciliter la bidouille. En ligne de commande, on peut déjà construire ou installer un paquet en précisant un code source différent et/ou des dépendances différentes :

```
# Compile une "release candidate" de Emacs.
$ guix build emacs --with-source=./emacs-25.1rc2.tar.gz

# Recompile Git en remplaçant OpenSSL par LibreSSL dans tout
# son arbre de dépendances.
$ guix build git --with-input=openssl=libressl
```

Puisque toutes les structures de données et interfaces de programmation de Guix sont exposées, on peut aussi définir des variantes de paquets existants. Par exemple, pour créer une variante de Emacs qui ne dépende pas de D-Bus, on peut définir une variable `emacs-sans-dbus` dont la valeur est un paquet qui hérite du paquet `emacs` mais retire la dépendance sur D-Bus, puis ajouter le fichier à `GUIX_PACKAGE_PATH` :

```
$ cat > /tmp/my-emacs.scm <<EOF
(define-module (my-emacs)
  #:use-module (guix packages)
  #:use-module (gnu packages emacs)
  #:use-module (srfi srfi-1))  ;manipulation de listes

(define-public emacs-sans-dbus
  (package (inherit emacs)
    (name "emacs-sans-dbus")
    (inputs (alist-delete "dbus" (package-inputs emacs)))))
EOF

$ export GUIX_PACKAGE_PATH=/tmp
$ guix package --list-available=emacs
emacs   24.5    out     gnu/packages/emacs.scm:69:2
[...]
emacs-sans-dbus 24.5    out     my-emacs.scm:7:2
$ guix build emacs-sans-dbus --dry-run
La dérivation suivante serait compilée:
   /gnu/store/...-emacs-sans-dbus-24.5.drv
```

Les commandes ont automatiquement pris en compte notre Emacs personnalisé, et celui-ci va magiquement suivre les changements et mises à jour faites au paquet `emacs` de la distribution, avec juste notre modification.

Les usagers peuvent donc facilement maintenir un dépôt de paquets privé avec leurs personnalisations, à la *personal package archive* (PPA), ou encore publier leurs propres définitions de paquets. Un aspect que nous n'avons qu'effleuré ci-dessus est la « programmabilité » : on peut par exemple écrire des fonctions qui renvoient des paquets en fonction des paramètres, ou encore réécrire le graphe de dépendance d'un paquet donné, comme le fait l'option `--with-input` ci-dessus. La frontière entre utilisation et développement de la distribution est floue !

10 Conclusion

GNU Guix permet à des utilisateurs non privilégiés d'installer des paquets de façon transactionnelle, de créer des environnements logiciels contrôlés, et de personnaliser la distribution. Guix fournit actuellement plus de 3200 logiciels libres. Guix et GuixSD sont encore considérés en version beta mais s'approchent dangereusement de la 1.0. La dernière version est sortie fin

mars 2016, fruit du travail d'une cinquantaine de personnes — rejoignez nous ! Dans un prochain article, nous verrons comment cela se généralise à une distribution tout entière avec GuixSD.