Un prototype de cache de métadonnées pour le passage à l'échelle de NixOS-Compose

Compas 2024, Nantes

Dorian GOEPP[1], Samuel BRUN[1], Quentin GUILLOTEAU[2], Olivier RICHARD[1]

[1] Université Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

[2] University of Basel, Switzerland

2024-07-04

Contexte & Motivation

Construction et déploiement d'environnements logiciels de systèmes distribués : itératif, difficile

↪ contre-incitation à la reproductibilité

Contexte & Motivation

Construction et déploiement d'environnements logiciels de systèmes distribués : itératif, difficile
↪ contre-incitation à la reproductibilité

NixOS-Compose

- itérations rapides en local,
- puis un déploiement du logiciel à l'identique ;
- s'appuie sur Nix et NixOS $\Longrightarrow$ environnements logiciels déterministes et reproductibles ;
- cibles actuelles : docker, machines virtuelles, Grid'5000 [a]

a. Si NixOS-Compose vous intéresse faites-nous savoir.

Contexte & Motivation

Construction et déploiement d'environnements logiciels de systèmes distribués : itératif, difficile
↪ contre-incitation à la reproductibilité

NixOS-Compose

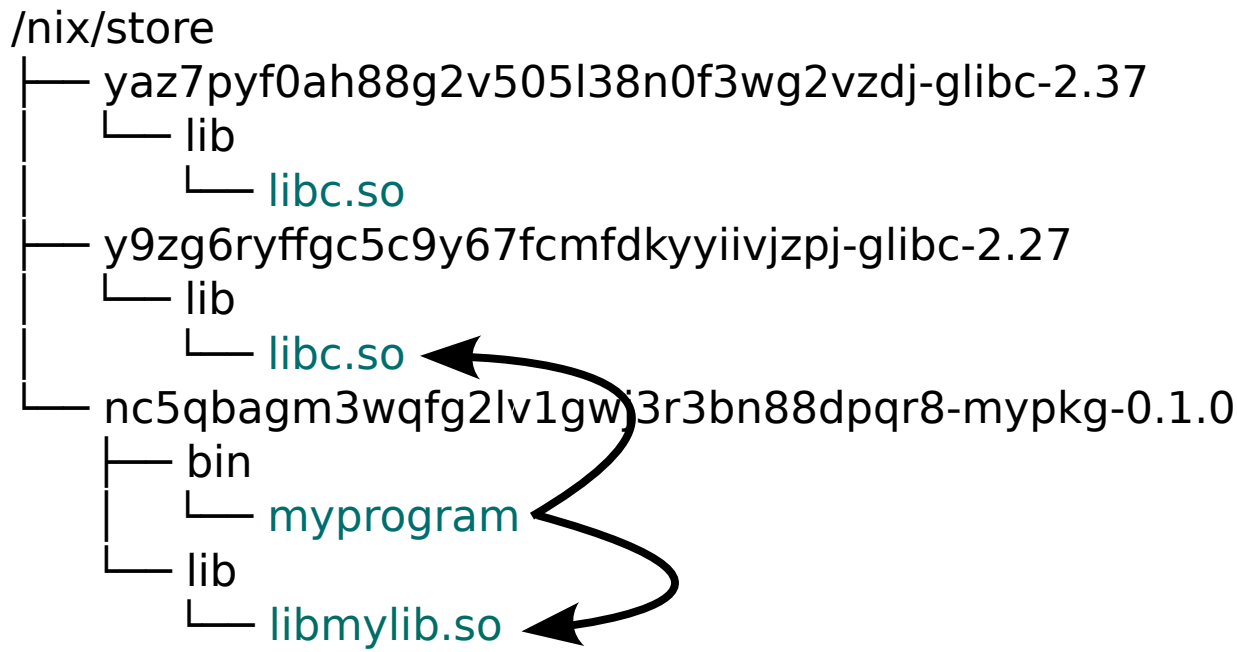
- itérations rapides en local,
- puis un déploiement du logiciel à l'identique ;
- s'appuie sur Nix et NixOS $\Longrightarrow$ environnements logiciels déterministes et reproductibles ;
- cibles actuelles : docker, machines virtuelles, Grid'5000 [a]

a. Si NixOS-Compose vous intéresse faites-nous savoir.

Cible `g5k-nfs-store` : minimise le temps de déploiement sur Grid'5000.
Mais, elle cause une déferlante de métadonnées.

Le magasin Nix

- Motivation : isoler les paquets, pour la reproductibilité et la cohabitations de versions incompatibles
- Chaque paquet est dans son propre sous-répertoire du magasin, selon le schéma `<hash>-<nom>-<version>`.
  ↪ Nix ne se conforme pas au *Filesystem Hierarchy Standard* (FHS).
- On ne peut qu'ajouter des paquets au magasin : pas de modifications



Le chargement de dépendances dynamiques

Cas des binaires ELF :

- `ld-linux.so` a besoin de la liste des dossiers des dépendances d'un programme ;

Le chargement de dépendances dynamiques

Cas des binaires ELF :

- `ld-linux.so` a besoin de la liste des dossiers des dépendances d'un programme ;
- il ne sait pas quel dossier contient quelle dépendances
  $\implies$ explosion combinatoire d'appels système `open()` et `stat()` ;
  N dépendances dans M dossiers : FHS $\to$ M $\ll$ N, Nix $\to$ N $\simeq$ M

Le chargement de dépendances dynamiques

Cas des binaires ELF :

- `ld-linux.so` a besoin de la liste des dossiers des dépendances d'un programme ;
- il ne sait pas quel dossier contient quelle dépendances
  $\Longrightarrow$ explosion combinatoire d'appels système `open()` et `stat()` ;
  N dépendances dans M dossiers : FHS $\rightarrow$ M $\ll$ N, Nix $\rightarrow$ N $\simeq$ M
- la plupart des appels système *échouent*
  $\Longrightarrow$ aucun transfert de données
  $\Longrightarrow$ saturation du système de fichiers distribué en *métadonnées*

Le chargement de dépendances dynamiques

Cas des binaires ELF :

- `ld-linux.so` a besoin de la liste des dossiers des dépendances d'un programme ;
- il ne sait pas quel dossier contient quelle dépendances
  $\implies$ explosion combinatoire d'appels système `open()` et `stat()` ;
  N dépendances dans M dossiers : FHS $\to$ M $\ll$ N, Nix $\to$ N $\simeq$ M
- la plupart des appels système *échouent*
  $\implies$ aucun transfert de données
  $\implies$ saturation du système de fichiers distribué en *métadonnées*

Phénomène amplifié quand NixOS-Compose déploie beaucoup de nœuds.
Problématique similaire pour les langages dynamiques.

Réalisation d'un `open()`

Chaque tentative donne un appel système `open()` ou `stat()` $\Longrightarrow$ plusieurs appels `lookup(<dossier>, entrée)` au système de fichier.

Exemple : ouvrir `/nix/store/0ad51r-zlib-1.3/lib/libjpeg.so`

1. `lookup(root_inode, "nix")` $\rightarrow$ `</nix>`

Réalisation d'un `open()`

Chaque tentative donne un appel système `open()` ou `stat()` $\Longrightarrow$ plusieurs appels `lookup(<dossier>, entrée)` au système de fichier.

Exemple : ouvrir `/nix/store/0ad51r-zlib-1.3/lib/libjpeg.so`

1. `lookup(root_inode, "nix")` $\rightarrow$ `</nix>`
2. `lookup(</nix>, "store")` $\rightarrow$ `</nix/store>`

Réalisation d'un open()

Chaque tentative donne un appel système `open()` ou `stat()` $\Longrightarrow$ plusieurs appels `lookup(<dossier>, entrée)` au système de fichier.

Exemple : ouvrir `/nix/store/0ad51r-zlib-1.3/lib/libjpeg.so`

1. `lookup(root_inode, "nix")` $\rightarrow$ `</nix>`
2. `lookup(</nix>, "store")` $\rightarrow$ `</nix/store>`
3. `lookup(</nix/store>, "0ad51r-zlib-1.3")` $\rightarrow$ `</nix/store/0ad51r-zlib-1.3>`

Réalisation d'un open()

Chaque tentative donne un appel système `open()` ou `stat()` $\Longrightarrow$ plusieurs appels `lookup(<dossier>, entrée)` au système de fichier.

Exemple : ouvrir `/nix/store/0ad51r-zlib-1.3/lib/libjpeg.so`

1. `lookup(root_inode, "nix")` $\rightarrow$ `</nix>`
2. `lookup(</nix>, "store")` $\rightarrow$ `</nix/store>`
3. `lookup(</nix/store>, "0ad51r-zlib-1.3")` $\rightarrow$ `</nix/store/0ad51r-zlib-1.3>`
4. `lookup(</nix/store/0ad51r-zlib-1.3>, "lib")` $\rightarrow$ `</nix/store/0ad51r-zlib-1.3/lib>`

Réalisation d'un open()

Chaque tentative donne un appel système `open()` ou `stat()` $\Longrightarrow$ plusieurs appels `lookup(<dossier>, entrée)` au système de fichier.

Exemple : ouvrir `/nix/store/0ad51r-zlib-1.3/lib/libjpeg.so`

1. `lookup(root_inode, "nix")` $\rightarrow$ `</nix>`
2. `lookup(</nix>, "store")` $\rightarrow$ `</nix/store>`
3. `lookup(</nix/store>, "0ad51r-zlib-1.3")` $\rightarrow$ `</nix/store/0ad51r-zlib-1.3>`
4. `lookup(</nix/store/0ad51r-zlib-1.3>, "lib")` $\rightarrow$ `</nix/store/0ad51r-zlib-1.3/lib>`
5. `lookup(</nix/store/0ad51r-zlib-1.3/lib>, "libjpeg.so")` $\rightarrow$ `</nix/store/0ad51r-zlib-1.3/lib/libjpeg.so>`

Déploiement NixOS-Compose sur Grid'5000

Anatomie d'un déploiement `g5k-nfs-store`

1. une image `ramdisk` Linux minimaliste, sans environnement logiciel ;
2. lancement avec `kexec` sur les machines cibles,
3. tout en préservant le montage NFS de Grid'5000 ;
4. le magasin Nix repose sur le montage NFS
5. `systemd` démarre les services ;
6. machines disponibles pour l'expérience

Déploiement NixOS-Compose sur Grid'5000

Anatomie d'un déploiement `g5k-nfs-store`

1. une image `ramdisk` Linux minimaliste, sans environnement logiciel ;
2. lancement avec `kexec` sur les machines cibles,
3. tout en préservant le montage NFS de Grid'5000 ;
4. **le magasin Nix repose sur le montage NFS** $\Longrightarrow$ saturation
5. `systemd` démarre les services ;
6. machines disponibles pour l'expérience

NixOS-Compose sur Grid'5000 cause une déferlante en métadonnées

En résumé :

1. Nix place chaque paquet dans son propre répertoire
2. `ld-linux.so` cherche les dépendances dans de nombreux répertoires
3. cette recherche produit de nombreux `open()` et `stat()` infructueux donc inutiles
4. NixOS-Compose démarre plusieurs machines NixOS simultanément
5. le magasin Nix est hébergé sur un serveur NFS partagé par les machines NixOS
6. le serveur de fichiers est inondé de ces requêtes sur métadonnées

Objectifs et contraintes de *Chorage*

Intention

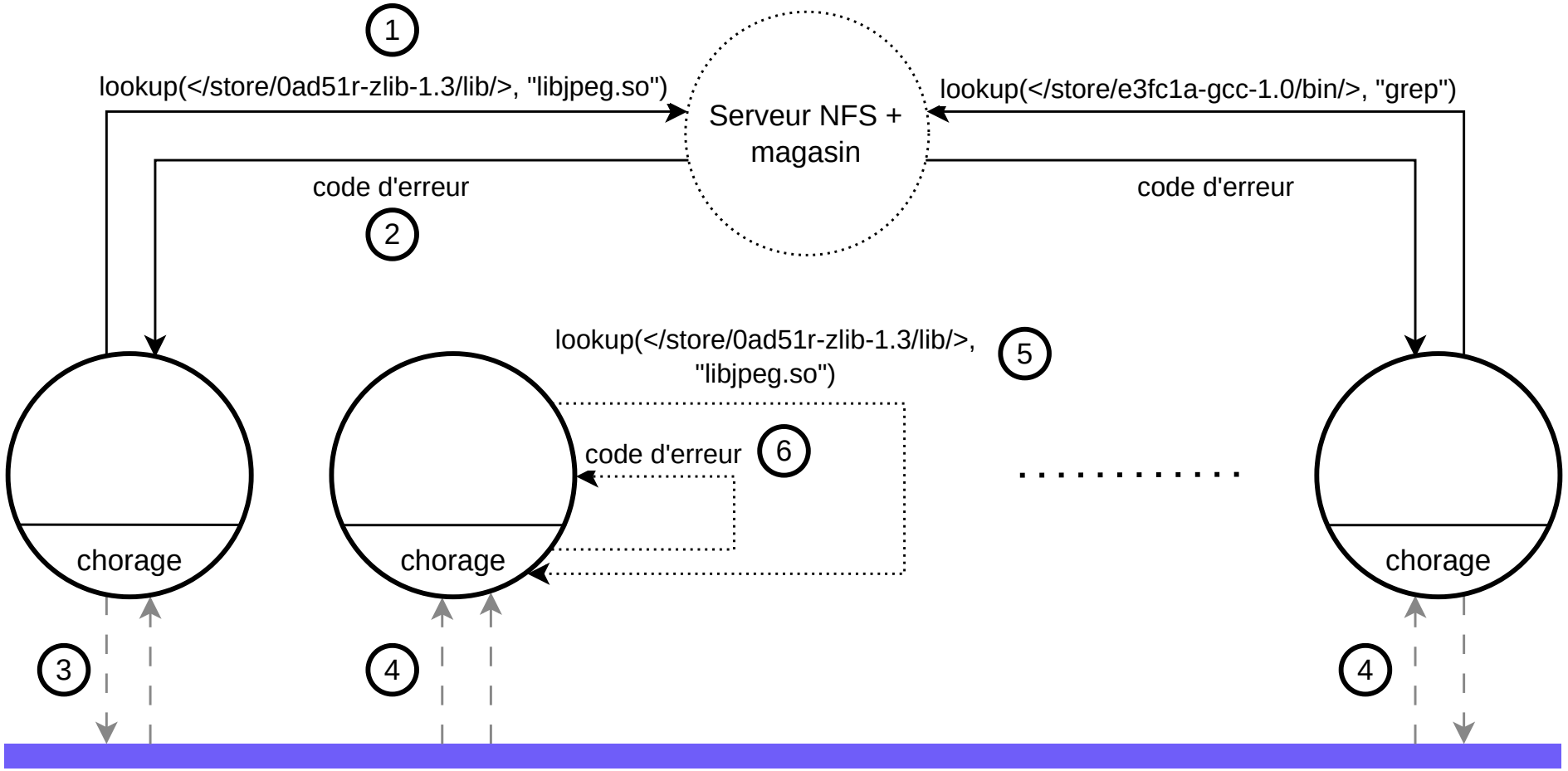
Filtrer les op. en métadonnées inutiles avant qu'elles ne passent par NFS

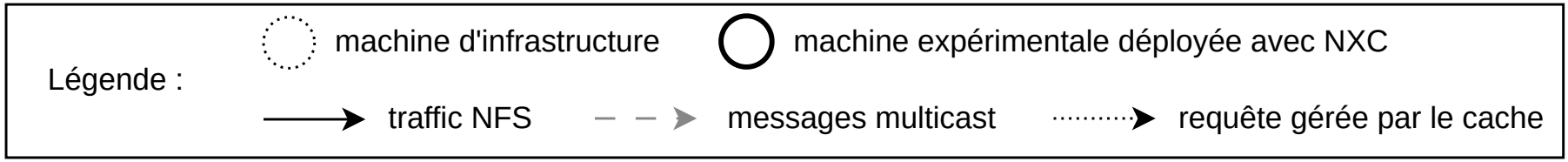
- `ld-linux.so` reste intact (compatibilité avec les langages dynamiques)
- le serveur NFS de Grid'5000 n'est pas modifiable (infrastructure)
- nous ne touchons pas à Nix (un trop gros projet)
- le magasin Nix est en lecture seule
- impossible de préchauffer le « cache » *Chorage* (composition à usage unique)
- partager le cache entre machines

État de l'art

réf.	approche	restriction	implantation
[5]	traces, préchargement de cache, métadonnées → données		FUSE
[3]	métadonnées → données		image disque
[4]	pré-chargement, cohérence faible, lot de requêtes	Lustre	modifications de Lustre
[2]	limite de débit (QoS) dynamique	invasif	interception POSIX
[1]	cache spécifique à `ld-linux.so` pour Guix	binaire ELF	dans Guix
[6]	chemins absolus pour chaque dépendance	binaire ELF	modification du binaire

Conception de *Chorage*





Implantation de *Chorage*

- FUSE, reprenant le code exemple `passthrough_hp.cc`
- ajout de 292 lignes aux 1038 lignes initiales
- cache : (*inode du répertoire, nom recherché*) $\rightarrow$ code d'erreur associé
- diffusion inter-machines par *multicast IP*

Plan d'expérience

- logiciel C construit avec 1000 dépendances (cas pathologique)
- architecture imitant le magasin Nix
- chemins des dépendances inscrites dans le `rpath` du binaire ELF
- un serveur NFS dédié pour ne pas affecter Grid'5000
- 10 clients exécutent simultanément le logiciel

Résultats

machine	en cache	hors cache	ratio de cache
gros-65	20 514	480 986	4,1 %
gros-66	46 020	455 480	9,2 %
gros-67	76 847	424 653	15,3 %
gros-68	74 893	426 607	14,9 %
gros-69	108 927	392 573	21,7 %
gros-70	48 040	453 460	9,6 %
gros-71	30 341	471 159	6,1 %
gros-72	127 889	373 611	25,5 %
gros-73	66 111	435 389	13,2 %
gros-74	88 123	413 377	17,6 %
total			13,7 %
médiane			14,1 %

en cache = opérations filtrées par le cache ;
hors cache = opérations relayées à NFS.

Résultats complémentaires

Travaux en cours d'affinage de notre analyse de *Chorage* :

- *Chorage* peut-il faire gagner du temps ?
  $\hookrightarrow$ si on pré-charge sur une machine ($\approx 39s$), les suivantes prennent $\approx 6s$

Résultats complémentaires

Travaux en cours d'affinage de notre analyse de *Chorage* :

- *Chorage* peut-il faire gagner du temps ?
  $\hookrightarrow$ si on pré-charge sur une machine ($\approx 39s$), les suivantes prennent $\approx 6s$
- Y a-t-il des messages *multicast* perdus ?
  $\hookrightarrow$ expérience en cours de préparation ; pour confirmer le problème de synchronicité

Résultats complémentaires

Travaux en cours d'affinage de notre analyse de *Chorage* :

- *Chorage* peut-il faire gagner du temps ?
  $\hookrightarrow$ si on pré-charge sur une machine ($\approx 39s$), les suivantes prennent $\approx 6s$
- Y a-t-il des messages *multicast* perdus ?
  $\hookrightarrow$ expérience en cours de préparation ; pour confirmer le problème de synchronicité
- Quels sont les gains en charge processeur du serveur NFS et les coûts en charge réseau ?

Conclusion

En somme :

- Introduction du phénomène de déferlante en métadonnées avec NixOS-Compose
- Travaux en cours sur un cache spécifique aux gestionnaires de paquets pour un déploiement massif

Pistes pour la suite :

- meilleure caractérisation de la charge avec et sans *Chorage*
- utiliser Nix pour avoir une liste des chemins existants
- intégrer au déploiement NixOS-Compose

Remerciements

Merci à tout le monde pour leur contribution :

- Olivier Richard, travaux initiaux sur NixOS-Compose, encadrement
- Samuel Brun, version précédente à *Chorage*
- Quentin Guilloteau, supervision de Samuel, conseils, relecture et présentation à Compas 2024

Merci aux relecteurs pour leurs commentaires pertinents, et tout particulièrement à Frédéric Le Mouël [1].

L'expérience présentées dans cet article a été réalisée en utilisant la plateforme d'essai Grid'5000, soutenue par un groupe d'intérêt scientifique hébergé à l'INRIA et incluant le CNRS, RENATER et plusieurs universités ainsi que d'autres organisations (voir `https://www.grid5000.fr`).

1. le seul à avoir décliné son identité, et dont les commentaires sont d'une qualité exceptionnelle

Bibliographie I

[1] Ludovic Courtès, *Taming the 'Stat' Storm with a Loader Cache — 2021 — Blog — GNU Guix*, https://guix.gnu.org/en/blog/2021/taming-the-stat-storm-with-a-loader-cache/ (visité le 03/10/2023).

[2] R. Macedo, M. Miranda, Y. Tanimura, J. Haga, A. Ruhela, S. L. Harrell, R. T. Evans, J. Pereira et J. Paulo, « Taming Metadata-intensive HPC Jobs Through Dynamic, Application-agnostic QoS Control », in 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) (mai 2023), p. 47-61.

[3] C. A. MacLean, H. Leong et J. Enos, « Improving the Start-Up Time of Python Applications on Large Scale HPC Systems », in Proceedings of the HPC Systems Professionals Workshop (12 nov. 2017), p. 1-8.

Bibliographie II

[4] Y. Qian, W. Cheng, L. Zeng, X. Li, M.-A. Vef, A. Dilger, S. Lai, S. Ihara, Y. Fan et A. Brinkmann, « Xfast : Extreme File Attribute Stat Acceleration for Lustre », in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '23 (11 nov. 2023), p. 1-12.

[5] T. Shaffer et D. Thain, « Taming metadata storms in parallel filesystems with metaFS », in Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (12 nov. 2017), p. 25-30.

[6] F. Zakaria, T. R. W. Scogland, T. Gamblin et C. Maltzahn, « Mapping out the HPC Dependency Chaos », in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '22 (18 nov. 2022), p. 1-12.

Le chargement de dépendances dynamiques

Dans un chemin du magasin, disons `/store/0ad51r-zlib-1.3/lib/` on peut s'attendre à trouver un `libzlib.so` mais sans garantie.

Il pourrait aussi y avoir un fichier `libzlib-1.so` ou `libzlib-helper.so`.

Nous savons seulement que si `ld-linux.so` cherche dans ce chemin, il contient *au moins* une dépendance du binaire concerné, puisque chaque dépendance est ajoutée dans la liste des dossiers à parcourir.