NixOS

Outil de traduction de package

Ahmed NASSIK
Vincent TURRIN

Tuteur : Olivier Richard

I. Présentation de Nix/NixOS

Une majeur partie de notre projet a consisté à découvrir Nix et NixOS. Cette technologie assez pointue (qui s'adresse vraiment à des personnes qui savent ce qu'elles recherchent dans le développement) a été difficile à appréhender. Nix et NixOS était au départ totalement obscurs et on a dû revenir de nombreuses fois sur les mêmes choses avant d'en tirer quelque chose. Notre expérience peu poussée en système nous a poussé pas mal de soucis car on Nix nous faisait directement passer à un niveau au-dessus. Ceci est pas totalement négatif car au final Nix exige plutôt une manière propre de gérer son système et ses packages, là où une autre distribution qui utilise apt par exemple laisse plus de place au hasard dans ses installations.

a) Quelques notions de Nix

Commandes Principales :

- Nix-build : Permet de construire la dérivation. Si la dérivation est valide, un lien symbolique vers le résultat est créée.
- Nix-Env : Permet de manipuler l'environnement utilisateur Nix (installation, requêtes ...)
- Nix-shell : Permet de démarrer un shell Nix ou peuvent être chargée les dépendances de la dérivation pour reproduire l'environnement d'une dérivation (utile pour le développement d'un paquet).
- Nix-store : Permet de manipuler et d'adresser des requêtes au nix store (dossier contenant tous les paquets installés)

Fichiers importants:

- nix.conf : Fichier contenant des paramètres persistant de nix (Installation de KDE pour tout les utilisateur, création d'utilisateurs ...). Le contenu est évalué puis pris en compte en utilisant la commande *nixos-rebuild switch* suivi d'un redémarrage.
- all-packages.nix : recense les appels aux paquets crées.

b) Présentation condensée de Nix

Dans cet partie nous essayons de donner un des informations sur Nix, clairement et simplement, dans l'objectif de saisir le réel intérêt de Nix et son fonctionnement. Nous ne réutilisons quasiment pas le manuel pour écrire cette partie, nous utilisons ce que nous avons assimilé de Nix.

Pour traiter le gestionnaire de paquets Nix nous avons décidé d'utiliser NixOS qui est entièrement basé sur Nix, également dans certains aspects de sa configuration (on peut revenir dans des configurations systèmes précédentes facilement).

Nix est un gestionnaire de paquet. NixOS est une distribution de Linux qui utilise le gestionnaire de paquet Nix.

Nix est un gestionnaire de paquet très puissant qui est entièrement fonctionnel. C'est à dire que pour n'importe quelle construction à partir d'un package et les différentes entrées, le résultat sera toujours le même : déterministe. Le fait qu'il n'y est pas de hasard est un élément extrêmement important. Sur des distributions plus classiques, lors d'un build, les dépendances par exemple, peuvent ne pas toujours être les mêmes (recherche dans /usr/bin qui peut amener à des dépendances non choisies par l'utilisateur).

Les constructions à partir de ces paquets Nix sont stockées dans le nix store : /nix/store. Chaque construction (unique) à son propre dossier à l'intérieur de celui-ci :
ex: */nix/store/b6gvzjyb2pg0kjfwrjmg1vfhh54ad73z-firefox-33.1/*
b6gvzjyb2pg0kjfwrjmg1vfhh54ad73z → est la hash générée à partir des différents paramètres du package : dépendances, sources, options de compilations, script de build etc… en clair tout.

Deux paquets identiques mènent à la même Hash : déterminisme.

Un aspect à avoir en tête est que chaque construction dans le nix store est totalement indépendante. Une désinstallation ou installation ne peut rien "casser".

Cette indépendance mène à un grand avantage : des utilisateurs sans privilèges peuvent installer des paquets sans problème. Si cet utilisateur désire installer quelque chose qui a déjà été installé, un simple lien sera créé vers celui-ci et l'installation ne sera pas faite de nouveau. Cependant s'il installe un paquet qui diffère de la quelconque des manière : une nouvelle hash et un nouveau répertoire sera créé dans le nix/store.

L'indépendance de chaque paquet permet de créer une infinité de version d'un même package sans aucun problème.

L'indépendance et le déterminisme de ces paquets exigent d'avoir 100% des informations nécessaire à la construction du paquet : on oublie pas de dépendance et on s'assure de la portabilité de celui-ci sur une autre machine.

Sur Nix, lorsque vous lancez une console, vous êtes dans un environnement quasiment vide (à l'exception de quelques paquets que vous avez décidé d'avoir dans celui-ci). Il est possible de charger un environnement avec la commande **nix-shell** qui va charger les dépendances voulues pour facilement développer.

Les deux autres commandes vitales sur Nix, sont **Nix-env** qui va permettre toutes les installation/désinstallation dans les profils utilisateur et **Nix-build** qui permet de construire un paquet sans pour autant l'installer.

c) Notre environnement de travail et développement de packages Nix

Un channel est en quelque sorte une base de données de paquets à laquelle on peut s'abonner et qui contient différents paquets.

Pour développer ses propres packages on a plusieurs choix.

1. On peut s'abonner à un channel officiel. Quand on veut développer notre propre package, on modifie le fichier configuration.nix pour spécifier l'override du paquet présent de le répertoire Nixpkgs officiel.
2. On développe avec le dépôt de paquets officiel en local, et on peut le modifier directement, ou bien ajouter des paquets et les spécifier dans le fichier all-packages.nix[1]

Dans notre cas nous travaillons en utilisant comme channel, le répertoire git officiel des packages que l'on a cloné en local.

Nix est présenté comme très bon pour charger n'importe quelle version d'un package. C'est un peu moins pratique que prévu. Seuls les paquets à jour sont gardés dans les dépôts officiels, les versions obsolètes ne sont pas gardées. Les seuls qui sont conservées sont ceux qui ont une réel utilité : compatibilité, assurance d'être stable. Une simple mise à jour n'est pas gardé car sa mise à jour est considérée comme son amélioration. C'est la politique qui est appliquée. Pour travailler avec des versions très spécifiques, il faut créer ses packages en local. Cependant la modification de packages à jour est très simple grâce aux overrides.

Pour utiliser un répertoire de paquets local on doit le spécifier dans le fichier configuration.nix
nix.nixPath = ["/home/guest/Builds/" "nixos-config=/etc/nixos/configuration.nix"]; dans le fichier configuration.nix situé dans /etc/nixos/.
Test d'un paquet

Un paquet Nix est une dérivation contenue dans un fichier default.nix. Cette dérivation contient toutes les informations nécessaires à la construction du paquet. Cette dérivation attend en paramètre les appels des dépendances nécessaire. Si l'on se contente d'exécuter la dérivation, Nix ne saura pas où chercher les paquets. C'est pour cela qu'il faut soit ajouter notre paquet créé au fichier all-packages.nix puis l'éxecuter, soit charger celui-ci.

nix-build -E "let nixpkgs = import <nixpkgs> {}; f = import ./default.nix; in nixpkgs.callPackage f{}"
On appelle notre derivation dans l'ensemble de dérivations connu : all-packages.nix.

Avec cette commande on peut tester directement notre dérivation default.nix, sans aller modifier le fichier all-packages.nix.

Pour tester un package, on ne peut pas le faire en exécutant la dérivation

[1] all-packages.nix est le fichier qui contient l'emplacement de toutes les dérivations. Il contient les alias qui vont réaliser l'appel du package correspondant.
ex :
drumkv1 = callPackage ../applications/audio/drumkv1 { };

Les appels de dépendances dans les packages font références aux alias du fichier all-packages.nix.

d) Gestion des utilisateurs

Dans Nix chaque utilisateur, même sans privilège peut installer des packages. En effet les installations sont sans aucun effet de bord.
→ Tous les utilisateurs partagent le même Nix store où sont stockées toutes les installations. Chaque utilisateur comprend une sorte de chemin qui constituent l'ensemble des installations propre à son environnement.
Ce qui caractérise un profil utilisateur est son ensemble de liens symboliques. Une installation rajoute juste un lien vers le bon dossier du Nix store (celui-ci n'est pas recréé s'il existe déjà). Une désinstallation enlève un lien symbolique. Une opération de rollback dans Nix va simplement enlever le dernier lien symbolique ajouté.

http://datakurre.pandala.org/2015/10/nix-for-python-developers.html
export $NIXPKGS=...
commandes importantes de test sous nixos : nix-build $NIXPKGS -A hello
Problème de build package.

Debug un paquet Nix : https://nixos.org/wiki/Debugging_a_Nix_Package

II. Transformation de paquets Arch (PKGBUILD) en paquet Nix (default.nix)

Le sujet initial consistait à un portage de NixOS sur carte TegraX1. Le sujet a été redirigé car nous l'avons jugé trop difficile et fastidieux : en clair trop mécanique et moins appréciable à traiter et trop fermé, même si ça reste un challenge technique.

Après avoir longuement étudié Nix et NixOS le sujet a été redirigé ainsi : développer un outil de traduction de paquets Arch Linux en paquets Nix.

L'objectif suivant était de tracer tous les curseurs du sujet en faisant quelques recherches pour savoir quels sont les aspects les plus intéressants/faisables, menant à un résultat conséquent.

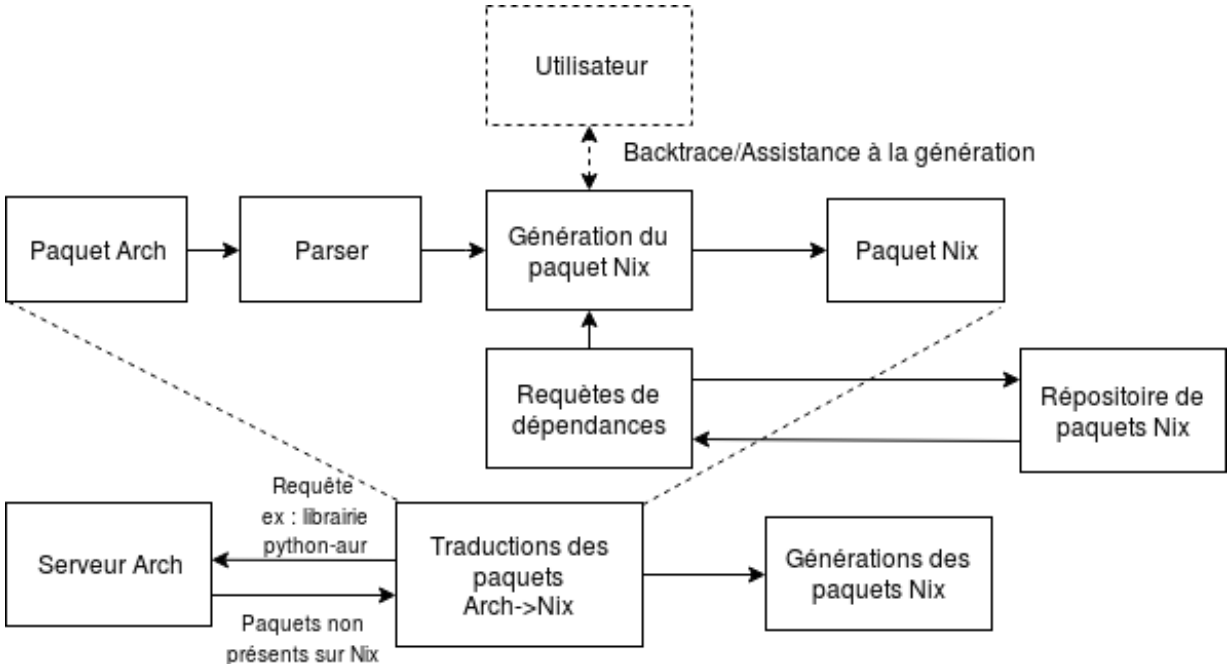

Figure : Structure globale de l'outil

Afin de réutiliser nos connaissances sur Nix, nous avons décidé de nous focaliser dans un premier temps sur la création de fonctions de traduction de package et avons choisi comme langage Python3.

Cette partie comporte tout d'abord un Parser, mécanique, qui peut être réalisé complètement. La génération de paquet Nix cependant n'est pas mécanique et nécessite bien plus de travail à la conception. Les champs ne sont pas les mêmes, la structure générale du paquet n'est pas la même (paquet Nix récursif), et les fonctions de constructions sont différentes (les répertoires d'installation, sources... diffèrent).

Les paquets **Arch** sont plus complets/explicites que les paquets **Nix**.

Des informations peuvent être omises dans les paquets Nix alors que tout est explicitement complétés dans les paquets Arch. Le portage Arch-->Nix semble donc possible.

C'est le cas par exemple au niveau du building des paquets. Sur Nix, rien qu'en connaissant le répertoire où sont situées les sources, le building est souvent automatiquement effectué (notamment en détectant le moyen de build : make, cmake …). Nix effectue alors les commandes nécessaires (configure, make, install pour un makefile par exemple). Nix a beaucoup de fonctionnalités implicites que nous connaissons pas, celles-ci ont beaucoup de sens pour les personnes qui ont l'habitude et l'expérience du paquetage d'un projet logiciel, librairies.

https://git.archlinux.org/svntogit/packages.git/tree/trunk/PKGBUILD?h=packages/freeglut

a) Structures des paquet Arch et Nix

Arch :

Quelques ressources utiles pour comprendre la structure d'un paquet Arch :

https://wiki.archlinux.fr/Standard_paquetage
https://wiki.archlinux.org/index.php/creating_packages
https://www.archlinux.org/pacman/PKGBUILD.5.html

Exemple de paquet Arch:
https://git.archlinux.org/svntogit/packages.git/tree/trunk/PKGBUILD?h=packages/freeglut

Chaque champ et leur signification : https://wiki.archlinux.fr/PKGBUILD :

Normalement le créateur du paquet ne rajoute pas de variables : si jamais c'est le cas les variables sont préfixées de "_"

Nix :

Un paquet Nix est un ensemble d'expression Nix. Sa structure est un peu plus compliquée que le paquet Arch, mais aussi plus puissante. Les dépendances doivent être appelées explicitement.
http://nixos.org/nix/manual/
http://lethalman.blogspot.fr/2014/07/nix-pill-1-why-you-should-give-it-try.html
https://docs.google.com/spreadsheets/d/1jwpYHKQeEmR8Hy19MdvwkYN6abVrjdyQcFQuSyFI7j8/edit#gid=0 : quelques notes sur la traduction des champs

Nous avons procédé de manière incrémentale pour développer cette traduction de paquet.

Outil complet de traduction :

b) Parser

Nous sommes partis de la traduction d'un package Arch classique avec des Champs classique.
Sur les paquets Arch on distingue plusieurs formes:

pkgname=NAME
Ces variables sont soit des champs par défauts soit des champs personnalisés précédés par un "_". Les champs personnalisés servent à éviter la répétition d'un élément dans le fichier (en utilisant *${_nomduchamp}* pour y faire appel).

depends=('elem1' 'elem2' 'elem3')
Il s'agit d'une liste d'éléments (dans ce cas la il s'agit d'une liste de dépendances)

build() {

```
    cd ${_pkgname}-${pkgver}/build
    export ...
}
```

Il s'agit de commandes bash génériques contenu entre accolades.

L'étape du parser permet au final de générer un dictionnaire Python dans lequel l'outil de traduction fera ses recherches d'informations.

c) Génération des paquets

Tout d'abord, à partir du parsing est généré un squelette du paquet dont les champs mécaniques sont remplis, par exemple les patch, nom, version ...

Aspect de traduction à traiter : les dépendances.

Sur Nix elles sont citées explicitement. Sur Arch, le standard efforce qu'elles soit toutes citées même si ce n'est pas vital à la construction. Nous n'avons donc pas de problème pour Nix. Sur Arch les dépendances sont présentes dans les champs depends et makedepends (rarement quelques autres champs comme optional depends).

Les phases de préparation/construction

Sur Arch on observe généralement ces trois phases : **prepare() build() package()**

Pour la gestion de la construction du paquet il semble y avoir deux aspects. Nix semble assez "intelligent" pour utiliser les commandes qu'il faut (il est nécessaire d'avoir les flags du make/cmake.. tout de même).

Une idée est de d'abord mettre en place une traduction qui ne reprend même pas le builder du PKGBUILD. Voir si la version marche, si ce n'est pas le cas on passerait à une traduction du builder.

En fait ces scripts n'ont pas à être traduits, Nix attend surtout des paramètres en entrée.

Approfondissement d'un aspect, la gestion des dépendances :

Les dépendances d'un paquet à traduire pouvant être présentes ou non dans le répertoire des paquets Nix, il est essentiel d'effectuer des requêtes pour savoir si elles existent, et si oui, quel est leur nom.

Pour cela vérifier si la dépendance trouvée est valide, choisir parmi plusieurs paquets ou suggérer une autre recherche, l'assistance de l'utilisateur est nécessaire.
L'outil va effectuer des requêtes au répertoire des paquets de nix et demandera à l'utilisateur de le guider dans sa recherche si les paquets obtenus ne correspondent pas à la dépendance recherchée.

Conclusion et perspectives

Nous avons, par l'occasion de ce projet, compris certains aspects de Nix. Nous avons su nous diriger vers les aspects où nous allions le plus avancer et construire le résultat le plus conséquent.

Durant notre projet, nous avons passé du temps sur les phases fastidieuses : parsing, analyse de la correspondance entre les paquets Nix / Arch. Les aspects que nous avons traités sont les plus mécaniques. La traduction des builders n'est pas mécanique.

Notre projet est facilement améliorable, en continuant le travail que l'on a déjà fait : gestion de nouveaux champs…

Avec une gestion du portage des scripts des builds de Linux vers Nix le code peut amener de nombreux packages à fonctionner. Nix attend surtout des paramètres en entrée pour la compilation, tels que les flags. Dans ces scripts il faudrait changer peu de choses, comme les dossiers de destination (pas dans tous les cas) $out. C'est une phase du projet à étudier.