Institut **S**upérieur
d'**I**nformatique de
Modélisation et de
leurs **A**pplications

24, Avenue des Landais
BP 10 125
63 173 AUBIERE cedex

Organisation **E**uropéenne
pour la **R**echerche **N**ucléaire

CERN
CH-1211 Genève 23
Switzerland

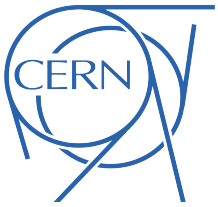
Rapport d'ingénieur
Stage de 2ème année
Filière Génie Logiciel et Systèmes Informatiques

Améliorer le fonctionnement sans disque de la distribution Red Hat grâce à la fusion de système de fichiers.

Présenté par : **Pierre Schweitzer**

Responsable ISIMA : **Emmanuel Mesnard**
Responsable entreprise : **Loïc Brarda**

9 Septembre 2011



Institut **S**upérieur
d'**I**nformatique de
Modélisation et de
leurs **A**pplications

24, Avenue des Landais
BP 10 125
63 173 AUBIERE cedex

Organisation **E**uropéenne
pour la **R**echerche **N**ucléaire

CERN
CH-1211 Genève 23
Switzerland

Rapport d'ingénieur
Stage de 2ème année
Filière Génie Logiciel et Systèmes Informatiques

Améliorer le fonctionnement sans disque de la distribution Red Hat grâce à la fusion de système de fichiers.

Présenté par **: Pierre Schweitzer**

Responsable ISIMA **: Emmanuel Mesnard**
Responsable entreprise **: Loïc Brarda**

9 Septembre 2011

Remerciements

Par la présente page, je tiens à remercier toutes les personnes qui m'ont aidées d'une façon ou d'une autre dans la réalisation de mon travail. J'aimerai remercier dans un premier temps Loïc Brarda et Niko Neufeld pour m'avoir accepté pour cette mission et m'avoir encadré pendant ces cinq mois au gré des mes errances et problèmes. Leur présence et leur soutien m'ont beaucoup aidé, de même que leurs conseils m'ont permis d'en apprendre énormément sur les systèmes Linux. Par ailleurs, sans le temps précieux qu'ils m'ont offert, il m'aurait été impossible d'aller aussi loin dans la réponse à la problématique du stage et encore moins au delà de celle-ci. Leur bonne humeur a aussi été d'un grand secours !

Je tiens également à remercier deux anciens de l'ISIMA, Jean-Christophe Garnier et Christophe Haen pour leur aide et le temps passé à regarder les problèmes que je rencontrais et à dénicher les erreurs que j'avais écrites.

De façon générale, je veux remercier l'ensemble de l'équipe Online de LHCb (dont il serait trop long d'énoncer tous les noms) pour son accueil, sa bonne humeur et ma rapide intégration dans leur « famille ».

Je souhaite aussi remercier Emmanuel Mesnard pour sa visite de stage au CERN que nous avons réussi à transformer en visite de l'expérience.

Et pour finir, je tiens à adresser des remerciement tous particuliers à Murielle Mouzat pour ses réponses à mes mails pour tous mes problèmes rédactionnels.

Acknowledgments

Using the following page, I would like to thank all the people who helped me whatever the way to achieve my work. First, I want to thank Loïc Brarda and Niko Neufeld for having accepted me for this mission and for having supervised me during those five months dealing with my wanderings and errors. Their presence and their support helped me a lot, as well as their advice that let me learn a lot about Linux systems. Moreover, without the precious time they offered me, I could not have went that far in the answer to the internship problem and even less far going beyond it. Their good mood was great help.

I also would like to thank two old guys from the ISIMA, Jean-Christophe Garner et Chistrophe Haen for their help and the time they spent to look at the issues I was facing and to spot the errors I had written.

In a more general way, I have to thank the whole LHCb Online Team (from which it would be to long to enumerate all the names) for its greeting, its great mood and my quick integration in their « family ».

I also want to thank Emmuanuel Mesnard for his internship visit to the CERN that we managed to turn into an experiment visit.

And finally, I want to address particular thanks to Murielle Mouzat for her answers to my mails regarding all my writing issues.

Résumé

Le travail durant ce stage a permis de revoir, toute la gestion de la ferme de calcul **sans disques** qu'utilise l'expérience LHCb du CERN. Celle-ci est maintenant entièrement administrable par l'intermédiaire du logiciel de gestion **Quattor** et autorise à la fois l'administration du Hight Level Trigger (HLT) mais également du trigger L0 (contrairement à ce qui était initialement prévu). Cette nouvelle gestion, fortement indépendante des version du système Linux utilisé, **Scientific Linux CERN** (SLC), permet le passage des CCPCs de SLC4 à SLC5 très facilement.

Un résultat important est l'absence d'utilisation des outils Red Hat pour administrer cette ferme, ce qui ouvre la voie à une futur **migration** de l'ensemble de la ferme à SLC6, quand celle-ci sera prête. L'abandon des outils Red Hat a nécessité l'utilisation de la **fusion de systèmes de fichiers** pour assurer l'unicité d'un répertoire racine partagé par plusieurs machines. Jusqu'alors la fusion de systèmes de fichiers n'était pas utilisée, et seuls certains fichiers étaient modifiables sur les nœuds. La fusion apporte donc une plus grande souplesse.

La fusion de systèmes de fichiers « standard » telle qu'on peut la trouver dans les pilotes déjà existant n'étant pas suffisante, une nouvelle solution, le système de fichiers **PierreFS**, a été développée pour répondre aux besoins spécifiques de l'expérience LHCb.

Mots clés : sans disques, Quattor, Scientific Linux CERN, migration, fusion de systèmes de fichiers, PierreFS.

Abstract

Work made during this internship let CERN LHCb experiment rethink the way its complete **diskless** farm server for physics computation was administrated. Now, the complete farm can be administrated using the **Quattor** management software and allows LHCb Online Team to administrate both High Level Trigger (HLT) and L0 trigger (even if it was not initially targeted). This new management method, mostly independent from the release of Linux system used, **Scientific Linux CERN** (SLC), enables the switch of the CCPCs from SLC4 to SLC5 easily.

One major achievement is that Red Hat tools are no longer required to administrate this farm. That way, this allows a future **migration** of the complete farm to SLC6, when the release will be ready. Quitting Red Hat tools has created the need of **file systems union** to ensure that a single shared root file system could remain. Up to now, file systems union was not used, and only a few files were open to modifications on the nodes. Here, file systems unions brings higher flexibility.

« Standard » file systems union, as it can be found implemented in already existing drivers was not sufficient. Then, a new solution, matching exactly the LHCb specific needs has, have been developed. It is a new file system called **PierreFS**.

Keywords: diskless, Quattor, Scientific Linux CERN, migration, file system union, PierreFS.

Table des matières

Table des illustrations

Introduction

L'expérience LHCb au CERN, pour effectuer des traitements sur les résultats obtenus lors de l'acquisition de données sur les collisions, utilise une ferme de serveurs. Ces serveurs puisqu'ils sont dédies au calcul n'embarquent aucun dispositif de stockage. En d'autres termes, on ne trouve pas de disque dur sur ces machines. Tout est déporté. C'est le principe de machines diskless. Un serveur central est chargé de fournir à tous ces serveurs clients tout ce dont ils ont besoin pour fonctionner. Notamment un système d'exploitation et de quoi stocker des données.

Ce système repose sur une méthode décrite, documentée par Red Hat[1], fournisseur de la distribution Linux principale dont dérive celle du CERN, SLC (Scientific Linux CERN). Red Hat fournit également des outils permettant le déploiement d'une telle infrastructure.

Néanmoins, cette méthode présente des défauts, et manque de souplesse. Par ailleurs, la nouvelle distribution Red Hat, dans sa version 6 abandonne les outils de création de serveurs et de clients diskless[2].

L'opportunité est donc donnée pour reprendre le principe, mais de zéro pour l'intégrer proprement à l'outil de gestion des serveurs (Quattor) qu'utilise le CERN. Et profiter de cette occasion pour améliorer le fonctionnement du diskless.

Comme il existe déjà des solutions permettant de répondre à certaines problématiques du diskless rencontrées au CERN, dans un premier temps, il a fallu les tester, et sélectionner celle qui présente le plus d'avantages pour le CERN. Il n'était pas question initialement de concevoir et de développer une solution spécifique au CERN. Une fois démonstration faite des capacités, il fallait intégrer l'ensemble dans le processus de déploiement de nouveaux serveurs, en limitant les problèmes et surtout les complications dans ledit processus, déjà lourd.

1 Environnement de travail

1.1 Le CERN

Le CERN, Organisation Européenne pour la Recherche Nucléaire est une organisation fondée en 1954. Aujourd'hui 20 états européens[1], connus sous le nom d'états membres, forment le conseil d'administration et de direction du CERN. Plusieurs autres états sont soit observateurs, soit participants.

Le CERN a plusieurs missions. Sa première mission est la recherche scientifique. Le but est de comprendre l'univers et connaitre sa création. Sa seconde mission est de pousser les technologies a leurs limites et donc de les repousser. C'est ainsi que l'on a eu la création du web. La troisième mission est une mission de collaboration, amener différents pays à travailler ensemble. Enfin, la dernière mission et pas des moindres, est de former les futures générations de scientifiques.

Le CERN est notamment connu pour les découvertes qui y ont été faites et qui ont eu des application concrètes hors des laboratoires du CERN, bien qu'elles ne soient pas nécessairement dans le domaine de la physique.

À l'heure actuelle, on parle beaucoup du CERN en raison de son outil principal, le LHC, des missions qui lui sont rattachées et des nouveaux records régulièrement établis.

1.2 Le LHC

Le LHC, Large Hadron Collider (en français : grand collisionneur de hadrons) est un accélérateur de particules et même le plus grand au monde. Il se trouve en Suisse et en France, à 175 mètres sous le sol à l'emplacement de l'ancien accélérateur du CERN, le LEP (Large Positron Electron collider – en français : grand collisionneur de positrons et d'électrons). Le principe de ces instruments est de faire tourner des particules à grande vitesse, proche de la lumière si possible. Au LHC, ce sont des hadrons (protons ou ions de plomb) qui sont mis en rotation, puis dont on observe les collisions. Le but de ces collisions est de recréer les conditions de naissance de l'univers (entre autres) et de confirmer/infirmer des théories physiques et des postulats sur l'existence ou non de particules (telle que le célèbre mais hypothétique boson de Higgs).

Le LHC doit ainsi permettre de répondre à un certain nombre de questions. Ce qui implique donc que différentes observations doivent être faites sur les collisions, toutes les informations n'étant pas nécessaires pour répondre à une question donnée. Et surtout parce qu'il serait impossible de tout capturer à un endroit précis, étant donné la nature différente des données.

On trouve donc six détecteurs sur le LHC. Ceux-ci sont situés sous terre, sur le LHC. Ces détecteurs sont ALICE, ATLAS, CMS, LHCb, LHCf, TOTEM[2,3].

1 D'ici 2015, ce nombre sera porté à 23. Par ailleurs, tous les états membres ne seront plus européens, avec l'arrivée d'Israël (en 2013).

2 Un septième détecteur a été approuvé en décembre 2009. Il s'agit du détecteur MoEDAL (Monopole and Exotics Detector At the LHC). Sa mise en place fonctionnelle devant être achevée cette année.

3 Les abréviations signifiants respectivement : A Large Ion Collider Experiment, A Toroidal LHC ApparatuS, Compact Muon Solenoid, Large Hadron Collider beauty, Large Hadron Collider forward, TOTal Elastic and diffractive cross section Measurement.

1.3 L’expérience LHCb

LHCb est le nom à la fois d’une expérience, mais aussi du détecteur associé. Il signifie Large Hadron Collider beauty. Le but de l’expérience est de voir les différences entre la matière et l’antimatière en passant par l’étude d’un type de particules : le quark beauty (aussi nomme quark b, en français quark beauté). On trouve bien évidemment son oppose l’antiquark b (antimatière). Le LHC recréant les conditions de genèse de l’univers (où il y avait matière et antimatière), il est donc possible de capturer les deux.

Illustration 1 : Salle de contrôle de l’expérience LHCb

Pour ce faire, LHCb utilise un détecteur, constitue de sous-détecteurs spécialisés (RICH, MUON, etc), au cœur duquel des collisions ont lieu. L’électronique des sous-détecteurs récupère les informations. Ceci génère un trafic très important de l’ordre 50Mo/s par collision. Il faut donc une capacité de calcul suffisante pour pouvoir gérer ce flux de données convenablement. D’autant que toutes les données ne sont pas nécessairement utiles à l’expérience et qu’un tri s’avère obligatoire.

1.4 La ferme diskless

La ferme diskless est un ensemble de 1400 serveurs. Ces serveurs constituent ce qu’on appelle le HLT (High Level Trigger, en français déclencheur de haut-niveau). On le nomme déclencheur de

haut-niveau, puis qu'à la sortie du détecteur, avant cette ferme, on trouve un premier trigger, de niveau zéro (Level-0 trigger) constitué de micro-électronique sur-mesure (que l'on trouvera désignée sous le nom de TELL1, eux-mêmes contrôlés par des PC embarqués au format carte de crédit, en anglais, Credit Card PCs - CCPCs). Le but de ce premier déclencheur est de faire le tri parmi les données des collisions de particules, afin de voir quelles sont les bonnes candidates pour analyse. Si tel est le cas, alors, les données des collisions sont envoyées par chaque TELL1 par l'intermédiaire d'un switch gérant 900 connexions vers un des serveurs de la ferme diskless dont une des instances de calcul est disponible. Celle-ci prend la relève, pour confirmer l'hypothèse et recréer l'ensemble de l'évènement.

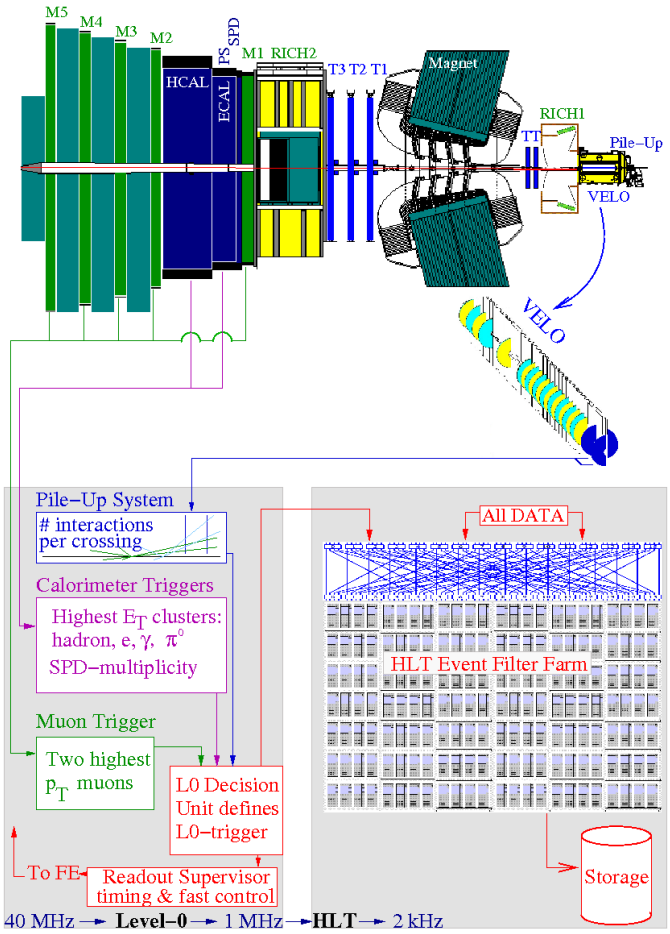


Illustration 2 : Le détecteur et les déclencheurs

Pour gérer le fonctionnement de ces serveurs diskless, on trouve 50[1] serveurs qui vont servir de point central. Ces serveurs fournissent aux clients le système d'exploitation via un partage NFS. Puis, sur chaque serveur sont reliées 28 machines clientes (ou encore nœud « node » en anglais).

1 Au 5 mai 2011. Ce chiffre évolue régulièrement.

Illustration 3 : Quelques-uns des nœuds

Le système d'exploitation fourni aux machines clientes est présent de façon unique sur chaque serveur et est distribué aux 28 nœuds. Il s'agit de la distribution Red Hat Enterprise Linux 5 (RHEL5), avec des modifications propres au CERN, ce qui a conduit à l'appellation Scientifc Linux CERN 5 (SLC5). Ce système d'exploitation est donc fourni en lecture seulement pour éviter qu'un client n'influence les autres. Cependant, un système d'exploitation, pour fonctionner, a besoin d'écrire dans certains fichiers afin, notamment, de conserver un état entre chaque redémarrage[2]. Sur le serveur central, une portion d'espace disque, appelée snapshot, disponible en écriture et propre à chaque client est également exportée. Le snapshot se superpose au système d'exploitation pour permettre au système en fonctionnement de stocker ses données et d'accéder à sa configuration spécifique (principalement réseau). Pour parvenir à ce résultat, jusque là, un procédé plutôt délicat était utilisé. On fournissait à chaque client une liste des fichiers qui étaient disponibles en lecture et écriture. Puis, les clients montaient lesdits fichiers par dessus les fichiers en lecture seule pour permettre l'enregistrement et la restauration d'état.

On voit donc ici apparaitre un défaut flagrant. Seuls les fichiers désignés sont disponibles en écriture. Toute tentative d'écriture ailleurs (même dans un nouveau fichier) est donc impossible. Cela limite donc les possibilités et noie l'utilisateur sous des informations pas nécessairement pertinentes

2 Il est également possible de ne pas conserver d'état. Dans un tel cas, on parle de machine diskless stateless. Mais, alors, à chaque redémarrage, la machine repart de zéro, comme si elle n'avait jamais fonctionné. Ceci n'est pas le comportement recherché par le CERN. Par ailleurs, le stateless ne permet pas de donner une identité à chaque machine, vu qu'elles partagent toutes absolument les mêmes fichiers.

(voir : Annexe 1).

1.5 Le projet

Le but du projet soumis était donc d'améliorer le fonctionnement de la ferme diskless, afin de supprimer les inconvénients qu'elle présente à l'heure actuelle. Néanmoins, toutes les méthodes ne sont pas bonnes, et une avait été sélectionnée en particulier par le CERN : la fusion de système de fichiers (attention, en anglais « file systems union » et non « merge »).

La fusion de système de fichiers permet, de façon transparente à l'utilisateur final, d'avoir plusieurs sources de données réunies dans un seul endroit. C'est-à-dire qu'un dossier *a* résultant d'une fusion de systèmes de fichiers peut contenir un fichier *b* provenant du disque dur de l'ordinateur local et un fichier *c* provenant d'un partage réseau. L'utilisateur lui ne voit pas quel fichier vient d'où et tout fonctionne comme si le répertoire venait de la même source.

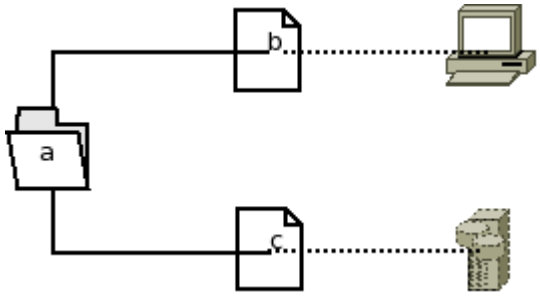


Illustration 4 : Vue simplifiée de la fusion de systèmes de fichiers

Il fallait donc, dans un premier temps prendre en main le concept même de système diskless, puis reprendre la configuration du CERN, la modifier, déployer la fusion de système de fichiers, et rendre le tout facilement déployable. Du moins, c'était la théorie.

Le cahier des charges lié à ce projet était simple. Il fallait avoir un système diskless plus performant et plus souple en n'utilisant que la fusion de système de fichiers (qu'importe le type de fusion utilisée). Ainsi, la liste des fichiers qui peuvent ou non être écrit devait disparaitre au profit d'un système totalement transparent. Tout fichier est potentiellement modifiable de façon séparée et unique sur chaque machine cliente. L'autre contrainte était que la méthode employée devait s'appliquer à la fois sur la ferme actuelle, mais sur la ferme dans le futur, à la mise à jour du système d'exploitation. Enfin, la contrainte finale était la possibilité d'intégration de la configuration de l'ensemble dans l'outil de gestion du parc informatique (de production) de LHCb : Quattor.

Une demande sous-jacente était donc la réécriture du composant Quattor en charge de la gestion des fermes diskless pour qu'il soit lui aussi plus souple et plus performant, et que, bien évidemment, il supporte la fusion de systèmes de fichiers.

Ce projet était donc un grand saut dans l'inconnu. La nouvelle version de la distribution Linux du CERN, SLC6, n'ayant encore jamais été testée pour du diskless, les fusions de systèmes de fichiers non plus, et encore moins le déploiement d'un système diskless sans outil de configuration. De ce fait, il était impossible d'établir à l'avance un diagramme de Gantt prévisionnel sur le travail à fournir et les étapes à franchir, vu que le travail à faire était jusque là inconnu, évoluant au jour le jour en fonction des résultats et la correspondance aux contraintes.

2 Solutions logicielles et administration système

2.1 Diskless sous SLC6

2.1.1 Spacewalk

La première étape du projet consistait donc à réaliser du diskless sous SLC6. D'après le document[2] statuant de la disparition de l'outil de configuration utilisée pour le diskless, il était possible d'utiliser un produit Red Hat pour le faire : Satellite. Ce produit, comme beaucoup de produits Red Hat est payant, je me suis donc tourne vers la solution open-source et gratuite (mais néanmoins supportée par Red Hat) sur laquelle il est basé : Spacewalk. Spacewalk est donc, à l'instar de son équivalent payant, une solution qui permet à partir d'un serveur central d'administrer et de gérer plusieurs clients. Cela allant de l'installation[1] au déploiement de nouveaux paquets sur toutes les machines, tout en s'assurant qu'elles sont à jour (d'un point de vu logiciel), le tout avec une interface de gestion de paquets et de configurations.

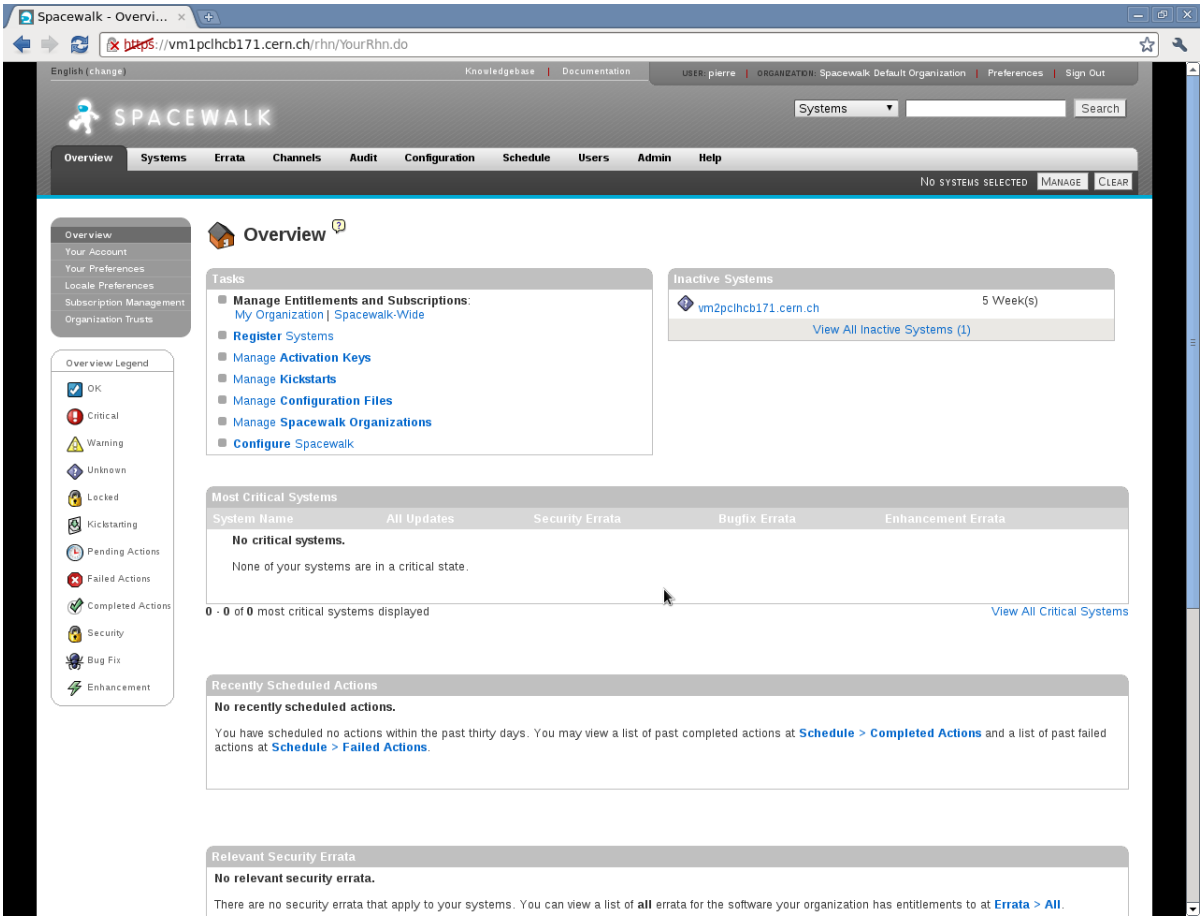


Illustration 5 : Accueil de Spacewalk

1 Le terme installation est ici erroné, puisqu'en raison d'un bug au niveau de l'interface de Spacewalk dans l'ajout des systèmes qu'il administre, seule la réinstallation d'un système déjà installé et administré par Spacewalk est possible. Le bug est déjà âgé.

Spacewalk doit donc être installe sur un serveur central. La majeur partie de la configuration de cet outil passe par l'utilisation d'une interface web. C'est-à-dire qu'il suffit de lancer un navigateur web sur n'importe quelle machine et d'aller à l'adresse de la machine serveur spacewalk.

Par la suite, à chaque installation d'une nouvelle machine, si on désire qu'elle soit administrée par Spacewalk, il faut également installer Spacewalk dessus. Néanmoins, il s'agit juste de la partie cliente de Spacewalk ici, bien plus légère. C'est elle qui communique avec le serveur Spacewalk. L'interconnexion initiale entre les deux se fait d'une seule commande. Afin d'identifier le groupe de machines auquel ajouter le nouveau client, Spacewalk utilise des clés uniques d'enregistrement. Ainsi, sur la machine cliente, lors de la saisie de la commande d'enregistrement, il suffit de fournir la bonne clé. Puis, le logiciel client se connecte au serveur pour effectuer son enregistrement.

Mais, s'il s'avère qu'effectivement, Spacewalk rempli certaines des fonctionnalités attendues du diskless (gestion centralisée, installation facilitée, gestion de configuration unique, logiciels identiques), il faut néanmoins un disque dur sur chaque machine, ce qui viole donc l'une des contraintes du projet. Par ailleurs, les nombreux bugs rencontres lors de l'utilisation de Spacewalk, plus une lourdeur liée au Java nécessaire pour faire tourner Spacewalk m'ont conduit à l'abandon de cette solution. Il fallait tout reprendre à partir de zéro.

2.1.2 Installation manuelle

Arrivé ici, il fallait donc faire « manuellement » un système diskless. En soit, cela n'est pas insurmontable. Il suffit de configurer un serveur DHCP, un serveur NFS, et enfin un serveur TFTP. Puis, de déployer un noyau et une image de démarrage. Fort heureusement Red Hat fournit une documentation[3] sur comment réaliser un tel système diskless et donc la solution au problème de l'image de démarrage.

2.1.2.1 Serveurs DHCP, TFP et NFS

Les étapes de la documentation concernant la mise en place d'un serveur DHCP n'ont pas été utilisées. En effet, pour simplifier la recherche, tout a été fait en machines virtuelles, via la librairie de virtualisation libvirt, son interface graphique fournie par Red Hat, ainsi que l'interface en ligne de commande : virsh. Cette dernière permet de proprement configurer le réseau auquel sont rattachées les machines virtuelles. Et donc, de configurer le serveur DHCP de libvirt. Ainsi, sur la machine hôte, aucun besoin de serveur DHCP.

La configuration du serveur TFTP est elle très simple et proprement interfacée avec libvirt qui fournit des passerelles. L'installation et la configuration du serveur NFS étant suffisantes dans leur version standard, celles-ci ont été conservées. Seuls les exports ont été ajoutés.

2.1.2.2 Création du système d'exploitation partagé

La documentation impose qu'un système soit déjà installé sur une machine et qu'on en fasse une copie. Dans mon cas, il était plus simple de partir d'une installation vierge. Un outil tel que yum permet de simuler l'installation de la distribution dans un répertoire à l'aide d'une commande telle que : `yum --installroot=/scratch/netboot/root groupinstall Base`. Dans cette ligne de commande, l'option installroot permet de spécifier le répertoire racine (le rootfs) dans lequel va travailler yum, puis l'option groupinstall permet de sélectionner un ensemble de paquet à installer (ici le groupe « Base » contient le minimum nécessaire pour faire tourner une distribution SLC6). Cette

commande crée donc entre autres l'arborescence et tous les fichiers de configuration nécessaires. C'est donc cette commande que j'ai utilisée. Et j'ai ensuite exporté ce répertoire dans NFS.

L'installation par yum présente néanmoins quelques défauts auxquels il faut palier avant d'avoir un système utilisable. Tout d'abord le groupe Base ne contient pas le serveur SSH[1]. Il m'a donc fallu l'installer avec la commande : `yum –installroot=/scratch/netboot/root install openssh-server`. Syntaxiquement, on retrouve la même chose que précédemment, à la différence près que install a remplacé groupinstall, puisqu'un seul paquet est à installer. Par ailleurs, un autre paquet vient automatiquement avec le groupe Base, il s'agit de yum-autoupdate. Ce paquet installe juste un script de démarrage qui permet de mettre à jour le système via yum dès qu'il démarre. Ceci ne présente aucun intérêt sur un système diskless. Il est en effet plus judicieux de réaliser la mise à jour sur le serveur qui possède le système de fichiers exporté (avec une commande comme : `yum –installroot=/scratch/netboot/root update`). Je l'ai donc supprimé grâce à la commande `yum : yum –installroot=/scratch/netboot/root remove yum-autoupdate`. Enfin, le dernier probleme mais pas des moindres : lors de l'installation de tout ordinateur (que ce soit sous Linux, Windows ou un autre système d'exploitation) un premier utilisateur est créé et son mot de passe est demandé à l'utilisateur. Avec le groupinstall de yum, l'utilisateur yum est bien créé, mais sans mot de passe ni possibilité de se connecter. Il faut donc impérativement définir un mot de passe pour activer le compte et permettre son utilisation. Dans la majorité des cas, les mots de passes sont stockés de façon chiffrée sous Linux, dans un fichier nomme shadow dans le répertoire /etc. Afin de ne pas laisser trainer le mot de passe en clair partout, il existe une commande qui permet de définir le mot de passe directement en fournissant la version chiffrée. Ainsi donc, il m'a suffit de faire la commande suivante pour changer le mot de passe : `echo root:$1$ls/e3eCd$prLP8Ao23Fh/OdL6rO7XW1 | chroot /scratch/netboot/root chpasswd -e`

Cette commande est intéressante à plus d'un titre et permet de bien comprendre le mécanisme. Dans un premier temps, on demande l'écriture du compte (root) et du mot de passe, séparés par « : ». Ceci permet donc d'avoir ce duo dans la sortie standard. À ce moment, on utilise une redirection « | » qui permet de rediriger la sortie standard vers l'entrée standard. Ainsi, la sortie de la commande echo (le texte qui suit) ne sera pas affiché, mais envoyé directement à la commande qui suit. Cette commande suivante est chroot. Elle signifie « change root ». À l'instar de installroot pour yum, elle permet de changer le répertoire racine de travail d'une commande. Ici, on change donc le répertoire racine vers /scratch/netboot/root pour la commande chpasswd. Et c'est donc cette commande chpasswd qui va effectuer le changement de mot de passe, dans le répertoire racine indiqué, en lisant l'entrée standard (en recevant le résultat de la commande echo). L'option -e permettant d'indiquer que le mot de passe est chiffré, qu'il n'y a donc pas besoin de le chiffrer.

Enfin, il restait une dernière modification à faire. Les systèmes Linux disposent d'un fichier nommé fstab (pour File Systems TABle) situé dans le répertoire /etc. Ce fichier, comme son nom l'indique, contient l'ensemble des systèmes de fichiers à monter automatiquement[2] au démarrage. Si sa présence n'est pas obligatoire, elle permet d'éviter des problèmes et des messages d'erreurs. J'ai donc rajouté un fichier fstab contenant le minimum requis pour SLC6 :

```
devpts          /dev/pts        devpts  gid=5,mode=620  0 0
tmpfs           /dev/shm        tmpfs   defaults        0 0
sysfs           /sys            sysfs   defaults        0 0
proc            /proc           proc    defaults        0 0
```

On ne trouve dans ce fstab que des systèmes de fichiers virtuels qui n'ont aucune existence physique. Ils ne fonctionnent qu'en mémoire et leurs données disparaissent en même temps que la

1 Une caractéristique particulièrement surprenante, d'autant plus que sous SLC5 le serveur SSH est dans le groupe Base.

2 Sous réserve néanmoins que le système de fichier n'ait pas l'option noauto dans ses options.

machine est éteinte. Cela rend donc ce fstab quelque peu « universel ».

Un message d'erreur n'allait pas disparaitre, c'est celui du répertoire racine manquant. Mais ici, il était impossible de le mettre, puisque ce répertoire résulte d'une fusion.

Fort de toutes ces modifications, le répertoire racine partage était prêt à l'utilisation.

2.1.2.3 Configuration du démarrage réseau

La configuration du démarrage des machines via PXE, devait se faire « à la main », contrairement à l'ancien système où elle était générée. Néanmoins, de ce côté, rien ne changeait (si ce n'est une syntaxe plus moderne et plus simple).

Voici le fichier d'une des machines diskless :

```
DEFAULT SLC6_diskless
LABEL SLC6_diskless
      SAY Now booting the kernel from SLC6...
      LINUX vmlinuz-2.6.32-71.el6
      APPEND initrd=initramfs-2.6.32-71.el6.img
root=nfs:192.168.122.1:/scratch/netboot/root selinux=0 rdshell rdinitdebug
      TIMEOUT 30
```

Si on le détaille :

- DEFAULT indique quel label utiliser par défaut ;
- LABEL indique une configuration (donc SLC6_diskless) ;
- LINUX indique quelle image du noyau Linux utiliser. Cela permet donc d'avoir plusieurs noyaux qui cohabitent ;
- APPEND contient toute une ligne de commande qui sera transmise et utilisée par le noyau. Elle contient notamment l'image de démarrage et le root file system à utiliser. On voit bien le diskless puisqu'il est indiqué un serveur NFS et une localisation ;
- TIMEOUT est le temps pendant lequel l'utilisateur peut intervenir pour changer une configuration lors du démarrage avant que ne démarre le système par défaut. C'est une valeur en dixièmes de secondes. Donc, ici, cela correspond à 3 secondes.

2.1.2.4 Génération de l'image de démarrage

Ce dernier point nécessite un peu plus de travail. En effet, une image de démarrage contient tout ce dont le noyau a besoin pour démarrer proprement et initialiser le système d'exploitation correctement. Si cette image est incorrecte, ou absente, le noyau arrêtera de fonctionner immédiatement après son démarrage. Dans le cas du diskless, cette image contient toute la configuration et les applicatifs nécessaires pour que le noyau puisse monter le système de fichier racine à partir d'un serveur distant. Dans la majorité des cas, cette image est un ramfs (généralement nommée initramfs.img). Autrement dit, un système de fichier monté en RAM, qui est libéré une fois que le système a fini de démarrer.

C'est alors que le point important dans la documentation est traité. La génération de l'image de démarrage. Un nouvel outil a fait son entrée dans SLC6, il s'agit de dracut. Il remplace tous les anciens outils de génération d'image de démarrage (mkinitrd, mkinitramfs). Sa force est sa souplesse. Il est entièrement configurable à l'aide de scripts qui modulent ce qui va être sur l'image et la façon dont le démarrage va se produire. En effet, pour démarrer, Linux a besoin d'un script, l'initscript qui réalise

toutes les initialisations (matérielles et logicielles) pour rendre le système opérant. Avec dracut, il est possible de simplement modifier cet initscript grâce à des hooks qui seront lances à des instants précis (et choisis). Et par défaut[2], dracut embarque d'emblée tous les outils nécessaires pour faire démarrer un système diskless. Il est à noter que dans mon cas, vu que je virtualisais avec du materiel « non standard » virtio, Linux était incapable de gérer le matériel proprement. Là, dracut a encore prouvé toute sa force, puisqu'il est possible de lui indiquer des pilotes à embarquer. Dans ce cas, dracut les rajoute à l'image de démarrage et s'assure qu'ils soient rapidement démarrés pour que les services qui en dépendent ne souffrent pas de leur absence.

Fort de tout cela, pour avoir une image de démarrage utilisable pour du diskless, il suffisait de taper :

```
dracut -f -k /scratch/netboot/root/lib/modules --add-drivers 'virtio virtio_pci virtio_ring virtio_net virtio_blk' /scratch/netboot/tftpboot/initramfs-2.6.32-71.el6.img 2.6.32-71.el6
```

Où :

- -f force dracut à recréer l'image si elle existe déjà ;
- -k suivi du chemin indique où récupérer les pilotes et modules à mettre dans l'image pour le bon fonctionnement du noyau ;
- --add-drivers et la liste qui suit indiquent quels pilotes supplémentaires inclure (ici, ceux pour la virtualisation) ;
- le chemin suivant indique où mettre l'image résultante ;
- le numéro qui suit est le numéro de version du noyau.

J'avais enfin du diskless sans aucun outil graphique de configuration (sans outil de configuration en général même !). Ce qui est intéressant, c'est de constater que toutes les étapes réalisées ici (sauf celles avec virsh, mais le CERN ne virtualise pas HLT et cela n'est pas en projet) auraient pu l'être via un script. Il est donc aisé de réaliser un script qui permette d'automatiser la création d'un nœud diskless et la configuration pour les clients.

J'ai donc créé deux machines diskless. J'ai pu constater qu'elles partageaient les mêmes clés publiques SSH. En toute logique.

Sur la machine serveur, l'arborescence utilisée est simple. Dans /scratch/netboot/root se trouve le système partagé par tous les nœuds diskless. Chacun y accède en lecture-seule. Puis dans /scratch/netboot/tftpboot se trouve tout le matériel nécessaire pour un démarrage sur le réseau. L'image du noyau, mais également l'image de démarrage et les configurations pour les machines (dans le dossier pxelinux.cfg). Ce répertoire restera utilisé pour tous les essais. Le système de démarrage réseau reste le même, seule la configuration des machines change.

2 Sous réserve néanmoins d'avoir installé les bons paquets sur la distribution, dracut-network dans notre cas. Dans le cas d'une installation à partir du site internet de dracut, tout est fourni dans une seule et même archive.

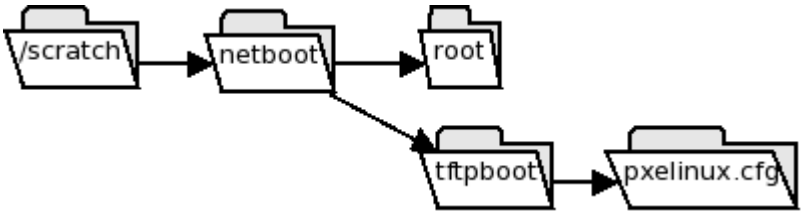


Illustration 6 : Arborescence initiale

2.2 Fusion de systèmes de fichiers

Après quelques recherches sur internet, il apparait que deux solutions viables existent aujourd'hui pour la fusion de systèmes de fichiers. Toutes deux sont un système de fichiers qui encapsule les autres systèmes de fichiers.

La première solution et plus ancienne est unionfs. Ce pilote existe depuis 2003. C'est lui qui a donné naissance à la deuxième solution : aufs (Another UnionFS). Si aufs était d'abord une amélioration de unionfs, c'est aujourd'hui une réécriture. Les deux ne fournissent pas les mêmes fonctionnalités (même si certaines se recoupent, évidemment).

Il existe encore d'autres solution, mais qui s'appuient sur FUSE. Dans le cas du CERN, une telle solution ne semble pas intéressante au premier abord, ni même réalisable puisque le montage et la fusion se font au démarrage, pour tous les utilisateurs avant même que le système ne soit complètement initialisé (avant même le « switch root », voir : Annexe 2).

Dans Linux, on trouve une interface commune pour tous les systèmes de fichiers appelée VFS (Virtual File System, en français Système de Fichiers Virtuel). C'est par cette interface que transitent tous les appels systèmes pour gérer les fichiers. Ainsi dès qu'une application tente d'ouvrir un fichier, via l'appel fopen, par exemple, l'exécution se termine dans VFS. C'est alors VFS qui va faire la redirection vers le bon système de fichiers, qui pourra alors traiter la requête[4].

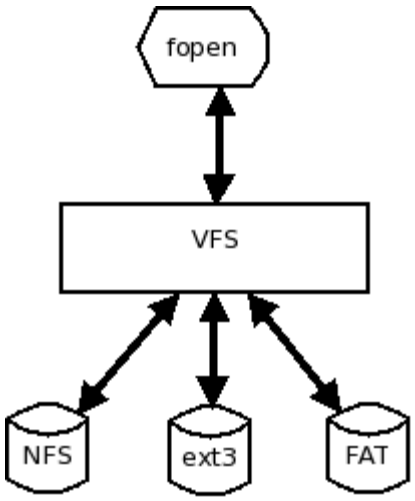


Illustration 7 : Le rôle d'interface de VFS

La fusion de système de fichiers rajoute une couche dans ce mécanisme et apporte la notion de pile de systèmes de fichiers. Quand une application tente d'accéder à un fichier dans un répertoire qui résulte de la fusion de deux autres répertoires, la demande va être transmise à VFS selon le principe standard. Puis, VFS va faire le relai vers le pilote de système de fichiers responsable de la fusion (unionfs ou aufs). Celui-ci va alors déterminer de quel fichier il s'agit et surtout de son emplacement réel. À partir de là, il va effectuer une descente dans ladite pile et transmettre la demande au pilote du système de fichiers qui dispose réellement du fichier (dans notre cas, le pilote NFS). Celui-ci gérera alors la requête sans même savoir que cela n'est pas VFS qui transmis la requête. Évidemment, ceci peut poser des problèmes, les pilotes de systèmes de fichiers n'étant pas conçus pour une telle mise en couche. Mais nous y reviendrons plus bas.

Il existe des cas où la fusion de systèmes de fichiers est particulièrement ardue. On pensera notamment aux cas où : on supprime un fichier qui est sur une branche en lecture-seule, on renomme un fichier sur une branche en lecture-seule, ou encore on modifie un fichier sur une branche en lecture-seule. Ici, le pilote de fusion de systèmes de fichiers doit garder l'état, sans pour autant pouvoir modifier le fichier de base. Dans ce genre de situations, il crée un fichier sur la branche[1] disponible en écriture. Trois cas de figures se présentent.

Dans le cas de la modification, le pilote va réaliser un « copyup ». Ce qui consiste ni plus ni moins à copier le fichier depuis la branche en lecture-seule vers la branche qui est disponible en écriture pour effectuer la modification dessus.

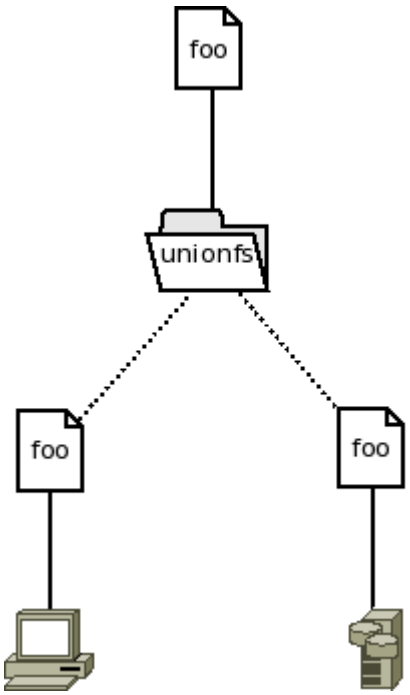


Illustration 8 : Fichier en lecture-seule foo modifié

1 Dans le cas où plusieurs branches sont disponibles en écriture pour une même fusion, le comportement varie selon le pilote. Alors qu'unionfs privilégie la première branche disponible, aufs utilise le principe de load-balancing et utilise la branche la moins chargée, et la plus disponible. C'est l'une des forces de ce pilote.

Ici, encore, si le fichier est déjà éditable (donc sur une branche accessible en écriture), alors le pilote ne fera aucun copyup.

Dans le cas de la suppression, le pilote va créer un fichier spécial « whiteout »[1]. Cela signifie que pour la fusion, le fichier n'existe plus. Et quand l'utilisateur liste le contenu du répertoire, le pilote ne fait plus apparaitre ni le fichier, ni le whiteout. Le fichier de whiteout utilise la syntaxe suivante : .wh.<nom du fichier d'orgine>. Ainsi, si le fichier foo[2] sur la branche en lecture-seule doit être supprimé, alors un fichier .wh.foo sera crée sur la branche accessible en écriture.

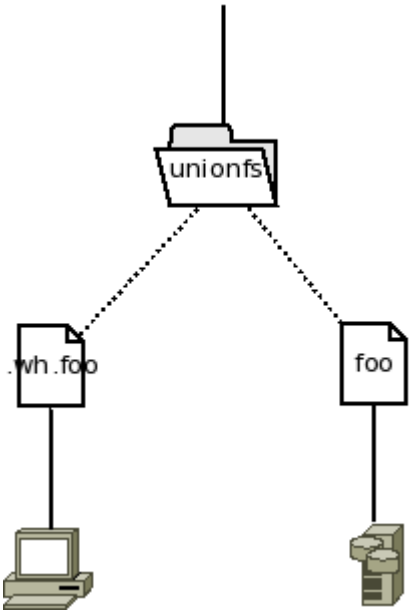


Illustration 9 : Fichier en lecture-seule foo supprimé

Il va de soit que dans le cas d'un suppression d'un fichier existant uniquement sur une branche où il est possible d'écrire, le pilote supprime directement le fichier sans passer par le mécanisme de whiteout. Il n'y a aucune difficulté ici.

Le pilote pour gérer simplement le copyup et le whiteout utilise un mécanisme de priorités qu'il affecte aux branches. La priorité est représentée par un nombre (entier et positif). Plus le nombre est faible, plus la priorité est élevée, à l'instar du facteur de nice sous Linux. Par conséquent, dans une fusion avec une branche en lecture-seule et une branche disponible en écriture, cette dernière récupère la priorité la plus élevée. Ainsi, quand une demande pour un fichier est réalisée par une application, le pilote va d'abord regarder dans la branche de priorité la plus élevée et s'il ne trouve rien, descendre dans la suivante et ainsi de suite. Donc, lorsque dans la branche de priorité la plus haute se trouve un fichier whiteout, le fichier est considéré comme supprimé, et si un fichier portant le nom est disponible, c'est celui-ci qui est retourné au demandeur.

1 En référence à la gamme de liquides correcteurs « wite out » (rachetée par Bic depuis).
2 On reprend ici les conventions de nommage anglaises utilisées dans la documentation de unionfs.

Dans le cas du renommage, le pilote va fonctionner en deux temps. Dans un premier temps, il va réaliser un copyup du fichier sur la branche disponible en écriture. Puis, il va renommer ce fichier vers le nom demandé. Enfin, pour que le renommage soit effectif, il va créer un whiteout, pour cacher le fichier de la branche en lecture-seule qui existe toujours. De ce fait, le renommage est terminé. L'utilisateur ne voit plus que le fichier renommé.

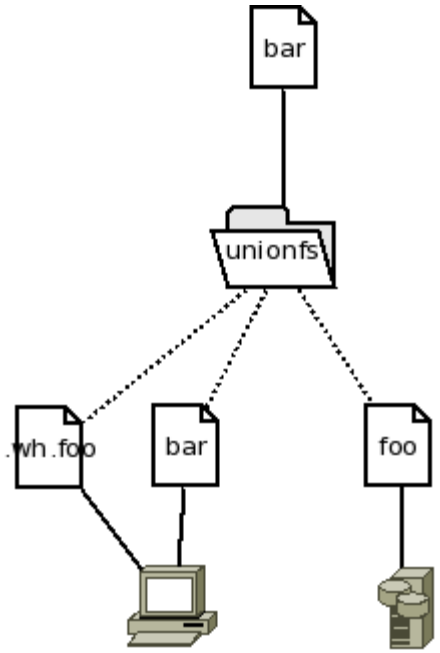


Illustration 10 : Cas du fichier foo en lecture-seule renommé en bar

2.2.1 Another Union FS (aufs)

Comme il fallait faire un choix entre les deux systèmes de fichiers disponibles, le choix a été guidé par le maintien des logiciels. Ceci n'a rien d'un choix arbitraire. Il permet d'assurer une certaine qualité au composant, et surtout un support, un suivi et des correctifs en cas de bugs découverts lors de l'utilisation. À première vue, le développement de unionfs semblait plus ou moins au point mort, le site également. C'est pourquoi, je me suis tourne vers aufs.

Aufs semblait également présenter un grand avantage : il pouvait être compilé comme module de Linux. En effet, Linux supporte deux types de pilotes. Historiquement, il supporte ce qui porte aujourd'hui la véritable appellation de pilote. Il s'agit en réalité de pilotes « built-in ». Ils sont directement intégrés dans le noyau, compilés en même temps que le noyau. Il va sans dire que la souplesse n'est pas de rigueur. Arrivé plus récemment, on trouve le second type de pilotes. Ceux-ci sont appelés modules. Il s'agit de pilotes que l'on peut compiler sans recompiler tout le noyau, et qu'on peut installer, charger, et décharger à notre guise[1]. Ce qui permet une plus grande souplesse et permet

1 Ceci est également possible avec les pilotes en réalité. Même si leur compilation ne peut être indépendante du noyau, ils peuvent être compilés en mode module. Ce qui permet d'avoir les mêmes fonctionnalités qu'un module. Sans la souplesse et la facilité de compilation.

d’éviter de devoir fournir un nouveau paquet du noyau.

Je voyais donc cette possibilité comme une grande force de aufs. Pas besoin d’avoir de noyau particulier pour faire de la fusion de systèmes de fichiers.

La réalité était, elle, complètement différente. Pour pouvoir fonctionner correctement, même si aufs pouvait être un module, il fallait modifier Linux. Des exports étaient manquants (autant fonctions que variables). Résultat, qu’on le compile en pilote ou en module, dans tous les cas, il allait falloir fournir une nouvelle version du noyau. Ce prix étant jugé acceptable par mon tuteur, j’ai continué dans cette direction. Il s’agissait juste d’appliquer deux patchs au noyau et de le compiler. Ce fut chose faite rapidement.

Il fallait maintenant que les machines diskless puissent utiliser ce système. Donc, techniquement, il fallait que dracut le supporte. L’argument de vente principal de dracut est sa souplesse et le fait que peu de choses soient écrites en dur dans ses fichiers de démarrage. Par extension, cela signifie qu’il est facile de rajouter des fonctionnalités spécifiques au démarrage. Pour ce faire, il suffit de rajouter un script (comme expliqué plus haut) également appelle module dracut. Je me suis donc attelé au développement d’un module pour supporter le snapshot et l’union. Afin de conserver une certaine compatibilité avec l’ancien système, il fallait également que le snapshot puisse être passé en ligne de commande via le paramètre snapshot.

La documentation pour le développement de modules dracut n’étant pas des plus complètes, le plus simple était de passer par les autres modules fournis. Il apparait qu’en réalité, trois scripts (sur le modèle bash) étaient nécessaires :

- module-setup.sh : c’est le script qui permet l’installation du module dracut. Il définit les dépendances et les scripts à ajouter à l’image de démarrage. Et les conditions dans lesquelles ajouter le script à l’image de démarrage. Ici, je me suis contenté de demander à dracut de l’ajouter à chaque fois, si le module NFS était présent.
- parse-snapshot.sh : ce script permet de *parser* la ligne de commande dracut afin d’en extraire le snapshot à utiliser, fournit par le paramètre snapshot=<chemin à utiliser sur le serveur NFS>
- snapshot.sh : ce script utilise réellement le snapshot pour le fusionner avec le système d’exploitation. C’est donc lui qui réalise l’union et permet le démarrage sur le root file system résultant.

Techniquement, voici les changements apportés au montage réalisé par dracut. Normalement, dracut monte l’image de démarrage en rootfs, monte le nouveau système de fichier dans /sysroot. Puis, une fois le démarrage complété, le switch root est effectué sur /sysroot et l’image de démarrage déchargée.

Ici, il fallait donc ajouter la fusion entre le montage et switch root, tout en gardant la compatibilité avec les étapes intermédiaires. Ainsi :

1 – Dans un premier temps, on monte le snapshot dans /.snapshot

2 – Puis, on déplace /sysroot dans /.ro

À ce stade, on a donc les deux branches à fusionner de disponibles.

3 – On fusionne les deux branches dans /unionfs, en précisant les droits d’accès pour les deux

4 – Puis, on déplace /.ro dans /unionfs/.ro pour s’assurer de la persistance du point de montage

5 – On fait de même pour /.snapshot dans /unionfs/.snapshot

6 – Arrivé ici, tout a fonctionné, on déplace donc /unionfs dans /sysroot. Les autres scripts pourront donc fonctionner sur /sysroot, sans voir aucune différence.

Enfin, il fallait modifier la configuration de démarrage PXE des machines. En ajoutant simplement l'argument snapshot=/scratch/netboot/snapshots/vmXpclhcb171 avec X correspondant au numéro de la machine. Et en modifiant le noyau à utiliser.

Soit, en reprenant le fichier plus haut, modifier les lignes LINUX et APPEND pour obtenir :

```
LINUX vmlinuz-2.6.32-71.el6_aufs
APPEND initrd=initramfs-2.6.32-71.el6_aufs.img
root=nfs:192.168.122.1:/scratch/netboot/root
snapshot=/scratch/netboot/snapshots/vm3pclhcb171 selinux=0 rdshell rdinitdebug
```

Voici l'arborescence utilisée pour ce diskless sur SLC6. Dans /scratch/netboot/root se trouve le système partagé, comme précédemment. Je n'ai pas changé ce point et l'ai réutilisé puisque le but n'est pas d'avoir du diskless « simple ». Puis, dans snapshots, on trouve un répertoire au nom de chaque machine, qui correspond à son snapshot ou elle pourra écrire. Chaque machine dispose de son propre snapshot et n'a pas accès aux autres.

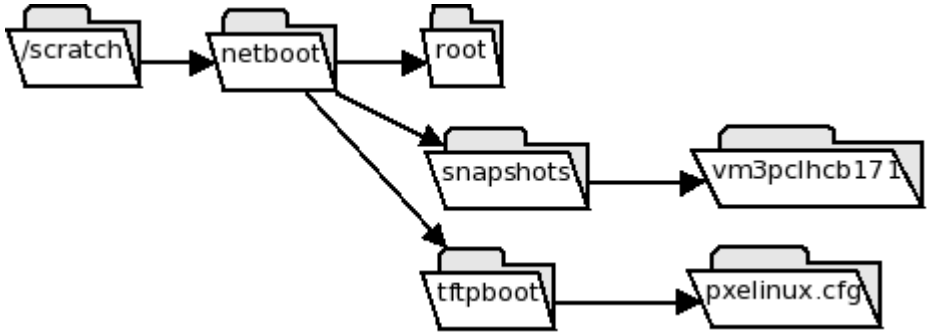


Illustration 11 : Arborescence avec aufs et unionfs sous SLC6.

Néanmoins, la modification de dracut n'a pas été suffisante. Si aufs réalise bien une union de systèmes de fichiers, il ne peut pas la faire sur n'importe quel système de fichiers. Pour pouvoir réaliser certaines fonctions, il a besoin de fonctionnalités précises sur la branche disponible en écriture. Or, il s'avère que NFS ne les supporte pas. Donc, aufs ne supporte pas NFS[5].

J'ai donc éliminé aufs des solutions utilisables.

2.2.2 Unionfs

Aufs ne faisant pas l'affaire, je me suis donc tourné vers unionfs par défaut. J'ai d'abord cherché d'éventuelles informations sur un non-support de NFS. N'ayant rien trouvé, et aufs semblant à l'origine supporter NFS[5], j'ai donc tenté le même travail avec unionfs.

La grande majorité du travail étant déjà faite, les syntaxes de unionfs et aufs pour le montage étant identiques, il suffisait d'ajouter le module unionfs. Ici, il n'était pas possible de compiler le pilote séparément, il fallait donc fournir également une nouvelle image noyau. J'ai donc lancé la compilation qui n'a posé aucun problème, puis, j'ai refais une image de démarrage avec dracut.

J'ai alors obtenu un diskless qui fonctionnait. Pour valider la fusion de système de fichiers, j'ai d'abord supprimé les clés SSH sur le système de fichiers partagé, pour que le serveur SSH les recrée

sur chacun des snapshots, et que chaque machine ait donc son identité propre. Au redémarrage suivant des deux machines, les clés ont effectivement été recrées. Il ne restait plus qu'à valider en se connectant aux deux machines.

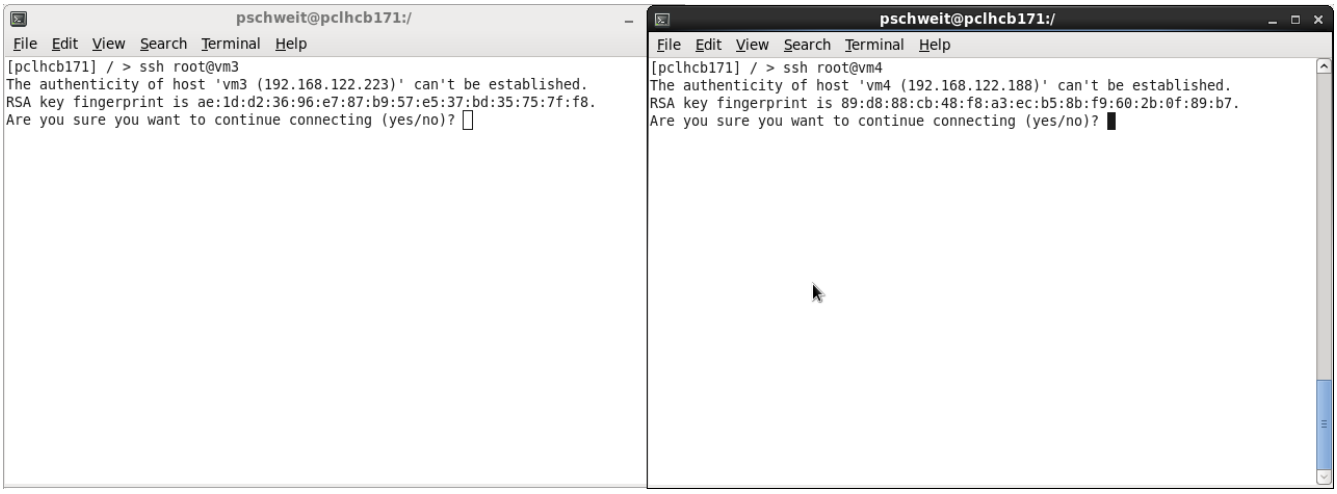


Illustration 12 : Les deux machines s'identifient différemment

Comme l'image ci-dessus le montre, les deux machines avaient donc leur propre identité. Cela étant, ceci était déjà possible avec l'ancien système diskless. Il fallait donc montrer par un test simple la force du nouveau système. Je me suis donc contenté de créer un nouveau fichier (nommé « bla ») sur le répertoire racine de la machine trois, de regarder s'il apparaissait. Et dans le même temps, de constater sur la machine quatre que le même fichier n'apparaissait pas. Ceci était purement et simplement impossible à cet emplacement avec l'ancien système.

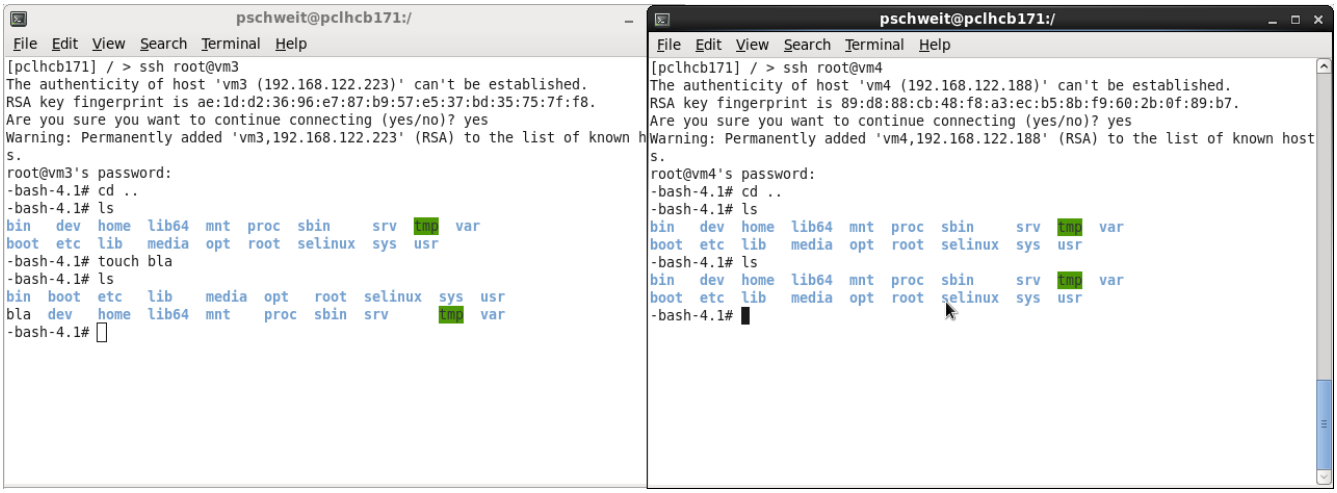


Illustration 13 : Sortie du test avec le fichier

La solution diskless est donc parfaitement fonctionnelle. Il ne restait donc plus que l'étape du paquetage. C'est-à-dire créer un paquet (connu sous le nom de RPM sous SLC6) qui permettrait facilement l'installation et la diffusion du nouveau noyau pour les machines diskless. Cette étape fut aisée, puisqu'il a suffit de reprendre le SRPM du noyau SLC6, à savoir le RPM qui permet d'installer les fichiers et les scripts qui ont permis la compilation du noyau, d'ajouter le patch pour unionfs et de refaire les paquets.

Il a fallu faire la même chose pour le module dracut (appelé pour l'occasion dracut-snapshot). Ce fut chose faite rapidement (même si là, je partais de rien). Le paquet a donc pour dépendance dracut-network et installe le nouveau module parmi ceux de base avec lesquels vient dracut.

2.3 Fusion de systèmes de fichiers sous SLC5

Sous SLC5, la partie diskless est aisée. Il suffit d'utiliser les outils fournis. Néanmoins, par rapport à SLC6, certaines choses restent constantes, notamment la création du système partagé et la configuration NFS.

Le système utilisé, en revanche, manque de souplesse. Pour pouvoir faire fonctionner le système diskless avec unionfs et en machine virtuelle, il a fallu modifier complètement le logiciel et en refaire un paquet puisque les scripts font partie intégrante du logiciel. Il n'existe par exemple pas d'option pour demander de charger des pilotes supplémentaires.

J'ai donc modifié l'application pour ajouter dans le script de démarrage tout le nécessaire pour réaliser la fusion (effectuée de la même façon que sur dracut, la méthode ayant été validée). Et j'ai également modifie le script de création de l'image de démarrage pour qu'il ajoute le pilote unionfs[1].

Par ailleurs, l'outil de Red Hat imposant une arborescence (root pour le système partagé qui était donc ici déjà pris, snapshot pour les snapshots), j'ai du prendre cette contrainte en compte pour mon serveur, pour aboutir à l'arborescence suivante :

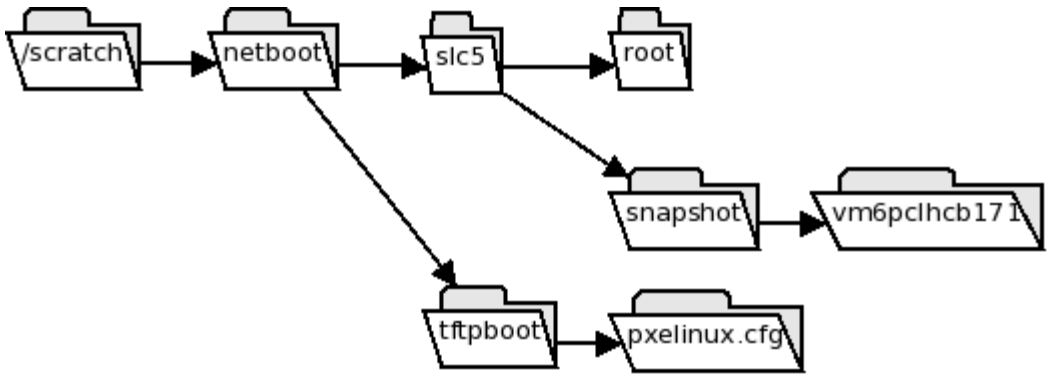


Illustration 14 : Arborescence avec unionfs sous SLC5

Le plus dur a eu lieu après. A l'instar de SLC6, unionfs était fourni sous forme d'un patch pour le noyau. Mais le noyau Linux de SLC5 est une ancienne version du noyau (la 2.6.18), la plus ancienne supportée par unionfs d'ailleurs. Mais en réalité, si cette version est ancienne, le noyau ne l'est pas tant que cela. Red Hat fournit régulièrement des mises à jour (correctifs et sécurité) pour le noyau en gardant pour base la version 2.6.18. Ils réalisent donc des *backports*. Ce qui a certes pour effet d'avoir

1 Ainsi que les pilotes pour libvirt, évidemment. Tout ceci est toujours virtualisé.

un noyau avec moins de bugs et moins de failles, mais surtout d'avoir un code du noyau qui ne ressemble plus nécessairement partout au noyau 2.6.18. L'application du patch fut donc laborieuse, en raison des conflits naissants des backports également réalises par le patch unionfs. Afin de le faire dans les conditions les plus efficaces possibles, j'ai finalement décidé de créer un repository SVN, d'y envoyer le code source du noyau, d'appliquer l'ensemble des patchs de backports dessus (voir : Annexe 3). Puis, une fois ceci fait de me pencher sur une application manuelle du patch unionfs. Ceci me permettant en cas d'erreurs trop importantes sur l'application du patch unionfs, de renverser les changements liés à celui-ci, et de recommencer avec tous les backports déjà présents. Cela offrait également la possibilité de facilement pouvoir recréer un fichier de patch applicable sur le noyau avec tous ses backports.

Une fois le patch opérationnel, le noyau unionfs pour SLC5 n'était pas encore assuré. Il fallait qu'il compile, et surtout que les différents paquets liés au noyau soient réalisables. Et enfin, que le noyau en lui-même fonctionne. Si la compilation a rapidement été acquise, la création des paquets liés l'a été beaucoup moins rapidement. Quand un SPRM du noyau est compilé, on récupère en sortie plusieurs RPM. Il y a de toute évidence le noyau, mais également un paquet contenant l'ensemble des en-têtes exportés par le noyau pour permettre le développement noyau (et notamment la compilation). Sous Linux, l'export des en-têtes correspond à un mélange prononcé. Pour compiler le noyau, il y a en effet deux types de fichiers d'en-tête. Le premier type, les internes qui ne concernent que des déclarations que le noyau utilisera qui n'ont aucun intérêt à être disponible pour les utilisateurs vu que ce qui y est mis n'est pas accessible hors du noyau. Le second type, les externes qui contiennent tout ce que le noyau exporte pour le développement noyau. Toujours est-il que ces en-têtes sont situes au même endroit dans l'arborescence du noyau. Seul un fichier permet de savoir lesquels sont exportes. De toute évidence, un en-tête exporté ne peut pas inclure d'en-tête non exporte (cela poserait de sérieux problèmes a la compilation). Or, le patch unionfs produisait cette contradiction. J'ai donc du travailler à la supprimer. Dans le même temps, j'ai travaillé à rendre le patch plus proche du noyau 2.6.32 (utilisé dans SLC6) pour une meilleure compréhension, et faciliter l'intégration des prochains backports. La compilation était donc fonctionnelle.

Pour s'assurer une compilation plus rapide (le noyau Linux est très long à compiler sur une machine « standard »), je passais via des serveurs de calcul (nommés plusXX ou XX correspond à un nombre de 01 à 20) dont dispose l'expérience LHCb. Ces serveurs, sous SLC5 étaient donc parfaitement a même de compiler le noyau SLC5. Ce qu'elles ont fait avec brio. Néanmoins, toutes les compilations de SLC5 ont conduit à un noyau non opérationnel. Le démarrage se terminait en kernel panic à chaque fois, dans le pilote unionfs. J'ai donc tenté de nombreux changements dans le patch, pour aboutir au verdict que c'était soit la compilation qui posait problème, soit unionfs de façon générale. N'ayant rien à perdre, j'ai décidé de prendre une nuit pour le compiler sur mon ordinateur au CERN. Et il s'est avéré qu'il s'agissait bien de la compilation sur les machines plus qui posait problème[1].

Je disposais donc maintenant d'un diskless SLC5 avec unionfs fonctionnel.

2.4 Retour vers FUSE et fusion côté serveur

Arrivé à ce stade, l'objectif principal du stage était rempli. Mais néanmoins pas de la façon la plus optimale. Il fallait en effet déployer un nouveau noyau, qu'il fallait d'abord compiler. Ce qui impliquait donc de maintenir un noyau pour deux versions de SLC. Mais également un module dracut

1 Même si l'origine du problème n'a pas été cherchée en détails, il semblerait qu'il venait du compilateur C utilisé. Sur les machines plus se trouve un compilateur C diffèrent, spécifique au CERN. Une compilation de GCC plus récente avec d'autres options.

et un outil Red Hat modifié. Le gain en souplesse était payé par un cout élevé de maintenabilité.

Une autre idée a vu le jour. Pour la méthode avec unionfs, il fallait exporter le système partagé, les snapshots, et fusionner le tout côté nœud. Autant envisager la possibilité dans l'autre sens, à savoir une fusion côté serveur et exporter le résultat fusionné aux clients. Pour ce faire, l'idée de FUSE rejetée plus tôt a été retenue. L'idée étant de s'affranchir de la complication du module noyau, mais également de la modification de l'outil de Red Hat ou encore de dracut.

Pour la fusion de système de fichiers sur FUSE, deux librairies existent : funionfs et unionfs-fuse. Encore une fois, ici, il a fallu faire un choix. Les deux semblaient abandonnés. Néanmoins, funionfs est déjà disponible sur SLC6 (mais pas sur SLC5) ce qui permet d'éviter le paquet CERN pour cet OS. Je me suis donc tourné vers cette solution.

2.4.1 Fusion sous SLC6

Pour pouvoir tester, je suis reparti de rien, avec une nouvelle hiérarchie, plus « adaptée ».

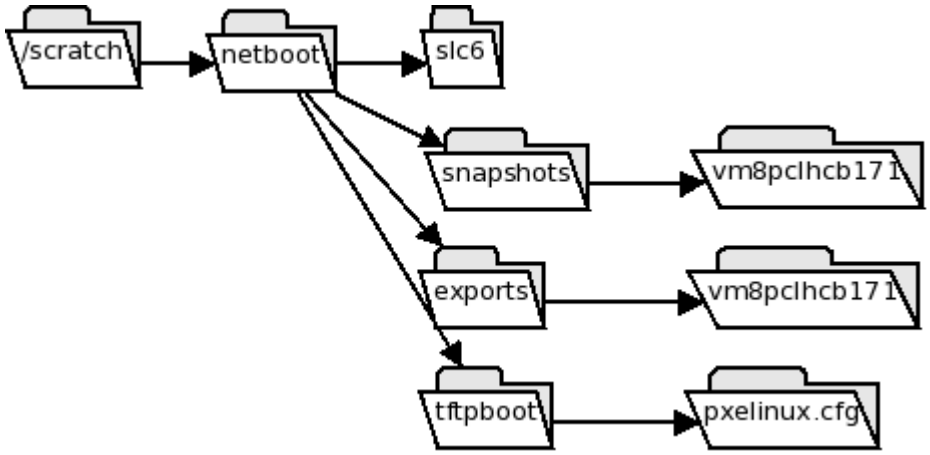


Illustration 15 : Arborescence pour fusion avec FUSE

Le répertoire slc6 contient le système de fichier partagé, à l'image de root utilisé précédemment. Le dossier snapshots n'a pas changé de fonctionnalité et contient toujours les snapshots propres à chaque machine. La nouveauté est le dossier exports. Il contient le résultat de la fusion des systèmes de fichiers. Son nom vient du fait que c'est ce répertoire et ce répertoire seulement qui sera exporté via NFS. Tout le reste n'est plus nécessaire.

L'installation de funionfs était donc très simple, en passant par le système de paquets de SLC6 : `yum install funionfs`. La configuration des exports elle, était évidente aussi, grâce à une syntaxe proche de unionfs. Comme beaucoup de systèmes de fichiers sur FUSE, un programme est déjà fourni permettant de ne pas passer par la commande mount. Ainsi la commande suivante :
`funionfs -o dirs=/scratch/netboot/slc6=RO -o allow_other /scratch/netboot/snapshots/vm8pclhcb171 /scratch/netboot/exports/vm8pclhcb171`
Permettait de faire la fusion. L'argument -o signifiant qu'une option va suivre ; l'option dirs permet de

lister les répertoires à fusionner. L'option allow_other permet à d'autres utilisateurs d'accéder au répertoire fusionné (autrement le répertoire n'est monté que pour l'utilisateur courant). Puis, suit le chemin de la branche supérieure, c'est à dire celle de priorité la plus élevée, qui sera donc fusionnée avec les branches fournies avec l'option dirs. Puis, comme d'habitude, on retrouve le point de montage.

Ainsi donc, la fusion fonctionnait. Il n'y avait plus qu'à exporter ce répertoire avec NFS. Il est possible d'exporter un point de montage FUSE avec NFS, moyennant un ajout dans la configuration. Il faut en effet fournir un FSID (File System ID). En temps normal, lorsque NFS exporte un répertoire, il utilise un nombre basé sur des informations du périphérique contenant les données. Dans le cas de FUSE, de toute évidence, la chose est impossible. Ce FSID ne servant qu'à identifier les exports, il est alors possible d'en fournir un manuellement pour chaque export FUSE. Il suffit d'entrer n'importe quel nombre de façon arbitraire (et unique, il s'agit d'un identifiant !). Je me suis donc retrouvé avec la ligne suivante :

```
/scratch/netboot/exports/vm8pclhcb171
192.168.122.0/24(rw,no_root_squash,sync,no_subtree_check,fsid=54[1])
```

Rappelons qu'ici, aucune modification de noyau n'est nécessaire, pas plus que de modification sur le système de fichiers. Néanmoins, par soucis de simplification (et par volonté de ne pas perdre de temps futilement), je me suis contenté d'utiliser le « vieux » ramfs d'unionfs et le même noyau. Néanmoins, j'ai effectué deux modifications dans la configuration PXE. D'une part, j'ai signalé que le rootfs reçu par dracut est disponible en écriture, et surtout qu'il n'y a pas de snapshot (pour prévenir toute utilisation du script dracut les gérant ainsi que tout utilisation d'unionfs).
On se retrouve donc avec en partie le fichier suivant :

```
APPEND initrd= initramfs-2.6.32-71.el6_unionfs.img
root=nfs:192.168.122.1:/scratch/netboot/exports/vm8pclhcb171:rw selinux=0 rdshell
rdinitdebug
```

Ceci fait, la machine démarrait. Mal. S'il lui était possible d'écrire et le lire sur son rootfs, chaque tentative (même si elle fonctionnait !) retournait une erreur. Afin de confirmer que le problème vient bien de funionfs, j'ai tenté les mêmes opérations en local. J'ai obtenu les mêmes erreurs.

C'est à l'issue de cet échec avec funionfs que je me suis tourné vers unionfs-fuse, funionfs n'étant plus maintenu (sinon, j'aurais rédigé un rapport de bug pour voir si le bug était connu/corrigeable). Ici, la situation était plus complexe à mettre en œuvre. La première étape était donc de compiler la librairie pour FUSE, puis de l'installer. L'étape de compilation et d'installation n'ont posé aucune difficulté, grâce à une utilisation de l'outil CMake. Pour l'utilisation, j'ai conservé l'arborescence décrite ci-dessus, je n'avais donc plus qu'à fusionner les branches, mais avec unionfs-fuse.

La syntaxe à employer ici était quelque peu différente de celle de funionfs :

```
unionfs -o cow,allow_other
/scratch/netboot/snapshots/vm8pclhcb171=RW:/scratch/netboot/slc6=RO
/scratch/netboot/exports/vm8pclhcb171
```

On reconnait ici le paramètre -o qui permet de spécifier des options (séparées par des virgules ici). L'option cow est particulièrement intéressante et est un héritage de unionfs. La signification est Copy-On-Write. C'est l'opération qui résulte en la création d'un *copyup* à chaque fois qu'un fichier non

1 En réalité, ce 54 n'est pas aussi arbitraire qu'il en a l'air. Afin de m'assurer de l'unicité de mon identifiant, j'ai utilisé le réseau IP (Internet Protocol) sur lequel je travaillais. En effet, toutes les machines virtuelles étaient sur le réseau 192.168.122.0/24. Chaque machine est identifiée de façon unique par son adresse IP, et plus spécifiquement par le dernier nombre de celle-ci (grâce à la configuration du réseau). C'est donc lui que j'ai utilisé comme identifiant. L'IP de la machine était donc 192.168.122.54.

disponible en écriture est modifié (voir le paragraphe 2.2). Après les options, on retrouve les deux branches à fusionner (ainsi que leur mode de fusion), puis le point de montage.

C'était le seul changement à effectuer.

À l'issue de cela, j'ai obtenu un diskless qui marchait parfaitement, sans modifications chez les clients donc.

Comparativement, que la fusion soit faite côté client ou côté serveur, les performances sur le client sont équivalentes. La charge également. La seule différence notable est la réduction de la taille mémoire occupée par le système quand la fusion est prise en charge côte serveur. Ceci n'est pas négligeable quand on travaille sur des systèmes ou la mémoire est un composant critique et limite.

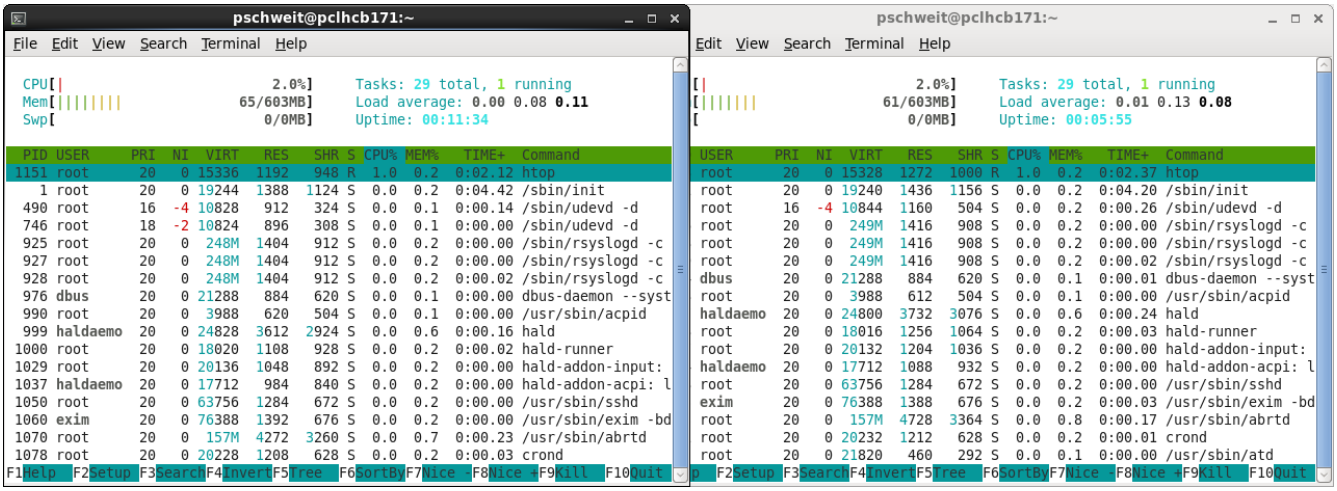


Illustration 16 : SLC6 avec fusion côté client (gauche) et côté serveur (droite)

2.4.2 Fusion sous SLC5

Pour la fusion avec SLC5, deux cas de figures se sont présentés. Mon PC au CERN étant sous SLC6, c'est lui qui réalisait les exports et les fusions pour toutes les machines clientes. Il ne s'agissait donc pas de faire une fusion sous SLC5, mais d'abord de faire fonctionner SLC5 en diskless sur un répertoire fusionné. Le deuxième cas de figure est alors de faire la fusion sous SLC5 puis de l'exporter via NFS.

Pour le support d'un système fusionné par SLC5, la grande majorité du travail était déjà faite. Il s'agissait juste de modifier quelque peu les scripts fournis par Red Hat pour monter le rootfs en écriture et non en lecture-seule, donc de changer une option dans la commande mount, en changeant l'option forçant la lecture-seule en écriture-lecture[1]. On obtenait alors sans difficultés un système SLC5 diskless avec fusion. Il n'y avait même pas besoin de modifier la configuration PXE par rapport à un autre diskless.

En revanche, pour le support de la fusion sous SLC5, les choses étaient différentes. La première

1 De façon assez logique, il fallait également modifier les scripts pour qu'ils prennent en charge les pilotes virtio. Tout ceci étant toujours virtualisé.

étape consistait bien évidemment en la compilation de unionfs-fuse sur SLC5. Ici, cela ne provoquait aucun changement par rapport à SLC6. En revanche, l'export via NFS a posé problème. En effet, par défaut, FUSE sous SLC5 n'est pas compilé de façon à permettre les exports[6]. Comme l'indique le fichier README.NFS de l'aide NFS, il s'agissait donc de compiler FUSE en lui passant la bonne option : --enable-kernel-module lors de la configuration de la compilation. Le tout en s'assurant que FUSE n'est pas déjà compilé en tant que pilote dans le noyau Linux. Fort heureusement, ici, tel n'était pas le cas. Autrement, il aurait fallu compiler également le noyau SLC5 et distribuer un noyau modifié (ce qui n'aurait pas présenté un grand intérêt puisque l'idée première était justement de se débarrasser de cette dépendance au noyau compile soi-même). La compilation pouvait donc entièrement marcher. Ce qui fut le cas. Une fois le FUSE fourni par la distribution remplacé par le mien, l'export était devenu possible, de la même façon qu'il l'était sous SLC6.

Ici aussi, je me suis permis de faire la comparaison de performances selon sur quelle machine avait lieu la fusion. Il s'est avéré que la différence était encore plus notable avec SLC5. On constate déjà que lorsque la fusion a lieu sur le client, le système consomme 14Mo de mémoire en plus (ce qui représente tout de même une hausse de consommation de plus de 31% !). Par ailleurs, la charge du système est bien plus élevée (avec notamment 11 processus supplémentaires qui tournent). S'il est assez difficile de trouver une explication à toutes ces observations, elles viennent conforter l'utilité de la fusion côté serveur.

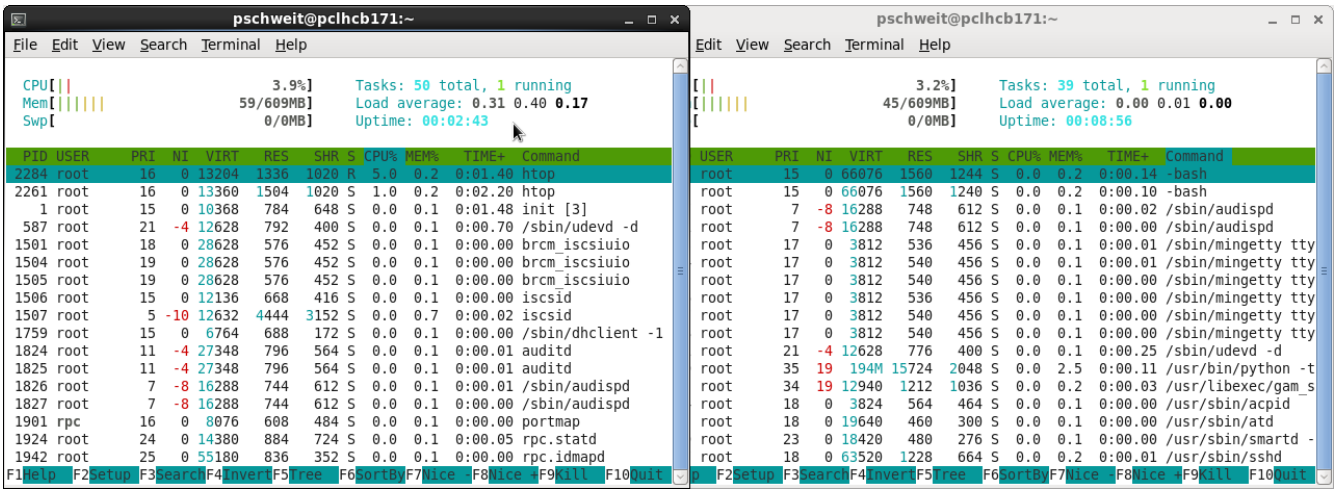


Illustration 17 : SLC5 avec fusion côté client (gauche) et côté serveur (droite)

Également, les machines diskless sous SLC5 étaient incapables de s'éteindre. Qu'il y ait fusion, ou que ce soit l'ancien système utilisé par le CERN. Dans tous les cas, un souci avait lieu avant la fin, aboutissant à une absence d'extinction électrique. Le CERN avait donc recours à IPMI pour éteindre les machines diskless. Ici, la commande poweroff fonctionne à nouveau correctement. La machine s'éteint d'elle-même.

Si la consommation est en baisse du côté des clients, elle est évidemment en hausse du côté du serveur. Néanmoins, il est difficile d'estimer correctement l'importance de cette hausse et son impact. Des tests « grandeur nature » seront faits ultérieurement pour s'assurer que le serveur ne sera pas

surchargé.

Afin de m'assurer également que tout fonctionne au mieux, avant ces tests de résistance, j'ai passé le code de unionfs-fuse en revue, afin de m'assurer qu'il n'y avait ni fuites mémoire, ni fuite de ressources. Ceci m'a permit d'en corriger et de les intégrer dans le paquet du CERN. Le patch a également été envoyé à l'auteur de unionfs-fuse. Celui-ci l'a rapidement accepté et mis en place dans le processus de développement de unionfs-fuse[7].

2.4.3 Résolution des problèmes liés à la fusion

Si la fusion répondait à la problématique posée (quel que soit le côté où elle est effectuée), elle posait tout de même des problèmes d'ordre pratique.

L'un des objectifs premiers de la fusion est de permettre l'administration facile d'un parc informatique. Il suffit de modifier le système de fichiers partagé pour que toutes les machines bénéficient de la modification.

Cependant, à l'aide la fusion de systèmes de fichiers avec copie sur la branche disponible en écriture en cas de besoin, un nouveau problème s'est posé : les copies trop fréquentes rendaient finalement chaque système quasiment indépendant et non administrable par la partie commune. Ce qui ne remplissait donc plus l'objectif de commodité cherché.

J'ai donc modifié le pilote FUSE unionfs-fuse pour régler ce problème. Plutôt que de modifier l'option de copy-on-write, j'ai ajouté une nouvelle option à utiliser en plus de l'option cow (lors du montage). L'option, nommée originalement « cern » permet de moduler l'option cow, voire d'offrir des fonctionnalités supplémentaires quand cette dernière n'est pas utilisée. L'objectif de cette fonction est de fortement modérer la copie et de ne la restreindre qu'aux cas réellement nécessaires. Dans les autres cas, une parade est trouvée ou le succès est simulé.

Le développement de pilote pour FUSE repose notamment sur les opérations à effectuer sur les fichiers (ouverture, renommage, suppression, etc). C'est donc vers ces opérations que je me suis tourné pour insérer mes modifications.

La première modification à apporter était le changement de mode sur un fichier (droits d'écriture, de lecture, etc). Ceux-ci sont stockés avec le fichier. Donc, si on désire modifier les modes sur un fichier qui n'est pas accessible en écriture, c'est impossible. Unionfs-fuse réalisait donc une copie du fichier pour pouvoir changer le mode. Avec l'option CERN, cette copie n'est plus réalisée. Et si le changement de mode est impossible, alors une erreur est retournée. Néanmoins, afin de limiter les erreurs, un changement de mode vers un mode déjà existant (une action qui n'aurait donc rien changé) ne retourne aucune erreur.

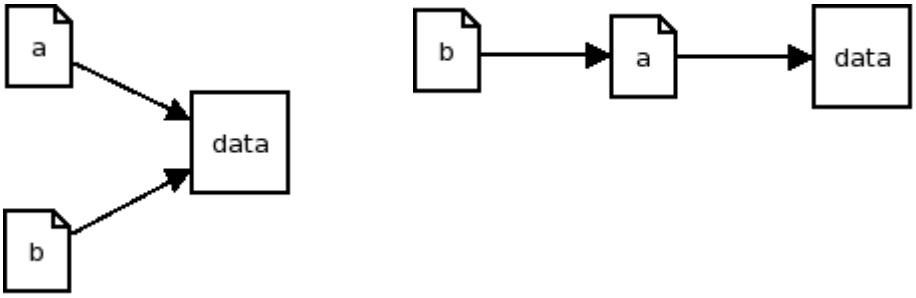
Le principe se logeant derrière est très simple. Normalement, unionfs-fuse parcourt toutes les branches disponibles pour trouver le fichier. S'il est introuvable, une erreur est retournée. Si le fichier est trouvé sur une branche en lecture-seule, alors il est copié vers une branche où il est possible d'écrire. C'est le fameux procédé de copy-on-write. Ici, j'ai modifié le processus pour que la recherche s'effectue sur toutes les branches, mais qu'en cas d'échec à trouver le fichier sur une branche disponible en écriture, aucune copie ne soit effectuée. À ce moment là, il y a alors une tentative de récupérer les droits actuels du fichiers et si c'est un succès, alors ils sont comparés. À aucun moment il n'y a de copie.

De la même façon, un fichier dispose d'une personne et d'un groupe qui le possèdent. Cette information est également stockée avec le fichier. J'ai donc ici appliqué le même changement qu'avec

la gestion des droits. Et la même gestion des erreurs.

La création (ou l'ouverture) d'un fichier (mais également d'un répertoire, voire d'un nœud) étant typiquement l'opération ou l'utilisation de copy-on-write peut être nécessaire, je n'ai apporté aucune modification à ce fonctionnement.

La gestion des liens a été plus astucieuse. Rappelons-le un lien est ce qui permet de lier un fichier à un autre. On pourrait rapprocher cela d'un « raccourci » (sous Windows). On trouve deux types de liens : les liens physiques (hard links) et les liens symboliques (symbolic links voire symlinks). Les liens symboliques sont faciles à gérer. Il suffit juste de créer un fichier et d'inscrire dedans l'information selon laquelle il pointe vers un autre fichier. Il va de soit que si la cible est supprimée, le lien devient caduque, mais de part sa nature existe toujours. Le lien physique est plus complexe et nécessite la modification du fichier cible. En réalité, sur un disque les données sont stockées à un endroit (à une adresse) et les fichiers pointent vers cette adresse. Le principe du lien physique est de rajouter un fichier pointant sur cette adresse. Ce qui impose donc la première limitation : lien physique et données doivent être sur le même support physique (d'où le nom). Par ailleurs, pour assurer la pertinence des liens (aucun n'ayant une prédominance) un compteur de référence sur les données est gardé avec les données et tient compte du nombre de liens pointant sur les données. Tant que ce compteur n'atteint pas zéro, les données ne sont pas supprimées. Mais donc, cela implique de modifier les données. Dans le cas d'une branche en lecture-seule, c'est donc impossible. Unionfs-fuse a recours à la copie du fichier. Avec l'option CERN, une différence a lieu. Lorsqu'il faut créer un lien physique sur un fichier, mais que ce fichier est sur une branche en lecture-seule, alors, le fichier cible n'est pas copié, mais un lien symbolique est crée.



La modification majeure qui a été apportée a été la gestion propre du renommage de fichier. Comme expliqué précédemment, le renommage d'un fichier avec la fusion de systèmes de fichiers est une opération ardue. Or, le but était ici de rendre l'opération plus souple (donc, encore plus complexe).

Plusieurs cas de figures se présentent. Dans le cas où l'on renomme un fichier non modifiable, alors la méthode décrite précédemment est toujours appliquée. Dans le cas où l'on renomme un fichier éditable, le nouveau nom de fichier est d'abord cherché pour voir si un fichier avec le même nom existe. Ce fichier n'est cherché que sur les branches en lecture-seule. En effet, dans le cas où il existerait déjà sur une branche modifiable, alors il serait purement et simplement écrasé. Reste donc le cas de la lecture-seule.

Ce dernier cas est important et intéressant puisqu'il offre la potentielle opportunité de supprimer un copyup et surtout d'éviter un phénomène de doublon. En effet, si on prend un fichier *a* dans une branche en lecture-seule que l'on renomme en *b*, *a* sera copié puis renommé en *b*. Puis, si on renomme

à nouveau *b* en *a*, alors le copyup réalisé précédemment sera conservé. Alors que techniquement, il n'y a aucune raison (les deux fichiers sont identiques !). C'est donc sur ce phénomène que je me suis penché afin de l'éviter.

Ainsi, quand un fichier cible existe déjà sur une branche en lecture-seule alors il va être ouvert par le pilote unionfs-fuse pour calculer une somme de contrôle des données. L'algorithme utilisé ici est la fonction de Adler-32. Ce choix a été motivé par des raisons de performances. La fonction de Adler-32 est plus rapide que ses équivalentes et offre, a priori, un résultat suffisant. Il s'agit en effet de ne pas impacter de façon trop importante sur les performances du pilote. Une deuxième somme de contrôle est calculée sur le fichier source. Les deux sommes sont alors comparées. En cas d'égalité des deux sommes de contrôle, les fichiers cible et source sont considérés comme identiques. Au lieu de renommer le fichier copyup, qui masquera alors le fichier en lecture-seule, le pilote supprime d'abord le copyup, puis supprime le whiteout qui servait à cacher le fichier en lecture-seule. Le renommage est alors effectif et a permis de supprimer l'usage d'un copyup.

Ce mécanisme de renommage a pu être testé très rapidement et validé. De façon périodique, sous Linux, une tache dénommée prelink est exécutée. Celle-ci consiste en l'ouverture de tous les fichiers binaires exécutables (et librairies) afin d'y effectuer une opération de maintenance visant à accélérer le fonctionnement des exécutables. Le fonctionnement de l'opération est le suivant : d'abord une copie de l'exécutable est effectuée. La copie est nommée comme le fichier d'origine suffixé par prelink et un numéro unique. La copie est alors ouverte et d'éventuelles modifications y sont apportées. Une fois ces modifications apportées, la copie est renommée avec le nom du fichier d'origine pour le remplacer (par écrasement). Si l'on rapporte cela avec la fusion, on se retrouve avec le schéma suivant. Le fichier exécutable, qui est un fichier système, est sur une branche en lecture-seule. La copie en revanche est réalisée sur une branche disponible en écriture. Si des modifications ont lieu dans la copie, alors, ladite copie reste sur la branche accessible en écriture et masque le fichier système d'origine. En revanche, s'il n'y a pas de modifications, alors la copie est supprimée, et le fichier d'origine en lecture-seule toujours utilisé.

Ceci a permis d'éviter un phénomène fortement gênant sur les machines diskless. En effet, prelink tournant sur tous les nœuds diskless, on se retrouvait à terme, avec toutes les machines diskless qui avaient une copie des fichiers systèmes sur leur snapshot. Ce qui ne présentait aucun intérêt et était plutôt gênant vu les attentes placées dans le diskless. Ainsi, il n'était plus possible de mettre à jour le système pour tous les nœuds en passant par le système de fichiers partage, puisque tous les nœuds en avaient une copie. Grâce à cette modification de la fonction de renommage, la chose a été à nouveau rendue possible.

Pour la gestion de la modification des dates modification et de dernier accès d'un fichier, la modification que j'ai apporté est simple. Si le fichier n'est pas disponible en écriture, alors aucun copyup n'est fait, et l'opération retourne le succès. Ce choix a été fait puisque la modification des dates de dernier accès sur un fichier diskless ne présente pas grand intérêt et la date de dernière modification n'est changée qu'en cas de modification du fichier. Et si le fichier est modifié, il est forcément disponible en écriture.

Les dernières modifications portent sur la gestion par unionfs-fuse des attributs étendus sur un fichier. Tous les fichiers possèdent de base des attributs simples (notamment les droits d'accès, les dates d'accès, de modification, etc). Certains systèmes de fichiers disposent également d'attributs étendus, souvent abrégés en xattr (pour eXtended ATTRibutes). Même si NFS ne supporte pas ce mécanisme d'attributs étendus[8], il me semblait important de faire un travail complet, ouvrant la voie au support de nouvelles fonctionnalités (si par exemple, NFS venait à supporter les attributs étendus).

Les attributs se présentent sous la forme suivante : nom:valeur. On accède donc à un attribut par son nom pour récupérer sa valeur. Quatre opérations sur ces attributs sont disponibles : suppression d'un attribut, modification d'un attribut, lecture d'un attribut, listage des attributs. Les deux dernières opérations ne posent aucun problème puisqu'il est uniquement question de lecture d'attributs. Le fichier peut donc être en lecture-seule. Les deux premières opérations nécessitaient un peu de travail dans le cas d'un fichier en lecture-seule.

Pour la suppression de l'attribut, le pilote unionfs-fuse va chercher l'attribut qu'il faut supprimer. Si la recherche échoue parce que l'attribut est manquant, alors la fonction va retourner le succès. L'attribut n'est pas (plus) là, comme le souhaite l'appelant.

Pour la modification de l'attribut, le pilote ici aussi cherche l'attribut dont il faut modifier la valeur. S'il ne le trouve pas, il va retourner une erreur. Ce choix a été sciemment fait. En effet, dans les faits, il est possible de créer un nouvel attribut en utilisant la fonction de modification (l'idée sous-jacente étant : si l'attribut existe, je défini cette valeur, s'il n'existe pas, alors je le crée avec cette valeur. Dans tous les cas, à la fin, l'attribut existe et a cette valeur), sauf que la création de l'attribut impliquerait la création du copyup, ce que l'on tente d'éviter. Cette première recherche, en plus de s'assurer de l'existence de l'attribut, permet de récupérer la taille de la valeur de l'attribut. Si celle-ci correspond à la taille de la nouvelle valeur, alors le pilote lit la valeur de l'attribut. Une fois la lecture faite, les deux valeurs sont comparées bit à bit. Si les deux valeurs sont identiques, alors l'opération est un succès. Autrement, c'est un échec. Quoi qu'il en soit, en cas de succès, l'attribut a bien la valeur que l'appelant voulait qu'il ait.

La fusion de système de fichiers via FUSE posait également un autre problème. En effet, comme explicité précédemment, les fusions sont réalisées à l'aide d'une commande tapée dans un terminal. Le problème étant qu'en cas de redémarrage des serveurs, les administrateurs ne peuvent pas taper les commandes nécessaires (il y en aurait trop). Il fallait donc s'assurer d'une fusion automatique au démarrage. C'est notamment ce à quoi sert le fichier fstab (cf : 2.1.2.2).

La syntaxe des données à mettre dans le fichier fstab suit une logique précise, ordonnée par colonnes. La première colonne contient le périphérique physique à monter. Dans le cas où il ne s'agit pas d'un périphérique, il est coutume de mettre le système de fichier monté (voire « none »). Dans le cas de FUSE est unionfs-fuse, cette colonne doit suivre une syntaxe particulière. Tout d'abord, elle contient le système de fichiers : unionfs, puis un dièse, suivi des branches à fusionner avec le mode de fusion. La seconde colonne contient le point de montage du système de fichiers. Ici, de toute évidence, selon l'arborescence choisie précédemment, il s'agira de sous-répertoires de /scratch/netboot/exports. La troisième colonne contient le système de fichiers utilisé. Dans le cas présent, il s'agit donc de fuse. La quatrième colonne liste toutes les options à passer au système de fichiers pour réaliser le montage. C'est ce que l'on retrouvait après le -o dans le cas de unionfs-fuse (plus quelque unes de plus). Enfin, les deux colonnes suivantes ne présentent que peu d'intérêt dans notre cas, et sont souvent mises à zéros dans les autres cas. Elles correspondent respectivement à la fréquence à laquelle sauvegarder les données et à l'ordre de passage pour les vérifications du système au démarrage. Ici, tout étant déporté et donc géré par le serveur, ces options sont mises à zéros.

On se retrouve donc avec des lignes suivant ce schéma :

```
unionfs#/scratch/netboot/snapshots/vm8pclhcb171=RW:/scratch/netboot/slc6=RO
/scratch/netboot/exports/vm8pclhcb171 fuse
cern,cow,rw,nosuid,nodev,allow_other,default_permissions
```

Si on s'attarde sur la colonne des options, on reconnait les options déjà présentent dans la ligne de commande, mais également la nouvelle option CERN. D'autres options ont également été ajoutées. Celles-ci sont implicites lorsque l'on utilise la commande unionfs, mais doivent être précisées lors du

montage via fstab (qui n'utilise pas le programme unionfs pour le montage, mais passe directement par FUSE).

Un autre problème est apparu lors des essais sur la fusion de systèmes de fichiers. En effet, pendant que je testais la fusion de système de fichiers, j'abandonnais également les outils de Red Hat, et ce afin de ne pas avoir de modifications à faire sur ces outils, puisque de toutes les façons, nous devions les abandonner. Néanmoins, ces outils réalisaient des modifications sur le rootfs partagé, et l'absence de ces modifications pouvait conduire à des dysfonctionnements.

C'est ce que je pensais du problème suivant.

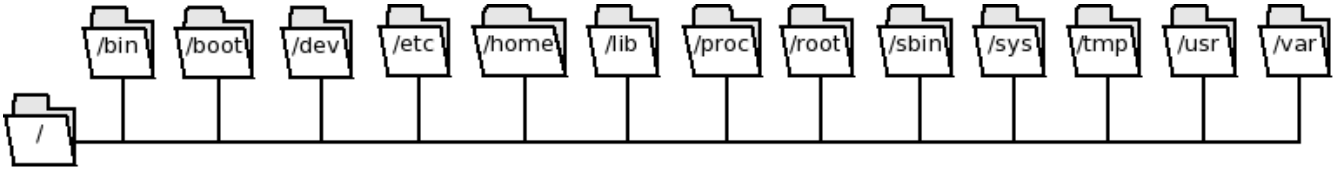
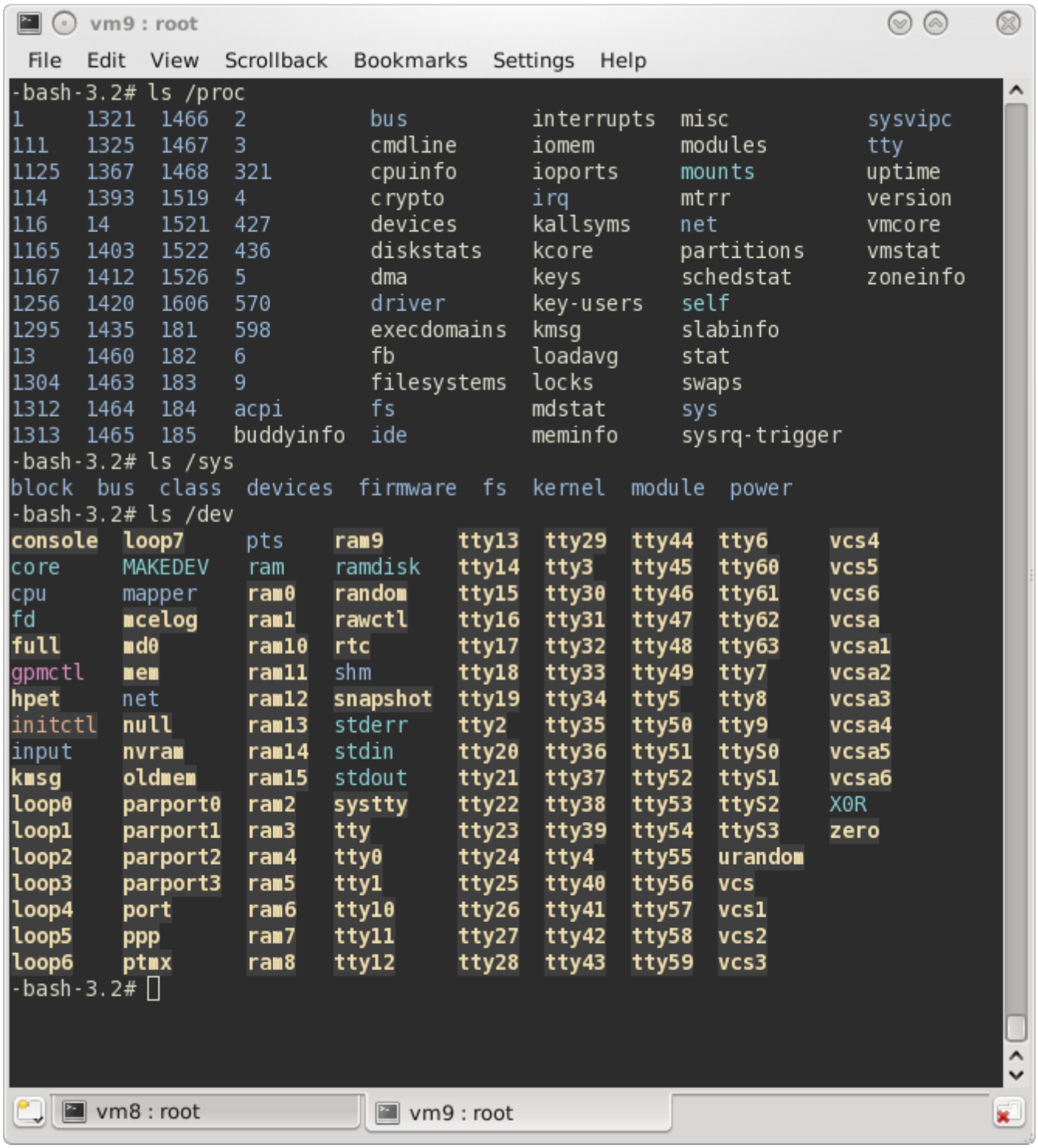


Illustration 20 : Arborescence de base de Linux

Au bout de quelques temps, le nœud se retrouvait avec deux répertoires vides : /proc et /sys. Et ces deux répertoires sont critiques pour le bon fonctionnement du système d'exploitation. Le premier contient toutes les informations sur les processus qui tournent sur le nœud, ainsi que leur configuration. Le second contient toutes les informations sur le système, y compris sa configuration. Pire encore, lors de tests sur du matériel réel et particulièrement performant, même /dev devenait vide. Ce répertoire est également critique pour le fonctionnement puisqu'il contient toutes les informations sur les périphériques du nœud (dev étant le diminutif de « device », en français périphérique). Il contient notamment les TTY, c'est-à-dire les consoles. Or dès qu'un utilisateur se connecte, d'emblée, une console lui est affecté. De facto, une machine sans /dev est une machine à laquelle on ne peut plus se connecter (localement et à distance). C'est donc une machine qui ne sert plus à rien.

Lors des tests virtualisés, j'avais mis un programme très simple (voir annexe 4), qui tournait en tâche de fond et qui surveillait le statut des deux premiers points de montage cites. S'il détectait qu'un d'eux était vide, alors il le démontait puis le remontait. Ceci mettait d'ailleurs en avant que les deux répertoires étaient vide et non démontés.

Sur les machines réelles, cette tentative de correction ne marchait simplement pas, puisque /dev était vide également. Et la seule solution viable était de les redémarrer. Sauf que cela impliquait que la durée de vie de ces machines avoisinait alors les 5 minutes. Rien d'utilisable donc.



Un autre « correctif » se trouvait en réalité dans les outils Red Hat. En effet, ceux-ci, lors de leur utilisation, en plus de préparer tout le nécessaire pour le démarrage sur le réseau, installaient un service de démarrage sur le système de fichier partagé qui allait être ensuite lancé au démarrage des nœuds. Ce service, en réalité un script très simple, réalise trois actions :

1 – Démontage de l'ancien rootfs (c'est-à-dire le ramdisk, avant le switch root)

2 – Suppression du ramdisk de la mémoire RAM.
3 – Création d'un verrou diskless.

L'utilisation de ce script permet non pas de régler le problème de /proc, /sys et /dev, mais néanmoins d'en retarder l'échéance. Ce script présente également un avantage non négligeable : il libère de la mémoire vive sur les nœuds (puisque le ramdisk n'y est plus conservé).

Différents tests réalisés par la suite semblent montrer que la responsabilité de ce problème incombe à FUSE et serait donc côté serveur. Pour isoler et affirmer ce fait, le test suivant à été fait.

Trois machines diskless sont préparées. Elles vont toutes fonctionner sur un modèle différent. La première aura son rootfs exporté directement par le serveur. Toutes les modifications qu'elle fera y seront immédiatement reportées. Il n'y aura ni fusion, ni FUSE. Elle a donc son propre rootfs via NFS.

La seconde, elle va récupérer son rootfs sur le serveur également. Elle sera la seule à l'utiliser, et il n'y aura pas de fusion non plus. Néanmoins, elle utilisera une couche FUSE tout de même, puisque le système de fichiers exporté ne sera qu'une « image » transparente du système réel à travers FUSE. Tout fonctionne à l'identique, à la différence près que l'on force l'utilisation de FUSE. Pour ce faire, un système de fichiers exemple de FUSE est utilisé : fusexmp_fh.

Enfin, la troisième est une machine diskless standard comme celles utilisées précédemment. C'est-à-dire shared root, avec fusion de systèmes de fichiers via FUSE.

Le résultat est sans appel et flagrant. Seule la première machine n'a pas rencontré le problème. Et malgré toute l'aide de l'équipe Online, nous n'avons pas réussi à comprendre d'où venait le problème avec FUSE.

2.5 Intégration à Quattor

Quattor est un ensemble logiciel qui permet à l'équipe LHCb d'administrer l'intégralité des machines du parc informatique facilement. En effet, l'un des problèmes lors de l'administration d'un grand parc informatique est qu'il devient difficile de gérer toutes les machines en même temps en les considérant de façon unique. C'est à ce problème que répondent des outils tels que Spacewalk ou encore Quattor.

Quattor arrive avec une approche complètement différente de celle de Spacewalk. Chaque machine est décrite par un fichier, écrit dans le langage Pan (développé pour Quattor). Ce fichier « template » contient toutes les informations sur les logiciels à installer sur la machine, ou encore quelle configuration utiliser pour lesdits logiciels ou le système. Des informations sur le matériel constituant la machine peuvent également être mis dans le template (on pense notamment aux adresses MAC ou encore à l'architecture).

Quattor permet aussi d'aller encore plus loin. Il est tout à fait possible de créer des templates qui ne correspondent à aucune machine mais qui correspondent à des particularités de configurations communes à plusieurs machines. Et dans ce cas, ces templates peuvent être inclus, via la directive adéquate dans le template des machines cibles. On se retrouve donc avec un vrai langage de programmation, modulaire, qui permet la factorisation de configurations.

Quattor permet également l'installation d'une machine à partir de zéro, juste en utilisant son fichier template. Il devient donc très facile pour les administrateurs LHCb de déployer une nouvelle machine. Il leur suffit d'écrire le template Quattor correspondant à leurs besoins, en incluant les templates qui offrent déjà une réponse partielle à leurs besoins, tout en complétant avec les informations requises. Une fois ceci fait, Quattor fait le reste, déchargeant d'une énorme et fastidieuse partie du travail.

Le fonctionnement de Quattor est le suivant : chaque machine, une fois décrite, voit son template compilé en XML. Une fois la compilation terminée, la configuration est envoyée sur un serveur distant, prête à être déployée. Une « impulsion » est donc envoyée sur chaque machine, sur laquelle un client Quattor est à l'écoute. Ce client va alors lancer un programme qui se chargera de récupérer la nouvelle configuration sur le serveur et d'appliquer les changements nécessaires en comparant le nouveau fichier XML à l'ancien en cours d'utilisation. Cela passe donc par la modification de fichiers de configuration, ou encore l'installation de nouveaux paquets.

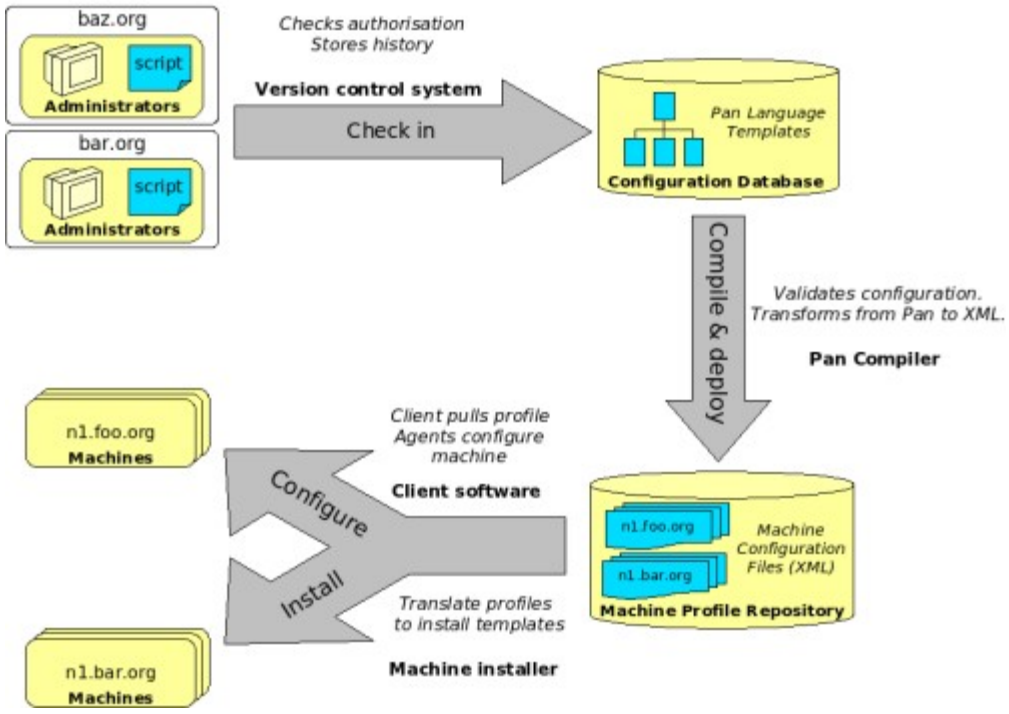


Illustration 22 : Schéma du fonctionnement de Quattor

Quattor lui-même est décomposé en composants fonctionnels qui chacun réalisent une tache bien précise, configurent un élément cible de la machine. Ce sont ces composants que l'on configure lors de l'écriture du template, afin qu'ils effectuent les bonnes actions. Les composants sont en réalité des modules Perl sur lesquels s'appuie alors Quattor. Ces composants viennent également avec un fichier template qui décrit la configuration qu'ils attendent en entrée. C'est-à-dire, la façon dont les templates pour les machines doivent être écrits pour que le composant puisse réaliser correctement les actions.

Pour la gestion du diskless, Quattor dispose d'un composant dédié. Ce composant permet de générer la configuration DHCP du serveur central mais également de configurer tout ce qu'il faut exporter pour le bon fonctionnement en réseau des nœuds. Néanmoins, ce composant souffre de plusieurs faiblesses majeures. Tout d'abord, il ne gère pas du tout la création du système de fichiers partagé. Celui-ci doit donc être créé de façon « bricolée ». De plus, ce composant s'appuie sur les outils Red Hat qui ont disparu dans SLC6. Enfin, ce composant ne gère pas les Credit Card PCs (CCPCs).

Ces PCs, qui correspondent à des caractéristiques matérielles très spécifiques, forment le niveau

0 du trigger LHCb. En raison de leurs spécificités techniques, ils n'embarquent pas de disque dur. Ils fonctionnent donc également en diskless. Par ailleurs, ils ne fonctionnent que sous SLC4. Le composant ne gère pas cette particularité non plus.

Il s'avère donc que si le composant est effectivement configurable, les options et le support qu'il offre sont en réalité très faibles. Plutôt que de le modifier pour qu'il supporte les étapes nécessaires à la création d'un système diskless avec fusion de systèmes de fichiers, il a été décidé de réécrire totalement ce composant (en partant de zéro) avec le cahier des charges suivant, qui répond aux problèmes exposés précédemment :

- Prise en charge de SLC5 et SLC6
- Prise en charge des CCPCs
- Prise en charge de la fusion de systèmes de fichiers
- Prise en charge de la création du système partagé
- Abandon des outils Red Hat (paquets system-config-netboot)

Enfin, puisqu'il s'agissait d'une réécriture, j'avais plus de liberté pour faire ce que je voulais, d'autant que seul LHCb semble utiliser ce composant, quitte à perdre la compatibilité avec les configurations et templates précédents. Il a donc été décidé d'aller encore plus loin dans le composant :

- Suppression de l'utilisation de données de configuration obsolète
- Support de plusieurs configurations diskless sur une machine
- Génération de fichiers de configuration mieux structurés
- Possibilité de défaire la configuration

Néanmoins, même s'il s'agissait d'une réécriture, les fonctionnalités qu'offrait l'ancien composant devaient être conservées. Notamment la gestion dynamique des nœuds, qui permettait d'avoir une modification efficace de la configuration pour ajouter ou supprimer un nœud au serveur (sans effacer toute la configuration sur la serveur et la réécrire).

La réécriture s'est effectuée en plusieurs étapes. Dans un premier temps, j'ai commencé par implémenter les fonctions qui faisaient défaut de façon critique : le support SLC5 sans dépendance aux outils Red Hat, le support SLC6 ainsi que la création du système partage.

Les premiers problèmes se sont alors posés. Si pour SLC6, créer une image de démarrage est aisé, grâce à dracut, pour SLC5, c'est autrement plus compliqué. L'ancien composant se résumait sur ce point comme un « wrapper » autour des outils fournit dans system-config-netboot (à savoir pxeos et pxeboot). Ces outils permettent entre autre de créer l'image de démarrage. Un de objectifs du nouveau composant étant de ne plus avoir cette dépendance, j'ai décidé d'offrir un compromis. Plutôt que de tout réinventer pour avoir une image de démarrage (et prendre le risque d'introduire des bugs face à système déjà éprouvé), j'ai décidé de prendre le script chargé spécifiquement de la création (ainsi que le script de démarrage) et de les utiliser. Ainsi, il suffit de les mettre dans un paquet RPM à part, et de les maintenir, hors du paquet system-config-netboot, maintenant obsolète. Le composant se contente donc d'appeler le script de création de l'image (updateDiskless) légèrement modifié (plus de souplesse au niveau des constantes utilisées pour l'adapter au composant) pour avoir le résultat.

Le composant crée également, et là sans l'aide d'aucun script, les fichiers de configuration nécessaires au démarrage réseau pour chacun des nœuds. L'ancien composant utilisait également les outils pour ce point.

La seconde étape était l'intégration des CCPCs. Je me suis appuyé sur les travaux réalisés l'année dernière par Sylvain Planche à ce sujet, qui devait porter SLC5 pour les CCPCs. J'ai utilisé son

rapport[9] comme documentation et utilisé les paquets qu'il a réalisé. De ce fait, le composant gère les quelques légères modifications à apporter au système partagé pour les CCPCs. Afin de simplifier la gestion de ces différences, j'ai introduit deux fonctionnalités. La premier consiste en un méta-paquet. Ce paquet qui ne contient aucune donnée, aucun fichier et se contente d'avoir en dépendance tous les paquets dont un CCPC a besoin pour fonctionner. Il suffit donc d'installer ce paquet via yum pour s'assurer que tous les autres paquets seront installés. La seconde fonctionnalité, fortement inspirée par l'architecture de dracut est l'appel d'un hook-script. Le composant Quattor peut donc, à la fin de la création du système partagé, appeler un hook-script, spécifié dans le template pour effectuer des actions supplémentaires. Le hook-script pour les CCPCs est donc une partie d'un script qu'avait écrit Sylvain Planche pour s'assurer que le système consommerait le minimum de ressources possible.

La troisième étape fut la réécriture de la gestion des nœuds. L'ancien composant utilisait également les outils Red Hat pour générer et gérer une liste des nœuds. Cette liste de nœud (au format XML) permettait alors de savoir quels nœuds ont été supprimés ou ajoutés. Dans mon cas, les outils n'étant plus à disposition, il fallait que je trouve un autre système. J'ai donc opté pour un système de cache situé dans l'arborescence centrale. En effet, pour le composant, j'ai repris une arborescence similaire (mais configurable !) à celle décrite l'illustration 13. Ainsi, au même niveau que les dossiers snapshots et exports se trouve un fichier de cache, dénommé .nodes. Ce fichier contient la liste des nœuds, ainsi que leur root directory et leur IP (en format hexadécimal). On se retrouve avec des entrées du type : quattortest01;0A801050;/diskless/ccpc/root

Via ce système de cache que le composant commence par lire avant de gérer la liste des nœuds du template, il est possible de savoir quel nœud a été ajouté et supprimé. Ainsi quand le composant a fini de créer tous les répertoires pour les nouveaux nœuds et de mettre à jour la configuration, il réalise une seconde passe pour comparer le cache à la configuration. Si une entrée apparait dans le cache, mais pas dans la configuration, alors il considère le nœud comme supprimé et le supprime du serveur (répertoires et configuration).

Une fois ces étapes remplies, j'avais un composant théoriquement fonctionnel qui comblait déjà des manques. Cependant, il a été décidé de continuer le développement jusqu'au bout pour ne tester qu'à la fin (en raison de tests un peu lourds à mettre en œuvre). La gestion de plusieurs versions de systèmes d'exploitation partagés (appelés protonodes) par serveur a donc été ajoutée. Cette possibilité de configuration multiple offre notamment la capacité à un serveur de faire à la fois serveur pour des serveurs standards ou des CCPCs, ou encore d'utiliser plusieurs systèmes d'exploitation (SLC5 en production, SLC6 en test, par exemple). Dans la foulée, la réécriture de l'écriture du fichier de configuration du serveur DHCP s'est imposée. Le code était vieux, pas réellement propre, et produisait en sortie un fichier guère mieux, et peu facilement interprétable au premier abord. Grâce à la réécriture, les configurations sont regroupées au sein d'un même groupe. Cela permet de repérer bien plus facilement les nœuds rattaches au même protonode.

Enfin, la dernière étape fut l'intégration propre de Quattor dans Quattor. En effet, le serveur qui contient les protonodes est géré par Quattor (d'où l'utilité dudit composant...). Mais également, par soucis de simplicité, les protonodes eux-mêmes sont gérés par Quattor. Ainsi, le composant, par un changement de répertoire racine (opération connue sous le nom de chroot – change root) lance Quattor dans celui-ci. Par sécurité, pour éviter tout problème avec les verrous de Quattor, et les démarrages de services diverses et variés, deux mesures ont été prises. Le répertoire /proc est monté lors du changement de répertoire racine. Cela assure aux applications tournant dans le nouvel environnement

d’avoir accès aux informations sur les processus déjà existants. Également, des modifications de certains applications sont installés. Il s’agit d’anciens scripts de LHCb, connus sous le nom de « wrap kill & co ». Le principe est simple, pour tout lancement de commandes (kill, service, killall) qui devraient s’effectuer sur le protonode, il y a une vérification du nom d’hôte. S’il s’agit du protonode, la commande n’est pas lancée. Sinon (il s’agit donc d’un nœud), elle est correctement lancée.

2.6 Prise en charge des CCPCs

Si le composant Quattor gérait effectivement, en théorie, les CCPCs, il fallait encore valider cette théorie par la pratique. C’est une fonctionnalité très attendues puisqu’elle simplifierai la gestion de ces CCPCs, actuellement pas vraiment gérés via Quattor, et présentant une mise en place quelque peu chaotique et complexe (et toujours sous SLC4 !).

Pour comprendre le fonctionnement de ces CCPCs, j’ai repris les travaux de Sylvain Planche, qui avait travaillé dessus l’année dernière. L’objectif étant de reprendre totalement son travail pour éviter une duplication d’effort et avoir la certitude d’obtenir quelque chose qui fonctionne. La majeure difficulté ici venait du fait que le kit de développement pour CCPC utilisé l’année dernière n’était plus trouvable. Le kit de développement permettant surtout d’interagir directement avec le CCPC. Il est utilisé à des fins de développement, mais surtout à des fins de débogage. Faute de kit, il fallait donc que cela marche. À terme, une solution avec un CCPC particulier, disposant d’une liaison série (permettant l’accès direct la console du CCPC) a été trouvée.

Dans un premier temps, il fallait créer les profils Quattor pour ces CCPCs (dont un pour leur protonode). La technique employée a été relativement simple. J’ai repris le profil des nœuds standards et supprimé les logiciels qui n’existent pas sur l’architecture des CCPCs. Puis, afin de privilégier la légèreté du système, j’ai également supprimé un grand nombre de logiciels et de services qui auraient conduits à l’encombrement de la mémoire.

Il a ensuite fallu valider le composant Quattor. Il s’est avéré que le noyau que fournissait Sylvain présentait des problèmes au niveau de la gestion des numéros de version et le composant Quattor reposant sur ceux-ci, il a fallu compiler à nouveau le noyau pour la modifier. Cela a provoqué des erreurs en cascade au niveau des autres paquets qui reposaient sur l’usage de cette numérotation. J’ai alors du également corriger et compiler une nouvelle fois ces paquets.

Néanmoins, cela ne m’a pas permis immédiatement d’avoir un CCPC qui démarre. Beaucoup de travail était encore nécessaire. Dans un premier temps, il a fallu également modifier le script générant les images de démarrage. Celui-ci cherchant un grand nombre de module pour le noyau, il ne pouvait les trouver pour les CCPCs, puisque ceux-ci étaient des pilotes directement compilés dans le noyau. J’ai donc modifié le script pour qu’il ignore ces erreurs, si on lui signale qu’on le lance pour un CCPC.

Pourtant, malgré ces corrections, s’il était possible d’installer le protonode pour les CCPCs, il était impossible d’en faire démarrer un. En effet, le script de démarrage rencontrait presque immédiatement une erreur, en signalant qu’il ne trouvait pas le rootfs. Les CCPCs ne démarrent pas exactement de la même façon que les serveurs standards. Le protocole de démarrage PXE n’existe pas pour eux, et le démarrage se résume à l’utilisation d’un fichier NBI (Network Boot Image) qu’ils récupèrent sur le réseau. Ce fichier contient à la fois le noyau, l’image de démarrage et les arguments à

passer au noyau (par analogie, il inclut en quelque sorte le fichier de configuration que l'on crée dans pxelinux.cfg). On ne peut donc pas personnaliser cette ligne de commande pour chaque nœud (à moins de créer une image NBI par nœud). L'idée qui a été trouvée est donc la suivante, dans la ligne de commande, on met un NFSROOT avec une variable %HOSTNAME% dont le nom est explicite. Dès lors, le script de démarrage remplacera cette variable par le nom d'hôte récupéré via le DHCP. Ainsi, il est possible de définir l'adresse du rootfs de façon « dynamique ».

Cela n'a pas réglé les problèmes de démarrage du CCPC, celui-ci se faisant presque immédiatement tuer par son watchdog (chien de garde) matériel. Cela m'a donc amené à l'évidence qu'il y avait un soucis avec le démon du watchdog. Le démon du watchdog est un logiciel qui tourne en tâche de fond et qui régulièrement va demander au pilote du watchdog d'envoyer un signal au watchdog pour que celui-ci ne redémarre pas le CCPC. Démon et pilote communiquent par le biais d'un périphérique nommé /dev/watchdog. C'est en regardant de plus près de ce côté que je me suis aperçu que par défaut, ledit périphérique n'existait pas. Le script de démarrage avait donc pour mission de le créer.

Malheureusement, ce changement n'était pas suffisant. Le démon démarrait trop tard (en théorie, vu qu'en pratique, il ne pouvait plus démarrer : le CCPC était déjà redémarré). J'ai donc modifié la procédure d'installation du démon pour que le démarrage de celui-ci se fasse juste après la fin du script d'initialisation du système (/etc/rc.d/rc.sysinit). Néanmoins, udev un démon lancé par ce script est très long à charger, cela ne suffisait donc pas. J'ai donc décidé d'effectuer le démarrage du démon juste avant le démarrage de udev. Ainsi, udev pouvait prendre tout le temps nécessaire. Néanmoins, le démon du watchdog étant géré en dehors de rc.sysinit (directement par le processus init), il fallait faire un choix. Ou supprimer cette gestion externe, ou s'arranger pour que l'instance de rc.sysinit ne gène pas. La gestion externe présentant bien plus d'avantages (notamment la possibilité d'une relance automatique du démon en cas de crash de celui-ci), j'ai encore modifié le script rc.sysinit pour qu'il tue le démon juste avant de se terminer.

J'ai d'ailleurs été quelque peu surpris de constater qu'il faut tuer le démon sans l'en informer (à savoir lui envoyer un signal de fin SIGKILL), plutôt que de lui demander de s'arrêter. Si celui-ci reçoit le signal de fin, alors il coupe aussi le pilote watchog et le redémarrage est assuré !

Une subtilité à laquelle j'ai également du faire face sur les CCPCs est la libération du disque de démarrage afin de récupérer la mémoire RAM qu'il utilise (inutilement). Comme je l'avais expliqué dans le 2.4.3, il existait un script dans les outils Red Hat qui permet justement d'enlever le disque de démarrage de la mémoire. Techniquement, celui-ci se sert d'une entrée dans /dev (/dev/ram0), symbolisant l'espace mémoire dans lequel le disque est chargé, pour demander la libération grâce à un appel système. Sur les CCPCs, en raison du noyau minimaliste, une telle entrée n'existe pas, l'appel échoue donc. L'astuce consiste alors à demander au noyau Linux de libérer le cache, en lui mettant à vrai, la variable /proc/sys/vm/drop_caches, en utilisant la commande suivante : `/sbin/sysctl vm.drop_caches=1`. On se retrouve alors avec un CCPC qui a 30 Mo de mémoire de libre (sur 64 !). J'ai modifié le script pour qu'il le fasse à chaque démarrage.

Fort de ces modifications, j'ai obtenu un CCPC qui a démarré mais surtout qui fonctionnait et consommait peu de mémoire (le principe étant d'en laisser suffisamment pour que les programmes de contrôle puissent fonctionner). Le tout étant géré par Quattor !

Pour en arrivant là, il m'a justement fallu modifier le composant diskless de Quattor, pour qu'il désactive les scripts de démarrage de Quattor sur les CCPCs, mais les laisse actifs sur le protonode. S'il est possible de faire tourner Quattor sur le protonode, il n'est pas envisageable de le faire tourner sur

les CCPCs, vu sa consommation mémoire et CPU en comparaison du peu de ressources disponibles.

2.7 PierreFS

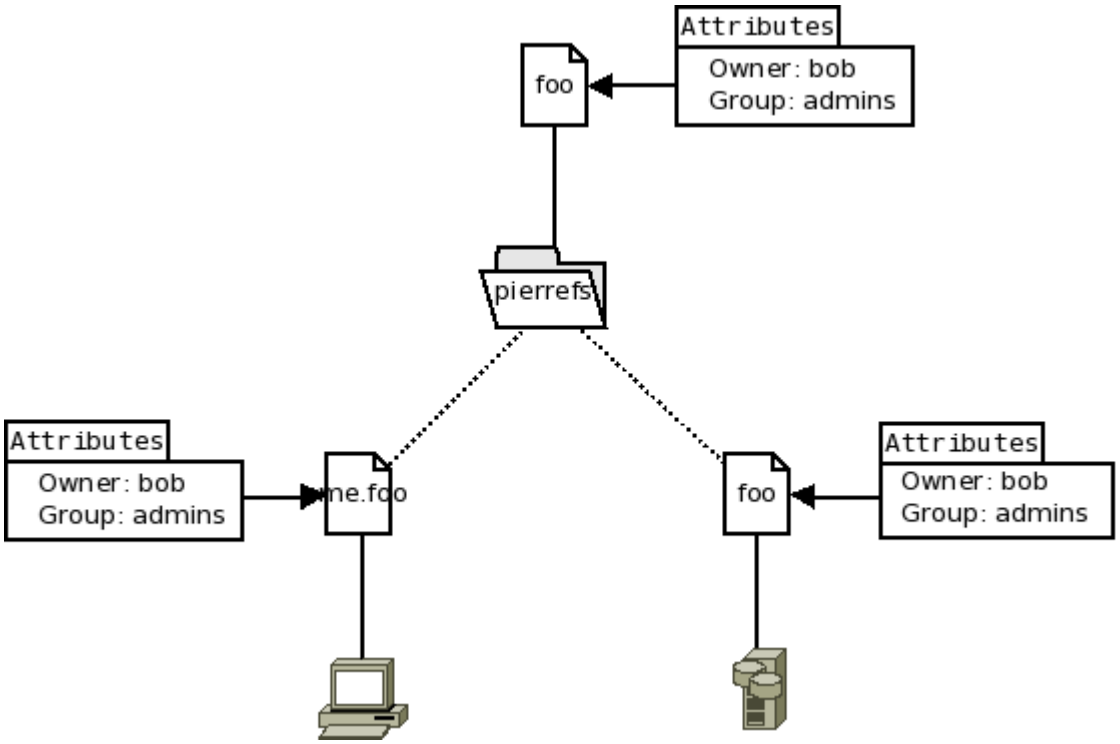
Au fur et à mesure de l'utilisation et des tests des nœuds diskless, j'ai pu m'apercevoir d'un certain nombre de faiblesses liées à la fusion de systèmes de fichiers en elle-même. Je les avais effleurées et ignorées lors de la correction des problèmes dans unionfs-fuse, considérant qu'a priori, ceci n'avait pas un impact important.

Néanmoins, il s'avère qu'à l'utilisation d'un nœud diskless, on se retrouve quand même dans la situation où l'on sent les limites de la fusion. On ne peut pas changer les attributs d'un fichier ou d'un dossier, ou alors, on se retrouve avec un copyup qui rend le système plus difficile à administrer. Dans notre cas, là, le choix avait été fait d'avoir un système facile à gérer. Si le système était facile à administrer, on se retrouvait avec un système de fichier pauvre en fonctionnalités (et de surcroit limitées). Il y avait certes, toujours plus de fonctionnalités qu'avec l'ancienne méthode, mais ici, le système d'exploitation ne considérait pas qu'il était sur un répertoire racine en lecture-seule. Ce qui changeait radicalement la donne.

La décision a donc été prise de développer un nouveau système de fichiers, répondant au doux nom de PierreFS (à défaut de trouver mieux...[1]). Celui-ci devait se présenter comme la solution « ultime » aux problèmes que la fusion de systèmes de fichiers traditionnelle pouvait poser. Pour résumer les fonctionnalités, celui-ci offre de façon transparente toutes les fonctionnalités qu'offre un système de fichiers standard, mais également les fonctionnalités liées a la fusion de systèmes de fichiers et ce tout en limitant le nombre de copyup.

On se souvient bien de l'utilisation des copyup pour unionfs : il s'agit d'autoriser des modifications sur un fichier (que ce soit les données ou les métadonnées). Néanmoins, le copyup parait dangereux (et inutilement consommateur de ressources) quand on ne souhaite modifier que les métadonnées (ou encore attributs) d'un fichier. C'est ici qu'à l'instar des whiteouts, un nouveau fichier caché, propre au système de fichiers est apparu. Il s'agit du fichier de métadonnées. Celui-ci ne réalise en réalité qu'un copyup des métadonnées. On se retrouve donc avec un fichier vide qui porte les métadonnées. Cela permet de réduire la redondance de données. Le fichier de métadonnées utilise la syntaxe suivante : .me.<nom du fichier d'origine>. Me étant la forme raccourcie de metadata (métadonnées en anglais).

1 À noter que le LHCb utilise déjà un système de fichiers nommé RainerFS tiré du prénom de son développeur au sein de l'équipe Online.



Parmi les autres modifications, on trouve également ce que l'on pourrait qualifier de régression : le système de fichiers ne gère plus que deux branches. En effet, dans le cadre du CERN, la gestion des branches multiples alourdit inutilement le système de fichiers vu qu'elle n'est pas utilisée.

Enfin, on retrouve également les modifications qui avaient été apportées à unionfs-fuse visant à réduire le nombre de copyup par suppression des copyup devenus inutiles. Ces modifications doivent aussi faire l'objet d'optimisation. En effet, au lieu d'ouvrir les deux fichiers à chaque fois, un systeme de cache sera mis en place (via des fichiers .ca.) contenant des informations sur le fichiers d'origine lors de sa copie.

Pour des raisons de simplicité (et donc de temps), et pour essentiellement montrer les capacités du système de fichiers, celui-ci a été développé sous FUSE.

3 Résultats et livrables

3.1 Paquets livrés

L'objectif premier du stage, était donc d'améliorer le HLT pour qu'il fonctionne avec la fusion de systèmes de fichiers, et surtout sans les outils Red Hat. L'objectif secondaire était, si possible bien évidemment, l'intégration dans Quattor, puisque ce dernier ne prenait pas en charge via son composant diskless_server la fusion et était dépendant des outils Red Hat.

La fusion a donc été livrée par l'intermédiaire d'un pilote FUSE, déjà existant, unionfs-fuse qui néanmoins vient avec des modifications importantes, visant à correspondre au maximum aux besoin de l'expérience LHCb. Ce pilote, écrit en C, est fournit sous forme de paquet RPM pour faciliter son déploiement.

J'ai également été obligé de modifier le pilote FUSE. Par défaut, la version de FUSE utilisée sur SLC5 n'autorise pas l'export sur le réseau. J'ai donc du la compiler à nouveau pour permettre cela. De plus, la version fournie était ancienne, et des bugs étaient présents. J'ai donc pris la dernière version que SLC5 pouvait supporter pour la compiler moi-même et la mettre à disposition de LHCb. Enfin, comme je leur fournissais déjà un paquet non officiel, j'ai décidé d'y inclure des backports de correctifs, afin de s'assurer d'avoir un FUSE fonctionnant correctement, cette partie étant critique. Ici, je n'ai pas réellement écrit de code, j'ai simplement fait de l'audit de code (pour corriger quelques bugs) et j'ai surtout fait le backport. Tout cela étant bien évidemment en langage C également.

Cet ensemble, même s'il est livrable, ne correspond pas totalement aux demandes formulées par l'équipe LHCb Online notamment en raison du bug tenace de /proc et /sys (qui reste non corrige à l'heure actuelle) qui ne semble affecter que les nœuds standards.

Un autre élément livrable important (même si non disponible sous forme de RPM pour l'instant, vu que je continue à le modifier) est le composant diskless_server pour Quattor. Celui-ci, qui s'est vu complètement réécrit pour rajeunir le code, apporter plus de souplesse à l'utilisation, mais également une configuration plus simple, comporte aussi tout le support nécessaire pour la mise en place d'une ferme avec fusion de systèmes de fichiers, que ce soit pour les CCPCs que pour des serveurs standards. Les versions 5 et 6 de SLC étant supportées. Ce composant, écrit en perl est la clé de voute du travail réalisé, il permet de faire fonctionner tous les éléments malgré leurs architectures très différentes et offre la possibilité de déployer très rapidement de nouveaux nœuds dans la ferme.

Pour fonctionner proprement, il s'appuie sur des scripts récupérés des outils abandonnés de Red Hat écrit en bash. Ceux-ci réalisent notamment l'image de démarrage, ou l'initialisation à partir du disque de démarrage. Je les ai bien sur modifiés pour qu'ils supportent la fusion et les CCPCs.

Tous ses éléments seront donc installés sur le serveur servant les nœuds diskless.

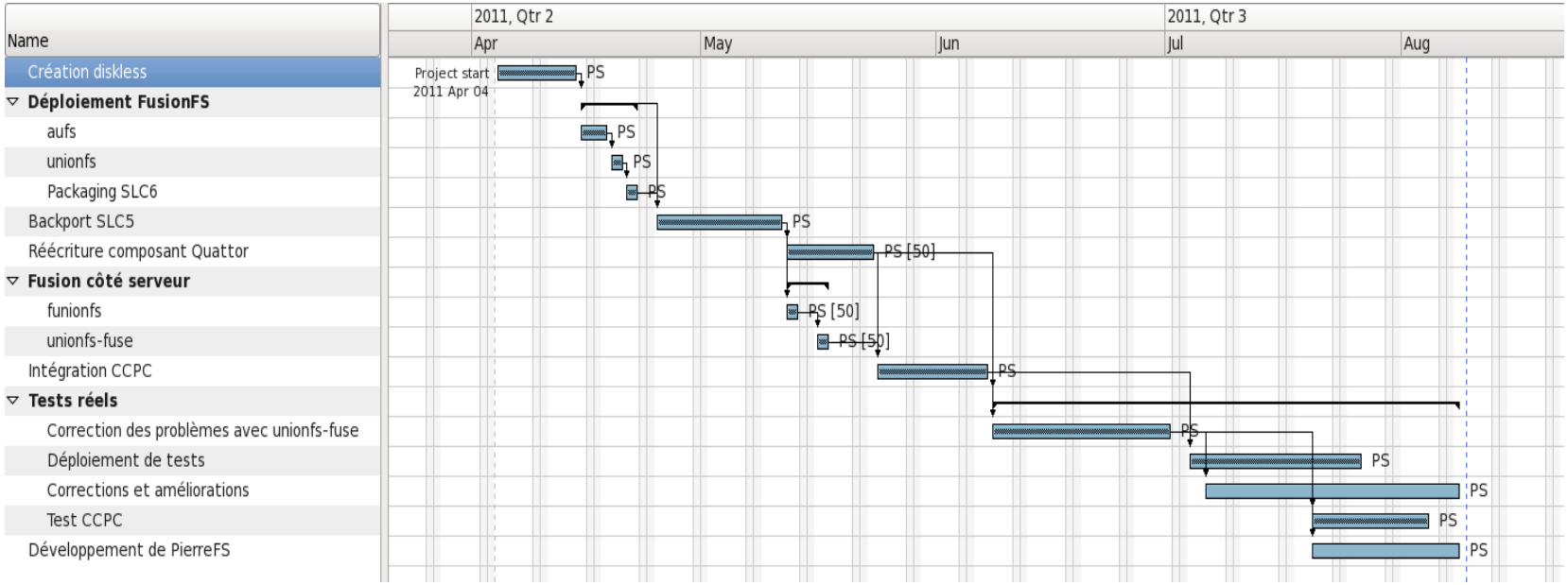
Le composant, lui fonctionne conformément aux attentes. Une modification pourrait néanmoins avoir lieu pour prendre en compte une subtilité technique des CCPCs (au niveau de leur sortie série).

Enfin, au moment de la rédaction de ce rapport, le développement de PierreFS n'a pas été terminé. Cependant, certains tests ont déjà été réalises, notamment sur l'utilisation des fichiers .me., mais une mise en application n'est pas encore possible.

3.2 Planning de réalisation

Si, comme je l'ai expliqué au début, je n'ai pu fournir de planning prévisionnel de travail, j'en ai maintenu un durant l'ensemble du stage, afin de montrer les passages difficiles, ou au contraire, les éléments qui ont été réalisés. Cela permet notamment de tirer un enseignement important : le diskless sous SLC6 est aisé !

On peut également constater que sur la fin, j'ai réalisé plusieurs tâches en parallèle. Plusieurs raisons expliquent cela. Tout d'abord, je tenais à répondre aux demandes de fonctionnalités qui découlaient de l'avancée de mon travail, tout en réglant les problèmes que je pouvais constater sur l'état actuel. Mais également, certains tests de correction, notamment pour le bug de /proc et /sys étant longs, il fallait en lancer plusieurs en parallèle (pour voir quelle solution pouvait fonctionner). Aussi, la volonté de développer PierreFS qui n'était à l'origine pas demandé s'est rajouté à cette parallélisation.


Name
2011, Qtr 2
2011, Qtr 3
Apr
May
Jun
Jul
Aug
Création diskless
Déploiement FusionFS
aufs
unionfs
Packaging SLC6
Backport SLC5
Réécriture composant Quattor
Fusion côté serveur
funionfs
unionfs-fuse
Intégration CCPC
Tests réels
Correction des problèmes avec unionfs-fuse
Déploiement de tests
Corrections et améliorations
Test CCPC
Développement de PierreFS
Project start
2011 Apr 04
PS
PS [50]

Conclusion

À l'origine, l'objectif premier du stage était d'améliorer le fonctionnement des fermes et tout particulièrement du HLT. À l'issue de ce stage, cet objectif n'est pas complètement rempli, mais en bonne voie. Quelques bugs empêchent encore la bonne marche du projet. Néanmoins, le stage ne s'est pas arrêté à cet objectif et a été élargi. La gestion via Quattor a été améliorée et surtout, il y a été ajoutée une gestion des CCPCs. Si ceux-ci ne pouvait jusque là pas être administrés par Quattor, cela est maintenant possible et cela permet de finir les travaux du stage de Sylvain Planche, puisqu'ils fonctionnent sous SLC5. C'était un des progrès les plus attendus par le responsable des CCPCs.

Par ailleurs, la force de ce qui a été réalisé est la possibilité d'évoluer. En effet, le nouveau composant Quattor gère aussi bien SLC5 que SLC6, et ce, sans réelle dépendance à de quelconques outils, de sorte que le jour où la release 6 aura été validée, l'équipe LHCb Online pourra passer sans difficultés de SLC5 à SLC6.

S'il y a encore du travail à accomplir pour régler les problèmes (principalement /proc et /sys) qui empêchent la mise en production du stage sur HLT, celui-ci peut être réalisé à distance, et c'est ainsi que j'assurerai également le support pour ce qui a été fait jusque là.

Pour les CCPCs, ces problèmes ne se présentant pas, les résultats du stage seront mis en production au prochain grand arrêt technique de deux semaines du LHC, en décembre.

Enfin, le travail réalisé ici fera l'objet d'une présentation (orale, ainsi qu'un poster, accompagnés de la publication d'un article) à la conférence ICALEPCS 2011 qui aura lieu à Grenoble cette année.

Lexique

Backport : le backport (on trouve la traduction rétro-portage en français) est le fait de porter un correctif prévu et développé pour une version précise d'un logiciel vers une version plus ancienne du logiciel (et plus nécessairement maintenue par ses développeurs). C'est une pratique courante dans le monde du libre, notamment pour la gestion des failles de sécurité.

Dynamic Host Configuration Protocol (DHCP) : C'est un protocole sur le réseau IP qui permet de configurer automatiquement un hôte qui rejoindrait le réseau. Il est donc constitue d'un serveur qui va fournir aux hôtes clients une adresse IP, voire un nom, voire un emplacement de démarrage (bien que dans la majorité des cas, l'IP soit suffisante). La configuration peut être statique, auquel cas, le serveur en fonction de l'hôte fournira une configuration prédéfinie ou dynamique, où les informations transmises sont « aléatoires ».

Filesystem in USErspace (FUSE) : Ce pilote de système de fichier Linux représente une bibliothèque de fonctions de pilote de système de fichier générique du noyau qui permet ensuite de développer des systèmes de fichiers juste via des librairies en mode utilisateur. Cela permet donc d'ajouter le support de nouveaux systèmes de fichiers à Linux sans avoir besoin de modifier le noyau.

Fork : En informatique, le fork est la séparation d'un projet en deux projets distincts. L'ancien, et le nouveau qui prend pour base l'ancien. Généralement, un fork se produit sur impulsion d'un ou plusieurs développeurs qui désirent faire renaitre un projet mort, ou qui sont en désaccord avec les plans de développement actuels du projet. On peut trouver le terme « embranchement » en français pour designer une telle séparation.

Hook : Le hook (ou encore « crochet » en français) est une technique de personnalisation simple des applications. Il s'agit d'un bout de code, d'un script qui sera exécuté à un instant donné par l'application (avant l'appel d'une fonction, après l'appel d'une fonction, etc.). Généralement, les hooks sont des fichiers qui sont enregistrés à un endroit précis, avec un nom correspondant à une règle utilisée par l'application. Mais il peut également s'agir d'hooks enregistrés dynamiquement dans l'application.

Intelligent Platform Management Interface (IPMI) : IPMI est une interface matérielle fournie avec certains ordinateurs (généralement les serveurs pour les entreprises) qui permet, à distance, de vérifier l'état de la machine et d'intervenir dessus, et ce, qu'elle soit allumée ou non. On peut donc allumer, redémarrer ou éteindre une machine via IPMI. Mais également voir ce que la machine afficherait à l'écran si on été devant (et qu'elle avait un tel écran).

Kernel Panic : Le kernel panic est la réaction du noyau Linux à un enchainement d'instructions non supporté aboutissant à un état dans lequel le noyau n'est plus programmé pour agir. Dans une telle situation, alors le noyau arrête le système abruptement, pour éviter la mise en péril des données. L'arrêt peut être provoqué soit par le noyau, soit par un pilote qui rencontre le même type de problème et qui demande l'arrêt du système faute de pouvoir continuer. C'est l'équivalent des écrans bleus de Windows.

Network File System (NFS) : NFS (en français Système de Fichiers Réseau) est un système de fichiers qui, comme son nom l'indique, permet le travail en réseau. Sa fonction première est le partage de fichiers à travers un réseau. Il permet ainsi à un serveur d'exporter des fichiers, des dossiers, voire une arborescence complète pour la rendre disponible à des clients. De l'autre côté, les clients peuvent

accéder et monter les données exportées par le serveur, et s'en servir comme si elles étaient résidentes.

Patch : Le terme patch désigne trois choses intimement liées mais non égales en informatique. Dans la majorité des cas, le terme patch (traduisible en français par correctif) désigne une modification du logiciel visant à supprimer un ou plusieurs bugs. Dans les autres cas, il désigne un fichier (traduisible en français par différentiel et parfois nommé diff en anglais) dans lequel sont écrites toutes les modifications à apporter au code source. Ce fichier tire son nom de l'application à utiliser pour appliquer automatiquement les modifications contenues dedans sur le code source : patch (qui est donc la troisième chose).

Random Access Memory (RAM) : Mémoire à accès rapide (comparativement à un disque dur) de l'ordinateur. Les données qui y sont stockées sont perdues à chaque redémarrage de l'ordinateur.

Ramfs : Ramfs (pour RAM File System, « système de fichiers RAM » en français) est un système de fichiers où les données ont la particularité de ne pas être persistantes. Elles sont stockées dans la mémoire vive (RAM) de l'ordinateur, et seront perdues à l'extinction de l'ordinateur. Néanmoins, ce système possède de nombreux avantages : stockage sur un support standard et accès rapide aux données.

Rootfs : Rootfs (pour Root File System, « système de fichiers racine » en français) est le répertoire racine d'un système d'exploitation basé sur Linux. On le note /. Tous les autres fichiers et répertoires ont pour point d'origine ce répertoire. Ces systèmes d'exploitations offrent également la possibilité d'utiliser plusieurs rootfs, via la commande chroot qui permet de passer d'un rootfs à l'autre le temps de l'exécution d'une commande.

Secure SHell (SSH) : Ce protocole de console sécurisée (en français) est la technique d'administration système la plus fréquente pour les systèmes Linux. Elle permet de prendre à distance le contrôle d'un ordinateur (nomme hôte) en se connectant à l'ordinateur comme un utilisateur local (à l'hôte, qui doit donc avoir un compte). Les échanges entre l'hôte et l'ordinateur prenant le contrôle sont chiffrées, ce qui explique la nécessité de clés de chiffrements, qui permettent entre autres d'authentifier l'hôte (duo nom d'hôte et clé).

SubVersioN (SVN) : SVN est un outil de gestion de version. Cela signifie que l'on peut envoyer des documents sur un serveur et ceux-ci sont alors stockés, annotés d'un numéro (de version) unique. Chaque changement provoque une incrémentation de ce numéro. Il est possible de voir les différences entre les versions, voire de consulter l'état d'un fichier à une version donnée.

Trivial File Transfert Protocol (TFTP) : Le protocole trivial de transfert de fichiers est un protocole de transfert de fichiers sur le réseau particulièrement simple et peu verbeux. Venant sans authentification, chiffrement, ou même listage de fichiers présents sur le serveur. Son utilisation repose sur une configuration précise des clients qui savent ce qu'ils cherchent sur le serveur et où.

Références bibliographiques

[1] Diskless Environments [Ressource électronique]. [Raleigh, Caroline du Nord] : Red Hat, Inc. [réf. du 5 mai 2011]. Disponible sur : http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/3/html/System_Administration_Guide/ch-diskless.html

[2] Package And Driver Changes [Ressource électronique]. [Raleigh, Caroline du Nord] : Red Hat, Inc, 2010. [réf. du 5 mai 2011]. system-config-netboot. Disponible sur : http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Migration_Planning_Guide/chap-Migration_Guide-Package_Changes.html

[3] Setting Up A Remote Diskless System [Ressource électronique]. [Raleigh, Caroline du Nord] : Red Hat, Inc, 2009. [réf. du 5 mai 2011]. Setting Up A Remote Diskless System. Disponible sur : http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Storage_Administration_Guide/nfs-diskless-systems.html

[4] BOVET Daniel, CESATI Marco, Understanding the Linux Kernel, Third Edition, O'Reilly Media, Inc. , Sebastopol , 2006.

[5] aufs2 on nfs3 root failing [Ressource électronique]. J. R. Okajima, 2010. [réf. Du 23 mai 2011]. Disponible sur : http://sourceforge.net/mailarchive/message.php?msg_id=24353606

[6] README.NFS [Ressource électronique]. Miklos Szeredi, 30 septembre 2006. [réf. Du 20 juin 2011]. Disponible sur http://fuse.git.sourceforge.net/git/gitweb.cgi?p=fuse/fuse;a=blob_plain;f=README.NFS;hb=4a541a9617e2f6cf6762d78dd8554d7b92e09485

[7] Removed resource leaks [Ressource électronique]. Pierre Schweitzer, 6 juin 2010. [réf. Du 7 juin 2011]. Disponible sur http://hg.podgorny.cz/unionfs-fuse/rev/da9fa1d0736d

[8] ATTR [Ressource électronique]. Andreas Gruenbacher. [ref. Du 20 juin 2011]. DESCRIPTION. Disponible sur http://www.linuxmanpages.com/man5/attr.5.php.

[9] PLANCHE Sylvain, Création d'une distribution basée sur Scientific Linux 5 pour les processeurs embarqués de l'expérience LHCb, ISIMA, Aubière, 2010.

Annexes

Table des annexes

Table des illustrations

Annexe 1 : Le diskless avec les outils Red Hat

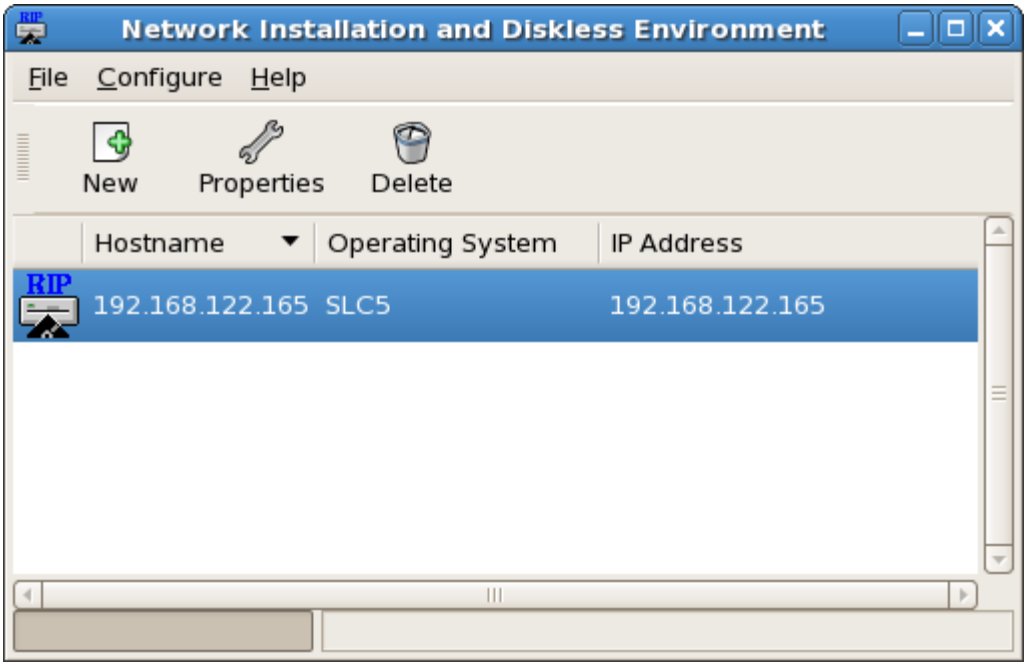
Jusqu'à RHEL (Red Hat Entreprise Linux, distribution sur laquelle est basée SLC) version 5, Red Hat fournissait des outils afin de rendre disponible et facile le diskless. Avec RHEL6, ils ont totalement abandonné et recommandent maintenant d'utiliser des outils tels que Satellite.

Pour cette annexe, nous allons donc faire un saut dans le passé et regarder comment cela marchait avec SLC5. L'outil graphique s'appelait system-config-netboot. Un outil en ligne de commande était également fourni : system-config-netboot-cmd. Ces outils ne géraient pas que le diskless mais permettait également de faire une installation sur le réseau. Dans notre cas, seul le diskless importe.

Ces outils requéraient qu'une architecture soit déjà en place. Un serveur NFS devait exister et exporter un répertoire dans lequel se trouvait un sous-répertoire root. Et dans ce root, on trouvait le rootfs partage. Pour la création du rootfs partagé, toutes les méthodes étaient bonnes, même si Red Hat conseillait d'utiliser la copie d'un système déjà existant et proprement installé.

Il fallait alors lancer l'outil system-config-netboot sur la machine qui accueillerait le serveur TFTP. Un assistant aidait à choisir le serveur pour régler les données. Il suffisait de lui donner l'adresse du serveur NFS et le chemin du répertoire exporté. De la, il créait l'image de démarrage, tout en effectuant quelques modifications sur le shared root pour faciliter le diskless. Puis, il copiait le noyau dans le répertoire pour le TFTP. Dans le même temps, il, configurait aussi le serveur TFTP.

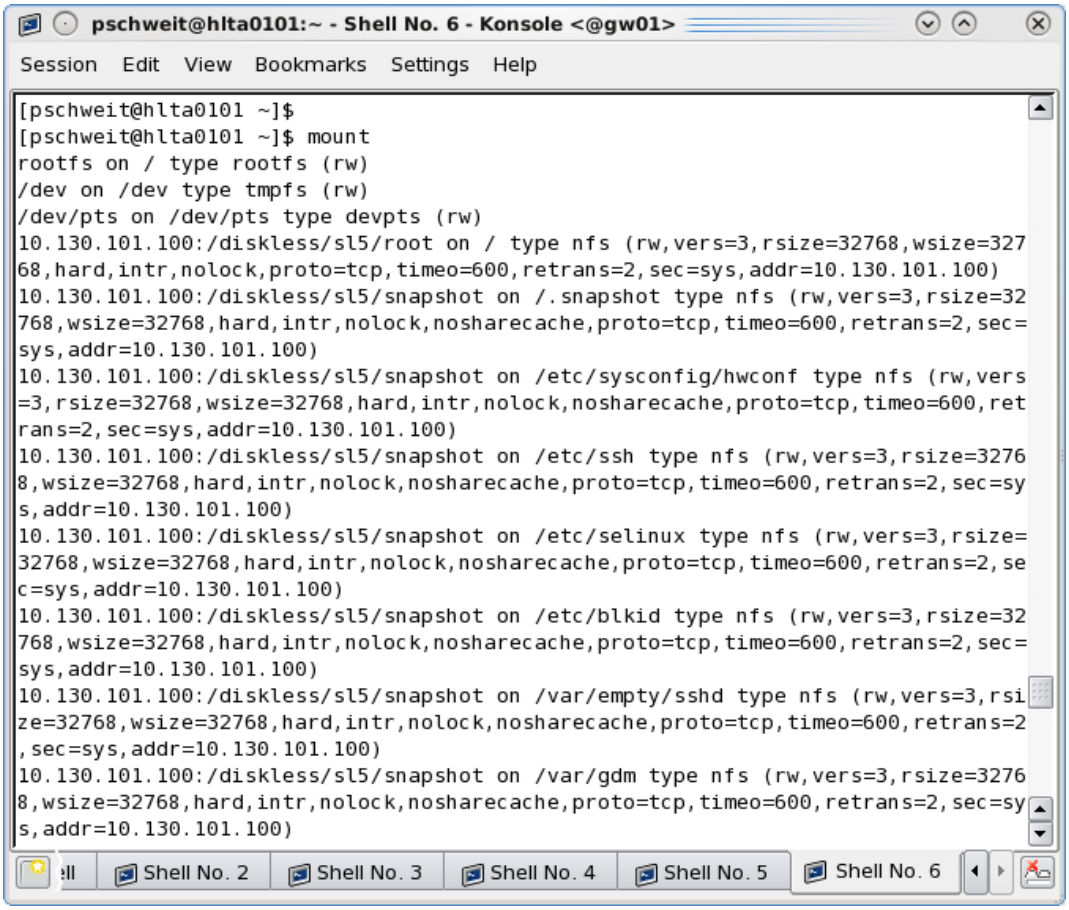
Une fois cette étape accomplie, il était possible d'ajouter des clients (puisque, ne l'oublions pas, chaque client doit avoir son fichier de configuration dans le répertoire TFTP) que l'outil configurait proprement.



On voit donc que cet outil était surtout une aide pour les gestes « finaux » et non une intégration complète telle que le composant Quattor permet de le faire.

Par ailleurs, ce diskless était très contraignant. On a déjà pu voir que l'arborescence était imposée. De plus, le shared rootfs était partagé par tous les clients. Mais pourtant, il n'y avait pas de fusion (ou partielle...). Le script de démarrage utilisé par system-config-netboot avait donc recours à une astuce pour offrir au système d'exploitation la possibilité d'écrire sur certains fichiers. À côté du shared rootfs, on trouvait des snapshots, un par client. Chacun contenait les fichiers qui pouvaient être écrits. Et surtout, à la racine, un fichier contenant la liste de ses fichiers existait. Ainsi, au démarrage, le script montait le shared root, puis montait chacun des fichiers de la liste séparément en les remontant par dessus les fichiers du shared root. De sorte que le noyau écrive bien dans les fichiers du snapshot. On aboutissait donc à une solution lourde et peu efficace. Pour pouvoir ajouter un nouveau fichier en écriture, il fallait par exemple redémarrer les nœuds.

De plus, sous Linux, il est possible de lister tous les éléments qui sont actuellement montés (ce qui est utilisé pour régler certains problèmes). Comme ces fichiers étaient montés séparément, on aboutissait à une noyade sous les informations (non intéressantes) lors de la demande la liste puisque tous les fichiers accessibles en écriture étaient listés.



Enfin, pour pouvoir parvenir à cette astuce, en réalité, le script de démarrage ne montait pas le

shared root, mais le répertoire exporté contenant le shared root (pour permettre le remontage à partir du répertoire snapshot). De sorte qu'en réalité, chaque nœud pouvait accéder au snapshot de tous les nœuds.

Annexe 2 : Démarrage de Linux

Le démarrage de Linux se déroule toujours de la manière suivante et ce, que le démarrage ait lieu sur le réseau, ou non.

En effet, la première étape primordiale est celle du chargeur de démarrage (*bootloader*, en anglais). Celui-ci va récupérer les informations de démarrage : quel noyau et son emplacement, quelle image de démarrage et sa localisation, et quelle ligne de commande fournir au noyau. Une fois ces trois informations récupérées, le chargeur de démarrage va récupérer (sur le réseau, ou sur un support magnétique ou optique) et placer en mémoire l'image de démarrage ainsi que le noyau et lancer ce dernier avec la ligne de commande fournie.

Le noyau va tout d'abord fonctionner seul et commencer par s'initialiser et initialiser une première partie du matériel « de base » (carte-mère, processeur, bus, etc). Théoriquement, une fois cette étape terminée, on pourrait dire que le démarrage de Linux (le noyau) est terminé. Néanmoins, le démarrage du système d'exploitation, lui, n'est pas terminé.

En effet, le noyau va alors lire l'image de démarrage et s'en servir comme répertoire racine. Il va alors lancer un script d'initialisation sur ce rootfs. Celui-ci est livré avec l'image de démarrage (et est généralement nommé /init).

Ce script est très important, il est chargé de terminer l'initialisation du matériel dont il a besoin, en chargeant si nécessaire les pilotes et les modules qui font défaut (et qui sont bien évidemment présents dans l'image de démarrage). Ce script n'a qu'un seul objectif : monter le véritable répertoire racine. Et c'est pour cela qu'il a besoin de compléter l'initialisation du matériel, puisque ce montage passera fatalement par l'interrogation du matériel (disque, carte réseau, etc).

Ce script est donc dépendant du montage à effectuer. Chaque configuration de démarrage a un script diffèrent, les actions à réaliser n'étant pas les mêmes, le type de matériel impliqué changeant également.

Dans notre cas, ce script charge par exemple le pilote nécessaire au bon fonctionnement de la carte réseau, puis il charge le pilote NFS pour pouvoir monter le rootfs. Il récupère alors les informations du DHCP. Il est ici inutile de charger le pilote du disque dur (si tant est qu'il y en ait un !).

Une fois que l'initialisation est véritablement terminée, le script monte le rootfs réel. Cette étape est critique. Si elle échoue, le script retourne une erreur et le noyau provoque un kernel panic. Comme le noyau a déjà un rootfs et ne peut en avoir deux, le rootfs réel est monté dans un sous-répertoire du répertoire racine de l'image de démarrage.

Dans notre cas, c'est ici que le script interroge le serveur et monte le répertoire exporté par NFS.

Suite à cela, l'exécution du script n'est pas encore terminée, celui-ci réalise encore des actions post-mount. Cela peut être de changer la configuration sur le rootfs réel. Par exemple, quand la fusion avait lieu sur chacun des nœuds (avec unionfs), c'est ici qu'avait lieu la fusion. Le snapshot et le shared rootfs ayant été montés préalablement.

Le script va alors réaliser ses dernières actions afin d'assurer une continuité dans le fonctionnement du noyau. Il va lier le répertoire /proc qui se trouvait sur l'image de démarrage à celui du rootfs réel, de même pour /dev et /sys. Ces répertoires dynamiques contiennent des informations sur la configuration du noyau et sur le matériel, c'est pourquoi ils doivent perdurer. Le reste de l'image de

démarrage n’a pas d’importance vu qu’il ne contient que le nécessaire au démarrage (dont on trouvera finalement une copie sur le rootfs réel).

C’est alors que vient l’étape finale qui achève le démarrage : le switch root. C’est à ce moment que le script informe le noyau (grâce à un binaire) qu’il doit changer de répertoire racine, et qu’il doit exécuter un nouveau script de démarrage. Le nouveau répertoire racine est bien évidemment celui qui a été monte précédemment. Quand au script, dans la majorité des cas, il s’agit du script d’init du système d’exploitation lui-même.

Quand le switch root débute, le démarrage de Linux est terminé. Le noyau est passé sur le rootfs qu’il utilisera jusqu’à ce qu’on l’éteigne et la distribution (l’ensemble du système d’exploitation) peut enfin démarrer. C’est à ce moment que sont alors lances les services, les démons. Une fois ceci terminé, l’utilisateur peut enfin se connecter à sa machine.

Annexe 3 : Gestion des patchs du noyau SLC5

Pour les pilotes de fusion de systèmes de fichiers, j'avais besoin de modifier le noyau de SLC5, autant pour que le pilote fonctionne que pour compiler le pilote dedans. Le principe de création des RPMS est simple : il y a une archive qui contient le code, optionnellement des patchs, et un fichier texte d'extension .spec (comme spécifications) qui décrit le paquet à réaliser, les patchs à appliquer ainsi que la procédure à suivre pour construire le paquet (il peut également contenir du code bash à exécuter). Lors de la génération du paquet RPM, cette archive est extraite, les patchs appliqués, le code compilé, pour finalement donner le RPM. J'aurais donc pu modifier cette archive directement, et appliquer les changements de code dessus. Mais cela aurait été une pratique particulièrement contre-productive puisque personne n'aurait pu voir (ni corriger le cas échéant) les changements effectués.

De plus, dans le cas du noyau SLC5, il y a évidemment l'archive contenant le code, mais il y a également un grand nombre de patchs (plus de 5000) qui sont appliqués avant la compilation. Et de ce fait, une fois tous ces patchs appliqués, le patch de unionfs ne s'appliquaient plus. J'ai également tenté dans l'autre sens, en appliquant d'abord le patch de unionfs puis les autres, mais cela ne donnait pas non plus de résultat. Je n'avais donc d'autre choix que d'appliquer tous les patchs sans passer par la procédure RPM, et de refaire un patch pour unionfs. Procédure particulièrement fastidieuse. Donc, afin de faciliter cette étape (pour moi et ceux qui pourraient avoir besoin de le maintenir), j'ai mis au point un parser en perl. Le parser va donc lire le fichier spec, établir la liste des patchs à appliquer (certains dépendants de paramètres précis) puis générer un fichier qu'il suffira de lancer pour que les patchs soient appliqués.

Ce script va un peu plus loin en intégrant aussi certains scripts bash qui seraient inclus et nécessaire pour l'application des patchs.

```
#!/usr/bin/perl
# File:    generate_patcher.pl
# Author:  Pierre Schweitzer <pierre.jean.schweitzer@cern.ch>
# Created: 27-Apr-2011
# Purpose: Generate a bash script to apply patches given a spec file

use strict;
use warnings;

# Globals
my %defines_table = ("nil" => "",
                     "_tmppath" => "/tmp",
                     "_default_patch_fuzz" => "2");
my %patches_table = ();
my $skip = 0;
my $patching = 0;
my $arch = "x86_64";
my @skips = ();

# Get date
my @months = qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec);
my ($mday, $mon, $year) = (localtime())[3,4,5];
$year += 1900;

# Open required log files
open(my $specs, "<", "kernel-2.6.spec") or die "Can't open kernel specs: $!";
```

```
open(my $bash, ">", "apply_patches.sh") or die "Can't open bash script: $!";

# Print script header
print $bash "#!/bin/sh\n";
print $bash "# File:    apply_patches.sh\n";
print $bash "# Author:  Pierre Schweitzer <pierre.jean.schweitzer\@cern.ch>\n";
print $bash "# Created: $mday-$months[$mon]-$year\n";
print $bash "# Purpose: Apply all patches to the kernel\n";
print $bash "# Warning: Automatically generated file. DO NOT EDIT\n";
print $bash "#          Building everything for $arch\n\n";

print $bash "# Execute everything in kernel dir\n";
print $bash "cd ./SLC5_kernel/\n";

while (<$specs>)
{
  # Remove end of line chars
  (my $line = $_) =~ s/(\v*)$//;
  my @exploded_line = split(/ /, $line);

  # First, handle if-condition
  if ($line =~ /^(\%else)/)
  {
    $skip = !$skip;
  }
  elsif ($line =~ /^(\%endif)/)
  {
    if ($#skips != -1)
    {
      # Restore previous state
      $skip = pop(@skips);
    }
    else
    {
      $skip = 0;
    }
  }

  next if ($skip == 1);

  # If line is a if condition
  if ($line =~ /^(\%if )/)
  {
    # Save previous state
    push(@skips, $skip);

    # Handle the not
    if ($exploded_line[1] =~ /^(!)/)
    {
      $exploded_line[1] = substr $exploded_line[1], 1;
      $skip = 0;
    }
    else
    {
      # Handle optionnal
      $exploded_line[1] = substr $exploded_line[1], 1 if ($exploded_line[1] ↵
```

```
=~ /^(0)/);
      $skip = 1;
    }

    # Only handle vars right now
    if ($exploded_line[1] =~ /^(\%{)/)
    {
      my $define = substr $exploded_line[1], 2, -1;
      # If starting with ? finding can fail
      if ($define =~ /^\?/)
      {
        $define = substr $define, 1;
      }
      else
      {
        # Cannot fail. Ensure var exists
        if (!exists($defines_table{$define}))
        {
          @exploded_line = split(/ /, $line);
          print "Error ($.): Var for condition \"$exploded_line[1]\" was not ↵
found!\n";
          &quit(0);
        }
      }

      # Is it somehow true?
      $skip = !$skip if (exists($defines_table{$define}) && ↵
$defines_table{$define} != 0 && $defines_table{$define} !~ //);
    }
    else
    {
      print "Warning ($.): Un-handled condition \"$exploded_line[1]\"!\n";
    }
  }
  # If line is a ifarch condition
  elsif ($line =~ /^(\%ifarch )/ || $line =~ /^(\%ifnarch )/)
  {
    # Save previous state
    push(@skips, $skip);

    # Handle non-arch
    $skip = ($line =~ /^(\%ifarch )/) ? 1 : 0;

    # If compare is done with a var
    if ($exploded_line[1] =~ /^\%/)
    {
      my $define;

      # For whatever reason, it sometimes comes without {}
      if ($exploded_line[1] =~ /^\%\{/)
      {
        $define = substr $exploded_line[1], 2, -1;
      }
      else
      {
        $define = substr $exploded_line[1], 1;
```

```
      }

      # If starting with ? finding can fail
      if ($define =~ /^\?/)
      {
        $define = substr $define, 1;
      }
      else
      {
        # Cannot fail. Ensure var exists
        if (!exists($defines_table{$define}))
        {
          @exploded_line = split(/ /, $line);
          print "Error ($.): Var for condition \"$exploded_line[1]\" was not ↵
found!\n";
          &quit(0);
        }
      }

      # Is it somehow matching arch?
      $skip = !$skip if (exists($defines_table{$define}) && ↵
$defines_table{$define} =~ /$arch/);
    }
    else
    {
      $skip = 0 if ($exploded_line[1] =~ /$arch/);
    }
  }
  # If line is a define
  elsif ($line =~ /^(\%define )/)
  {
    # Check whether it was already defined
    if (exists($defines_table{$exploded_line[1]}))
    {
      # Warn if already defined and if not default var
      print "Warning ($.): " . $exploded_line[1] . " was already defined!\n" ↵
unless ($exploded_line[1] =~ /^(_)/);
    }

    # Really define
    $defines_table{$exploded_line[1]} = &get_define_value(@exploded_line);

    print $bash "# Defined at line $.\n";
    print $bash "#$exploded_line[1]=$defines_table{$exploded_line[1]}\n";

    # Small hack to get RPM constants working
    if ($exploded_line[1] eq "rpmversion")
    {
      $defines_table{"PACKAGE_VERSION"} = $defines_table{$exploded_line[1]};
      print $bash "#PACKAGE_VERSION=" . $defines_table{"PACKAGE_VERSION"}. "\n";
    }
    elsif ($exploded_line[1] eq "release")
    {
      $defines_table{"PACKAGE_RELEASE"} = $defines_table{$exploded_line[1]};
      print $bash "#PACKAGE_RELEASE=" . $defines_table{"PACKAGE_RELEASE"}. "\n";
    }
```

```
  }
  # If line is a patch
  elsif ($line =~ /^(Patch)\d+(: )/)
  {
    # Get patch number
    my $patch = substr $exploded_line[0], 5, -1;

    # Check whether it was already defined
    print "Warning ($.): patch $patch was already defined!\n" if ↵
(exists($patches_table{$patch}));

    # Really define
    $patches_table{$patch} = $exploded_line[1];
  }
  elsif ($line =~ /^(\%patch)\d+( )/)
  {
    # Get patch number
    my $patch = substr $exploded_line[0], 6;

    # Check if patch exists
    unless (exists($patches_table{$patch}))
    {
      print "Error ($.): patch $patch was required but not defined!\n";
      &quit(0);
    }

    # Handle -b with suffix
    my $error = 0;

    # Prepare bash line
    my $bash_line = " | patch ";
    shift @exploded_line;
    foreach my $arg (@exploded_line)
    {
      $error = 0;
      $bash_line .= $arg . " ";
      if ($arg eq "-b")
      {
        $bash_line .= "--suffix ";
        $error = 1;
      }
    }

    if ($error == 1)
    {
      print "Error: -b arg given without suffix!\n";
      &quit(0);
    }

    $bash_line .= "--fuzz=" . $defines_table{"_default_patch_fuzz"} . " -s\n";

    my $value = $patches_table{$patch};
    # Special case where patch could have been compressed
    if ($value =~ /(.bz2)$/)
    {
      $bash_line = "bzip2 -dc ./../patches/$value" . $bash_line;
```

```
    }
    else
    {
      $bash_line = "cat ./../patches/$value" . $bash_line;
    }
    # Put patch in bash + comment with name & number
    print $bash "# Patch $patch: $value\n";
    print $bash "echo \"Applying patch $patch: $value\"\n";
    print $bash $bash_line;

    # Define whether we are in patching mode
    $patching = ($patch >= 99990) ? 0 : 1;
  }
  # Once %build is reached, patching is over for good
  elsif ($line =~ /^(\%build)/)
  {
    print "Information ($.): reached end of patching. Leaving\n";
    &quit(1);
  }
  # Handle bash script that could be needed for patching
  # We don't want: empty line, empty comment line, commented stuff, spec line
  elsif ($patching == 1 && $line !~ /^$/ && $line !~ /^(#)$/ && ↵
$line !~ /^(#\S)/ && $line !~ /^(\%)/)
  {
    # Write down the line
    print $bash "# kernel-2.6.spec line $.\n";
    print $bash $line . "\n";
  }
}

print "Error: abnormal termination!\n";
&quit(0);

sub get_define_value
{
  # Get list
  my @values = @_;
  # Shift define & define name
  shift @values;
  my $define = shift @values;
  # Make single line
  my $value = join(' ', @values);

  # Get rid of optional
  $value =~ s/\%\{\?/\%\{/g;

  # Small hack to handle parameters checks
  if ($value =~ /(\%\{\w+:\s+\d\}\s\%\{!\w+:\s+\d\})$/)
  {
    print "Information: faulting input parameter $define!\n";

    if ($value =~ /(\%\{_without_\w+:\s+0\})/)
    {
      return "1";
    }
    elsif ($value =~ /(\%\{_with_\w+:\s+1\})/)
```

```
    {
      return "0";
    }
    else
    {
      print "Error: unhandled default configuration!\n";
      &quit(0);
    }
  }

  # Now we must evaluate any of the %{define} in value
  $value =~ s/\%\{\w+}/$defines_table{substr $&, 2, -1}/g;

  return $value;
}

sub quit
{
  # In case of success, complete file
  if ($_[0] == 1)
  {
    print $bash "\n# Get rid of unwanted files resulting from patch fuzz\n";
    print $bash "find . \\( -name \"*.orig\" -o -name \"*~\" \\) -exec rm -f ↵
{} \\; >/dev/null\n";
    print $bash "# Go back to previous dir\n";
    print $bash "cd -\n";
  }

  # Close files
  close $specs or die "Can't close kernel specs: $!";
  close $bash or die "Can't close bash script: $!";

  exit;
}
```

Annexe 4 : Surveillance de /proc et /sys

Le démontage intempestif de /proc et /sys posant un problème (le système d'exploitation n'est plus réellement utilisable), il me fallait trouver une solution. Pendant que j'en cherchais une pour lutter contre le démontage, j'ai du mettre au point une solution temporaire (élégamment nommée *hackfix* en anglais) qui consiste simplement à remonter ces répertoires quand ils sont démontés. Ce programme doit tourner en tant que démon (programme qui tourne en tâche de fond) sur les machines, étant donné qu'une fois /proc et /sys partis, il n'est plus possible de se connecter à la machine.

Ce programme est très simple. Dans un premier temps, il s'initialise en tant que démon, puis, se lance dans un boucle infinie. Pour détecter le démontage, régulièrement, il tente d'ouvrir un fichier sur /proc et un sur /sys. Si l'ouverture échoue, alors, il démonte les répertoires (afin de repartir sur une base saine) et force le remontage (en donnant tous les arguments pour éviter un recours au fichier /etc/fstab dans lequel les entrées proc et sys pourraient être absentes).

Pour des raisons à la fois d'information mais surtout de statistiques (pour tenter de comprendre le phénomène...), le programme enregistre dans un fichier toutes les fois qu'il rencontre un répertoire de démonté et aussi s'il n'arrive pas à le remonter.

```
/**
 * File:    checkproc.c
 * Author:  Pierre Schweitzer <pierre.jean.schweitzer@cern.ch>
 * Created: 17-Jun-2011
 * Purpose: Program for checking for both /sys & /proc mount points
 *          and remount them if required.
 */

#include <time.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mount.h>
#include <sys/types.h>

#define UNUSED_PARAMETER(p) (void)p

static void write_pid(pid_t pid)
{
    int fd;
    size_t len;
    char pid_s[0x6];

    fd = open("/var/run/checkproc.pid", O_WRONLY | O_TRUNC | O_CREAT);
    if (fd == -1)
    {
        /* Ignore failure */
        return;
    }

    len = snprintf(pid_s, sizeof(pid_s), "%d", pid);
```

```
    write(fd, pid_s, len);
    close(fd);

    return;
}

static void append_log(int fd, const char * text)
{
    size_t len;
    time_t rawtime;
    char buffer[0x50];
    struct tm * timeinfo;

    /* Get current date */
    time(&rawtime);
    timeinfo = localtime(&rawtime);

    /* Get a representation */
    len = strftime(buffer, sizeof(buffer), "[%x %X] ", timeinfo);
    if (len == 0)
    {
        return;
    }

    /* Write date to log */
    write(fd, buffer, len);

    /* Write text to log and append line break */
    len = strlen(text);
    write(fd, text, len);
    write(fd, "\n", 1);

    return;
}

int main(int argc, char ** argv)
{
    pid_t child;
    int ret, log;

    UNUSED_PARAMETER(argc);
    UNUSED_PARAMETER(argv);

    /* Fork to run as deamon */
    child = fork();
    if (child == -1)
    {
        /* Write error to stderr since someone may be reading output */
        fprintf(stderr, "Failed starting deamon. Staying foreground\n");

        /* Write current pid */
        write_pid(getpid());
    }
    else if (child > 0)
    {
        /* Write pid and quit */
```

```
        write_pid(child);

        /* Quit parent */
        exit(EXIT_SUCCESS);
    }

    /* First of all, open log */
    log = open("/var/log/checkproc", O_WRONLY | O_APPEND | O_CREAT);
    if (log == -1)
    {
        /* Should we continue or quit? */
        exit(EXIT_FAILURE);
    }

    /* Get new session ID */
    if (setsid() < 0)
    {
        /* More or less ignore failure */
        append_log(log, "Failed getting new session ID. Falling back to parent ↵
ID");
    }

    /* Change working dir to a dir that won't go away */
    chdir("/");

    /* We don't need them any longer */
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);

    append_log(log, "Deamon started...");

    /* Now start infinite loop */
    for (;;)
    {
        /* Check for /proc */
        if (access("/proc/stat", F_OK) == -1)
        {
            /* Log the failure */
            append_log(log, "/proc file system missing");

            /* This call can fail, we don't care */
            umount("/proc");
            ret = mount("proc", "/proc", "proc", 0, "");
            if (ret == -1)
            {
                append_log(log, "Failed remounting /proc");
            }
        }

        /* Check for /sys */
        if (access("/sys/kernel/vmcoreinfo", F_OK) == -1)
        {
            /* Log the failure */
            append_log(log, "/sys file system missing");
```

```
            /* This call can fail, we don't care */
            umount("/sys");
            ret = mount("sysfs", "/sys", "sysfs", 0, "");
            if (ret == -1)
            {
                append_log(log, "Failed remounting /sys");
            }
        }

        /* Sleep for a while */
        sleep(1);
        }

    /* We should never ever reach that point */
}
```