Étude du phénomène Stat-Storm - Limitation des appels systèmes pour les systèmes de fichiers distribués de type store

Samuel Brun

Laboratoire d'Informatique de Grenoble

Samuel BRUN
Informatique - POLYTECH Grenoble
Rapport de stage de quatrième année.

Étude du phénomène Stat-Storm; Limitation des appels systèmes pour les systèmes de fichiers distribués de type store

Tome Principal
ET
Annexe

Année universitaire : 2022/2023
17 avril 2023 - 28 juillet 2023

Remerciements

Avant de commencer ce rapport, je souhaite exprimer ma gratitude envers l'ensemble des personnes qui ont participé, de près comme de loin, au bon déroulement de mon stage.

Mes remerciements s'adressent tout d'abord à M. **Oliver Richard**, mon tuteur de stage, enseignant-chercheur de l'équipe DATAMOVE du LIG, pour m'avoir accordé sa confiance afin de mener à bien la mission qui m'a été confiée et pour m'avoir permis de découvrir le monde de la recherche. Je souhaite également remercier M. **Quentin Guilloteau**, doctorant au sein de la même équipe, pour son expertise, ses conseils et son aide précieuse tout au long du projet; sans Quentin le projet ne serait pas ce qu'il est aujourd'hui.

Parallèlement, je tiens à exprimer ma reconnaissance envers M. **Pascal Sicard**, enseignant-chercheur au LIG, pour ses conseils et son écoute en tant qu'enseignant référent. Dans la même lancée, j'aimerais remercier l'équipe enseignante de Polytech Grenoble, dont M. **Jean-François Méhaut**, pour m'avoir transmis les compétences nécessaires au bon déroulement de ce stage et pour avoir répondu à mes diverses interrogations.

Je souhaiterais par ailleurs remercier mes collègues de bureau, **Adrien**, **Alexandre**, et **Maxime**, pour leur bonne humeur et leur disponibilité au quotidien. Je tiens à ajouter deux dernières personnes à ces remerciements, M. **Pierre Neyron**, pour m'avoir permis de visiter le datacenter du LIG et M. **Grégory Mounié**, pour m'avoir permis d'encadrer une activité d'initiation à l'informatique à une classe de secondes.

Enfin, je souhaite exprimer ma gratitude envers ma famille et mes amis, qui m'ont conseillé et relu lors de la rédaction de ce rapport, et qui m'ont soutenu et inspiré tout au long de cette belle aventure.

Résumé

Dans le cadre de ma quatrième année d'études en informatique à l'école Polytech Grenoble, j'ai eu l'opportunité d'effectuer un stage de recherche d'une durée de 15 semaines au sein de l'équipe DATAMOVE du Laboratoire d'Informatique de Grenoble (LIG). Ce rapport représente la synthèse des travaux effectués pendant cette période.

Cette étude se place dans un cadre précis, celui du *High Performance Computing* (HPC), et vise à proposer une solution générique à un problème spécifique appelé **Stat Storm**. Ce phénomène apparaît dans certaines architectures systèmes et se caractérise par une multitude d'*appels systèmes*, dont la plupart sont infructueux. La solution envisagée propose l'utilisation d'un *cache* par *nœud de calcul*, qui serait consulté avant d'effectuer les appels systèmes afin d'éviter les répétitions. Les résultats des différents appels seraient propagés par le biais d'un *réseau à diffusion*, permettant ainsi à chaque nœud de remplir son propre cache. Bien entendu, pour que le système soit fonctionnel, certaines hypothèses doivent être établies, notamment la nécessité que les appels systèmes soient déterministes.

Ma mission s'est décomposée en trois phases. La première a été dédiée à la compréhension du sujet, à l'apprentissage des outils et à l'étude de l'état de l'art adjacent au problème. La seconde phase a été axée sur la conception et le développement de la solution. Enfin, la troisième phase a été consacrée à l'expérimentation et à l'exploitation des résultats.

Mots-clés : HPC, architectures de type store, Stat Storm, systèmes de fichiers, NFS, FUSE, caches, multicast, appels déterministes.

Abstract

In the context of my fourth year of studies in computer science at Polytech Grenoble, I had the opportunity to carry out a 15-week research internship within the DATAMOVE team at the Grenoble Informatics Laboratory (LIG). This report represents the synthesis of the work conducted during this period.

This study is situated within a specific framework, that of *High Performance Computing* (HPC), and aims to propose a generic solution to a specific problem called **Stat Storm**. This phenomenon occurs in certain system architectures and is characterized by a multitude of *system calls*, most of which are unsuccessful. The proposed solution suggests the use of a *cache* per *compute node*, which would be consulted before making system calls to avoid redundancies. The results of the different system calls would be disseminated through a *multicast* network, enabling each node to populate its own cache. Naturally, for the system to be functional, certain assumptions must be established, including the requirement that system calls be deterministic.

My mission was divided into three phases. The first phase was dedicated to understanding the subject, learning the tools, and studying the relevant state-of-art. The second phase focused on the design and development of the solution. Lastly, the third phase was devoted to experimentation and result analysis.

Keywords : HPC, store-type architectures, Stat Storm, file systems, NFS, FUSE, caches, multicast, deterministic system calls.

Table des matières

Table des figures

I. Introduction

Plongé au cœur de ma quatrième année d'étude d'ingénieur en informatique à l'école Polytech de Grenoble, j'ai eu l'opportunité de réaliser un stage d'une durée de 15 semaines avec l'équipe DATAMOVE du LIG **(A)**, le Laboratoire d'Informatique de Grenoble. Débutant le 17 avril 2023, et s'achevant le 28 juillet de la même année, j'ai eu un aperçu de ce qu'est le monde de la recherche en informatique.

À l'heure où les installations *HPC* **(B)** ne cessent d'accroître leurs capacités de calculs **[1]**, que ce soit par l'ajout de cœurs, d'accélérateurs ou de technologies innovantes et puissantes comme les *FPGA* **(C)** ou les *GPU* **(D)[2]**. Il est important que le système de stockage ne devienne pas le goulot d'étranglement des performances de ces systèmes *HPC*. Fournissant d'excellentes performances en termes d'entrées/sorties, les systèmes de fichiers en parallèle sont populaires sur ce genre d'installation, **[3][4]**. Cependant, les performances d'accès aux *métadonnées* restent un problème pour la plupart de ces systèmes de stockage. Là où l'accès aux données des fichiers peut être accéléré, les *métadonnées* n'ont pas ce luxe, car nécessitent un degré élevé de cohérence **[5]**.
De nos jours, il est fréquent de trouver des applications avec plusieurs centaines, voire milliers de dépendances. Au démarrage de ces applications, lorsque ces dépendances sont résolues, un grand nombre d'appels système (*stat* et *open*) sont effectués, mais la plupart d'entre eux se soldent par des erreurs. Un exemple illustrant ce phénomène est celui-ci, *emacs --version*, où plus de 2 200 fichiers sont recherchés avec près de 67% d'erreurs **[6]**. Ce phénomène est connu sous le nom de "Stat Storm", ou "tempête de métadonnées". La tempête de métadonnées a un effet notable sur les performances des applications, notamment dans le cas de filesystems lents ou distribués, tels que *NFS* **(E)**, ainsi que dans les situations d'applications parallèles où des latences pouvant durer plusieurs minutes peuvent être observées **[6]**. Bien que le problème ait été adressé sur les systèmes classiques *UNIX-like* **(F)**, il devient exacerbé dans le cas des systèmes orientés *store* (cf Partie 3.A), où l'on observe une explosion combinatoire du nombre d'appels systèmes.
La gestion efficace des dépendances et la résolution du problème du Stat Storm deviennent ainsi des enjeux cruciaux, en particulier dans le contexte des systèmes orientés store, qui sont de plus en plus utilisés dans le domaine du *HPC* **[7].**

Le sujet de mon stage se concentre sur l'étude du phénomène Stat Storm afin de proposer une solution générique à ce problème dans le domaine du *HPC* utilisant des architectures type store. Cette solution vise à maintenir sur chaque nœud de calcul un cache, qui sera rempli par les appels systèmes et qui partagera ces informations avec les autres nœuds de calcul par le biais d'un réseau *multicast*. Ainsi, si un appel a déjà été effectué par l'un des nœuds, il sera intercepté par notre solution et ne sera pas soumis au filesystem.
Ce rapport se décompose en trois parties. Premièrement, l'étude de l'état l'art autour du sujet (Partie IV.A), deuxièmement, la conception et l'implémentation de la solution proposée (Partie IV.B) et enfin une phase d'expérimentation et d'exploitation des résultats (Partie IV.C). Ce travail marque la fin de ces 15 semaines de stage. Il relate l'intégralité de mes missions ainsi que divers bilans personnels et informations sur le contexte du stage.

II. Présentation et contexte de travail

A. Le LIG

Le Laboratoire d'Informatique de Grenoble (**G**) est l'établissement qui m'a accueilli durant toute la durée de mon stage. Sous la direction de **Noël De Palma** depuis 2021, le laboratoire joue un rôle essentiel dans la recherche en informatique sur le bassin grenoblois. Fondé en 2007, il compte aujourd'hui près de 200 chercheurs et pas moins de 150 doctorants, formant ainsi une équipe experte et pluridisciplinaire.

Figure 1 : Le bâtiment IMAG

Principalement implanté au cœur du campus de Saint-Martin-d'Hères, le LIG est installé dans le bâtiment IMAG (**H**), qui héberge également d'autres laboratoires, qu'ils soient de mathématiques, avec le LJK, ou d'informatique avec VERIMAG et GRICAD. Cette diversité d'environnement en fait un espace propice aux échanges et partages intellectuels. De nombreux événements sont organisés au sein du bâtiment, dont des conférences et des activités introductives. Au cours de ces dernières semaines, j'ai eu la chance de suivre certaines d'entre elles, notamment à l'activité d'informatique débranché à laquelle j'ai participé en tant qu'intervenant ayant pour but d'introduire l'informatique à des élèves de seconde. J'ai également assisté à une introduction à la régulation dans le milieu informatique, ainsi qu'à diverses conférences comme celle sur les FPGA ou celle établissant le lien entre HPC et cloud.

Au-delà de l'acquisition de compétences précieuses, ce stage m'a surtout permis de faire des rencontres enrichissantes et d'établir des liens avec des experts passionnés. L'expérience acquise au sein du LIG restera sans aucun doute un précieux atout pour mon parcours professionnel futur.

B. Le contexte de travail

Le LIG est un regroupement d'une vingtaine d'équipes de recherche, couvrant ainsi un large éventail de domaines de l'informatique actuel, allant de l'intelligence artificielle aux systèmes en temps réel. Au cours de mes 15 semaines de stage, j'ai rejoint l'équipe DATAMOVE, une équipe spécialisée dans l'étude des mouvements de données pour le calcul

haute performance. Elle est dirigée par **Bruno Raffin** et compte près de 35 membres (chercheurs, doctorants, ingénieurs et assistants).
L'objectif principal de DATAMOVE est donc de relever les défis autour du déplacement massif des données dans des infrastructures de calcul en rapide évolution. La loi de Moore en est le parfait exemple, les puissances de calcul ne cessent de croître, il est donc impératif de faire en sorte que le déplacement des données ne soit pas un goulot d'étranglement des performances globales. L'équipe développe donc des solutions innovantes d'optimisation de transfert de données pour les supercalculateurs. Un aspect parallèle aux performances, mais tout aussi crucial, est aussi mis en avant par l'équipe; la dimension écologique. Il s'agit de réduire le coût énergétique des supercalculateurs en optimisant les déplacements de données.
Enfin, au-delà de ces enjeux, DATAMOVE prépare les techniques de stockage de demain en explorant de nouvelles méthodes d'optimisation et de gestion des données, notamment via des algorithmes d'ordonnancement complexes.

Parmi les projets actuels de l'équipe DATAMOVE, on peut citer :

- OAR, un gestionnaire de tâches pour les grappes de calcul intensif.
- NXC, pour NixOs-Compose, un outil visant à réduire la complexité de la mise en place de systèmes distribués éphémères grâce à NIX **(I)** et NixOs **(I)**. La finalité de l'étude derrière ce stage pourrait s'inscrire dans ce logiciel.
- Melissa, une solution pour l'analyse de sensibilité qui met en œuvre des algorithmes pour calculer des champs statistiques spatio-temporels sur les résultats d'études de sensibilité à grande échelle.

C. Conditions de travail

Au cours de mes semaines de stage, j'ai partagé mon bureau avec plusieurs personnes dont Adrien, un ingénieur de talent, ou encore Alexandre, Maxime, Abel et Adélaïde, eux aussi stagiaires au LIG. Bien que les bureaux soient chauds en période de canicule, nous avons été soutenus par un environnement de travail particulièrement fonctionnel et stimulant avec des espaces de repos, de partage intellectuel et une cafétéria. Les horaires n'étaient pas fixes et variaient entre 8h30 et 9h pour le début de journée jusqu'à 16h30 ou 17h pour la fin. Ces conditions de travail favorables et stimulantes ont assurément contribué à la réussite de mon stage.
Durant ce stage, j'ai utilisé trois outils essentiels qui ont grandement facilité mon travail. Le GitLab de l'Inria m'a permis de collaborer efficacement avec mes encadrants. Telegram, une application de messagerie, a été notre principal moyen de communication instantanée. Enfin, *Grid'5000* **(J)**, une infrastructure de recherche, m'a offert la puissance de calcul nécessaire pour déployer des expériences et tester l'application à grande échelle avec un caractère reproductible. Ces outils ont participé à la réussite de mon stage, favorisant notamment la communication et l'efficacité de l'équipe.

III. Préambule technique et origine du problème

Cette partie regroupe l'ensemble des notions utiles pour comprendre en profondeur l'origine du Stat Storm.

A. Les hiérarchies orientées type store

Avant de pouvoir entrer en profondeur dans le sujet, il faut comprendre précisément comment et pourquoi le phénomène **Stat Storm** apparaît. Pour cela, il faut déjà comprendre en quoi consistent les architectures de type store, comme peuvent l'être *Nix*, *Guix* ou *Spark*. Ces architectures ont la particularité de ne pas suivre le modèle classique *FHS* (Filesystem Hierarchy Standard) **(K)** proposé sur la plupart des systèmes *Linux* et *UNIX-like*. Ce modèle définit la localisation des dossiers principaux dans une hiérarchie système, ainsi que leurs contenus (/etc, /bin, /usr, …). C'est typiquement via ce modèle que l'on sait où sont stockés les *librairies* et packages sur notre machine (/lib ou /usr/lib). L'avantage de ce modèle est qu'il permet une gestion simple et cohérente des packages, mais l'inconvénient est que dans la grande majorité des cas, une seule version d'un même logiciel (ou d'une même bibliothèque) est acceptée. Cette technique rend les mises à jour de sécurité plus simples, puisqu'elles n'impliquent qu'un seul fichier à mettre à jour, mais peut aussi provoquer des incohérences lors de celles-ci. En effet, si une mise à jour écrase une ancienne version du même nom alors que celle-ci était utilisée par un processus ou un programme, des risques de corruption logiciels émergent. De plus, dans ce contexte, très peu de logicielles versionnent leurs paquets, c'est-à-dire qui indiquent clairement avec quelles versions de quels paquets le logiciel est en mesure de suivre sa spécification. Cela implique des difficultés lors des déploiements de l'application, tout en multipliant l'effort du personnel chargé de tester le programme.

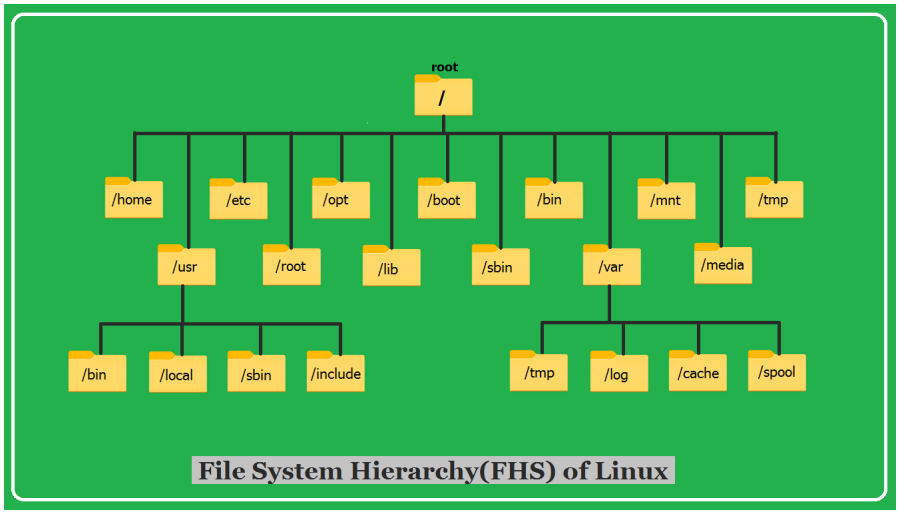


Figure 2 : Schéma d'un FHS (https://tecadmin.net/filesystem-hierarchy-structure-in-linux/)

Avec une architecture de type *store*, toutes les librairies (et versions de ces librairies) sont stockées dans des répertoires différents sous un même grand répertoire **immuable** communément appelé le "*store*" (en nix : /nix/store). Par rapport au modèle classique, ce modèle a l'avantage d'être un excellent gestionnaire de version. Là où dans le système classique, il est complexe, voire impossible, de faire cohabiter différentes versions d'une même librairie ou du même logiciel, le paradigme du store permet d'assurer que les versions peuvent et pourront toujours cohabiter et être indépendantes les unes des autres. Cela est rendu possible par le biais de la méthode de stockage; chaque package est installé dans le store, dans un

répertoire indépendant des autres, suivant une norme de nommage précise et unique. Il est donc tout à fait possible d'avoir plusieurs versions des mêmes librairies.

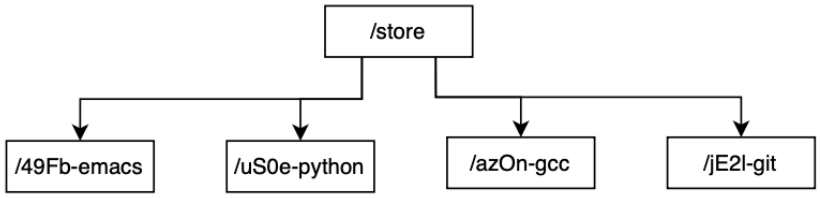


Figure 3 : Schéma d'un FS type store

Sachant les avantages des gestionnaires de version typés store, il est facile de concevoir l'intérêt des centres *HPC* pour ces technologies [7]. En effet, ces centres se voulant les plus libres possible, imposer une seule version d'un logiciel aux utilisateurs semble inconcevable et limitant. Le choix de la version des librairies utilisée doit rester celui du programmeur, surtout sur ce type d'architectures servant notamment aux tests. Ceci justifie l'attrait des centres HPC envers cette technologie novatrice.

B. La technologie NIX

Lors de ce stage, nous avons utilisé la technologie *NIX* (**1**). *NIX* est un système de gestion de paquets ainsi qu'un environnement d'exécution pour les systèmes d'exploitation basés sur *Unix*, suivant le modèle store présenté ci-dessus.
NIX se caractérise par son approche fonctionnelle et déclarative. Cela permet à ses utilisateurs de décrire leur configuration système de manière reproductible et tout en assurant l'absence de conflits. Avec *NIX*, les paquets logiciels sont isolés les uns des autres, évitant ainsi les problèmes d'interférences et de dépendances contradictoires. Cela signifie que *NIX* est construit de façon à obliger les développeurs à spécifier de manière explicite l'ensemble des dépendances lors du *build* d'une application, ce qui permet de lier les paquets avec une totale liberté quant aux versions utilisées. Cette approche garantit une gestion cohérente des versions logicielles. Évidemment, la conséquence de ceci est qu'il faut rebuild l'entièreté de l'application lors de la mise à jour d'une simple dépendance. *NIX* est également réputé pour sa robustesse, sa sécurité et son extensibilité, en faisant un outil apprécié des développeurs et des administrateurs système.

L'ensemble du stage a été réalisé en utilisant *NIX* mais les solutions proposées pourraient être adaptées pour répondre au même problème sur d'autres systèmes orientés store.

C. Dépendances statiques et dynamiques

Un autre point clé, important à comprendre, est la mécanique de résolution des dépendances dynamiques d'une application. Un exécutable peut avoir deux types de dépendances, celles dites *statiques* et celles dites *dynamiques*. Les dépendances statiques sont établies au moment de la *compilation* du programme, les bibliothèques ou les modules nécessaires sont alors intégrés directement dans le binaire. Cela signifie que toutes les dépendances statiques sont liées de manière permanente à l'exécutable, ce qui rend l'application volumineuse. Les avantages des dépendances statiques résident dans leur portabilité, le programme pouvant ainsi être exécuté sur des systèmes sans avoir besoin de bibliothèques externes potentiellement absentes, mais cela conduit à des mises à jour complexes.

De leur côté, les bibliothèques *dynamiques* sont établies au moment de *l'exécution* du programme. Ainsi, plutôt que d'être intégrées directement dans le binaire, les bibliothèques externes nécessaires sont chargées au lancement du programme. Cela permet de réduire la taille du binaire, car il ne contient que le code du programme lui-même, tandis que les bibliothèques externes sont stockées séparément sur le système (et par conséquent peuvent très bien être utilisés par d'autres programmes, typiquement la libc). Les dépendances dynamiques rendent également les mises à jour plus faciles, du fait que les bibliothèques peuvent être mises à jour indépendamment sans avoir besoin de recompiler le programme principal, sauf, comme expliqué ci-dessus, dans le cadre d'une architecture type store (cf. Annexes.C). Cependant, les liaisons dynamiques impliquent que pour fonctionner, le programme doit être capable de trouver les bibliothèques externes spécifiées. De nos jours, les liaisons dynamiques sont considérées dans la plupart des cas comme la meilleure alternative parmi les deux possibles. Du point de vue d'un logiciel, les dépendances dynamiques forment un graphe orienté acyclique. Ainsi, la mise à jour d'un paquet de ce logiciel revient simplement à modifier ce graphe.

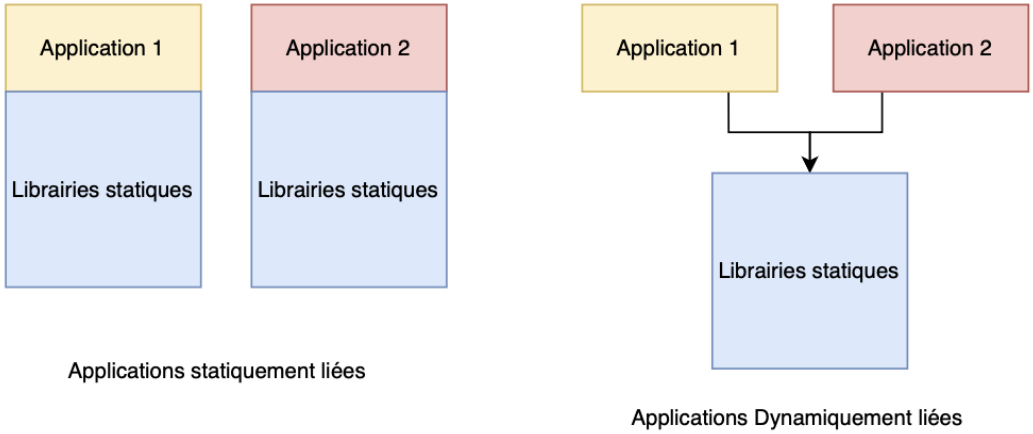


Figure 4 : Liaisons Statique et Dynamiques

D. Résolution des dépendances dynamiques

Classiquement, la phase de recherche des dépendances dynamiques s'appelle la phase de *résolution des dépendances*. Une fois trouvées, ces librairies sont chargées par le *dynamic loader*, le programme chargé de la résolution, (qui est lui-même une librairie partagée, chargée en plusieurs phases au début du programme). Ce programme a pour objectif de localiser les ressources dynamiques et d'injecter leurs symboles dans *l'espace d'adressage* du processus pour les rendre accessibles en *runtime*. Pour localiser les bibliothèques, dans les cas des FHS, le loader cherche dans les répertoires classiques et va optimiser ces recherches en utilisant un cache, le fichier */etc/ld.so.cache* qui mappe les libraires avec leurs emplacements dans le filesystem. Toutefois, dans un contexte de store, cette méthode n'aurait pas lieu d'etre; la recherche dans le store entier prendrait beaucoup trop de temps et l'utilisation du fichier ld.so.cache n'aurait pas de sens, étant donné que ce paradigme a pour objectif premier d'isoler chaque paquet dans un répertoire avec ses dépendances. Ainsi, le passage par un cache pourrait mener à une situation où si un paquet A dépend d'une version spécifique d'une bibliothèque X, tandis qu'un autre paquet B dépend d'une version différente de la même bibliothèque X, nous ne saurions pas quelle version mettre en cache.

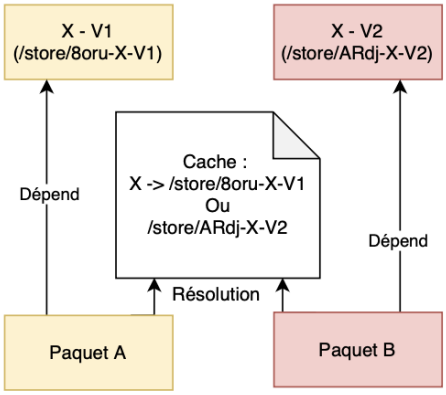


Figure 5 : Incohérence de ld.so.cache en type store

Mettre les deux versions semble non-pertinent et aurait pour conséquence de remplir le cache et de créer un nombre de *caches miss* démesuré. C'est pourquoi les systèmes type store désactivent cette fonctionnalité, ce qui rend le phénomène stat storm apparent.
La méthode adoptée par ce type de système est la méthode classique et non-optimisée: pour chaque dépendance, le loader cherche dans la *variable d'environnement LD_LIBRARY_PATH* **(L)** (ou dans le fichier associé ld.so.conf); s'il ne trouve rien, il cherche dans les métadonnées *RUNPATH* **(L)** ou *RPATH* **(L)** de l'exécutable. Ces variables contiennent une liste de chemins où il est possible de trouver l'une des dépendances. Le loader va donc, pour chaque fichier, essayer de l'ouvrir (open ou stat) avec les chemins possibles afin d'observer une erreur si le fichier n'existe pas. La manipulation va se répéter ainsi jusqu'à la résolution de l'entièreté des dépendances. Les variables *RUNPATH* et *RPATH*, si elles sont définies, le sont au moment du build de l'application. Les commandes unix **ldd (M)** ou **objdump (N)** permettent notamment de lister les dépendances d'un objet et la commande **patchelf (O)** peut être utilisée pour fixer *RUNPATH* ou *RPATH* une fois le build effectué.

En somme, peu importe l'architecture système utilisée, la méthode de recherche s'effectue globalement de la sorte :

- Recherche dans *RUNPATH* et *RPATH*
- Recherche dans *LD_LIBRARY_PATH*
- Recherche dans les répertoires standards (/lib ou /usr/lib)
- Enfin, si rien n'est trouvé, recherche dans le répertoire courant

À noter tout de même que cet ordre/ces lieux de recherche peuvent varier en fonction du système et de sa configuration.

E. RPATH et RUNPATH

RPATH est un mécanisme ancien et qui vise à devenir obsolète, bien qu'il soit toujours pris en charge pour des raisons de compatibilité. Lorsque *RPATH* est utilisé pour spécifier les chemins de recherche des bibliothèques partagées, il peut contenir non seulement les chemins vers les bibliothèques partagées directement nécessaires par l'exécutable, mais également les chemins des dépendances transitives de ces bibliothèques. Cette liste étendue peut ralentir la recherche, mais peut aussi et surtout la rendre moins prévisible.
C'est là l'une des principales motivations de l'apparition de *RUNPATH*. Il permet de spécifier explicitement les chemins de recherche pour une bibliothèque ou un exécutable sans inclure les dépendances transitives, ce qui améliore la prévisibilité et la séparation entre les chemins de recherche des différentes bibliothèques et exécutables. À noter que pour que cela fonctionne, il

faut que *RUNPATH* prenne le pas sur le *RPATH*. Ainsi, la règle suivante a été instaurée; si *RUNPATH* est défini, *RPATH* est ignoré.

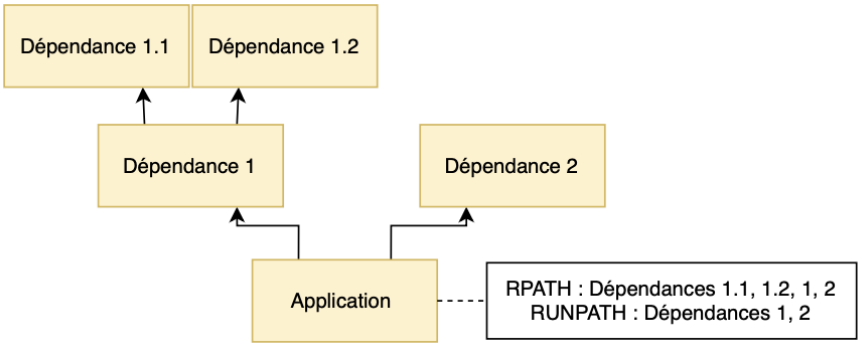


Figure 6 : Différences RUNPATH/RPATH

Malgré les correctifs apportés par la métadonnée *RUNPATH*, un problème de prévisibilité demeure encore. Admettons une application qui a besoin d'une librairie nommée lib, et admettons que deux répertoires se trouvent dans le *RUNPATH* (ou de façon équivalente dans *RPATH*). Si les deux répertoires possèdent une librairie nommée lib, celle sélectionnée sera la première vue.

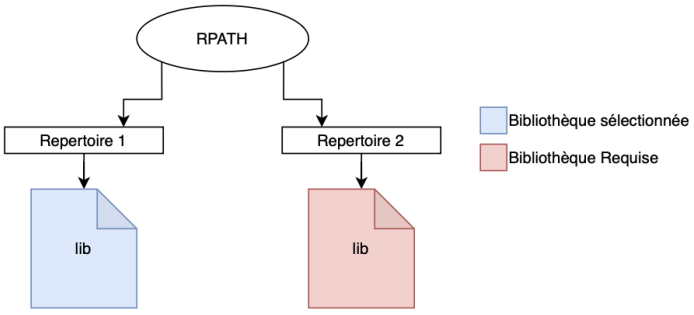


Figure 7 : Les problèmes des RPATH et RUNPATH

L'exemple montre à quel point l'utilisation de *RPATH* plutôt que *RUNPATH* peut amplifier le problème. Il y a un désir général d'éviter l'utilisation de *RPATH* (ld.so du man linux indique qu'il est obsolète). Cependant, en contradiction totale, la bibliothèque Qt demande d'utiliser *RPATH* et non *RUNPATH* pour pouvoir trouver la bonne version de la bibliothèque demandée. Si l'application a un *RUNPATH* la recherche n'inclut que les chemins définis dans la biblio de *Qt*, cela est une fonctionnalité tant que l'appli n'est pas un plugin. Si elle en est un, quand elle sera chargée par un programme A, les chemins de recherche de A ne pourront être fournis par *RUNPATH*, il faudra alors passer par *LD_LIBRARY_PATH* **(P)**.

C'est en observant toutes ces divergences d'utilisations que l'on se rend compte qu'une telle gestion des dépendances transitives n'est pas adaptée, en particulier pour les distributions par répertoire (store).

F. Le Stat Storm

À ce stade, nous pouvons voir le problème arriver. Pour localiser les bibliothèques, le loader teste un à un les différents répertoires qu'il rencontre dans ces variables, via les *syscalls stat* ou *open*, d'où le nom Stat Storm. Une fois qu'il a trouvé une dépendance, il passe à la suivante et ainsi de suite jusqu'à qu'il résolve l'entièreté des dépendances (et les transitives).

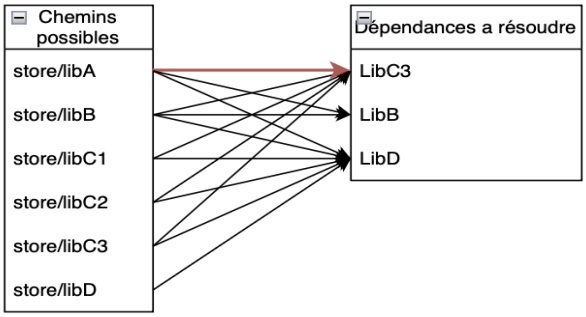


Figure 8 : Interprétation graphique de Stat Storm

Dans cet exemple basique, l'application a trois bibliothèques à trouver, chacune pouvant être dans l'un des chemins possibles. Ici, libX se trouve dans le répertoire store/libX. Chaque flèche représente un test du type "La librairie X est-elle dans le répertoire Y ?". Par exemple : "la librairie LibC3 est-elle dans le répertoire store/libA ?".
Ainsi, si l'on regarde d'un point de vue complexité, notons D le nombre de dépendances à résoudre, C le nombre de chemins possibles, alors sur une machine la résolution se fait en *O(C*D)* opérations sur le filesysteme. Si l'on regarde d'un point du HPC, avec N nœuds, la résolution devient de l'ordre de *O(N*C*D)*.

Maintenant que le problème est clairement établi, il apparaît que la performance des métadonnées reste un goulot d'étranglement important dans les systèmes de fichiers parallèles. En particulier, lorsque des applications complexes démarrent sur de nombreux nœuds à la fois, la tempête se produit lorsque chaque instance et parcourt le système de fichiers à la recherche d'exécutables, de bibliothèques et d'autres composants nécessaires à l'exécution. Non seulement cela retarde l'application en question, mais cela peut rendre l'ensemble du système inutilisable par d'autres clients.

IV. Travaux réalisés

A. L'état de l'art

Le problème que nous adressons dans ce stage comporte plusieurs dimensions et angles d'attaques. De nombreux travaux ont déjà été réalisés à ce sujet.

		1 nœud	>1 nœuds
Offline	Par loader	- Champs DT_NEEDED [8]	
	Par environnement	- Self-Referential Model [10] and Hermetic Root Model [10] - Dependency Views [10]	
	Par exécutable	- Shrinkwrap [10] - Injection Soname [11]	
Runtime		- Cache par application [6]	- Spindle [9] - Amélioration I/O [9,12,13,14]

Figure 9 : Etat de l'art

1. Modifier le loader pour améliorer les performances

Modifier le *loader* peut sembler être une solution convenable. Il serait possible de s'inspirer du **cache** déjà existant dans les FHS pour créer, non pas un cache global, mais un cache par application, mappant dépendances et chemins [6]. Au moment du build, tous les chemins pourraient être enregistrés dans un cache qui serait consulté par le loader. Le système s'adapte particulièrement bien aux dépendances transitives. En plus de régler le Stat Storm, cette approche corrige le caractère imprévisible de la résolution.

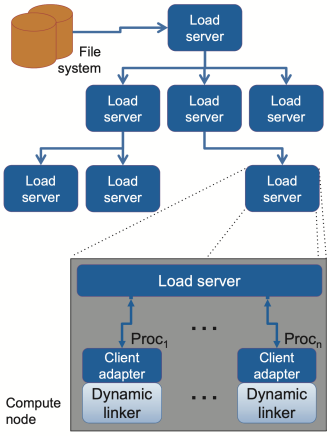
Une autre idée pourrait être d'utiliser le champ ***DT_NEEDED***, déjà existant des fichiers *ELF*, afin d'y stocker chemins réels lors de la compilation et de rendre *RUNPATH* redondant et inutile. Ainsi, le loader pourrait résoudre les dépendances directement depuis ce champ [8].

Ces deux propositions s'appuient sur le fait que dans les systèmes comme *NIX*, l'ensemble des dépendances sont connues et sont explicites au moment du build. Néanmoins, le désavantage commun de ces solutions est la modification du code source du loader pour l'ajout de prise en compte de la solution. Ce changement, aussi infime soit-il, est un problème en lui-même puisqu'il faut valider le loader et le faire utiliser par tous pour profiter de l'amélioration. De plus, dans le cas d'applications interprétées ou en runtime, ce genre de mécanisme devient déprécié dû à l'absence de chemin de recherche comme l'est le *RUNPATH*.

D'autre part, dans une dimension *HPC*, des solutions comme ***collfs*** [9] proposent de personnaliser le loader pour permettre le chargement synchronisé de bibliothèques. Bien que cela puisse accélérer certains programmes, cela change la sémantique du chargement et peut nécessiter des modifications spécifiques aux applications.

2. Utilisation de l'interface du loader

Une proposition intéressante nous est présentée par l'outil ***Spindle*** (Q). Il tire profit de l'organisation des *HPC* et de l'interface du loader, pour intercepter les demandes faites aux *FS* par ce dernier et les traite en cachant et en diffusant l'information [9]. *Spindle* a l'avantage de proposer plusieurs modes, en fonction de l'utilisation souhaitée des nœuds de calcul.



[9]

Figure 10 : Spindle

Spindle établit un réseau de communication indépendant, optimisant les charges et les accès au système de fichiers. Les clients *Spindle* vont rerouter les demandes de leur chargeur vers les nœuds serveurs *Spindle* via ***rtld-audit***, l'interface du *loader* (K). Le serveur de charge gère les sauvegardes locales en RAM de table pour les métadonnées, avec une réplication pour optimiser les performances malgré la réduction d'espace RAM disponible. Cependant, l'inconvénient principal de cette approche réside en la spécification des nœuds de calcul.

3. Changements de l'environnement d'exécution

Une autre solution pour résoudre le problème pourrait résider en la modification de l'environnement d'exécution des applications.

Le **Self-Referential Model [10]** est une approche regroupant les dépendances et le logiciel lui-même, dont le chemin de recherche donne la priorité aux bibliothèques locales. Cette méthode contredit le principe de store, mais a l'avantage d'une meilleure expérience utilisateur, car le logiciel peut résider n'importe où sur la machine et avoir plusieurs versions. De plus, l'installation et la suppression des paquets deviennent atomiques, ce qui facilite la gestion des paquets. Cependant, cette méthode présente des inconvénients, notamment la duplication des bibliothèques, la taille accrue des *bundles* et la difficulté d'ajouter des extensions ou du code utilisateur.

L'**Hermetic Root Model [10]** vise à exploiter certaines règles implicites de *FHS* pour améliorer la sécurité et l'atomicité. Elle ajoute des couches au système de fichiers, déployables de manière similaire à *Git*, permettant la navigation entre les versions de ces couches. Cependant, cette méthode est très complexe d'utilisation et rend, par exemple, difficile la création d'images à l'intérieur des ressources de calcul.

La méthode **Dependency Views [10]** veut, au lieu de définir des *RPATH* ou *RUNPATH* sur l'exécutable et chaque bibliothèque, associer chaque dépendance à un seul *RPATH* ou *RUNPATH* vers un répertoire local contenant un système de fichiers de style *FHS* peuplé de liens symboliques vers les dépendances réelles du package. Cela réduit considérablement la taille du *RPATH*, mais peut entraîner un nombre conséquent d'inodes et ne permet pas à un paquetage de dépendre de plusieurs relations.

4. Une solution modifiant l'exécutable.

Une approche alternative pourrait consister à effectuer des modifications au niveau de l'exécutable. À cet égard, ***Shrinkwrap*** **[10]** paraît être l'un des outils le plus au point. Son objectif est de remplacer la liste de noms de bibliothèques dans la section *DT_NEEDED* par des chemins absolus. Cette approche permet non seulement de réduire les conflits de noms, mais aussi de résoudre et d'inclure toutes les dépendances (même transitives) dans le binaire grâce à des chemins absolus. Cependant, cet outil ne prend pas en charge certains linkers Linux actuels, et le fait de référencer les dépendances par leur chemin absolu peut rendre impossible le remplacement de ces dépendances par d'autres bibliothèques en utilisant, par exemple, *LD_LIBRARY_PATH*.

Une autre solution consiste à modifier le ***soname*** (nom de la librairie) de la bibliothèque en incluant un chemin absolu après son installation **[11]**. En faisant cela, le loader pourra charger directement la bibliothèque en utilisant le chemin absolu spécifié dans le soname, contournant ainsi les limitations liées aux dépendances et aux chemins de recherche standard.

5. Amélioration des I/O

Une proposition alternative pourrait être d'utiliser des systèmes de fichiers parallèles tels que ***Lustre*** **(S)**, pour permettre un accès parallèle aux ensembles de données répartis sur plusieurs disques **[12]**. Cependant, ayant trop peu de serveurs de métadonnées, le chargement parallèle ne profite pas pleinement de ce type d'accès, où chaque processus accède aux mêmes petits fichiers.

Certaines approches utilisent des nœuds *d'I/O* dédiés pour le chargement en les faisant agir comme des caches de chargement. Toutefois, ces approches nécessitent souvent une étape préalable de *staging*, ce qui peut être fastidieux et non transparent pour les utilisateurs. D'autres

approches utilisent des services tels que *DVS* [13,14], un service de transfert de données, pour rendre le processus plus transparent, mais elles peuvent toujours être limitées par la nécessité de stage préalable.

Enfin, une architecture *peer-to-peer* [15] a été proposée pour distribuer les bibliothèques partagées sur un réseau. Cette approche présente des limitations, notamment en nécessitant que les utilisateurs spécifient toutes les dépendances de bibliothèques à l'avance et en ne traitant pas efficacement le problème de métadonnées lors du démarrage d'applications à grande échelle.

6. Synthèse

L'étude de l'état de l'art pour améliorer la gestion des dépendances et résoudre le problème du Stat Storm a permis d'identifier plusieurs approches prometteuses, répondant toutes à des contextes précis. Que la solution mise sur la modification du loader, le changement d'environnement ou encore l'amélioration des I/O, toutes ces approches trouvent des avantages et des inconvénients. Il est important de noter que certaines solutions ne sont pas toutes incompatibles, mais toutes s'appuient sur le fait qu'il est très peu possible et souhaitable modifié l'éditeur dynamique. De plus, ils s'accordent sur le fait que *RPATH*, *RUNPATH*, *LD_LIBRARY_PATH* sont simplement des listes de répertoires et, plus il y a de dépendances, plus la liste est longue et donc moins l'application passera à l'échelle.

B. La solution proposée

1. La théorie

Avant d'entrer dans le vif du sujet, il faut remarquer la chose suivante; dans notre contexte, la plupart des objets demandés ont souvent déjà été demandés par un processus voisin. Il est donc possible d'exploiter la probabilité qu'un nœud voisin est déjà demandé afin de ne pas faire appel à chaque fois au *FS*. De plus, les demandes sont axées au démarrage d'un programme, pendant un temps court, et fréquemment sur un nombre faible de fichiers.
Notre cas d'utilisation typique est le suivant: un programme identique qui s'exécute sur N nœuds de calculs en parallèle. Ces nœuds partagent le même système de fichier par le biais de systèmes comme *NFS*. Actuellement, chaque nœud va réaliser la même résolution de dépendances, imposant une pression intense sur le système de fichier et donc de grandes latences d'utilisation.

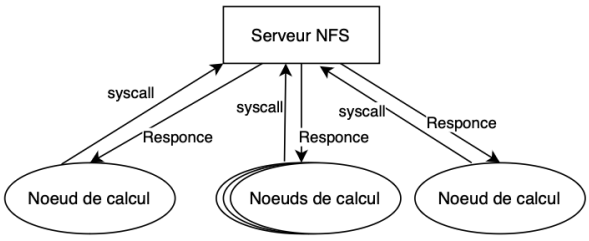


Figure 11 : Représentation initiale de la tempête en HPC

L'idée suggérée par le sujet est de partager des informations aux nœuds voisins exécutant le même programme afin de leur éviter des appels systèmes inutiles. Les informations seront stockées dans un *cache* qui sera consulté avant d'exécuter un appel système.

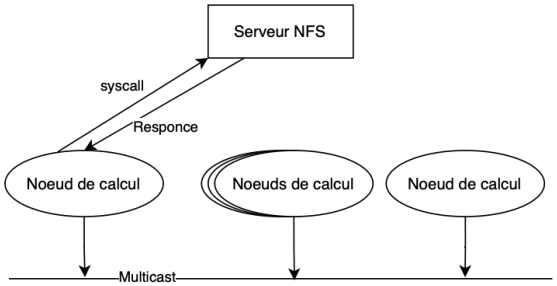


Figure 12 : Représentation graphique de la solution

À noter que n'importe lequel des N nœuds peut être à l'origine du *syscall* de la figure 12 ci-dessus.
Pour aller plus en profondeur, l'idée est de monter, sur chaque nœud, un FS en espace utilisateur **(T)** permettant d'intercepter les appels, de consulter le *cache* et, sur un *cache miss*, de déclencher l'appel sur le filesystem distribué. L'intérêt d'utiliser ce genre de filesystem est qu'il n'y a pas besoin d'écrire du code noyau difficile à tester.

2. Les systèmes de fichier en mode utilisateur

Pour expérimenter ces idées, nous avons utilisé la technologie *FUSE* (Filesystem in Userspace). Elle permet de monter dans l'espace utilisateur, des FS s'appuyant sur le code noyau. L'avantage ici est que ce code noyau n'a pas à être modifié. L'objectif premier est donc de monter un système FUSE dit *passthrough* **(U)**, c'est-à-dire qui permet de transférer directement des opérations système vers le système de fichiers utilisateur sans passer par le noyau.

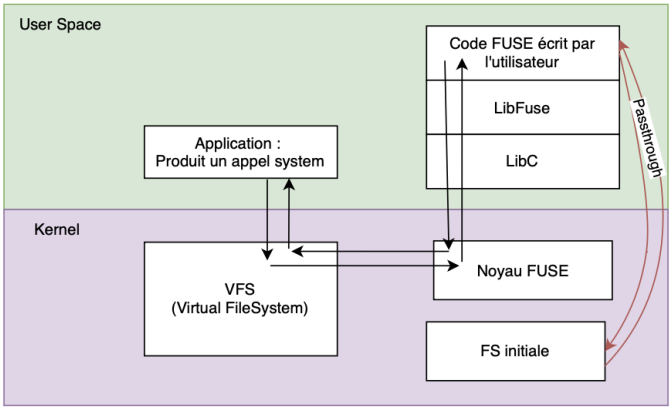


Figure 13 : Fonctionnement de FUSE

Avec *passthrough*, la source du *montage* est le système de fichiers utilisateur où les données sont stockées, et la destination est le point de montage dans le système de fichiers principal où les données du système de fichiers utilisateur sont rendues accessibles.
Au début du stage, les tests ont été réalisés en utilisant un *wrapper* Python de haut niveau. Cependant, ce *wrapper* s'est révélé inadéquat, car il ne permettait pas de tracer de manière précise les appels effectués. La solution a été d'opter pour une implémentation en C beaucoup plus bas niveau du *passthrough*, plus proche des comptes attendus, mais qui ne permet pas de retrouver facilement les chemins des fichiers utilisés lors des appels systèmes (seuls les *inodes* sont disponibles).

3. Les inodes

Un *inode* [16] est une structure de données utilisée dans les systèmes de fichiers pour stocker les *métadonnées* d'un fichier. Chaque fichier ou répertoire dans un système de fichiers possède son propre *inode*, qui contient des informations telles que le numéro d'inode, la taille du fichier, les autorisations d'accès, les dates de création et de modification, les pointeurs vers les blocs de données du fichier, etc. Les inodes permettent au système de fichiers de gérer efficacement les fichiers et de localiser les données associées à chaque fichier en utilisant des références indirectes. Dans le cas de systèmes *NFS*, la documentation indique que pour le même fichier, les inodes sont *identiques* sur toutes les machines montées.
Dans *FUSE passthrough*, un système d'inode est aussi présent. Les inodes *FUSE* sont également des structures, contenant notamment l'inode réelle du fichier. Cependant, à l'instar de *NFS*, deux montages *FUSE* différents d'un même répertoire donneront deux inodes différents.
En pratique, une liste d'inodes est maintenue par le noyau *FUSE*. Cette liste est remplie et vidée par le noyau. Il faut donc prêter une attention particulière à la suppression d'inodes dans cette liste.

4. Le système de cache

Pour ce qui est de la stratégie de cache, nous cherchons à réduire le nombre d'appels à *stat* et *open* sur le *NFS*; il faut donc pouvoir détecter à quel moment deux requêtes sont identiques, peu importe le nœud qui l'initie.

- S'il réussit, open retourne un *descripteur de fichiers*. Cette valeur est donc inhérente au nœud actuel et rien ne sert de la partager. L'information importante à partager ici est "le fichier X existe-t-il ?".
- Stat quant à lui, remplit une structure de diverses informations qui peuvent être cachées et partagées. À noter que les champs relatifs aux heures d'accès peuvent ne pas être à jour dans le cache, et cela ne pose aucun problème au vu de l'utilisation de ce dernier.

En résumé, sur le réseau, les messages doivent contenir une *clé*, typiquement le *chemin* du fichier testé, un booléen indiquant si oui ou non le fichier existe et s'il existe, le contenu de la structure stat.
Pour répondre aux besoins, deux caches sont disponibles. Le premier sert de cache à la fonction *open*. Il indique pour chaque nœud si le fichier existe et si oui, s'il a été ouvert. Pour ce faire, la clé est le chemin vers le fichier et la valeur est *WRONG_PATH* si le fichier n'existe pas à l'emplacement donné, *GOOD_PATH* si le fichier existe, mais qu'il n'a jamais été ouvert et sinon (existe et ouvert), le numéro *d'inode FUSE*. Le second est naturellement celui de *stat*, il contient pour clé *l'inode* réelle du fichier et pour valeur la structure habituellement envoyée par stat. À noter qu'à partir de l'inode FUSE, il est possible de retrouver l'inode réelle. Il est donc possible de retrouver toutes les informations utiles à partir d'un chemin. Le choix de séparer le système de cache en deux se motive par l'espace mémoire utile occupé.

En pratique, deux possibilités d'implémentation coexistent. Premièrement, utiliser une *HashTable* (typiquement celle fournie dans la *glib* (V)), l'avantage étant qu'elle est complète, dynamique et facile d'utilisation, l'inconvénient étant justement qu'elle est dynamique alors qu'un cache se voudrait plutôt de taille fixe, défini à la compilation. Deuxièmement, créer un cache statique, cette technique ayant l'avantage d'être statique, souple au niveau des points de comparaisons (pour la clé du cache) mais l'inconvénient d'être plus longue à implémenter. L'approche par *HashTable* a été celle que nous avons décidé d'explorer au vu de la durée du stage. Cependant, une *interface* de cache est disponible permettant l'implémentation de la

seconde méthode. Avec la *HashTable* de la *glib*, divers problématiques se sont ouvertes, par exemple celle de la libération des ressources ou encore les problèmes de concurrences nécessitant l'initialisation de *mutex*.

5. Le système Multicast

En complément du *cache*, il faut pouvoir propager par le réseau les informations découvertes depuis les différents accès au *FS* sous *NFS*. Pour ce faire, la politique retenue fut la suivante : quand un appel est réalisé, le résultat est mis en cache et est propagé sur le réseau *multicast*. Cette propagation est faite par le *thread* qui fait l'ajout local. Un autre *thread* s'occupe alors de lire ce qu'il voit passer sur le réseau et d'ajouter les infos dans les caches.

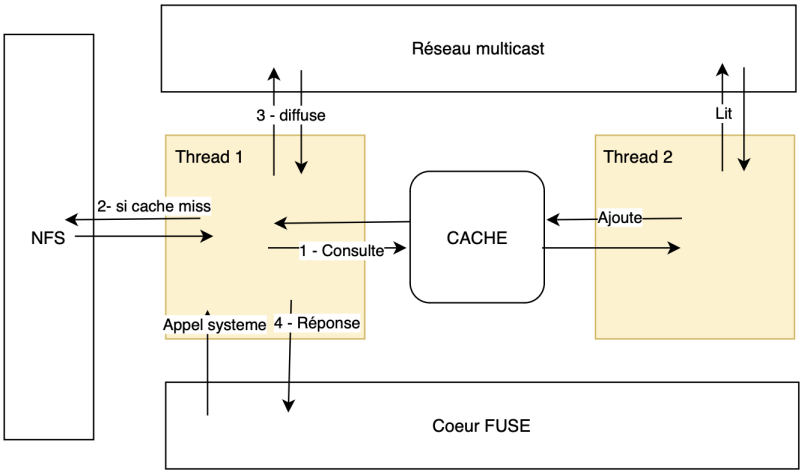


Figure 14 : Threads applicatifs

Une solution alternative serait d'avoir un buffer producteur consommateur qui réalise les différents envois sur le réseau et qui, par la primitive *SELECT*, fait les lectures ou écritures nécessaires sur le réseau.

Les messages transmis sur le réseau ont la structure prédéfinie suivante :

- Un *entier* pour la taille du chemin,
- Le *chemin* sous la forme d'un char*,
- Un *booléen* indiquant l'existence ou l'absence du fichier,
- La *structure* stat si le chemin existe (0 sinon)

L'intérêt de cette méthode est que l'ensemble des messages sont structurellement identiques, et qu'un message tient entièrement dans les données d'un paquet *UDP* (moins de 1472 octets). Ceci évite d'avoir plusieurs paquets pour un seul message et donc d'avoir un entremêlement des ressources reçues.

La contrepartie de cette solution est qu'il faut définir une taille maximale sur les chemins transmis, et si un chemin dépasse cette taille, le message ne sera pas propagé sur le réseau, donc l'ensemble des nœuds devra faire cet appel. Cependant, la taille max définie étant élevée, il est peu probable qu'un chemin dépasse cette taille.

6. Synthèse

La solution proposée est donc la suivante :

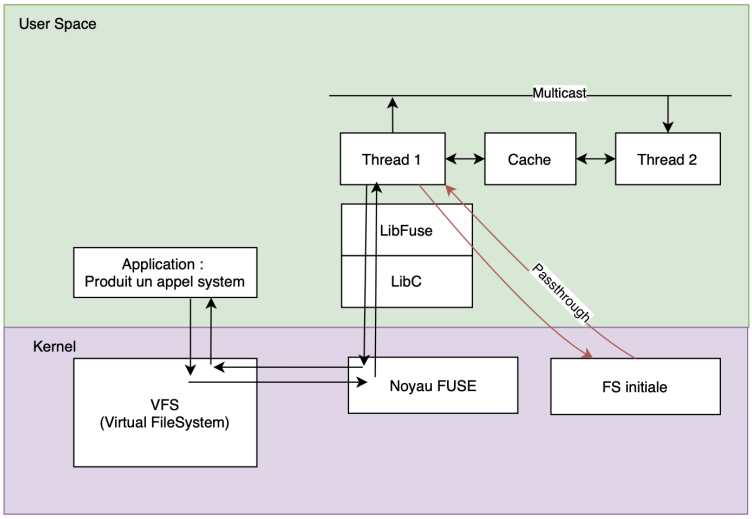


Figure 15 : Schéma applicatif

C. Expériences et résultats

L'ensemble des expériences ci-dessous ont été réalisés sur la plateforme *Grid'5000*, en utilisant le site de *Grenoble* et le cluster *dahu*. Il embarque 32 nœuds avec chacun :

- en *CPU*, 2 Intel Xeon Gold 6130, 16 cœurs, d'architecture X86_64
- en *mémoire*, 192 Gio
- en *stockage*, 240 Go + 480 Go de SSD, 4 To de HDD

le tout relié en *réseau* 10 Gbps. **(W)**
De plus, une application générant un exécutable a N dépendances a été développée par Quentin **(X)**. Il donne la main au développeur sur le nombre de dépendances à résoudre et sur leur localisation. Pour toutes les expériences, c'est cette application que l'on nommera ***dummy_app*** qui a été utilisée.

1. Local

La première expérience a été réalisée en local, c'est-à-dire sur un seul poste sans montage *NFS*. Son objectif est de tester le système afin d'observer le Stat Storm et d'évaluer le gain sur un seul poste avec le système de *cache*. Le protocole suivi est le suivant :

1. Génération d'une application a 10 dépendances dans le répertoire tmp :
   dummy_app build --nb_libs 10 --store tmp
2. Calcul, avec *strace* **(Y)**, de l'ensemble des appels de la tempête :
   strace -c tmp/app-10/bin/app
3. Création d'un dossier destination pour le système *FUSE* :
   mkdir dest
4. Calcul avec le système *FUSE* du nombre d'appels observés :
   ./passthrough_ll -f -o source=tmp dest
5. Dans un autre terminal, exécuter l'application depuis la destination montée. Pour cela, il faut redéfinir les chemins de recherche pour que la résolution se déroule dans le répertoire monté plutôt que dans la source. De plus, il faut prêter attention aux tabulations d'autocomplétion qui vont compter comme des appels :
   LD_LIBRARY_PATH=$(dummy_app ld_library_path --nb_libs 10 --store dest) dest/app-{nb_libs}/bin/app
6. Une fois l'exécution terminée, stopper le système *FUSE* et observer les résultats sur la sortie.

Pour calculer le coût de *l'overhead FUSE*, il est possible de simplement démarrer et arrêter le système *FUSE*.

Voici un tableau récapitulant les différentes opérations lors de l'exécution de l'application *dummy_app* résolvant 10 dépendances.

	Nombre total	Dont Erreurs
Stat	30	18
Open	1254	1243

Figure 16 : Appels et erreurs calculés par strace pour une application a 10 dépendances

	Nombre total	Dont Erreurs
Stat	44	0
Open	1258	1225

Figure 17 : Appels et erreurs calculés par le systeme FUSE sans caches pour une application à 10 dépendances

	Nombre total	Dont erreurs
Stat	34	0
Open	138	105

Figure 18 : Appels et erreurs calculés par le systeme FUSE avec caches pour une application à 10 dépendances

Nous pouvons observer que le système *FUSE* avec cache présente un avantage d'un facteur 10 en nombre d'appels à la primitive open par rapport au système de base. Le nombre d'appels à stat reste quant à lui du même ordre de grandeur. Le système semble donc avoir un intérêt en termes de nombre d'appels.

2. Les expérimentations

Une fois la version locale au point, nous avons expérimenté l'intégralité du système. Pour ce faire, la plateforme de tests *Grid'5000* et ses outils ont été utilisés. Pour faciliter les déploiements et les tests, j'ai utilisé l'outil *NixOS compose* **(Z)**, développé il y a peu par l'équipe DATAMOVE. Il permet de définir une configuration système pour diverses cibles (*VM*, *Docker*, ou *Grid'5000*) et offre ainsi une abstraction de la machine destination. Parallèlement, pour lancer les expériences, un script *Execo* a été écrit **(AA)**. Il permet entre autres de déployer des nœuds depuis un programme python.

Le protocole de l'expérience est le suivant :

- Déployer un nœud comme étant serveur *NFS* et N nœuds comme étant les clients NFS.
- Le serveur va générer un exécutable à partir de *dummy_app*
- Tous les clients vont faire leurs montages *FUSE*
- Une fois montés, les clients exécutent N fois l'exécutable de *dummy_app* (il est important d'exécuter plus d'une fois l'application dans le même montage pour observer les bénéfices du cache)
- Les clients redirigent dans un fichier le temps avant l'exécution, le résultat de l'exécution et le temps à la fin de l'exécution

Ce protocole se retrouve également dans le projet (Expérience).
Avec ce modèle, le calcul du nombre d'appels étant complexe à étudier (aucune manière directe de récupérer depuis le *FUSE* le résultat dans le script *Execo*), nous avons fait le choix de nous concentrer sur l'étude du temps d'exécution d'un programme à l'intérieur de notre système plutôt que sur son nombre d'appels.
Le script *Execo* permet de faire dérouler l'entièreté du protocole. Il prend, entre autres en entrée, le nombre de nœuds sur lesquels déployer, le nombre dépendances à générer pour *dummy_app* ainsi que le nombre de répétitions de l'exécution de l'exécutable de *dummy_app* sur le nœud.

3. Résultats et discussion

En déroulant l'expérience ci-dessus sur 20 nœuds de *Grid'5000*, nous obtenons les résultats suivants :

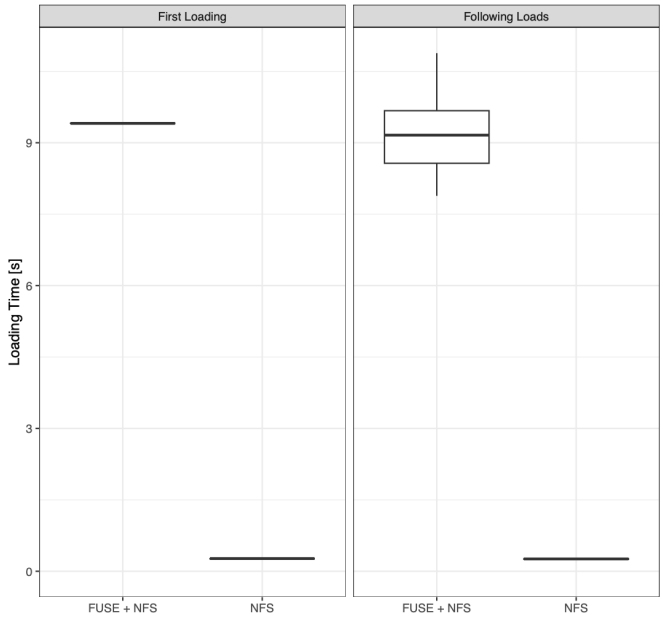


Figure 19 : Résultat d'expérience sur 20 nœuds et 50 librairies

Sur la figure 15, à gauche se trouve le temps d'exécution de la première exécution de l'application *dummy_app* et à droite se trouvent les temps des N-1 autres. Le temps *NFS* reste constant du fait de son cache interne (**AB**). Les temps d'exécution sont calculés en soustrayant le temps post-exécution avec le *temps* pré-exécution. L'utilisation des *barplots* permet de comparer de manière visuelle les temps *NFS* simple avec le montage *FUSE* par-dessus *NFS*.

Le *script R* utilisé pour l'exploitation des résultats est disponible en annexe (Script R).
La reproductibilité des expérimentations a été une préoccupation majeure tout au long de cette étude. Chaque étape méthodologique a été minutieusement documentée, permettant ainsi une vérification aisée des résultats obtenus. Néanmoins, il convient de noter que certaines concessions ont été faites, notamment en raison des contraintes temporelles du stage. Ces choix ont été guidés par la nécessité de garantir la cohérence et la pertinence des expérimentations dans le cadre imparti.
Contrairement aux expériences locales, le graphique ci-dessus ne semble pas montrer d'avantages significatifs. Au contraire, le surcoût induit par le système *FUSE* paraît trop important pour rivaliser avec la vitesse de *NFS* combiné à son cache. Le système dans son état actuel n'apporte ainsi aucun gain en termes de temps d'exécution. Pour diversifier ces propos et essayé d'apprécier la solution, cela peut être causé par un nombre de nœuds et de bibliothèques trop peu important pour affecter une pression suffisamment forte à *NFS* pour le

ralentir de façon notable. Une autre piste à explorer serait d'identifier les éléments du système *FUSE* étant les plus gourmands en temps afin de les optimiser.

V. Bilan

A. Appréciation du stage

L'appréciation de mon stage de recherche au sein du LIG relève d'une profonde satisfaction et d'une reconnaissance pour cette expérience enrichissante. Durant ces quinze semaines, j'ai pu m'immerger dans le domaine du HPC et j'ai été soutenu par un environnement de travail stimulant.

Les problématiques auxquelles j'ai été confronté ont attisé ma curiosité et au fil des phases de compréhension, de conception et de développement de la solution, j'ai pu mettre en pratique les connaissances et compétences théoriques acquises tout au long de ma formation. Les nombreuses activités et conférences organisées au sein du bâtiment IMAG ont enrichi mon expérience au LIG. Ces occasions de partage et d'apprentissage ont complété mon stage en me permettant d'élargir mes connaissances et de découvrir de nouvelles perspectives.

En conclusion, ce stage de recherche au LIG a dépassé le simple cadre professionnel pour devenir une aventure intellectuelle et humaine. Il m'a non seulement permis de mieux me connaître, mais a également renforcé mon intérêt grandissant pour la recherche en informatique et mon attrait pour les côtés systèmes de celui-ci. Cette expérience a solidifié ma détermination à poursuivre tôt ou tard une carrière dans le domaine de la recherche.

B. Pour aller plus loin

Le sujet abordé durant ce stage est extrêmement vaste et complexe, offrant de multiples perspectives de recherche. La problématique de Stat Storm dans les architectures systèmes HPC est encore en partie inexplorée, ouvrant la voie à divers axes d'études. À partir de la rentrée prochaine, une thèse sera entamée, portant en partie sur ce sujet, au sein de DATAMOVE. Elle aura pour objectif d'approfondir davantage les recherches amorcées lors de ce stage. Une première publication est envisagée dans les six premiers mois de celle-ci.

Durant ces quinze semaines, il a été nécessaire de faire des compromis afin de respecter les contraintes de temps imposées par le stage. L'un de ces compromis a été de privilégier une version dynamique du cache plutôt qu'une version statique, car celle-ci pouvait être mise en œuvre plus rapidement. Bien que cette version dynamique offre une base solide pour des améliorations futures et nous a permis d'effectuer les premiers tests de performance de la solution, une version statique serait appréciable parce qu'elle se rapprocherait davantage de la réalité d'usage.

De plus, il serait judicieux d'étendre les expérimentations en augmentant le nombre de nœuds de calcul, de dépendances ou d'itérations par exemple. Ces ajustements permettraient de renforcer la pression sur le NFS et donc de rendre nos résultats plus pertinents. De plus, explorer d'autres politiques d'ajout en cache pourrait également offrir des pistes pour améliorer l'efficacité du cache, et réduire plus encore la latence induite par le Stat Storm.

C. La gestion de projet

En ce qui concerne l'organisation du projet, M. Olivier Richard, M. Quentin Guilloteau et moi-même tenions chaque semaine une réunion de suivi, au cours de laquelle nous faisions un point sur l'avancement des travaux réalisés jusqu'alors. Ces réunions nous ont permis de

définir clairement les objectifs à atteindre pour la semaine suivante, tout en tenant compte des éventuels ajustements nécessaires en fonction des résultats obtenus. Elles nous ont aussi offert l'occasion de discuter des compromis réalisés et des problèmes rencontrés.
Grâce à cette approche itérative et collaborative, nous avons maintenu un rythme de travail et avons pu surmonter les obstacles rencontrés. De plus, la proximité géographique de nos bureaux, dans le bâtiment IMAG, m'a permis de rendre régulièrement visite à mes encadrants afin de débattre des solutions et compromis que je mettais en place au fur et à mesure de l'avancement du projet.
Cette gestion de projet efficace m'a permis de gérer mon temps de manière optimale, en accordant une attention particulière aux tâches prioritaires, pour être le plus efficace possible, et de rester concentré sur les objectifs fixés. Cela a grandement contribué au succès de mon stage de recherche et à la réalisation des missions qui m'ont été confiées.

D. Connaissance acquise

Au cours de ce stage, j'ai eu l'opportunité d'acquérir des connaissances et des compétences dans le domaine du HPC et des architectures de type store. J'ai découvert les problématiques liées aux Sttat Sorms et j'ai pu me familiariser avec les techniques et outils utilisés pour l'observer. La phase de compréhension du sujet m'a permis d'apprendre à lire des articles de recherches scientifiques, tandis que la phase de conception et de développement de la solution m'a permis d'appréhender les défis logiciels que cela requiert. J'ai notamment pu développer mes connaissances autours des librairies partagées, des systèmes de fichiers ou encore autour de nouveaux environnements comme NixOS.

De plus, les échanges réguliers avec mes encadrants et les autres membres de l'équipe DATAMOVE m'ont permis d'approfondir mes soft-skills, comme la communication et la collaboration au sein d'un environnement de recherche. Cette expérience m'a permis d'en savoir plus sur qui je suis et ce que je souhaite devenir, en particulier concernant ma passion pour l'informatique.

E. Compétences développées

En plus des nombreuses connaissances techniques que j'ai acquises durant ce stage, j'ai aussi développé plusieurs compétences clés pour un ingénieur en Informatique, particulièrement en administration d'infrastructures informatiques et automatisation du traitement de l'information.
En effet, j'ai eu l'occasion de réaliser de nombreux déploiements sur la plateforme de tests et d'effectuer à la préparation des plans de déploiement ainsi que des protocoles expérimentaux. Ensuite, j'ai suivi attentivement les étapes de mise en œuvre du déploiement, par le biais de l'outil NXC, et j'ai procédé à des tests et évaluations pour m'assurer du bon fonctionnement de l'infrastructure déployée.
Par ailleurs, j'ai également acquis des compétences supplémentaires en conception de solutions distribuées. J'ai ainsi établi un cahier des charges identifiant les besoins clés de l'application avant de sélectionner les technologies appropriées (FUSE, glib, Execo,…) à là distributions du programme dans le but de concevoir une solution adaptée aux besoins du stat storm.

Pour ce qui est de l'automatisation du traitement d'information, qui par ailleurs a été un aspect important de mon stage, j'ai réalisé des études de faisabilité et d'efficacité pour automatiser certaines tâches, comme le déploiement de l'application ou encore l'exploitation des résultats. En développant ces scripts, j'ai pu expérimenter l'application en modélisant les entrées afin de simuler et mesurer au mieux les performances de la solution mise en place.

En somme, grâce à ce stage et aux différentes expériences rencontrées, j'ai renforcé mes connaissances techniques et mes compétences d'ingénieur en informatique, notamment dans les domaines orientés systèmes. Ces compétences seront des atouts précieux pour ma future carrière d'ingénieur.

VI. Conclusion

En conclusion, ce stage LIG a été une expérience aussi enrichissante qu'inspirante. Durant ces quinze semaines intenses, j'ai eu l'opportunité de m'immerger pleinement dans le domaine passionnant du HPC et d'explorer en profondeur la problématique du "Stat Storm". J'ai pu mettre en œuvre les compétences théoriques acquises tout au long de ma formation et les appliquer à des situations concrètes et complexes.

Bien que les résultats globaux du système soient compromis par l'intensité temporelle de l'overhead FUSE, l'expérience a montré des points clés pertinents à développer. L'efficacité en termes de nombres d'appels systèmes est suffisamment performante pour continuer à développer le projet. Cette étude ne constitue qu'une première étape dans un domaine prometteur. Les résultats obtenus ouvrent la voie à des perspectives de recherche plus approfondies. La thèse qui débutera à la rentrée prochaine sera l'occasion d'explorer en profondeur les nombreuses facettes du "Stat Storm", en augmentant notamment le nombre de nœuds, les dépendances et les itérations dans les expérimentations. L'approfondissement de cette recherche ouvrira des horizons à la solution et pourra venir s'inscrire dans les outils NixOS compose pour accélérer les étapes de déploiement.

Ce stage de recherche a été une expérience formatrice, riche en enseignements, en découvertes et en échanges. Je tiens une nouvelle fois à exprimer ma gratitude envers toutes les personnes qui ont contribué à cette aventure.

VII. Références

[1] - Francieli Zanon Boito, Eduardo C. Inacio, Jean Luca Bez, Philippe O. A. Navaux, Mario A. R. Dantas, and Yves Denneulin. 2018. A Checkpoint of Research on Parallel I/O for High-Performance Computing. ACM Comput. Surv. 51, 2, Article 23 (March 2019), 35 pages. https://doi.org/10.1145/3152891

[2] - M. Véstias and H. Neto, "Trends of CPU, GPU and FPGA for high-performance computing," 2014 24th International Conference on Field Programmable Logic and Applications (FPL), Munich, Germany, 2014, pp. 1-6, doi: 10.1109/FPL.2014.6927483.

[3] - Lüttgau, J., Kuhn, M., Duwe, K., Alforov, Y., Betke, E., Kunkel, J., & Ludwig, T. (2018). Survey of storage systems for high-performance computing. *Supercomputing Frontiers and Innovations*, *5*(1).

[4] - Francieli Zanon Boito, Eduardo C. Inacio, Jean Luca Bez, Philippe O. A. Navaux, Mario A. R. Dantas, and Yves Denneulin. 2018. A Checkpoint of Research on Parallel I/O for High-Performance Computing. ACM Comput. Surv. 51, 2, Article 23 (March 2019), 35 pages. https://doi.org/10.1145/3152891

[5] - Dai, H., Wang, Y., Kent, K. B., Zeng, L., & Xu, C. (2022). The state of the art of metadata managements in large-scale distributed file systems—Scalability, performance and availability. IEEE Transactions on Parallel and Distributed Systems, 33(12), 3850-3869.

[6] - Ludovic Courtès. 2021. Taming the ‘stat’ storm with a loader cache. https://guix.gnu.org/en/blog/2021/taming-the-stat-storm-with-a-loader-cache/

[7] - Bruno Bzeznik, Oliver Henriot, Valentin Reis, Olivier Richard, and Laure Tavard. 2017. Nix as HPC package management system. In Proceedings of the Fourth International Workshop on HPC User Support Tools (HUST'17). Association for Computing Machinery, New York, NY, USA, Article 4, 1–6. https://doi.org/10.1145/3152493.3152556

[8] - Farid Zakaria. 2022. Making RUNPATH redundant for Nix. https://fzakaria.com/2022/09/12/making-runpath-redundant-for-nix.html

[9] - Wolfgang Frings, Dong H. Ahn, Matthew LeGendre, Todd Gamblin, Bronis R. de Supinski, and Felix Wolf. 2013. Massively parallel loading. In Proceedings of the 27th international ACM conference on International conference on supercomputing (ICS '13). Association for Computing Machinery, New York, NY, USA, 389–398. https://doi.org/10.1145/2464996.2465020

[10] - F. Zakaria, T. R. W. Scogland, T. Gamblin and C. Maltzahn, "Mapping Out the HPC Dependency Chaos," SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, 2022, pp. 1-12, doi: 10.1109/SC41404.2022.00039.

[11] - Harmen Stoppels. 2022. Stop searching for shared libraries. https://stoppels.ch/2022/08/04/stop-searching-for-shared-libraries.html

[12] - Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z. Ibrahim, and Jay Srinivasan. 2016. Scaling Spark on HPC Systems. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16). Association for Computing Machinery, New York, NY, USA, 97–110. https://doi.org/10.1145/2907294.2907310

[13] G. Johansen and B. Mauzy. Cray XT programming environment's implementation of dynamic shared libraries. In Cray User Group, Atlanta, Georgia, May 2009.

[14] S. M. Kelly, R. Klundt, and J. H. Laros III. Shared libraries on a capability class computer. In Cray User Group, Fairbanks, Alaska, May 2011.

[15] M. G. F. Dosanjh, P. G. Bridges, S. M. Kelly, and J. H. Laros III. A peer-to-peer architecture for supporting dynamic shared libraries in large-scale systems. In Fifth International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), 2012.

[16] - Tanenbaum, A. S. (2002). Modern operating systems Fourth edition. Herbert Bos.

VIII. Glossaire

HPC : High-performance computing, activitées sur les super-calculateurs
FPGA : Field Programmable Gate Arrays, Circuit programmables et reconfigurables
GPU : Graphics Processing Unit
Dynamic Loader : Programme chargé de résoudre les dépendances dynamiques
RTLD-audit : Interface programmable du loader
Inode : structures de données contenant des informations des fichiers stockés
VFS : Virtual FileSystem, une couche d'abstraction au-dessus d'un système de fichiers
FHS : Filesystem Hierarchy Standard, norme pour l'arborescence des fichiers
Dummy_App : Application pour créer une application a N dépendances
G5K : Plateforme de tests
NFS : Network FileSystem
Unix-Like : Système d'exploitation qui se comporte de façon semblable à un système Unix
NXC : NiXosCompose, outil de déploiement et de tests
Nix : Langage de programmation fonctionnel
Nix Os : Os intégrant Nix
store : Localisation de stockage des paquets
Guix/Spark : Alternatives a Nix
RUNPATH/RPATH : Métadonnée pour le stockage des localisations de dépendances
LD_LIBRARY_PATH : Variable d'environnement pour le stockage des localisations de dépendances
/etc/ld.so.cache : Fichier cache mappant dépendances et localisations

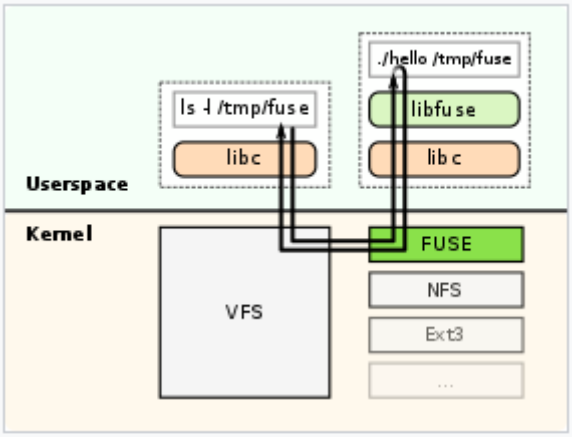
IX. Sources

(A) - DATAMOVE, https://www.inria.fr/fr/datamove
(B) - High-performance computing, https://en.wikipedia.org/wiki/High-performance_computing
(C) - FPGA, https://en.wikipedia.org/wiki/Field-programmable_gate_array
(D) - GPU, https://en.wikipedia.org/wiki/Graphics_processing_unit
(E) - NFS, https://www.ibm.com/docs/en/aix/7.1?topic=management-network-file-system
(F) - Unix-Like, https://www.computerhope.com/jargon/u/unix-like.htm
(G) - LIG, https://www.liglab.fr/
(H) - IMAG, https://batiment.imag.fr/
(I) - Nix/NixOs, https://nixos.org/
(J) - grid'5000, https://www.grid5000.fr/w/Grid5000:Home
(K) - FHS, https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard
(L) - Rpath/Runpath, https://en.wikipedia.org/wiki/Rpath
(M) - ldd, https://man7.org/linux/man-pages/man1/ldd.1.html
(N) - objdump, https://linux.die.net/man/1/objdump
(O) - patchelf, https://man.archlinux.org/man/patchelf.1.en
(P) - QT, https://www.qt.io/blog/2011/10/28/rpath-and-runpath
(Q) - SPINDLE, https://computing.llnl.gov/projects/spindle
(R) - Manuel RTLD-Audit, https://man7.org/linux/man-pages/man7/rtld-audit.7.html
(S) - Lustre, https://www.lustre.org/
(T) - FUSE, https://en.wikipedia.org/wiki/Filesystem_in_Userspace,
(U) - LibFUSE, https://github.com/libfuse/libfuse
(V) - GLib, https://fr.wikipedia.org/wiki/GLib
(W) - Hardware G5K, https://www.grid5000.fr/w/Grenoble:Hardware#dahu
(X) - Dummy App, https://gitlab.inria.fr/nixos-compose/stages/sbrun/-/tree/main/dev/nxc/pkgs/dummy_app
(Y) - Man Strace, https://man7.org/linux/man-pages/man1/strace.1.html
(Z) - NixOs compose, https://gitlab.inria.fr/nixos-compose/nixos-compose
(AA) - Execo G5K, https://gitlab.inria.fr/mimbert/execo
(AB) - Cache NFS, https://www.ibm.com/docs/fr/aix/7.3?topic=performance-nfs-tuning-client

X. Annexes

A. Le fonctionnement de FUSE

FUSE, est une technologie qui permet à un système de fichiers d'être implémenté en espace utilisateur plutôt qu'en noyau. Cela signifie que les opérations courantes sur les fichiers sont gérées par des processus en espace utilisateur, offrant une grande flexibilité et facilité de développement (notamment pour les tests) pour la création de systèmes de fichiers personnalisés.
FUSE fonctionne en interceptant les appels système du système de fichiers et en les redirigeant vers un programme en espace utilisateur qui les traite selon la logique du développeur. Cette approche permet de créer des systèmes de fichiers virtuels, tels que des systèmes de fichiers chiffrés, des systèmes de fichiers réseau personnalisés ou des systèmes de fichiers exotiques, sans nécessiter de modifications profondes dans le noyau du système d'exploitation.



A flow-chart diagram showing how FUSE works: Request from userspace to list files (ls -l /tmp/fuse) gets redirected by the Kernel through VFS to FUSE. FUSE then executes the registered handler program (./hello) and passes it the request (ls -l /tmp/fuse). The handler program returns a response back to FUSE which is then redirected to the userspace program that originally made the request.

Figure 20 : Fonctionnement de FUSE

B. A propos de NIX

Nix, NixOS et Nixpkgs constituent un écosystème de gestion de paquets logiciels et configuration système.
Nix est un gestionnaire de paquets fonctionnel qui garantit la reproductibilité des environnements logiciels en enregistrant les dépendances et les versions dans des espaces isolés appelés "store".
NixOS, est un système d'exploitation basé sur Nix, où chaque composant est défini de manière déclarative dans des fichiers de configuration. Cette approche permet des déploiements cohérents et prévisibles, ainsi que la possibilité de revenir à des versions antérieures du système.
Nixpkgs est la collection de paquets logiciels prêts à être installés avec Nix, et suit également une approche déclarative, permettant de spécifier précisément les dépendances et les configurations de chaque paquet.

Ensemble, ces éléments offrent un environnement puissant et fiable pour la gestion et la configuration des logiciels et des systèmes, avec un accent particulier sur la reproductibilité, la modularité et la flexibilité.

C. L'utilité des bibliothèques dynamiques dans les architectures type store

Dans un contexte de store, l'approche du dynamic linking revêt un intérêt significatif. Les librairies, bien que nécessaires, sont souvent utilisées par un nombre restreint de binaires spécifiques. Dans cette optique, conserver le dynamic linking s'avère avantageux, car il minimise la duplication de code et d'espace mémoire en permettant le partage efficace des objets partagés entre les applications. Cette approche se révèle particulièrement bénéfique dans un environnement où plusieurs cœurs de traitement collaborent sur un ensemble de librairies

communes. Les performances peuvent être calculées avec précision, car les interfaces préchargées améliorent l'efficacité et la rapidité des opérations. Le dynamic linking présente également l'avantage de permettre des ajustements plus fluides et des mises à jour sans nécessiter une recompilation complète, ce qui est crucial dans un contexte de store en évolution constante. Bien que le passage au linking statique puisse sembler séduisant, il peut potentiellement perturber la structure optimisée de l'environnement de store en brisant ses avantages du partage dynamique. Dans l'optique de la production finale, le dynamic linking émerge donc comme une solution stratégique, capable d'optimiser la performance, la flexibilité et la gestion des ressources au sein d'un contexte de store en évolution permanente.

D. Différence entre loader et linker

Un loader et un linker sont deux éléments essentiels du processus de compilation et d'exécution des programmes informatiques. Le linker, également appelé éditeur de liens, est responsable de la liaison des différents modules et bibliothèques d'un programme pour créer un exécutable final. Il résout les références entre les différentes parties du code et génère un fichier exécutable complet.
D'un autre côté, le loader est chargé d'exécuter un programme en mémoire. Il lit l'exécutable généré par le linker, charge le code et les données en mémoire et initialise l'environnement d'exécution nécessaire à l'application. Le loader joue un rôle crucial dans le chargement et l'exécution des programmes, assurant que le code soit placé correctement en mémoire et que toutes les dépendances soient satisfaites.

En résumé, le linker est responsable de la création d'un exécutable à partir de différentes parties du code, tandis que le loader s'occupe du chargement et de l'exécution effective du programme en mémoire.
Source : https://www.geeksforgeeks.org/difference-between-linker-and-loader/