Administration système reproductible avec GNU Guix

Julien Lepiller
Inria Rennes
263, avenue du général Leclerc
35042 RENNES Cedex - France

Résumé

GNU Guix est un gestionnaire de paquets transactionnel et sans état. Il permet de gérer un ou plusieurs profils, chacun avec son propre historique, de revenir en arrière ou en avant et prend aussi en charge les mises à jour, de manière atomique. En plus de cela, GNU Guix fournit des constructions reproductibles.

Guix est aussi un gestionnaire d'environnements logiciels. Cela signifie qu'il peut utiliser les paquets qu'il connaît pour créer des environnements logiciels, sous différentes formes : des environnements temporaires dans un seul shell, ou permanents comme avec pypi. Ils peuvent être isolés dans un conteneur Linux ou non, voire exportés dans des conteneurs docker, singularity, ou de simples archives. Guix peut même générer des systèmes d'exploitations complets, dans un conteneur docker, une machine virtuelle ou directement sur le métal.

Quel que soit le type d'environnement choisi, comme pour les paquets, sa création sera reproductible, sans état et transactionnelle. En plus, la configuration des environnements et des systèmes est entièrement déclarative, tout en permettant la puissance d'expression d'un vrai langage de programmation.

Mots-clefs

Reproductibilité, Administration Système, …

1 Introduction

Sur une distribution *GNU/Linux* « classique », on trouve un gestionnaire de paquets qui permet de mettre à jour et de personnaliser le système. Ces opérations sont toujours quitte ou double : il ne vaut mieux pas qu'une coupure de courant ait lieu en plein milieu d'une mise à jour, sans quoi notre système se retrouve dans un état intermédiaire, semi-installé. Et si la mise à jour se passe bien, il reste toujours la possibilité qu'un nouveau paquet introduise de nouveaux bogues. Il nous est alors difficile de revenir en arrière.

Un autre problème de la gestion « classique » des paquets est celui de l'unicité des paquets. Si deux applications sont intéressantes, mais qu'elles ont besoin chacune d'une

version différente d'un même paquet, qui installe des fichiers du même nom, alors il faudra choisir l'une ou l'autre, mais pas les deux.

Les différentes communautés de langages de programmation préfèrent aussi pour la plupart utiliser des gestionnaires de paquets dédiés, tels *bower*, *cabal*, *cpan*, *npm* ou encore *pip*. Ces gestionnaires de paquets apportent une réponse partielle aux problèmes posés par le gestionnaire de paquets de votre système : les paquets y sont acceptés plus vite, peuvent être installés dans des environnements séparés qui permettent aux auteurs des logiciels de choisir les versions de leurs dépendances qui fonctionnent bien.

Cependant, ces gestionnaires-là ont leurs propres limites : ce sont des outils supplémentaires qui contribuent à l'état du système, qui n'ont pas les mêmes standards en terme d'assurance qualité et peuvent interagir négativement avec le système. Certains paquets peuvent aussi faire des hypothèses erronées sur la disponibilité de telle ou telle version d'un compilateur ou d'un outil.

Parallèlement à ces problèmes, nous n'avons pas encore mentionné une propriété importante : la reproductibilité, qui signifie que deux constructions d'un même paquet résultent en le même paquet binaire, bit-à-bit, et par extension, qu'avec les mêmes étapes, on peut arriver à deux systèmes identiques sur plusieurs machines.

2 Gestion des paquets avec GNU Guix

La gestion des paquets avec *GNU Guix* est particulière, et assez différente de la gestion des paquets avec d'autres systèmes comme *apt* ou *dnf* : le gestionnaire de paquets utilise la modèle de gestion de paquets dite fonctionnelle (comme la programmation, pas uniquement parce que ça marche) initié par Eelco Dolstra dans sa thèse [1]. Ce paradigme de la gestion des paquets considère chaque construction de paquet comme une fonction : avec les mêmes entrées (dépendances, sources, recette), on obtient les mêmes sorties (le paquet compilé), mais si l'une des entrées change, alors la sortie est aussi différente. Ce sont ces travaux qui ont mené Eelco Dolstra à créer la première distribution fonctionnelle, *NixOS*. Bien que les deux soient très apparentés, *NixOS* introduit le langage d'expression *nix*, tandis que *GNU Guix* réutilise un langage existant, *Guile* (de la famille des langages *Scheme*, eux-mêmes dérivés du *Lisp*). *GNU Guix* s'inscrit donc dans un écosystème de bibliothèques et de programmes existants et propose un environnement de programmation unifié pour définir des paquets, des systèmes et interagir avec son *API*. Mais la différence essentielle est d'ordre philosophique plus que technique : *GNU Guix* propose uniquement des logiciels libres, et insiste énormément sur la reproductibilité [4] et « l'amorçabilité » [5].

La première différence majeure entre gestion des paquets classique et fonctionnelle réside dans la gestion des profils.

2.1 Profils et transactions

GNU Guix est un gestionnaire de paquets utilisateur, comme *pypi*, dans le sens où il est possible, en tant qu'utilisateur non privilégié, de gérer un ou plusieurs profils, indépendamment du système. Un profil est une collection de paquets, mais contrairement aux autres systèmes, il ne contient que les paquets explicitement installés.

Les dépendances ne sont pas directement disponibles dans le profil, bien que les logiciels installés soient capables de les trouver.

Ainsi, en partant d'un profil vide, on peut effectuer diverses opérations usuelles sur les paquets :

```
$ guix install hello          # installe le paquet « hello » dans
                              # le profil
$ guix pull && guix upgrade   # met à jour les paquets
$ guix remove hello           # supprime le paquet « hello » du
                              # profil
```

On peut aussi inspecter le contenu d'un profil, et des différentes générations. Ici, nous avons créé trois générations, en plus de la génération 0, qui est vide. La génération 1 suit l'installation de hello. La génération 2 suit sa mise à jour, et la 3 sa suppression.

Prenons maintenant un exemple plus complexe. Dans cet exemple, nous avons déjà un profil avec plusieurs paquets. Avec un gestionnaire de paquets standard, si je souhaite supprimer le paquet A puis ajouter le paquet B, je ne suis pas toujours certain d'obtenir le même résultat qu'en ajoutant d'abord B puis en enlevant A. En effet, si B dépend de A, dans le premier cas, A sera réinstallé, alors que dans le deuxième cas, B sera désinstallé, comme le montre la Figure 1.

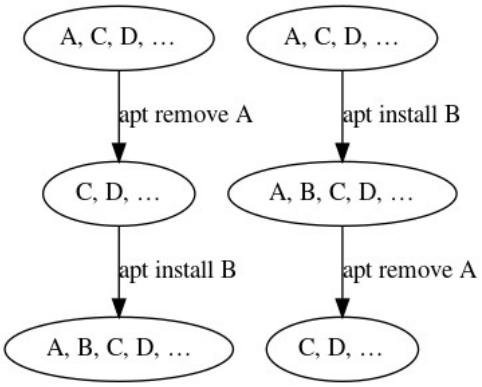

Figure 1 - L'ordre des opérations est important

Avec *Guix* cependant, on n'installe pas les dépendances dans le profil, bien que les paquets puissent toujours les utiliser. Ainsi, la Figure 2 montre comment, avec *Guix*, on peut obtenir les mêmes résultats dans les deux scenarii. Si on supprime d'abord A, comme précédemment, on obtient le même profil, privé de A, puis en installant B, on obtient un nouveau profil avec B, mais sans A. Si on commence par ajouter B, on obtient un profil avec à la fois A et B, puis en supprimant A, on obtient un profil sans A, mais toujours avec B.

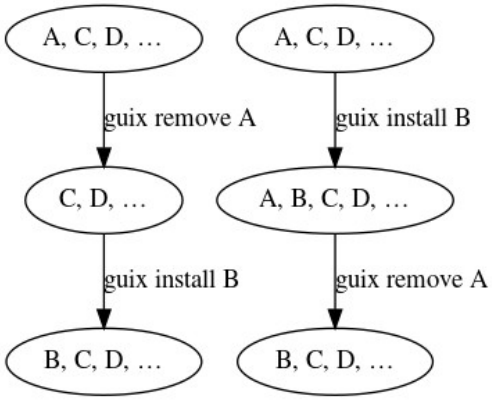

Figure 2 - L'ordre des opérations n'a aucune influence

Puisqu'on obtient le même résultat à la fin, il n'est pas nécessaire d'effectuer les opérations dans un ordre précis, ni même de les effectuer l'une après l'autre. *Guix* propose d'effectuer le changement en une seule et unique transaction (ce qui crée une génération de moins) :

```
$ guix package -i B -r A # on installe B et on supprime A en une
seule transaction
```

La raison pour laquelle tout cela fonctionne réside dans l'utilisation du dépôt.

2.2 Le dépôt

Le dépôt est le lieu dans lequel résident les paquets et leurs dépendances. Le profil est en réalité une collection de liens symboliques vers le dépôt. Ainsi, en installant A, on crée un profil qui contient des liens symboliques vers le répertoire de A dans le dépôt, et en supprimant B, on enlève les liens symboliques vers B dans le dépôt.

Mais comment B peut-il fonctionner, s'il dépend de A mais que A n'est pas installé dans le profil ? À la compilation de B, *Guix* a passé le chemin dans le dépôt de A, de sorte que B puisse chercher A directement dans le dépôt, plutôt que dans le profil. Pour des paquets écrits en *C* par exemple, cela est rendu possible par le *rpath* des fichiers exécutables.

La Figure 3 illustre cela dans l'exemple de l'installation de B puis de la désinstallation de A. Comme B fait toujours référence directement à A, à la génération 3, il peut être installé sans A, mais continuera de fonctionner.

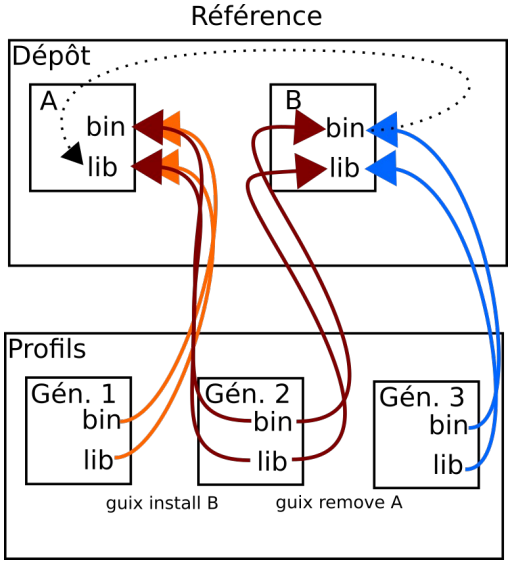

Figure 3 - Plusieurs générations ont des liens symboliques vers différents paquets, mais aucune ne nécessite d'installer la dépendance de leurs paquets, car ils y font référence directement

Une transaction s'effectue en deux étapes : la construction de notre nouvelle génération, dans le dépôt, en créant des liens symboliques vers les paquets installés à cette génération, puis le profil n'étant qu'un lien symbolique vers la génération actuelle, la transaction s'effectue en changeant ce simple lien, de manière atomique. Si une erreur se produit au cours de la construction de notre nouvelle génération, celle-ci n'est pas valide et *Guix* la supprimera du dépôt dès que possible, et dans tous les cas avant une nouvelle tentative, mais il gardera les paquets correctement installés dans le dépôt.

Toutes les modifications du dépôt que nous avons vues jusqu'ici ne faisaient que rajouter des paquets, sans jamais en modifier ni en supprimer. Si notre dépôt ne fait que grossir, n'est-ce pas une limite importante ? *Guix* propose en fait un ramasse-miettes qui s'occupe de supprimer les chemins non utilisés dans le dépôt, c'est-à-dire les chemins qui ne sont ni des racines (générations de profils ou racines manuelles), ni référencés par des chemins utilisés. Il faut donc pour que cela soit efficace, libérer ses anciennes générations avant de lancer « *guix gc* ».

2.3 Reproductibilité

Afin de garantir la reproductibilité des constructions, on doit assurer le suivi des versions. Pour cela, chaque paquet est installé dans un répertoire séparé du dépôt, dont le nom contient un *hash* de sa recette (la manière de construire le paquet), de ses sources et des *hashs* de ses dépendances. Ainsi, dès que l'un de ces paramètres change, *Guix* crée un nouveau répertoire du dépôt, au lieu d'écraser un répertoire existant.

Cela permet de s'assurer que la construction d'un paquet spécifique s'effectue toujours avec ses dépendances spécifiques, ses sources et sa recette spécifiques. Comme chaque version de *Guix* représente un graphe acyclique de dépendances entre paquets, on peut facilement reconstruire une version particulière des paquets représentés dans ce graphe, même plusieurs années après.

Cette séparation des paquets par *hash* permet aussi à *Guix* de reproduire localement des paquets. Ainsi, on peut vérifier, indépendamment, la validité des paquets binaires qu'on a reçus de la ferme de construction :

```
$ guix build --check hello # vérifie qu'on est capable de
reconstruire le même paquet, bit-à-bit
```

Guix propose aussi la *transparence binaire*, un concept important pour la confiance et éviter les portes dérobées dans les binaires. Elle ne peut venir qu'avec la reproductibilité : au lieu de faire confiance à la ferme de construction officielle, vous pouvez décider de construire vos paquets vous-mêmes (ou de créer votre propre ferme de construction). Le résultat de la construction locale étant le même, bit à bit, que celui de la construction de la ferme officielle, il est impossible de distinguer les deux cas. Cela permet de diminuer la confiance qu'il est nécessaire de placer en la ferme de construction. Les développeurs de *bitcoin-core* ont d'ailleurs sauté le pas pour utiliser *Guix* comme outil de vérification de leurs constructions [3].

Dans le même esprit, on peut « défier » des fermes de construction, officielles ou non, pour comparer les constructions entre elles et avec le système actuel :

```
$ guix challenge --substitute-urls='https://example.com
https://ci.guix.gnu.org'
```

2.4 Environnements de développement

Dans la plupart des langages, on a un gestionnaire de paquets qui permet de créer des environnements de développement : on télécharge les sources d'un logiciel, on lance une commande et on obtient un environnement de travail. Avec *Guix*, cela fonctionne de la même manière, mais unifié entre tous les langages de programmation, et même avec les langages qui n'ont pas de gestionnaire dédié. S'il existe un paquet dans la distribution, il suffit de lancer :

```
guix environment offlate
```

pour se mettre à développer et tester ce projet en *Python*, qui requiert normalement l'utilisation de *pip*. De même,

```
guix environment guile-torrent
```

permet de développer et tester ce projet en *Guile*, qui n'utilise pas de gestionnaire de paquets dédié. Un dernier exemple serait :

```
guix environment capstone python-capstone java-capstone ocaml-
capstone
```

qui permet d'avoir un environnement de développement pour les quatre paquets : *capstone* est écrit en *C* mais possède des liaisons *Java*, *OCaml* et *Python*. Un tel environnement nécessiterait de naviguer entre trois gestionnaires de paquets de langage, en plus du gestionnaire de paquets du système.

Si le paquet n'est pas disponible, on peut alors en créer un dans un fichier (par exemple *offlate* fournit son propre fichier avec sa propre recette) ou lister les dépendances explicitement avec le drapeau « *--ad-hoc* » :

```
guix environment --ad-hoc python python-pyqt5 python-lxml …
```

Guix et votre système sont indépendants, ce qui est une bonne chose puisque qu'il n'y a pas d'interférence entre les deux. Cependant, si vous essayez de compiler des programmes avec une partie des dépendances venant du système et l'autre de *Guix*, le résultat dépendra des deux. En plus, vous ne pourrez pas forcément déplacer ce résultat binaire sur un autre système, ni l'y reproduire. Il est plus prudent d'utiliser un seul environnement. Vous pouvez par exemple écrire une recette *Guix* qui compilera votre logiciel vers le dépôt, en garantissant sa reproductibilité.

2.5 Guix pour la science

Nous l'avons vu dans la partie 2.1, les paquets de *Guix* sont installés dans des profils gérés par les utilisateurs eux-mêmes, sans intervention d'un administrateur système. Les paquets sont indépendants du système installé et n'interfèrent pas avec lui. Les utilisateurs sont donc autonomes, sauf pour les logiciels nécessitant réellement les droits administrateurs. Historiquement en *HPC*, les administrateurs devaient proposer des scripts d'installation de logiciels pour les besoins particuliers des différents projets de chaque chercheur. *Guix* permet d'alléger et de partager cette tâche. Il est aussi possible pour un chercheur de reproduire exactement le même environnement logiciel qu'un collègue à l'autre bout du monde, simplement en connaissant la version de *Guix* utilisée et les logiciels installés. En effet, les recettes des paquets de *Guix* sont archivées dans un dépôt git, ce qui permet leur historisation. La quantité d'information à partager est très

réduite : le *hash* d'un commit *git* (celui du dépôt de *Guix*) et une liste de paquets, actuellement déjà fournie dans les papiers les plus sérieux.

La reproductibilité des environnements logiciels est d'ailleurs pérenne, puisqu'il suffit que *Guix* soit installé, de connaître la version utilisée (même ancienne) et les logiciels installés, et on peut reproduire bit-à-bit un environnement, même ancien, avec le même compilateur, les mêmes bibliothèques et les mêmes bogues ! Le plus simple pour cela est de revenir à la version de *Guix* utilisée et d'installer l'environnement décrit dans le papier :

```
guix pull --commit=738d0cd665ef89a77af0b1568aea68d29bc749d6
guix environment --ad-hoc mumps gcc-toolchain
# et maintenant, on peut relancer les mêmes expériences
```

La transparence binaire est aussi très utile dans ce but : si on veut reproduire notre environnement dans 20 ans, il n'est pas forcément dit qu'on sera en mesure de trouver un substitut binaire. Avec un peu de patience et de puissance de calcul, on pourra toujours reconstruire le binaire qui nous intéresse. Et si les sources elles-mêmes ont disparues ? *Guix* travaille main dans la main avec *Software Heritage* [6] pour tenter de préserver ces sources et garantir la reproductibilité dans le temps.

3 Administration Système

Maintenant que nous avons vu les concepts essentiels de *Guix*, pour rappel les transactions, les profils, les générations, la reproductibilité et qu'il ne s'agissait pas seulement de manipuler des paquets, mais aussi des environnements, il nous reste à voir comment *Guix* assemble ces différentes briques pour proposer un système d'exploitation complet.

3.1 Un système déclaratif

Dans *Guix*, on utilise un fichier de configuration dans lequel on déclare l'ensemble de son système d'exploitation. Il ne faut pas se laisser perturber par les parenthèses qui ne sont que de la syntaxe. Elles permettent de séparer les différents blocs. Les point-virgules introduisent des commentaires. Voici un exemple de déclaration de système d'exploitation tiré du manuel :

```
(use-modules (gnu))
(use-service-modules networking ssh)
(use-package-modules screen)

(operating-system
  (host-name "komputilo")
  (timezone "Europe/Paris")
```

```
  (locale "fr_FR.utf8")
  ;; Démarrage en mode « BIOS », en supposant que /dev/sdX est le
disque dur cible et
  ;; « my-root » est l'étiquette du système de fichiers racine.
  (bootloader (bootloader-configuration
                      (bootloader grub-bootloader)
                      (target "/dev/sdX")))
  (file-systems (cons (file-system
                                  (device (file-system-label
"my-root"))
                                  (mount-point "/")
                                  (type "ext4"))
                                %base-file-systems))
  ;; Ensuite, on déclare les comptes utilisateurs. « root » est
  ;; implicite, et il est initialement créé avec le mot de passe
  ;; vide, qu'il faudra changer. Il est aussi possible de
  ;; modifier %base-user-accounts pour indiquer un mot de passe.
  (users (cons (user-account
                (name "alice")
                (comment "Bob's sister")
                (group "users")
                ;; On ajoute le compte au groupe « wheel » pour
                ;; en faire un sudoer.
                (supplementary-groups '("wheel")))
               %base-user-accounts))
  ;; Paquets installés globalement
  (packages (cons screen %base-packages))
  ;; On ajoute des services à ceux de base : un client dhcp et
  ;; un serveur SSH.
  (services (append (list (service dhcp-client-service-type)
                          (service openssh-service-type
                                   (openssh-configuration
                                    (port-number 2222))))
                    %base-services)))
```

On le voit, la déclaration du système d'exploitation est relativement courte : un écran ou deux pour un système de base avec un peu de personnalisation. Comme avec les profils, *Guix* est capable de gérer les systèmes d'exploitation de manière reproductible, transactionnelle et générationnelle.

Ainsi, au démarrage, *Grub* vous proposera de démarrer sur la dernière génération en date, ou sur une autre génération. Pour rendre le fichier de configuration effectif à partir du média d'installation, on utilise la commande :

```
# guix system init /etc/config.scm /mnt
```

En supposant que votre fichier de configuration s'appelle */etc/config.scm* et que votre future racine est montée dans le répertoire */mnt*. À partir d'un système déjà installé, on

peut rendre une déclaration effective en créant une nouvelle génération du système d'exploitation avec la commande :

```
# guix system reconfigure /etc/config.scm
```

Comme pour les paquets, le système est construit par *Guix* dans le dépôt, ce qui inclut les fichiers de configuration. Cela signifie en particulier que la configuration est en lecture-seule. Pour modifier la configuration, il faut modifier le fichier de description du système d'exploitation. Cela garantit que la description du système correspond effectivement au système en cours d'exécution.

Actuellement, on ne peut reconfigurer que le système local : on doit lancer la commande sur le système à reconfigurer, et le fichier de configuration doit s'y trouver. On peut utiliser des outils de gestion de parcs pour lancer des mises à jour automatiques, ou le nouvel outil « *guix deploy* », qui permet de déclarer plusieurs systèmes au même endroit et de les reconfigurer en même temps tout en profitant de l'entièreté de l'*API* de *Guix*.

3.2 Analyse des fichiers générés

Bien sûr, comme avec tout système générant de la configuration pour l'utilisateur, il est légitime de se demander ce qui est produit et de vouloir y accéder. Sous le système *Guix*, très peu de configuration se trouve à un emplacement global, dans /etc. Certains fichiers essentiels sont bien copiés dans le dossier global (*passwd*, *shadow*, *group*, *profile*, etc.), mais la plupart du temps, les services sont paramétrés pour trouver leurs fichiers de configuration directement dans le dépôt.

Puisque les fichiers ne sont pas directement accessibles, il faut lancer une commande pour trouver l'ensemble des fichiers générés utilisés par la génération actuelle du système d'exploitation :

```
$ guix gc -R /var/guix/profiles/system | grep knot.conf
/gnu/store/...-knot.conf
```

En filtrant sur le nom, on peut facilement retrouver les fichiers qui nous intéressent, par exemple la configuration d'un serveur de noms. Comme la configuration est dans le dépôt, il n'est pas possible de l'y modifier directement : il faut pour cela déclarer le nouvel état du système que l'on souhaite obtenir et reconfigurer.

3.3 Réplication

Comme *Guix* propose des constructions reproductibles, on peut reproduire un système sur une autre machine. Il suffit pour cela d'avoir la même version de *Guix* (le même commit pour « *guix pull* ») et la même déclaration de système d'exploitation. Cela peut

être utile pour répliquer la même configuration sur plusieurs machines différentes, mais cela est d'autant plus intéressant, que la réplication est possible entre machine virtuelle et machine physique. Ainsi, avant de déployer une configuration sur une machine, on peut déployer cette configuration dans une *VM* de test temporaire, vérifier que tout fonctionne correctement, puis déployer cette configuration sur la machine physique.

Pour construire son système dans une *VM*, on peut utiliser la commande suivante :

```
$ guix system vm /etc/config.scm
```

Cela crée une *VM* pour *Qemu*. Bien que les droits super-utilisateurs ne soient pas nécessaires, il faut que votre utilisateur et les utilisateurs de construction de *guix* soient membres du groupe *kvm* sous certaines distributions. Cette commande renvoie un script qui nous permet de démarrer la *VM* et de passer des options à *qemu*, par exemple pour lui demander 1 Go de mémoire vive et l'accès au réseau :

```
/gnu/store/...run-vm.sh -n 1024 -net user
```

Cette *VM* n'a cependant accès qu'en lecture-seule à son système de fichiers, mais partage son dépôt avec celui de l'hôte, ce qui permet de préserver un peu de place.

Pour créer une image de *VM* modifiable, on pourra lancer :

```
guix system vm-image /etc/config.scm –image-size=20G
cp /gnu/store/… my-vm.img ; chmod 640 my-vm.img
qemu-system-x86_64 -net user net nic,model=virtio … my-vm.img
```

3.4 Retour en arrière

Il arrive parfois qu'une mise à jour ou une modification du système ne se passe pas comme prévu. Par exemple, notre nouvelle configuration fait planter un serveur ou la nouvelle version d'un logiciel a un bogue important.

Dans le cas de la gestion des paquets, nous avons vu que *Guix* préserve différentes générations de chaque profil, et permet simplement de revenir en arrière, sur l'une de ces générations :

```
$ guix package --roll-back
```

Guix ne traite pas différemment un système d'exploitation, et permet aussi de revenir en arrière, de deux manières différentes : à froid, via le chargeur d'amorçage qui propose de démarrer sur les anciennes générations encore présentes, ou à chaud via la ligne de commande :

```
# guix system roll-back
```

Cela permet de revenir très rapidement à une version du système que l'on sait fonctionnelle.

3.5 Variations sur un paquet

Parfois, la version d'un paquet utilisé par une distribution ne convient pas tout à fait au besoin, mais souvent cela aurait un coût trop élevé d'assurer le suivi d'une version du paquet modifié. *Guix* permet de partir de la définition existante d'un paquet et d'appliquer des transformations. Il y a deux manières de faire : on peut utiliser des transformations simples par la ligne de commande, par exemple pour utiliser une version différente des sources, ou de celles d'une dépendance, ici en spécifiant le *hash* d'un commit git différent du dépôt de *esmtp* et de *libetpan* :

```
$ guix package -i esmtp --with-commit=esmtp=01bf9fc
$ guix package -i claws-mail –with-commit=libetpan=4aee224
```

Ces options sont cependant limitées et il y a une deuxième manière de modifier un paquet : en en déclarant un nouveau, qui hérite du paquet de base. Par exemple, on peut ajuster une option passée au script configure :

```
 (define my-profanity
  (package
    (inherit profanity)
    (arguments
     `(#:configure-flags '("--enable-python-plugins")))))
(operating-system
  …
  (packages (cons* my-profanity ...)))
```

4 Travaux en cours

Même si cet article était très élogieux sur *Guix* jusqu'ici, il y a quelques limites, et des améliorations futures sont à prévoir. Par exemple, si on met *pkg-config* à jour, de

nombreux paquets qui en dépendent devront être reconstruits (parce que leur *hash* du dépôt change) même si leur contenu restera inchangé. Le modèle *intensionel*, introduit lui aussi dans la thèse d'Eelco Dolstra pourrait régler ce problème en partie en remplaçant le *hash* du dépôt par un *hash* du contenu, plutôt qu'un *hash* des outils de fabrication. Cela n'est actuellement pas implémenté, ni dans *Guix* ni dans *Nix*.

Grâce à la transparence binaire, les développeurs de *Guix* peuvent mettre à jour les recettes des paquets rapidement et les distribuer immédiatement aux utilisateurs, avant même que la ferme de construction n'ait commencé à les construire : s'ils ont réussi à construire le logiciel et qu'il est reproductible, la ferme de construction y arrivera aussi. Cela signifie que vous pourriez lors d'une mise à jour tomber sur un logiciel qui n'a pas encore été construit par la ferme de construction. Votre installation de *Guix* devra alors se mettre à construire le logiciel, ce qui introduit un délai et utilise la machine. Il n'y a pour l'instant aucun moyen de limiter une mise à jour aux paquets disponibles sous forme de substitut, mais il est possible d'utiliser « *guix weather* » ou « *--dry-run* » pour savoir ce qui devra être compilé localement.
La granularité de *Guix* se fait un niveau des paquets et les dépendances sont reconnues par la présence d'une référence dans le contenu du paquet. Cela n'est pas idéal pour des systèmes embarqués, qui ont souvent besoin d'une granularité plus fine, ou d'un ensemble de fonctionnalités facultatives plus réduites. Par exemple, un logiciel pourrait contenir un module permettant d'utiliser une base de données, qui fait référence à *mariadb*. Même si nous n'avons pas besoin de ce module, sans changer la recette *Guix* la référence sera trouvée et *mariadb* installé, même si ce logiciel est inutile. *Guix* permet de séparer les paquets en plusieurs chemins du dépôt (c'est-à-dire des sous paquets, comme *-bin*, *-dev*, *-lib*, etc.), ce qui permet de réduire la taille des paquets avec leurs dépendances, mais il reste beaucoup de travail à faire pour réduire la taille de nombreux logiciels courants.
La communauté est organisée autour des personnes ayant le droit de pousser des modifications dans les dépôts de *Guix*. Deux personnes sont aussi co-mainteneurs et représentent le projet auprès de *GNU*. Nous nous retrouvons régulièrement sur *IRC* pour discuter, mais aussi au cours d'autres événements comme le *Fosdem* et le *Reproducible Builds Summit*. Les correctifs sont bien entendu les bienvenus et passent toujours par un processus de revue par les pairs, aussi bien pour les contributeurs occasionnels que pour les plus expérimentés. C'est bien sûr là un appel à contribution !

5 Conclusion

Guix est donc à la fois un gestionnaire de paquet, installable sur n'importe quelle distribution et une distribution indépendante. Contrairement aux distributions classiques, les modifications de l'état du système et des profils de paquets se font de manière transactionnelle et permettent de revenir très facilement en arrière. Les transactions permettent aussi de s'assurer qu'une coupure de courant ne laissera pas le système dans un état intermédiaire. Une mise à jour peut bien introduire un bogue dans un programme, mais il est trivial de revenir à un état précédent qui fonctionne.

Les paquets ne sont pas traités différemment qu'ils aient le même nom ou non : deux paquets du même nom mais différant dans leurs dépendances, sources ou recettes ne

sont simplement pas installés au même endroit dans le dépôt et les paquets ne font référence qu'à la version avec laquelle ils ont été construits. Cela permet d'installer dans le même profil deux applications qui ont besoin de deux dépendances a priori incompatibles dans d'autres systèmes, même si installer explicitement deux paquets incompatibles n'est pas possible dans le même profil.

Guix est aussi un très bon candidat pour remplacer les gestionnaires de paquets dédiés des communautés de langage de programmation en implémentant des fonctionnalités équivalentes, unifiées et même en permettant de créer des environnements mêlant plusieurs langages.

Enfin, *Guix* propose des systèmes reproductibles par construction, qui permettent de se passer de la confiance dans une ferme de construction en particulier, et de vérifier grâce à la transparence binaire, qu'un paquet est bien issu de son code source.

Bibliographie

[1] Eelco Dolstra. The Purely Functional Software Deployment Model (PhD Thesis). Janvier 2006 ; http://nixos.org/~eelco/pubs/phd-thesis.pdf.

[2] Ludovic Courtès. Code Staging in GNU Guix. Septembre 2017. Dans 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17), https://arxiv.org/abs/1709.00833.

[3] Vlad Costea, Guix Makes Bitcoin Core Development More Trustless, article du journal Bitcoin Magazine. https://bitcoinmagazine.com/articles/guix-makes-bitcoin-core-development-trustless. Août 2019

[4] Danny Milosavljevic, Bootstrapping Rust, article du blog de GNU Guix. https://guix.gnu.org/blog/2018/bootstrapping-rust/, 2018

[5] Bootstrappable team, Reduced binary seed bootstrap, https://bootstrappable.org/projects/mes.html

[6] Software Heritage, Software Heritage and GNU Guix join forces to enable long term reproducibility, https://www.softwareheritage.org/2019/04/18/software-heritage-and-gnu-guix-join-forces-to-enable-long-term-reproducibility/. Avril 2019