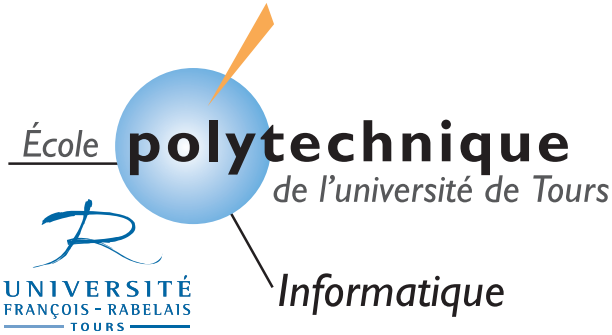


École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique
5^{e}année
2008-2009

Stage de Fin d'Études

Politiques hybrides d'ordonnancement dans un cluster Kerrighed

Encadrant
Matthieu Pérotin
matthieu.perotin@univ-tours.fr

Étudiant
Alexandre LISSY
alexandre.lissy@etu.univ-tours.fr

Université François-Rabelais, Tours

DI5 - Promo 2009

Version du 10 septembre 2009

Table des matières

Table des figures

Liste des tableaux

Liste des codes

Liste des algorithmes

1. Introduction

Dans ce chapitre, nous nous attacherons à présenter le contexte général du projet. Il s'agit à la fois de présenter les travaux qui ont été faits en amont et ont conduit à sa genèse, mais également de présenter le cadre plus général dans lequel il s'inscrit : le calcul haute performance.

Nous commencerons donc par définir les objectifs, en partant des travaux de thèses et projets de fin d'études précédents, en présentant les notions principales qu'ils amènent, et en indiquant les points qui justifie notre étude.

Ensuite, nous définirons le contexte large du projet, et les notions nécessaires qui manquent. Nous rappelons les besoins qui ont été déterminés lors de l'élaboration du cahier des charges, et qui découlent des travaux précédents.

Enfin, nous terminerons en présentant le plan du reste de l'étude.

1.1 Objectifs du projet

Nous allons présenter ici un résumé des travaux précédents. Il s'agit de la thèse de Matthieu Pérotin [Pér08] ainsi que du Projet de Fin d'Études de Mathieu Dabert [Dab08].

Nous présenterons ensuite ce que sont les systèmes à image unique.

1.1.1 « Calcul parallèle sur matériel générique »[Pér08]

L'objectif de ce travail de thèse est de répondre à la demande en calcul haute performance en proposant une solution pratique permettant d'utiliser les ordinateurs déjà en place dans l'université, par exemple au sein des salles de travaux pratiques. Le travail part ainsi d'un double constat :

1. D'une part les chercheurs, quel que soit leur domaine, font part d'un besoin croissant en puissance de calcul ;
2. D'autre part, les ressources informatiques d'une université sont sous-exploitées : les période de fermeture des bâtiments les rendent inacessibles jusqu'à 10 heures par jour, et elles sont souvent peu solicitées au cours des séances de travaux pratiques.

Ainsi, cette thèse montre que l'acquisition d'une plateforme de calcul dédiée, qui a un coût important, n'est pas forcément nécessaire, si tant est que l'on parvient à utiliser judicieusement les ressources déjà en place. Différents projets ont déjà montré la viabilité, dans un contexte particulier, de cette approche. Citons par exemple SETI@Home [ACK+02], BOINC [And04] ou encore Foling@Home [Uni].

Définition des acteurs

Trois types d'acteurs interagissant autours de la plateforme matérielle :
- L'utilisateur interactif ;
- L'utilisateur de Calcul Haute Performance ;
- L'administrateur du réseau.

Ces trois types d'acteurs ont des attentes différentes vis à vis du matériel. L'étude de leurs besoins permet de préciser le cahier des charges qu'une solution doit respecter.

L'utilisateur interactif Il s'agit d'une personne présente physiquement, ou à distance, sur une machine, par exemple, un étudiant au cours d'une séance de travaux pratiques. Son seul objectif est de pouvoir mener à bien son travail dans de bonnes conditions, il s'agit de l'utilisateur légitime de l'infrastructure. Ceci implique que la solution devra être non intrusive, à la fois en terme de disponibilité – la machine doit être accessible – mais également en terme de performances – la machine doit rester « utilisable ».

L'utilisateur de Calcul Haute Performance C'est quelqu'un qui a besoin de résoudre des problèmes tels qu'ils ne peuvent être résolus sur une simple machine, que ce soit à cause du temps d'exécution ou de la mémoire nécessaire. Il souhaite donc pouvoir résoudre soit des problèmes plus grands, soit les résoudre plus vite. Par ailleurs, ce chercheur n'est, par hypothèse, pas un expert en ordonnancement ni en parallélisation. Il est donc souhaitable que la solution soit la plus simple possible. Simple doit ici être entendu dans le sens où il ne doit pas être nécessaire d'apprendre de nouvelles bibliothèques pour procéder à la parallélisation. Une solution simple est par exemple le lancement simultané d'un programme sur des instances différentes. Cependant, l'utilisation de méthodes avancées telles que MPI doit rester possible.

Toujours dans l'objectif de rendre l'utilisation simple, il n'est pas acceptable que l'utilisateur ait à spécifier des informations sur ses applications. La durée d'exécution, l'occupation mémoire, ou encore les machines à utiliser sont autant d'éléments pour lesquels il est difficile (voir impossible) de fournir une information fiable.

L'administrateur du réseau Il s'agit du personnel qui est chargé de l'administration du réseau universitaire, dont on souhaite exploiter les machines. Le nombre d'administrateur est limité et ils sont souvent déjà fortement sollicités. Il est donc nécessaire que la mise en place d'un système de calcul haute performance n'introduise pas de surcout significatif sur la charge d'administration à réaliser. Il faut par ailleurs que le système s'appuie au maximum sur les outils déjà utilisés pour l'administration des machines.

La prise en compte de la multitude des systèmes d'exploitation présents au sein des salles machines est nécessaire, et l'administration doit rester la même quelque soit le système installé, Linux ou Windows.

Enfin, en cas de problème, le système doit être facilement débrayable de sorte à revenir à une situation stable.

Synthèse des contraintes

De cette analyse des besoins des acteurs, différentes contraintes qu'un système permettant la mise en œuvre de calcul parallèle doit respecter ont été dégagées :

1. Non intrusivité en terme d'accessibilité ;
2. Non intrusivité en terme de performances ;
3. Simplicité de développement des applications, sans empêcher les utilisations complexes ;
4. Simplicité du mécanisme d'exécution des applications ;
5. Surcharge d'administration limitée ;
6. Fonctionnement identique quelque soit les spécificités matérielles ;
7. Facilité de débrayage.

Classification des types de systèmes pour le Calcul Haute Performance

Après avoir opéré une étude des différentes solutions matérielles pour mettre en œuvre le parallélisme, le travail continue sur les outils d'aide à l'exécution. Deux classes sont définies pour les systèmes de calcul haute performance : les grilles de calcul et les clusters.

Contrainte	1	2	3	4	5	6	7
MPI	√						
BOINC	√	√				√	√
Corba	√	√				√	
Condor	√	√				√	√
OpenMosix	√	√	√	√			
Kerrighed	√	√	√	√			

TAB. 1.1 – Adéquation des solutions possibles aux contraintes

Grille de Calcul Une grille est un ensemble de machines, potentiellement très éloignées, à la fiabilité non garantie et inter-connectées avec un réseau non fiable, qui peuvent être dédiées, ou non, à cette utilisation. C'est par exemple le cas d'une solution comme BOINC exploitée au travers d'Internet.

Dans ce cadre, il existe des outils chargés de gérer la soumission par l'utilisateur des tâches à effectuer et ensuite de s'assurer d'une utilisation des ressources à sa disposition. Condor est un exemple d'outil de ce type.

Cluster Par cluster, on entend généralement des super-ordinateurs formés de machines homogènes, fiables et dédiées au calcul. Les nœuds qui forment ce super-ordinateur sont inter-connectés avec un réseau rapide et fiable.

Systèmes à Image Unique Ce sont des systèmes d'exploitation, dont l'objectif est de simuler une machine multiprocesseur virtuelle grâce à un réseau d'ordinateurs : quatre ordinateurs mono-processeurs dôtés chacun d'1 Go de mémoire vive seront vu de l'espace utilisateur comme une unique machine, quadri-processeurs, dotée de 4 Go de mémoire vive.

Ce type de système d'exploitation fait des hypothèses fortes sur la plateforme matérielle sur laquelle il est exécuté, notamment en terme de fiabilité des nœuds et de rapidité du réseau. Ainsi, une plateforme de type cluster est nécessaire. La section 1.1.4 s'attardera sur ce type de système en profondeur. Kerrighed et OpenMosix sont des exemples de ces systèmes.

Synthèse La conclusion de l'analyse de ces différentes solutions techniques est que seuls les systèmes à image unique permettent de vérifier les contraintes, comme le rappelle le tableau 1.1, extrait de la thèse [Pér08]. OpenMosix ayant été abandonné, la solution retenue est Kerrighed.

Justification de la Virtualisation

On remarque dans le tableau 1.1 que les trois dernières contraintes ne sont pas remplies par la solution retenue. Afin d'y parvenir, une autre proposition de cette étude est le recours à la virtualisation, avec VMware, puisqu'il s'agit de l'outil déjà déployé pour l'utilisation courante. Le recours à cette technique permet ainsi d'abstraire le matériel sur lequel tournera la machine virtuelle, et de conserver de bonnes performances. La machine virtuelle n'étant qu'un répertoire sur la machine physique, les outils de déploiement classiques peuvent toujours être utilisés. Enfin, il suffira de stopper les machines virtuelles qui s'exécuteront sur les ordinateurs des salles en cas de problème avec la plateforme de Calcul Haute Performance.

Synthétiquement, la pile logicielle complète est présentée dans la figure 1.1.1. Un schéma récapitulant l'architecture du système a été proposé dans [Dab08, Pér08], et est repris en figure 1.1.

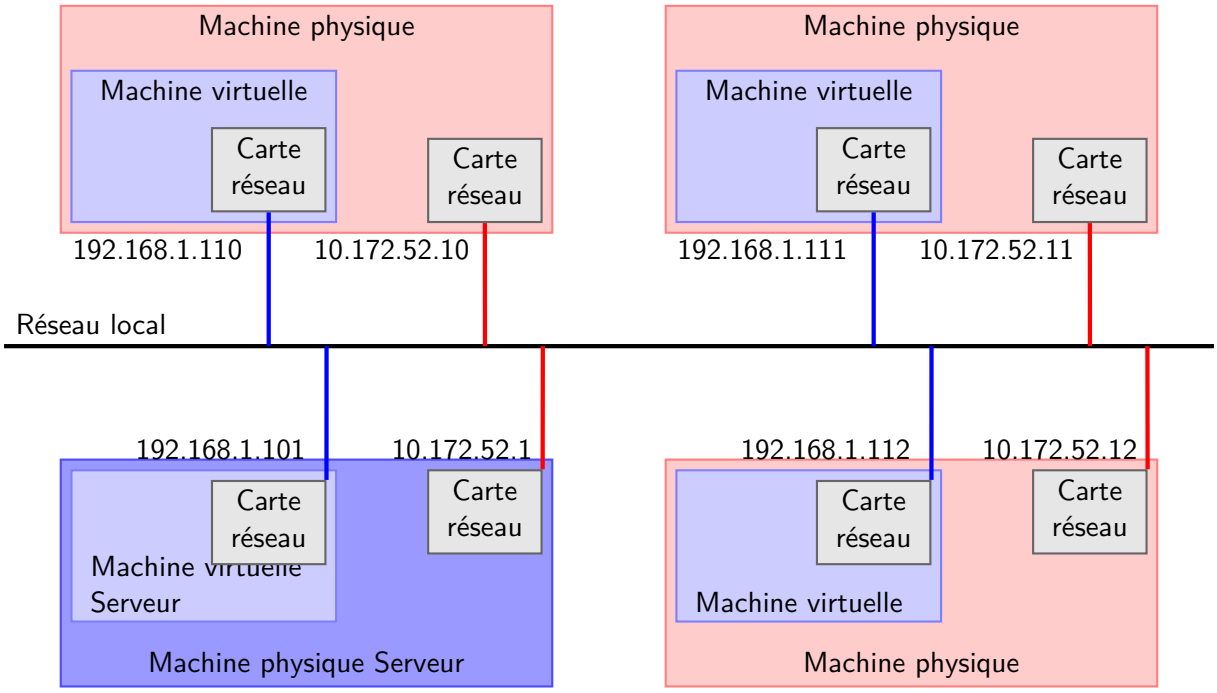


FIG. 1.1 – Architecture du système virtualisé proposé par [Dab08, Pér08]

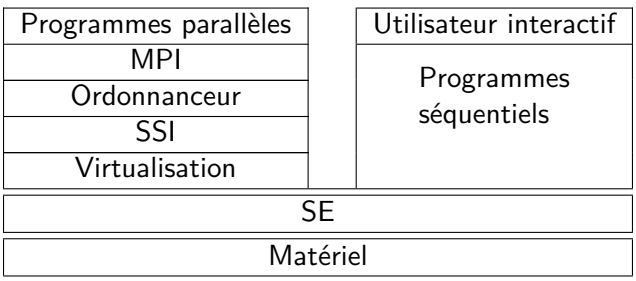


FIG. 1.2 – Pile logicielle retenue

Ordonnancement dans un Système à Image Unique

La suite de l'étude s'intéresse au problème de l'ordonnancement au sein d'un système à image unique tel que Kerrighed. L'objectif reste d'exploiter au mieux les ressources mises à disposition par tous les ordinateurs composant notre cluster : il est nécessaire d'équilibrer la charge au sein de ce dernier. Un algorithme d'équilibrage de charges est défini par un composant d'information et un composant de contrôles.

Il ressort également que l'élément clef de l'opération d'équilibrage de charges se situe dans la mesure de charge des nœuds. Pour répondre à ce problème, et considérer les contraintes de notre situation, une nouvelle mesure de charge est définie, découlant de la construction du problème LoadBalancing :

$$load_j = (1 + K \times state(m_j)) \frac{load^U_{m_j}}{speed(m_j)}$$

Dans cette mesure de charge, rappelons le rôle de certaines variables :

- $state(m_j)$ vaut 1 si la machine m_j est utilisée, i.e. qu'un utilisateur interactif est connecté dessus et 0 sinon ;
- K est un paramètre qui permet de pondérer la présence d'un utilisateur ;
- $load^U_{m_j}$ correspond au nombre de tâches s'exécutant sur la machine m_j ;
- $speed(m_j)$ représente la puissance de calcul de la machine m_j.

Notons au passage que l'utilisation conjointe de la virtualisation et de l'information de présence d'un utilisateur interactif implique une communication entre machine physique et machine virtuelle.

Composant d'Information Son rôle est de mettre à disposition de tous les nœuds du cluster des informations, principalement la charge de chacun.

Composant de Contrôle Son rôle est d'appliquer une politique d'ordonnancement, en décidant quel(s) processus migrer et à destination de quel(s) nœud(s). Il exploite les données mises à disposition par le composant d'information.

Algorithmes proposés

Après avoir posé le problème LoadBalancing, plusieurs algorithmes sont proposés pour le résoudre. Nous retiendrons particulièrement deux algorithmes, parce qu'ils sont les moins naïfs, RBT 2 et Rudolph 1.

Algorithme 1 Rudolph

si je suis la machine la plus chargée **alors**
  Soit i l'ordinateur avec lequel je peux échanger le plus de processus afin d'équilibrer nos charges
  Équilibrer ma charge avec i
end si

Algorithme 2 RBT

si je suis la machine la plus chargée **alors**
  Calculer la charge optimale en ayant relaxé la contrainte de mémoire
  Migrer autant de processus que possible vers les machines sous-chargées en essayant d'atteindre la charge optimale
end si

Expérimentation des algorithmes

Afin de voir laquelle des propositions d'algorithmes retenir, et même vérifier que proposer un algorithme plus complexe qu'une allocation au hasard est utile, des simulations ont été entreprises à l'aide de l'outil SimGrid. Cette simulation a été exécutée pendant vingt-quatre heures, à l'issue desquelles elle a été stoppée. Si un processus a eu le temps de s'exécuter avant coupure, il est considéré comme *Terminé*.

Les résultats ont montré plusieurs choses. D'une part, faire quelque chose pour réordonnancer les processus sur les nœuds du cluster a un intérêt. Ensuite, le faire mieux qu'au hasard a également un intérêt. L'aspect le plus intéressant est que les deux algorithmes proposés, RUDOLPH et RBT présentent des propriétés différentes.

En effet, on constate que RUDOLPH permet de lancer plus de processus, puisqu'il a un taux d'acceptation supérieur de 10 points à celui de RBT, à 88%. Par contre, l'ordonnancement proposé par RBT, puisqu'il accepte moins de tâches, permet de faire s'exécuter les tâches plus rapidement. Il en résulte que presque 61% des processus ont été en mesure de se terminer avec l'utilisation de RBT alors que RUDOLPH ne permet d'en accepter que 52%.

Il parait donc naturel de permettre d'utiliser plusieurs algorithmes d'ordonnancement au sein d'un même cluster, pour arriver à des objectifs différents.

1.1.2 Conclusions de la thèse de Matthieu Pérotin [Pér08]

À l'issue de cette thèse, plusieurs points importants ont été abordés. D'abord, l'étude réalisée a permis de dégager les contraintes fortes du projet, et elles-mêmes ont été à l'origine de la formulation et de la résolution du problème d'ordonnancement lié.

Ensuite, l'aspect technique n'a pas été négligé, et en considérant ces mêmes contraintes, la solution retenue impose l'utilisation d'un système à image unique, Kerrighed en particulier.

Enfin, notons que les expérimentations ont été menées avec un simulateur, et pas dans un véritable cluster.

Restait donc la question de savoir ce que donneraient ces résultats dans le contexte d'un système réel. Pour cela, il était d'abord nécessaire d'implémenter ces algorithmes au cœur d'un système à image unique existant : c'est l'objet du travail de fin d'études de Mathieu Dabert [Dab08].

Par ailleurs, ces expérimentations avec SimGrid ont mis en exergue le besoin de simplifier l'utilisation de cette bibliothèque. C'est l'origine de projet de fin d'études de Hamza Benarafa [Ben09].

Une autre question, qui a motivée la proposition du présent sujet, était de pouvoir être en mesure de savoir s'il est pertinent d'utiliser plusieurs algorithmes d'ordonnancement au sein d'un même cluster.

1.1.3 Projet de Fin d'Études de Mathieu Dabert [Dab08]

Comme cela a été précisé dans la section précédente, ce projet est né au cours du travail autour de la thèse [Pér08]. L'objectif de ce travail est de mettre en pratique ce qui a été pensé, à savoir à la fois fournir une API et une plate-forme pour être en mesure de tester, en pratique, les algorithmes d'ordonnancement.

Le rapport de ce projet couvre tous les aspects de la mise en œuvre de la plate-forme basée sur Kerrighed, et permet ainsi de mettre en place des machines virtuelles légères exécutant le système d'exploitation depuis un serveur de fichier situé sur le réseau.

Dans la suite, nous allons nous intéresser au cœur de ses travaux, à savoir la mise en place d'une API dans l'espace noyau, et les modifications nécessaires pour implémenter les algorithmes d'ordonnancement.

Mise en place d'une API en espace noyau

La partie qui nous intéresse concerne la définition d'une API. Le but de cette dernière est de permettre la remontée des informations qui sont nécessaires à l'implémentation des algorithmes RUDOLPH et RBT. Nous rappelons cette API dans le tableau 4.1, présent en page 48.

Implémentation des algorithmes dans Kerrighed 2.2.0

Lors du déroulement du projet de fin d'étude, la dernière version disponible de Kerrighed était la version 2.2.0. Dans cette version, le code gérant l'ordonnancement des processus au sein du cluster est concentré dans le fichier source `krg_scheduler.c`. De plus, il n'existe pas de moyen formalisé au niveau de Kerrighed de changer facilement cet ordonnanceur.

Une part importante de son travail a donc été l'identification des modifications à apporter de sorte à pouvoir implémenter les algorithmes. Le choix a été fait de fixer le type d'ordonnanceur a la compilation, sans qu'une justification précise ne soit énoncée. Cependant, on peut admettre que mettre en place le nécessaire pour opérer un changement d'ordonnanceur à chaud, dans un système qui n'est pas pensé dans cette optique, est quelque peu périlleux.

C'est donc dans le corps de la fonction `migration_manager_thread` que la constante `SCHEDULER_TYPE`, qui fixe l'ordonnanceur à exécuter, est évaluée à l'exécution.

Conclusion sur le Projet de Fin d'Études

La conclusion de cette implémentation ne porte pas mention des performances du système quant à la question de la répartition de la charge. Mais la description des modifications nécessaires pour être en mesure de tester plusieurs algorithmes suffit à montrer le manque de flexibilité de la version de Kerrighed utilisée : difficile, s'il faut recompiler la totalité du noyau du système, de pouvoir tester simplement des algorithmes d'ordonnancement.

La difficulté est d'autant plus accrue que le cadre opératoire n'est pas particulièrement restreint, il n'existe pas de framework dédié à l'ordonnancement au sein du système, et le développeur peut faire n'importe quoi, au sens premier. Sans oublier que tout le développement se fait en espace noyau, avec ses difficultés propres, tant en terme de debug, qu'en terme de lourdeur de développement : ce ne sont pas des modules noyau, il faut recompiler la totalité des sources à chaque modification.

Tous ces points ont amené à se poser la question de sortir de l'espace noyau, pour être en mesure de calculer et d'appliquer un ordonnancement depuis l'espace utilisateur. Idéalement, le mécanisme permettrait ainsi de pouvoir appliquer au sein d'un même cluster différentes politiques d'ordonnancement, de sorte à répondre à tous les besoins.

Un dernier aspect concerne la remontée d'information à propos de la présence d'un utilisateur interactif sur le nœud physique correspondant au nœud Kerrighed : cet aspect n'est pas pris en compte de manière pertinente dans le projet. Seule la connexion éventuelle d'utilisateurs au système virtualisé est considérée, en analysant la liste des processus : si un processus est en exécution, et que l'identifiant de son propriétaire est ≥ 1000, alors la machine est considérée comme étant utilisée par un utilisateur interactif. Cela permet néanmoins de valider le fonctionnement de la mesure de charge.

1.1.4 Systèmes à Image Unique

Les Systèmes à Image Unique sont une sous-famille des systèmes d'exploitation qui sont dédiés au calcul haute performance. L'idée générale qui sous-tend ces systèmes est : il s'agit de regrouper des ordinateurs « classiques » , au sens de machines de bureau que l'on trouve n'importe où par exemple, par opposition aux cluster construits à partir de matériel réalisé sur-mesure – par exemple BlueGene d'IBM ou Tera-10 de BULL –, en un seul gros élément.

Ainsi, le but à atteindre est de permettre à n'importe quel utilisateur du système d'exploiter au maximum les performances offertes par la combinaison de toutes les machines disponibles : accès à toute la mémoire de toutes les machines, accès à tous les processeurs, etc.

Plusieurs projets offrent la possibilité de mettre en place ce type de machines. Citons par exemple, et de manière non exhaustive :

- DragonFlyBSD [Dil] ;
- Kerrighed [Ker] ;

- LinuxPMI [Lon] ;
- MOSIX [BS] ;
- openMosix [Bar] ;
- OpenSSI [Wal] ;
- Plan 9 [Lab].

Le lecteur pourra se renseigner des fonctionnalités de chacun de ces projets sur leurs pages respectives. De même, une liste plus exhaustive est disponible sur la page Wikipedia correspondant aux Systèmes à Image Unique[1].

De manière rapide, ces systèmes fonctionnent en proposant plusieurs services, dont nous expliciterons succinctement l'objectif, parmi lesquels :

- Migration de processus et de threads ;
- Prise d'instantanées de processus (CheckPointing) ;
- Espace unique d'identifiants de processus au cluster ;
- Citons également la présence d'une racine du système de fichier unique, d'un espace d'entrées/sorties unique, et d'un espace de synchronisation unique.

Ainsi, le système dans sa totalité – le noyau, et les outils utilisateurs – sont en mesure d'agir sur tous les nœuds du cluster, et un processus peut être migré d'un nœud à un autre, ce qui permet aux utilisateurs – et surtout au système d'exploitation – d'utiliser au maximum la puissance de la machine multiprocesseur ainsi simulée.

Migration de processus et de threads

L'objectif de cette fonctionnalité mise à disposition par les systèmes à image unique est d'être en mesure d'exploiter correctement la totalité des processeurs au sein du système. Elle permet de déplacer un processus, ou un threads, d'un processeur sur un autre processeur. Ainsi, le système va décharger certains processeurs et en charger d'autres, de sorte à faire s'exécuter un maximum de tâches, et le plus rapidement possible.

Prise d'instantanées de processus

Cette fonctionnalité permet de réaliser une image d'un processus qui s'exécute actuellement sur un processeur. Par image, il est entendu qu'il s'agit de garder une trace de toutes les informations propres à ce processus, de sorte, par exemple, à être en mesure de pouvoir reprendre son exécution par la suite, en repartant du moment où il a été arrêté. La mémoire, l'état des registres, les descripteurs de fichiers, etc. peuvent donc faire partie de cette image.

Espace unique d'identifiants de processus

Afin de pouvoir mettre en commun les ressources de toutes les machines, le système à image unique offre une vue globale des identifiants des processus qui s'y exécutent, au lieu d'avoir des identifiants propres à une machine, ce qui pourrait engendrer des problèmes lors des migrations de processus. En pratique, Kerrighed utilise un mécanisme de masque qui dépend du nœud où a été lancé le processus. Ce masque s'ajoute – par une opération binaire – à l'identifiant de processus existant, et permet ainsi de donner un nouvel identifiant global, et unique sur tout le système à image unique.

1.2 Définition des besoins

Cette section se propose de répertorier les besoins qui ont été relevés lors de l'établissement du cahier des charges, et pour chacun, de rappeler brièvement l'objectif.

[1] http://en.wikipedia.org/wiki/Single-system_image (dernière consultation : 12 juin 2009)

1.2.1 Besoins fonctionnels

- Besoin 1 : Porter l'API développée dans le cadre du projet de fin d'étude précédent [Dab08] sur Kerrighed Trunk ;
- Besoin 2 : Porter les deux ordonnanceurs implémentés dans le cadre du projet de fin d'étude précédent [Dab08] sur la nouvelle architecture de Kerrighed Trunk ;
- Besoin 3 : Développer un module permettant une communication entre l'espace utilisateur et l'espace noyau, pour la notification de présence d'un utilisateur interactif. Ce module se limite à ouvrir une entrée `/proc` à définir, à laquelle on transmettra le nombre d'utilisateurs connectés ;
- Besoin 4 : Développer un démon Unix en espace utilisateur, écoutant l'interface réseau virtuelle du nœud de calcul, réalisant le travail de comptage du nombre d'utilisateurs connectés sur la machine physique, via un protocole de communication – à définir – lui transmettant l'information de connexion et de déconnexion d'un utilisateur. Ce démon notifiera par la suite le noyau via l'interface précédemment indiquée ;
- Besoin 5 : Développer un module PAM permettant la notification du système virtualisé de la connexion et de la déconnexion d'un utilisateur interactif ;
- Besoin 6 : Développer un module Winlogon permettant la notification du système virtualisé de la connexion et de la déconnexion d'un utilisateur interactif ;
- Besoin 7 : Implémenter une API noyau sous forme de module permettant la communication avec un service en espace utilisateur réalisant l'ordonnancement ;
- Besoin 8 : Spécifier et développer un service en espace utilisateur, dialoguant avec le noyau, et réalisant l'ordonnancement des processus suivant différents algorithmes implémentés sous forme de bibliothèque partagée.

1.2.2 Besoins non fonctionnels

- Besoin 9 : Identifier et documenter les changements entre Kerrighed 2.2.0 et la version Trunk avec SchedConfig ;
- Besoin 10 : Étudier la possibilité de faire fonctionner l'ordonnanceur en espace utilisateur ;
- Besoin 11 : Proposer et étudier de nouvelles heuristiques d'ordonnancement moins triviales que celles actuellement usitées.

1.2.3 Hiérarchisation des besoins

Nous reprenons ici le schéma synthétisant la hiérarchisation des différents besoins. Cette illustration est disponible en figure 1.3.

1.3 Présentation des besoins

Cette section vise à présenter chacun des besoins qui a été soulevé à la section précédente. Elle est directement reprise du cahier des charges.

1.3.1 Portage de l'API (Besoin 1)

Comme indiqué précédemment, il s'agit de transposer l'API qui a été implémentée au cours du Projet de Fin d'Étude [Dab08] sur la nouvelle infrastructure que propose la branche de développement officielle de Kerrighed : SchedConfig.

Cette API consistant uniquement en une remontée d'informations, il convient en pratique de développer de nouveaux modules de « probe » pour s'intégrer dans cette nouvelle manière de gérer l'ordonnancement.

FIG. 1.3 – Représentation de la hiérarchisation des besoins définie pendant l'élaboration de cahier des charges.

Après une première étude du fonctionnement de ces mécanismes, nous avons pu identifier les modules de récupération de données que nous devrons développer, et ceux déjà présents dans Kerrighed que nous aurons la possibilité de réutiliser :

- Obtention de la mémoire libre d'un nœud : besoin déjà comblé par le module `mem_probe` de Kerrighed ;
- Charge (au sens Unix) moyenne d'un nœud : besoin déjà comblé par le module `cpu_probe` de Kerrighed ;
- Consommation en Jiffies d'un processus : besoin déjà comblé par le module `mosix_probe` de Kerrighed ;
- Vitesse et nombre de processeurs d'un nœud : inexistant, création du module `cpuspeed_probe` ;
- Présence d'un ou plusieurs utilisateurs sur la machine hôte : inexistant à l'heure actuelle, création du module `user_probe` ;
- Taille d'un processus en nombre de pages : inexistant, création du module `processsize_probe` ;
- Charge d'un nœud en nombre de processus activables (état *runnable*) : inexistant, création du module `load_probe`.

1.3.2 Portage des ordonnanceurs (Besoin 2)

Dans la continuité de [Dab08] et du point précédent, nous allons également (re)développer sous forme de modules noyau les méthodes d'ordonnancement qui ont été évaluées précédemment, à savoir RBT et Rudolph.

Nous nous appuierons également sur l'infrastructure SchedConfig qui propose un système beaucoup plus souple et modulaire pour réaliser l'ordonnancement en espace noyau. Nous développerons donc deux modules, `rbt_policy` et `rudolph_policy`.

1.3.3 Signalisation de l'utilisation des machines (Besoins 3, 4, 5, 6)

Interface noyau (Besoin 3)

Il s'agit ici de développer un module en espace noyau, que nous nommerons `local_user_presence` , qui se contentera de définir un simple compteur, sous la forme d'un entier. Autour de ce compteur, on développera une petite API permettant de le manipuler, en exportant les symboles suivants dans le noyau, permettant ainsi d'incrémenter, décrémenter, etc.

Les symboles à exporter sont tout ceux nécessaires à la manipulation du compteur, à savoir :

- Connexion d'un utilisateur : `local_user_presence_user_connection` ;
- Déconnexion d'un utilisateur : `local_user_presence_user_disconnection` ;
- Nombre d'utilisateurs présents : `local_user_presence_user_connected` ;
- Le nœud est-il libre ? (`local_user_presence_node_free`) ;
- Le nœud est-il occupé ? (`local_user_presence_node_used`) ;

De plus, afin de permettre la communication des informations entre l'espace utilisateurs, nous implémenterons un autre petit module noyau, `local_user_notifier`, dont le rôle est de s'appuyer sur le précédent et d'offrir une interface dans le système de fichier `/proc`, à l'adresse `/proc/kerrighed/interactive_user`, et qui permette d'accéder à ces fonctionnalités du compteur.

Nous aurons ainsi une correspondance directe entre les entrées du répertoire cité précédemment et les fonctions exportées depuis le module de comptage :

- `local_user_presence_user_connection` ⇒ `/proc/kerrighed/interactive_user/connecion` ;
- `local_user_presence_user_disconnection` ⇒ `/proc/kerrighed/interactive_user/disconnection` ;
- `local_user_presence_user_connected` ⇒ `/proc/kerrighed/interactive_user/get` ;
- `local_user_presence_node_free` ⇒ `/proc/kerrighed/interactive_user/isfree` ;
- `local_user_presence_node_used` ⇒ `/proc/kerrighed/interactive_user/isused`.

1.3.4 Communication machine virtuelle (Besoin 4)

Nous développerons un simple démon, qui aura un rôle un peu étoffé. Tout d'abord, il réceptionnera les évènements des plugins PAM et WinLogon, et les relayera au noyau au moyen de l'interface décrite précédemment, et implémentée dans `/proc/kerrighed/interactive_user/`

Ce démon tournera dans la machine virtuelle, et écoutera un port utilisateur. Proposons le port 7919, qui semble libre d'après l'IANA[2].

Un protocole assez simple sera suffisant entre le démon et les modules. Proposons d'envoyer le caractère 'C' à la connexion d'un utilisateur, et le caractère 'D' à sa déconnexion.

Module machine réelle – Linux (Besoin 5)

Ce module PAM va avoir pour rôle d'intercepter les évènements de connexion et de déconnexion d'un utilisateur, et de les transmettre au démon unix précédemment présenté.

Il devra prendre en paramètre une chaîne permettant de décrire l'adresse IP du démon qu'il doit contacter, et ce de manière assez générique. On suppose en effet que les adresses des machines virtuelles sont « prédictibles » en ce sens où elles correspondront aux VLANs.

Par exemple, admettons que l'on ait un hôte en salle Unix B, ayant comme adresse IP *10.172.52.64*, la machine virtuelle qui sera hébergée sur cette machine aura comme adresse *192.168.52.64*.

Dès lors, on passera donc en paramètre le formatage de l'adresse IP cible, par exemple sous la forme suivante `ip=192.168.\$3.\$4`, en s'inspirant de ce que proposent les expressions régulières Perl pour la substitution de données.

Module machine réelle – Windows (Besoin 6)

Le besoin concernant le module pour WinLogon est symétrique.

1.3.5 Ordonnancement en espace utilisateur : nécessaire en espace noyau (Besoin 7)

Afin de prendre des décisions d'ordonnancement en espace utilisateur, il va être nécessaire de récupérer des informations depuis l'espace noyau. Les informations seront récupérées via le système des probes proposé par Kerrighed.

Il nous reste cependant à être en mesure de proposer à l'espace utilisateur d'accéder simplement à ces données. Une première approche consisterait à exploiter le système de fichier `/config`. Si cette idée est pertinente pour la récupération des informations de la machine locale, elle se heurte à une limite : impossible de récupérer les données à distance par ce biais.

Nous devons donc mettre en place un mécanisme qui permette à l'espace utilisateur de récupérer tout type de données proposé par toutes les probes. Idéalement, ces dernières ne devraient pas à être modifiées : le système permettant d'y accéder à distance depuis l'espace utilisateur doit être, autant que possible, transparent pour les modules de probe.

Par ailleurs, il serait très intéressant que ce système soit le plus flexible et le plus souple possible : découverte des probes disponibles « à chaud » par exemple.

1.3.6 Ordonnancement en espace utilisateur : mise en place (Besoin 8)

Pour calculer l'ordonnancement en lui-même, il nous faut un outil, un démon de préférence, qui aura à charge de prendre les décisions et d'effectuer lui-même les migrations, grâce à l'appel système `migrate` mis à disposition par Kerrighed.

Il sera nécessaire de poser les spécifications d'une bibliothèque ou d'une API permettant d'abstraire un minimum la récupération des données depuis les probes (masquer la complexité). Cette API aura en

[2] http://www.iana.org/assignments/port-numbers

charge non seulement la communication directe avec l'espace noyau, mais également le calcul de la fameuse Mesure de Charge.

Il faudrait également mettre en place un système de plugins, similaire à SchedConfig : chaque module d'ordonnancement sera un plugin du démon.

Les plugins n'auront d'autre but que de :

- Calculer le processus à déplacer ;
- Calculer le nœud où déplacer le processus.

1.3.7 Changements apportés depuis Kerrighed 2.2.0 (Besoin 9)

Il s'agit, non pas de fournir une documentation complète des changements opérés entre la version 2.2.0 de Kerrighed, sur laquelle étaient précédemment basés les travaux, et la version de développement incluant SchedConfig, mais plutôt d'expliciter :

- Les changements apportés par SchedConfig ;
- En quoi SchedConfig répond déjà en partie à nos besoins ;
- Présenter les éléments importants constitutifs de SchedConfig.

Ce besoin est nécessaire afin de pouvoir identifier le travail de portage qu'il sera nécessaire de réaliser.

1.3.8 Possibilité de réaliser l'ordonnancement en espace utilisateur (Besoin 10)

Il convient ici de s'intéresser à identifier les éléments qui sont nécessaires à permettre de réaliser les décisions d'ordonnancement en espace utilisateur. Il sera intéressant dans un premier temps de regarder ce que propose Kerrighed pour éventuellement déjà effectuer des migrations depuis l'espace utilisateur, et si ce n'est pas le cas, étudier et mettre en place le nécessaire pour le faire.

1.3.9 Proposition et étude de nouvelles méthodes d'ordonnancement (Besoin 11)

C'est, *in fine*, l'un des buts de ce projet. Actuellement, les algorithmes d'ordonnancement qui sont mis en place, que ce soit RBT ou RUDOLPH restent plutôt triviaux. C'est dû en partie aux contraintes inhérentes à l'espace noyau, où les facilités sont bien moindres.

Puisque nous sortons de l'espace noyau, nous pouvons nous permettre plus de latitude, c'est pourquoi nous devons proposer de nouveaux algorithmes d'ordonnancement, moins triviaux que les actuels.

Une fois l'ordonnancement en espace utilisateur fonctionnel, nous les testerons sur un cluster réel. En attendant que les outils soient prêts pour effectuer ce travail, les algorithmes seront évalués grâce à la plateforme SimGrid, développée par Hamza Benarafa au cours de son projet de fin d'études [Ben09].

1.4 Suite de l'étude

La présentation des travaux qui ont précédé à la réalisation de ce projet a permis de mettre en place les notions nécessaires, ainsi que le cadre dans lequel il s'inscrit. Nous avons également pu rappeler les raisons et origines d'une majorité des besoins qui ont été exprimés à la sous-section précédente, et qui sont détaillés dans le cahier des charges, disponible en annexe A.

Nous allons donc commencer par réaliser une étude visant à proposer de nouvelles méthodes d'ordonnancement que nous serions susceptible d'implémenter par la suite en espace utilisateur. Cette étude portera d'abord sur l'analyse des résultats de plusieurs projets de Programmation par Contrainte visant à modéliser et à résoudre le problème d'ordonnancement, en tenant compte des coûts de communication. Suivra un état de l'art au travers de quelques articles afin de trouver des pistes pour de nouvelles heuristiques. Dans un premier temps, nous proposerons un résumé des points principaux potentiellement intéressants dans ces articles. Puis, nous rappellerons nos contraintes et nous analyserons les hypothèses posées dans chaque cas,

afin de conclure sur la pertinence des solutions apportées par rapport à notre problème. En synthèse, nous proposerons une grille de lecture qui regroupera ces informations sous forme d'un tableau.

Dans un second temps, nous présenterons le contexte technique des travaux, en introduisant certaines notions nécessaires. Après avoir exposé ce qu'est Kerrighed, nous nous intéresserons aux composants qui nous seront utiles au sein du noyau Linux.

La suite de l'étude consistera à présenter la réponse apportée à chaque besoin soulevé dans le cahier des charges. Nous commencerons par présenter les changements opérés entre Kerrighed 2.2.0 et la version de développement actuelle. Ensuite, nous décrirons la manière dont a été portée l'API réalisée par Mathieu Dabert, puis nous nous attarderons sur l'implémentation des algorithmes d'ordonnancement, toujours en espace noyau, mais avec le nouveau mécanisme introduit pour Kerrighed 2.4. Une autre question avait été soulevée, celle de la notification de présence d'un utilisateur interactif sur une machine physique au nœud Kerrighed s'exécutant dans une machine virtuelle. L'approche choisie pour répondre à ce problème sera présentée avant que nous ne nous intéressions à la question d'opérer l'ordonnancement depuis l'espace utilisateur. Après avoir fait une étude sur l'étendue des modifications à opérer, nous commenterons celles qui ont été exécutées. Enfin, nous présenterons l'issue de réflexions portant sur la manière d'encadrer l'ordonnancement en espace utilisateur, en prenant en compte les modifications qui ont été opérées et ce qu'elles induisent.

La dernière partie s'attardera sur le suivi du projet. Nous commencerons par faire le point, besoin par besoin, sur leur état d'avancement, de sorte à identifier clairement ce qui est totalement réalisé de ce qui l'est partiellement, voire ce qui n'a pas pu être réalisé, ou ce qui reste à l'état de réflexion. Nous ferons ensuite le point sur les modifications apportées à Kerrighed et qui ont été reversées, ainsi que celles qui restent à faire remonter au projet. Nous nous attarderons ensuite à prendre un peu de recul par rapport au planning qui avait été prévu, et analyser le déroulement effectif.

Enfin, nous conclurons sur ce qu'a apporté ce projet à tous ses participants, et les évolutions que l'on peut en attendre à terme.

1.5 Continuité du projet dans le cadre du stage

Ce travail ne s'est pas arrêté une fois le Projet de Fin d'Études terminé. Dans le cadre du stage, ce projet a été continué, en parallèle avec un sujet sur le système de fichiers distribué XtreemFS.

Les principaux points de travail au cours de cette période ont été les suivants :

1. Débuggage du code : principalement, correction de fuites mémoires ;
2. Mise en place de l'infrastructure de monitoring web (Kerrighed Tracker Monitor) développée dans le cadre du projet d'Option Web ;
3. Mise en place et documentation d'une infrastructure de centralisation des données pour aider au debug et au suivi du cluster ;
4. Développement d'outils en espace utilisateur pour réaliser l'ordonnancement ;
5. Détection, mise en exergue et correction d'un bug critique dans la probe `cpu_probe`.

Le déroulement du projet nous empêche donc, à la date de la remise de ce rapport, de pouvoir répondre à la question initiale de l'intérêt de plusieurs politiques simultanées. Cependant, toutes les bases sont présentes pour le faire.

Afin de conclure sur cette partie, nous proposerons une conclusion séparée de la conclusion d'origine.

2. Nouvelles méthodes d'ordonnancement, en espace utilisateur

Le besoin 11, qui est exprimé en détails dans l'annexe C.8, stipule de proposer de nouvelles méthodes d'ordonnancement, que nous espérons plus pertinentes que celles actuellement utilisées.

L'idée principale est de pouvoir implémenter des mécanismes beaucoup plus complexes que ce qui se fait à l'heure actuelle en espace noyau. Non que ce soit impossible dans ce contexte, mais plutôt que grâce à la sortie en espace utilisateur, nous pouvons beaucoup plus facilement être en mesure de réaliser des mécanismes plus complexes.

2.1 Résolution du problème d'ordonnancement

Plusieurs projets antérieurs, qui ont été réalisés dans le cadre des mini projets de Gestion de Production Assistée par Ordinateur proposent des solutions au problème d'ordonnancement qui a été soulevé par les travaux de Matthieu Pérotin dans [Pér08].

Dans ce cadre, nous pouvons donc citer les rapports de Nicolas Chenné et Olivier Lesné [CL06], qui proposent une modélisation du problème sous forme de programmation par contrainte, tenant compte des communications réseaux.

Nous avons également à notre disposition les travaux réalisés par [GN07]. Ils reprennent le modèle précédent, mais incluent une minimisation des coûts de réseau, de sorte à essayer de limiter ces derniers.

Une première étape intéressante sera de se baser sur ces travaux pour réaliser un premier ordonnanceur. Les tests ayant été réalisés avec leurs implémentations (qui utilisent Choco) montrent des résultats intéressants : une solution optimale – en relaxant la contrainte mémoire – est trouvée, sans un temps de calcul trop long.

Autre voie intéressante à explorer, il s'agit de l'article qui décrit l'algorithme nommé Rudolph [RSAU91]. Celui-ci a déjà été implémenté dans Kerrighed 2.2.0 lors du projet de Mathieu Dabert [Dab08], et a montré des performances intéressantes. Un premier travail avec cet algorithme serait de l'implémenter en espace utilisateur, puis de l'améliorer graduellement.

2.2 Recherche de nouvelles heuristiques

Afin de mener une réflexion sur des heuristiques nouvelles, plusieurs articles ont été exploités :

- Approximation Algorithms for Multiprocessor Scheduling under Uncertainty [LR07] ;
- Scheduling and data redistribution strategies on star platforms [MRRV06] ;
- Bi-criteria Algorithm for Scheduling Jobs on Cluster Platforms [DEMT05] ;
- Libra : An Economy driven Job Scheduling System for Clusters [SAL+02] ;
- Simple and Effective Distributed Computing with a Scheduling Service [Mac01] ;
- Distributed and Multiprocessor Scheduling [Cha96] ;
- Randomized and adversarial load balancing [BFS99] ;
- The Load Rebalancing Problem [AMZ03].

2.2.1 Approximation Algorithms for Multiprocessor Scheduling under Uncertainty [LR07]

Cet article s'intéresse à l'ordonnancement de travaux dans une grille de calcul, à très grande échelle – typiquement Internet – telle que mise en place par des projets comme *SETI@HOME* ou *Folding@Home*. Plus particulièrement, la partie intéressante de ces travaux concerne l'aspect d'incertitude : il s'agit d'optimiser le temps d'exécution en tenant compte que, du fait de la nature même de la grille, certaines tâches pourraient ne pas se terminer – coupure réseau par exemple – comme prévu.

Les travaux sont décris par un graphe de dépendance qui permet de le découper en tâches à envoyer. Cependant, le cas de travaux indépendant est traité, et référencé comme problème SUU-I.

C'est ce cas particulier qui nous parait pertinent plus précisément dans le contexte de Kerrighed. Mais pas à court terme. Il s'agit de prévoir l'inclusion, dans quelques mois, du support de la perte de nœuds à chaud. Dans ces conditions, il parait assez pertinent de réaliser un ordonnancement qui en tienne compte, surtout si la solution retenue est déployée dans les salles de TPs sur l'École voire l'Université : il y aura nécessairement des pertes de nœuds assez régulières.

Les notions d'ordonnancement inconscient et de masse sont introduites, de sorte à définir le problème de maximisation de la somme des masses. Différents algorithmes sont ensuite proposés, pour résoudre ces problèmes et parvenir à une solution au problème SUU-I.

2.2.2 Scheduling and data redistribution strategies on star platforms [MRRV06]

Le sujet traité concerne la répartition de charge dans le cadre d'un système maître-esclave. Dans ce contexte, le maître découpe les travaux et les fait exécuter par les esclaves. Afin de s'adapter aux variations environnementales, il est nécessaire de redistribuer les tâches de sorte à améliorer la répartition du travail, pour qu'au final, il soit réalisé au plus tôt.

C'est ainsi qu'est introduit le problème d'ordonnancement pour des tâches maître-esclave sur une plateforme en étoile de processeurs hétérogènes – SPMSTSHP : Scheduling Problem for Master-Slave Tasks on a Star of Heterogeneous Processors.

Plusieurs algorithmes sont proposés. Le premier, Best Balance Algorithm est orienté pour les plateformes en étoile qui sont équipées de processeurs homogènes : à chaque itération, on regarde s'il est possible de finir plus tôt en déplaçant un travail. Si oui, elle est déplacée. Il est à noter que cet algorithme se place dans le cadre d'un réseau de communication symétrique et homogène.

Le second algorithme proposé est chargé de résoudre le problème dans le cas de processeurs hétérogènes, et est baptisé Moore based binary search algorithm, toujours dans le cadre d'un réseau de communication symétrique et homogène.

Par ailleurs, trois heuristiques sont proposées pour résoudre le problème dans le cas de processeurs hétérogènes. Les deux premières exploitent les algorithmes précédents, contrairement à la dernière proposition, Reversed Binary-Search Algorithm.

En dernière partie, les performances de ces algorithmes sont évaluées au moyen de la plateforme SimGrid. Ainsi, dans le cas de plateformes homogènes – communications réseaux et puissance de calcul –, BBA et MBBSA se comportent de manière identique et sont performants. R-BSA quant à lui est plus sensible aux caractéristiques de la plateforme.

Dans le cas de liens de communication homogènes, mais de puissance hétérogène, les résultats sont similaires au cas présent. Dès que les communications sont plus lentes que les calculs, R-BSA se montre plus performant.

Si la situation s'inverse, c'est-à-dire que les liens de communication deviennent hétérogènes et que la puissance de calcul devient homogène, l'algorithme BBA est le plus mauvais. À l'opposé, R-BSA se comporte bien, et surpasse en général MBBSA, sauf si les liens de communications sont rapides. Enfin, si les plateformes sont totalement hétérogènes, les résultats sont similaires.

2.2.3 Bi-criteria Algorithm for Scheduling Jobs on Cluster Platforms [DEMT05]

L'approche présentée dans cet article concerne l'ordonnancement de travaux sur une plateforme de cluster. L'idée principale consiste à ordonnancer les travaux, par lots, suivant leur date d'arrivée dans le cluster. Le but est d'obtenir un ordonnancement, optimisé sur deux critères : C_{max} et $\sum C_i$.

De cette manière, la méthode proposée permet de passer d'un algorithme hors ligne à garantie de performance de $\frac{3}{2} + \epsilon$ à un algorithme en ligne, dont la garantie de performance est de $3 + \epsilon$.

Ensuite, une borne inférieure pour le critère C_{max} est proposée, par approximation duale. Une relaxation du programme linéaire correspondant au problème permet de calculer une borne inférieure pour le critère $\sum C_i$.

2.2.4 Libra : An Economy driven Job Scheduling System for Clusters [SAL+02]

L'étude qui nous est proposée change un peu de point de vue par rapport à ce qui est habituellement formulé dans le contexte de l'ordonnancement de processus sur un cluster. En effet, au lieu de chercher à maximiser la performance processeur, ce système propose une autre approche, inspirée de l'économie, pour déterminer l'ordonnancement à exécuter, pour favoriser l'utilité du point de vue utilisateur en se basant sur son *budget* et la *date de fin* du travail.

Le coût d'un travail est calculé par la formule :

$$C = \alpha \times E + \beta \times \frac{E}{D}$$

où

- α, β sont des coefficients qui ajustent respectivement la tarification des travaux, et l'incitation à relaxer la date de fin ;
- E correspond aux nombres d'heures programmées pour la tâche ;
- D correspond à la date de fin demandée.

Ainsi, le premier terme correspond au coût en heures de calcul qui sera facturé à l'utilisateur, et le second permet de regarder la sympathie de ce dernier à l'égard du cluster. Il sera sympathique s'il relâche les contraintes sur la date de fin, par exemple.

2.2.5 Simple and Effective Distributed Computing with a Scheduling Service [Mac01]

L'étude proposée concoure à la réalisation d'un cluster de calcul, dédié à l'étude d'un problème précis – l'optimisation d'éléments d'optique diffractifs, grâce à une méthode de calcul basée sur les différences finies – qui requiert une grosse puissance de calcul, mais des besoins mesurés tant en consommation mémoire qu'en utilisation d'espace de stockage. Après études des différentes manières de paralléliser le problème, il est retenu de calculer une optimisation par la méthode des différences finies sur chaque machine à disposition. Il est en effet plus simple de procéder ainsi dans leur contexte.

La méthode de répartition des calculs est plutôt simpliste, et exploite le programmateur de tâches de Windows – l'environnement est une contrainte du problème. Un script accède à un partage réseau, et si le fichier `go.bat` est présent dans ce dernier, alors le calcul d'optimisation est lancé sur le nœud local.

Les résultats en terme d'amélioration de performances sont bons, puisque grâce au travail de séparation des tâches, et par l'aspect peu coûteux en communication des calculs, ces derniers se répartissent uniformément sur chaque machine. L'ajout de nœuds, dans les limites de leur utilisation, engendre une diminution directe du temps de calcul global. La conclusion naturelle est donc que leur méthode est performante, au regard de leurs besoins.

2.2.6 Distributed and Multiprocessor Scheduling [Cha96]

Le présent article propose une vue générale des problèmes liés à l'ordonnancement dans les systèmes destinés aux calculs intensifs. Après une présentation des différentes familles d'algorithmes dédiés à l'ordon-

nancement sur les systèmes parallèles et distribués, les bonnes pratiques de l'ordonnancement multiprocesseur sont présentées. Après une présentation des techniques propres aux systèmes parallèles, et aux mécanismes de passage de messages, l'étude porte sur les algorithmes distribués.

Dans la section abordant les algorithmes distribués, nous trouvons notamment un tableau résumant l'état de l'art en la matière, avec la liste des articles et les spécificités des algorithmes proposés : distribués, hétérogènes, capacité à gérer la montée en charge, coûts induits.

L'ensemble de méthodes qui nous intéressera le plus concerne les algorithmes distribués, coopératifs et suboptimaux.

Heuristiques de Blake Il s'agit de quatre algorithmes : Non-Scheduling (NS), Random Scheduling (RS), Arrival Balanced Scheduling (ABS) et End Balanced Scheduling (EBS). Seul le dernier est dynamique, et est à l'initiative des nœuds libres, et non chargés.

Méthodes de Ni et Abini Deux méthodes de répartition de charge pour des systèmes connectés localement sont proposées par Ni et Abini : Least Expected Delay et Shortest Queue. La première assigne la tâche à la machine qui sera libre le plus tôt, alors que la seconde méthode privilégie une machine dont le nombre de travaux en attente est le plus faible. Notons que l'information entre les nœuds se fait par broadcast, et que l'auteur signale les soucis de montée en charge qui y sont liés.

Proposition de Stankovic Parmi les deux méthodes proposées par Stankovic, l'une d'elle est un mécanisme dynamique d'automate à apprentissage stochastique. Elle exploite le principe de récompenses et de pénalités comme système de retour d'information. Là encore, l'auteur fait mention de communications exploitant la diffusion, ce qui pourrait poser des soucis de montée en charge.

2.2.7 Randomized and adversarial load balancing [BFS99]

La démarche suivie dans cet article est de s'intéresser à rétablir une situation de déséquilibre de charge manifeste dans un cluster. La question de la génération d'instances de processus favorisant un déséquilibre est également traitée. L'algorithme d'ordonnancement permet de rééquilibrer la charge en $poly(n)$ étapes.

Nous nous concentrons sur le cas du système à charge non bornée, qui correspond le plus à nos problèmes. Il est présenté dans la section 4 de l'article. L'algorithme de répartition proposé se découpe en quatre phases :

1. Estimation de la charge ;
2. Classification des nœuds chargés et non chargés ;
3. Assignation : mise en relation d'un nœud chargé avec un nœud libre ;
4. Transfert effectif des processus et de la charge liée.

2.2.8 The Load Rebalancing Problem [AMZ03]

Cet article, dont la lecture a été particulièrement recommandée, s'intéresse au problème de répartition de charge, ayant pour finalité l'optimisation – réduction – de la date de fin des tâches, sous contrainte de ne pas déplacer plus d'un nombre précis de travaux.

Dans cette optique, ils proposent un algorithme complexe –il est donc intéressant d'être à présent en mesure de travailler totalement en espace utilisateur – qui permet une 1.5-approximation par rapport à l'optimal. Et ceci avec une complexité temporelle de $O(nlog(n))$. Cet algorithme a été baptisé PARTITION et est proposé en page 31, en algorithme 3.

Ils proposent ensuite un autre algorithme, M-PARTITION, qui donne une $1.5 - approximation$ de la valeur d'OPT en $O(nlog(n))$. Ce second algorithme a la propriété de respecter au plus k déplacements de travaux.

Dans ce qui suit, les travaux de taille strictement supérieur à $\frac{1}{2}OPT$ sont considérés comme grands. Et tous les autres seront donc considérés comme des petits travaux. De plus, OPT correspond à la valeur optimale du C_{max}.

Enfin, un schéma d'approximation utilisant la programmation dynamique est proposé. Ceci nous donne une piste intéressante à explorer par la suite, pour implémenter une répartition de charge plus performante.

Ajoutons que L_T correspond au nombre total de grands travaux, m_L est le nombre de processeurs avec au moins un grand travail. Ainsi, $L_E = L_T - m_L$ donne le nombre de grands travaux en trop sur cet ensemble de processeurs.

Algorithme 3 PARTITION

1. Pour chacun des m_L processeurs ayant une tâche importante, retirer toutes les tâches importantes, sauf pour le plus petit des gros travaux.
2. Pour chaque processeur i, calculer les valeurs suivantes :
   - a_i : Le nombre minimal de petits travaux à retirer de sorte que la taille totale des travaux restant soit au plus $\frac{1}{2}OPT$.
   - b_i : Le nombre minimal de travaux (en considérant également les grands) à retirer de sorte que la taille totale des travaux restant soit au plus OPT.
   - $c_i = a_i - b_i$.
3. Choisir les processeurs L_T de plus petits c_i, en donnant priorité à ceux contenant de gros travaux. Retirer a_i *petits* travaux de ces processeurs, de sorte que la taille totale des petits travaux restant soit au plus de $\frac{1}{2}OPT$.
4. Sur les $m - L_T$ processeurs restant, retirer b_i travaux. S'il existe de gros travaux sur ces processeurs, ils seront retirés et réassignés, arbitraitement, à des processeurs sans gros travaux qui ont été créés à l'étape 3.
5. Assigner arbitrairement les gros travaux retirés à l'étape 1 aux autres processeurs sans gros travaux.
6. Pour les petits travaux retirés aux étapes 3 et 4, il convient de les assigner un par un aux processeurs de plus petite charge.

Algorithme 4 M-PARTITION

1. Utiliser la charge moyenne comme valeur de départ devinée pour OPT.
2. Calculer les valeurs de L_T, L_E, a_i, b_i et c_i en utilisant PARTITION3. Soit $\hat{k}$ le nombre total de mouvements nécessaires par cet algorithme.
3. **tant que** $\hat{k} > k$ **faire**
   - Augmenter la valeur devinée de OPT à la prochaine plus haute valeur.
   - Recalculer les valeurs de L_T, L_E, a_i, b_i et c_i en utilisant PARTITION3.
4. Retourner comme résultat la sortie de la dernière exécution de PARTITION3.

2.3 Recadrage des articles par rapport à notre problème

Nous allons maintenant proposer une « grille de lecture »des articles qui ont été résumés précédemment. L'idée générale est de rappeler pour chacun les hypothèses posées dans le cadre de l'article, et de les confronter aux nôtres. Ceci afin d'être en mesure de jauger la pertinence des solutions proposées dans ces derniers, relativement à nos problèmes.

2.3.1 Rappel de nos hypothèses

Avant de procéder à l'examen des hypothèses qui président dans les articles, il convient de rappeler celles qui sont liées à notre problème :

1. L'ordonnancement est distribué, chaque nœud effectue une décision locale ;
2. Les durées opératoires et la consommation mémoire des tâches, les contraintes de précédences entre les tâches sont acquises en-ligne ;
3. Nous n'avons pas d'information sur les dépendances entre les processus ;
4. Nous n'avons pas de date de fin prévue ;
5. L'information de durée d'exécution d'un processus est obtenue en-ligne ;
6. Les dates d'apparitions des processus sont acquises en-ligne ;
7. Les nœuds du système peuvent évoluer dynamiquement – ajout, retrait ;

2.3.2 Approximation Algorithms for Multiprocessor Scheduling under Uncertainty [LR07]

Dans cet article, nous pouvons relever les hypothèses suivantes, qui sont en contradiction avec les nôtres :

- C'est un algorithme pour grille ;
- Les processus sont munis d'un graphe de dépendances entre tâches ;
- Connaissance complète des travaux – durée d'exécution, date de disponibilité – non spécifiée ;
- Probabilité de réussite d'un travail sur un nœud ;

Les deux premiers points sont particulièrement critiques : ce n'est pas un algorithme distribué, et nous n'avons pas possibilité d'obtenir un graphe de dépendances entre les processus.

Par contre, il est intéressant de noter que la partie importante du travail consiste à définir une métrique, la masse, à partir des probabilités d'exécution réussie d'un travail sur une machine. Le but est ensuite de maximiser la masse. C'est ainsi qu'est traité l'aspect incertitudes du problème.

Il n'est par ailleurs nullement fait mention de connaissances nécessaires quant à la date de disponibilité et la durée d'exécution d'un processus. Assez peu d'hypothèses semblent en contradiction avec les nôtres, hormis l'aspect distribué.

2.3.3 Scheduling and data redistribution strategies on star platforms [MRRV06]

Les hypothèses suivantes ont été notées à la lecture de l'article, et nous nous proposons d'analyser leur pertinence par rapport aux nôtres :

- Mécanisme de distribution des données Maître/Esclave ;
- Connaissance complète des travaux (algorithme hors-ligne) ;
- Prise en compte d'une puissance de calcul hétérogène ou homogène ;
- Considération d'un réseau de communication de performance hétérogène ou homogène ;

Deux points particulièrement intéressants vis-à-vis de nos besoins sont les derniers. Nous sommes en effet dans le cadre d'un réseau à la fois homogène et hétérogène – considérons homogène au sein d'une salle de TP, et hétérogène au sein d'un bâtiment, puisqu'il y a changement de VLAN – et que, les puissances de calculs sont de la même classe. L'étude proposée à la fin de l'article en tenant compte de ces paramètres est donc pertinente.

Les algorithmes Best Balance Algorithm et Moore exploitent les dates de fin des travaux. Hors, il s'agit d'un paramètre que nous acquérons en-ligne dans notre cas. Il est donc difficile de les intégrer à notre démarche.

2.3.4 Bi-criteria Algorithm for Scheduling Jobs on Cluster Platforms [DEMT05]

Attardons nous sur les hypothèses qui sont nécessaires pour les travaux présentés dans cet article.

- Mécanisme de soumission sur le cluster par batch, sur un nœud précis ;
- Les nœuds constituant sont homogènes en puissance ;
- Mécanisme d'ordonnancement global, et hors-ligne ;
- Optimisation des critères C_{max}, $\sum C_i$;

Nous pouvons constater que toutes ces hypothèses contredisent largement les nôtres. Dès lors, les résultats présentés ne peuvent être considérés comme pertinents pour notre problème. De plus, l'algorithme proposé a besoin des dates de disponibilité des travaux à ordonnancer.

2.3.5 Libra : An Economy driven Job Scheduling System for Clusters [SAL+02]

Cet article s'impose les contraintes suivantes :

- Pensé pour s'intégrer dans un système de soumission de type « Batch » : l'utilisateur n'a qu'un seul point d'entrée pour envoyer des travaux sur le cluster ;
- Les nœuds du cluster sont dédiés uniquement à exécuter les travaux soumis ;
- Le système d'exploitation se doit d'accepter un paramètre qui correspond au pourcentage de temps CPU qui doit être alloué au processus, et doit être en mesure de s'assurer que cette valeur est respectée (tant que la somme de toutes les parts n'excède pas 100) ;
- Le temps d'exécution estimé, i.e. le paramètre E, est donné par l'utilisateur, et correspond à l'exécution d'un processus seul sur n'importe quel nœud du cluster ;
- Tous les nœuds du cluster sont homogènes. Si tel n'était pas le cas, alors E doit s'adapter pour refléter les capacités du nœud sur lequel le processus s'exécute ;
- Exploitation d'une métaphore économique : la loi de l'offre et de la demande ;

Alors que l'idée d'attribuer l'ordonnancement en utilisant une métaphore de l'offre et de la demande, à la lecture des hypothèses, il convient de constater que ces idées ne sont pas exploitables pour notre problème. Notamment, la nécessité d'estimer un temps d'exécution sur un nœud « type », la nécessité que le processus annonce au système d'exploitation la quantité de temps processeur nécessaire. De plus, le système a été pensé pour s'intégrer dans un mécanisme de soumission de tâches sur un cluster, plus précisément, l'implémentation a été réalisée avec PBS. Ceci viole donc également l'une de nos contraintes fortes.

2.3.6 Simple and Effective Distributed Computing with a Scheduling Service [Mac01]

Voici un article dont le titre était particulièrement intéressant. Ainsi que le résumé proposé. Mais après une lecture attentive, les différentes hypothèses et points clefs ont été relevés :

- Le système ne nécessite pas de modification sur les applications ;
- Fonctionne sous Windows 4.0, NT 4.0, NT 5.0 ;
- L'ordonnancement est en-ligne et distribué ;
- Faible intrusivité du mécanisme sur les machines utilisées ;
- Batch, Windows Tasks Scheduler ;
- Gros calculs, à l'initiative des nœuds ;
- Temps de communications négligeables ;

Force est de constater que le mécanisme de fonctionnement de ce système d'ordonnancement, s'il reste intéressant à être présenté, et surtout de savoir que cela fonctionne, n'est pas exactement ce que l'on était en droit d'attendre en lisant titre et résumé. Il n'y a aucun système d'ordonnancement distribué au sens où nous l'entendons.

2.3.7 Distributed and Multiprocessor Scheduling [Cha96]

Cet article, qui comme nous l'avons vu opère principalement un état de l'art de l'ordonnancement sur tous les systèmes parallèles, nous propose certains algorithmes, dont les hypothèses sont les suivantes :

- Ce sont des algorithmes distribués, et en-ligne ;
- Ils sont coopératifs – i.e. les décisions locales sont prises avec un but commun ;
- Les algorithmes proposés peuvent être à l'initiative des nœuds libres ;
- Certains transmettent les informations en broadcast ;

Ce sont les heuristiques de Blake, les méthodes de Ni et Abini et la proposition de Stankovic qui ont déjà été présentés dans la sous-section précédente. Ce sont les pistes qui nous semblent les plus prometteuses, étant donné que ces algorithmes ont déjà été répertoriés comme correspondant à nos critères.

Notons que les algorithmes proposés par Ni et Abini, et celui proposé par Stankovic exploitent une diffusion totale de l'information, par broadcast. Ceci pourrait être un frein notable à leur utilisation, puisque l'auteur note que cette méthode empêche une montée en charge correcte.

2.3.8 Randomized and adversarial load balancing [BFS99]

Analysons les hypothèses nécessaires qui sont exploitées dans cet article :

- L'ordonnancement est distribué sur tous les nœuds ;
- Mécanisme de rééquilibrage de charge ;
- Activation à partir d'une limite de charge ;
- Conservation au maximum des processus lancés sur une même machine ;
- Cas de la charge non bornée ;
- Méthode d'estimation de la charge du *système* ;
- À l'initiative des nœuds chargés ;

La partie de l'article qui nous intéresse concerne le cas de système à charge non bornée. En effet, dans ce contexte, toutes les hypothèses qui sont posées sont compatibles avec notre cas. Par contre, le mécanisme qui consiste à ce que les nœuds qui sont considérés comme chargés se trouvent un partenaire qui est considéré comme chargé est plus problématique. Cela nécessite un système où les nœuds capables de valider qu'ils sont appairés avec un autre.

Si ce dernier point est géré sur le plan technique, alors cette méthode devient intéressante. Il conviendra cependant de valider la pertinence et la qualité de la solution, puisque l'article n'aborde presque pas ce point, se contentant d'un laconique « You may or may not believe it, but simulations show that both algorithms perform pretty well ». Notons d'ailleurs que l'algorithme RUDOLPH dont il est fait mention dans [Pér08] est fortement inspiré de cet article.

2.3.9 The Load Rebalancing Problem [AMZ03]

Observons et commentons les hypothèses qui sont posées pour la résolution de ce problème :

- La taille des travaux est nécessaire pour opérer la classification grands/petits ;
- Optimisation de la date de fin, C_{max} ;

De par la présentation qui est faite du problème, dans l'introduction, ainsi que sa formalisation, et les hypothèses relevées précédemment, il est clair que les algorithmes proposés ne sont pas des algorithmes en-ligne. Ceci pose problème, puisque parmi nos hypothèses, nous sommes contraints d'acquérir certaines informations uniquement en-ligne. Or, la classification des travaux entre un ensemble de petits et de grands travaux nécessite de connaître à la fois la durée d'exécution totale de ces derniers, et la valeur optimale de la solution au problème formulé, notée OPT.

Cependant, la lecture de l'algorithme 4, et le lemme 5 de l'article, nous proposent de nous passer de connaître la valeur optimale. Si nous considérons les dates d'exécutions jusqu'au moment où l'algorithme est exécuté, alors cette solution est pertinente.

2.4 Grille de lecture synthétique

Nous proposons maintenant une « grille de lecture » des articles précédents dans le tableau 2.1. Son rôle est de synthétiser les remarques qui ont été faites dans le cadre de la section précédente. Les contraintes seront représentées par leur numéro.

Lorsqu'un article est explicitement compatible avec notre contrainte, nous l'indiquerons avec ☑. Si ce n'est pas mentionné, nous utiliserons □. Enfin, si la contrainte est violée, alors nous placerons le symbole ☒.

1. L'ordonnancement est distribué, chaque nœud effectue une décision locale ;
2. Les durées opératoires et la consommation mémoire des tâches, les contraintes de précédences entre les tâches sont acquises en-ligne ;
3. Les dates d'apparitions des processus sont acquises en-ligne ;
4. Les nœuds du système peuvent évoluer dynamiquement – ajout, retrait ;

	Contraintes			
Articles	1	2	3	4
Approximation Algorithms for Multiprocessor Scheduling under Uncertainty	☒	☒	☒	□
Scheduling and data redistribution strategies on star platforms	☒	☒	☒	□
Bi-criteria Algorithm for Scheduling Jobs on Cluster Platforms	☒	☒	☒	□
Libra : An Economy driven Job Scheduling System for clusters	☒	□	□	□
Simple and Effective Distributed Computing with a Scheduling Service	☒	□	□	☑
Distributed and Multiprocessor Scheduling	☑	☑	☑	□
Randomized and adversarial load balancing	☑	☑	☑	☑
The Load Rebalancing Problem	□	☒	□	□

TAB. 2.1 – Grille de lecture synthétisant les propriétés de chacun des articles étudiés.

3. Présentation du contexte technique

Nous allons commencer, dans un premier temps, par introduire les éléments techniques qui seront nécessaires à la bonne compréhension des besoins implémentés par la suite. Nous présenterons un peu plus en détails le projet Kerrighed, sur lequel nous nous sommes appuyés. Enfin, nous terminerons par de brefs rappels sur les fonctionnalités exploitées dans le noyau Linux, afin de développer ce dont nous avons eu besoin. Ces rappels n'ont bien sûr pas vocation à être exhaustifs, et il est recommandé de se reporter aux sources bibliographiques en rapport pour plus d'informations.

3.1 Kerrighed

Ce projet, démarré en 1999 au sein de l'INRIA, vise à fournir un ensemble de patchs pour le noyau Linux, permettant la mise en place d'un système à image unique au sens précisé dans la section 1.1.4, et fournissant un maximum de fonctionnalités.

Ainsi, le projet Kerrighed propose les fonctionnalités énumérées précédemment, à savoir :

- La capacité à migrer des processus et des threads ;
- La prise d'instantanées d'un processus, via le mécanisme de CheckPointing ;
- La vision globale du cluster comme une grosse machine SMP ;
- La gestion des identifiants de processus est globale à tous le cluster ;
- La gestion globale des entrées/sorties et des synchronisations ;
- Et le dernier point, à savoir une racine unique, n'est pas propre à Kerrighed (puisqu'il utilise simplement un démarrage réseau avec partage NFS).

À titre informatif, le précédent projet de fin d'étude, réalisé par Mathieu Dabert [Dab08], ainsi que le travail de thèse de Matthieu Pérotin [Pér08] exploitaient la version 2.2.0 de Kerrighed, qui était à l'époque la dernière version disponible. Depuis, de nombreuses avancées dans le projet ont été réalisées, et justifient une bonne partie de nos besoins préliminaires.

À l'heure actuelle, la dernière version stable est la 2.3.0, mais certains besoins et composants précis, sur lesquels nous reviendront dans la section 4.1 nous ont imposé de travailler sur la version dite « trunk », celle en plein développement. Il est à noter que, malgré le nom qui pourrait laisser à penser que le système résultant est instable, en réalité il s'avère que les processus de validation mis en place par le projet font que les sources disponibles dans cet arbre de développement se sont toujours révélées fonctionnelles.

Par ailleurs, Kerrighed est actuellement fait pour s'appliquer sur la version 2.6.20 du noyau Linux, et un travail de portage est en cours afin de le mettre à jour pour pouvoir être utilisé avec les dernières versions. Le noyau 2.6.29 est celui sur lequel ce travail s'effectue en ce moment. Maintenir un tel patch est un travail important, et c'est d'ailleurs le passage de Linux 2.4 à Linux 2.6 qui a participé à la mise en berne du projet openMosix, sur lequel se faisait précédemment les travaux de thèse de Matthieu Pérotin.

3.2 Linux

Il n'est pas nécessaire de présenter Linux, cependant, quelques notions propres au développement en espace noyau avec celui-ci vont être introduites. Il s'agit notamment de la réalisation de modules noyau, ainsi que d'une brève explication des systèmes de fichiers virtuels que sont ProcFS [BBN] et ConfigFS [Bec]. Nous présenterons aussi le framework de test Linux Test Project [SI].

3.2.1 Modules Noyau

Nous n'allons pas expliquer dans les détails comment réaliser et charger un module noyau, mais simplement de rappeler quelques éléments de base afin que le lecteur puisse approfondir si besoin, et dans tous les cas aient un certain nombre de repères. Une bonne source est l'ouvrage Linux Device Drivers [CRKH05a], considéré comme une référence de l'introduction au développement de modules noyau. Plus particulièrement, les bases sont détaillées dans [CRKH05b] et [CRKH05c].

Une autre approche plus détaillée de la création de modules noyau est par ailleurs disponible, sous forme d'un exemple concret avec l'écriture d'un pilote de périphérique USB, dans le rapport de mini-projet Unix [FL08]. C'est une source complémentaire à [CRKH05a], qui s'appuie d'ailleurs sur ce dernier.

Qu'est-ce qu'un module noyau Linux ? C'est une portion de code exécutable, qui peut-être chargée – et déchargée – à chaud, dans l'espace d'adressage mémoire noyau. C'est donc une partie intégrante du noyau, et en ce sens, il a accès à toutes[1] les fonctionnalités de ce dernier, et tous les symboles exportés.

Un symbole exporté est une fonction, ou une variable, pour laquelle on précise explicitement, via l'utilisation de la macro `EXPORT_SYMBOL` – ou `EXPORT_SYMBOL_GPL` si l'on souhaite restreindre la portée aux modules libres – qu'il est accessible depuis un autre module noyau. Sans cet export, il est impossible pour un module noyau d'utiliser une variable ou une fonction définie dans un autre. L'utilisation de symboles exportés par un autre module va introduire la notion de dépendance entre deux modules. Soit un module A, qui exporte le symbole `fonctionA()`. Si l'on écrit un module B qui fait référence à la fonction `fonctionA()`, alors le module B dépend du module A, et ce dernier doit impérativement être chargé avant notre module B. L'analyse et le calcul des dépendances entre modules se fait grâce à l'outil `depmod`.

Concrètement, un module noyau est constitué d'un ou plusieurs fichiers source C, compilés – et liés ensembles dans le cas où il a été décidé de décomposer le code source en plusieurs fichiers – en un fichier objet un peu spécial. Contrairement aux autres fichiers objets compilés qui sont nommés `.o`, ceux-ci sont nommés `.ko` – comme « Kernel Object » –, et surtout le code contenu est relocalisable. C'est-à-dire, que les adresses mémoires contenues dans ce code objet peuvent être déplacées, à la différence de celles d'un module compilé statiquement dans le noyau. La compilation du module repose sur l'infrastructure de compilation du noyau, et est explicitée dans [CRKH05c, FL08]. Une fois ce module compilé, il reste à le charger dans le noyau. Ceci est fait au moyen soit de la commande `insmod` – qui se limite à charger directement le module objet dans l'espace mémoire – soit avec la commande `modprobe` qui est en mesure de résoudre les dépendances entre modules. Ce dernier outil nécessite cependant que le module ait été installé, dans le répertoire `/lib/modules/‘uname -r‘/` grâce à l'utilisation de `make install` après avoir été compilé. Lors de l'installation, l'infrastructure de compilation calcule les dépendances entre modules grâce à l'utilitaire `depmod`, c'est le résultat de ce calcul qui sera réutilisé par la suite lorsque `modprobe` sera appelé.

Pour implémenter ce module, les sources n'ont finalement besoin que de peu de choses, dans le cas d'un pilote ne travaillant avec aucun sous-système particulier – comme le bus PCI, USB, etc. – puisque cela se limite à :

- Inclure des entêtes particulières :
  - `<linux/kernel.h>` ;
  - `<linux/module.h>` ;
- Définir quelques informations sur le module, via des macros explicites :
  - `MODULE_LICENSE` ;
  - `MODULE_AUTHOR` ;
  - `MODULE_DESCRIPTION` ;
- Et enfin, définir deux points d'entrées :
  - `module_init(fonction_init)`
  - `module_exit(fonction_exit)`

  où `fonction_init` et `fonction_exit` sont des pointeurs sur des fonctions de type respectifs :
  - `void __init fonction(void)`

[1] Sauf cas particuliers liés aux licences. Mais dans le cas d'un module sous licence GNU GPL, aucune restriction.

- `void __exit fonction(void)`

La fonction pointée par le paramètre qui est passé à la fonction `module_init` sera exécutée après avoir chargé le module. Pour le cas de la fonction `module_exit`, la fonction qui est pointée par le paramètre sera appelée avant de lancer le déchargement du module.

3.2.2 ProcFS

Le noyau Linux propose ce système de fichier virtuel, qui s'intègre à la racine sous l'arborescence `/proc` : c'est le « ProcFS [BBN] ».

Son rôle, à l'origine, est de permettre d'exposer un maximum d'informations de l'espace noyau à l'espace utilisateur, d'une manière qui soit simple à ré-exploiter, i.e. en restant dans la philosophie habituelle Unix de « tout est fichier ».

On y trouve par exemple :

- `/proc/meminfo` : Informations quant à la mémoire du système ;
- `/proc/loadavg` : Informations relatives à la charge du système.

Et beaucoup d'autres. Le lecteur pourra se référer à la documentation officielle [BBN] présente dans le noyau Linux. Cependant, avec le temps, l'utilisation de ce système de fichier s'est dévoyée[2], ce qui a poussé ses mainteneurs à une refonte. Plus exactement, à créer d'autres systèmes de fichiers virtuels : SysFS [Moc], et ConfigFS. Nous parlerons un peu plus en détail de ConfigFS dans la section 3.2.3.

L'utilisation, côté noyau, de ce système de fichier est relativement simple, et est plutôt bien documentée. Là encore, le lecteur pourra trouver des exemples concrets en se référant à [CRKH05a, FL08].

Quant à SysFS [Moc], il joue, *in fine* le même rôle que ProcFS : afficher à l'espace utilisateur des informations propres à l'espace noyau, à la différence qu'il est dédié aux éléments matériels du systèmes tels que les bus, les processeurs, etc.

3.2.3 ConfigFS

Ce système de fichier virtuel a été introduit en 2005, par un développeur d'Oracle. À ce jour, et en dehors de Kerrighed, les projets intégrés au noyau Linux qui suivent exploitent ConfigFS :

- OCFS2, un système de fichier développé par Oracle ;
- DLM, un système de gestion de verrou distribué, développé par RedHat ;
- NetConsole, qui redirige la console du noyau sur un port udp disant.

D'autres projets, qui ne sont pas intégrés au noyau, exploitent aussi ConfigFS. Citons par exemple `Target_Core_Mod`.

De manière succincte, ConfigFS permet la gestion d'objets de configuration en espace noyau, depuis l'espace utilisateur. Concrètement, il se présente sous la forme d'une arborescence dans le système de fichier, présente sous `/config`, et qui se composera donc d'un ensembles de répertoires et de fichiers, pour lesquels on pourra effectuer des opérations de lecture et d'écriture.

Cette arborescence est composée directement des objets de configuration que nous manipulons, et représente la hiérarchie d'héritage de ces derniers. Les objets sont créés et détruits en créant ou détruisant le répertoire correspondant dans l'arborescence. Ensuite, si par exemple un objet possède un attribut, il devient possible de lire sa valeur en lisant le fichier qui représente cet attribut. De même, pour changer la valeur de ce dernier, il suffit d'écrire dedans. En espace noyau, les objets qui sont manipulés sont des structures de type `struct config_item`. L'aspect le plus intéressant de ConfigFS, est que l'on peut **créer** les objets de configuration depuis l'espace utilisateur, là où SysFS ne permet que de manipuler des objets qui sont créés par le noyau.

En dehors de la documentation officielle [Bec] présente dans le noyau Linux, il est assez difficile de trouver de la documentation sur l'utilisation, au sens de la réalisation de module noyau exploitant une interface de

[2] « That debug code had originally used procfs to export a device tree, but under strict urging from Linus Torvalds, it was converted to use a new filesys- tem based on ramfs »[Moc]

configuration dans ConfigFS Le lecteur pourra constater la présence d'évolutions non négligeables entre la version de ConfigFS utilisée dans Linux 2.6.20, et celle actuellement disponible dans Linux 2.6.29. C'est très explicite avec les sources d'exemples d'utilisation de ConfigFS disponibles à côté de la documentation : impossible de faire compiler celles disponibles dans la dernière version avec notre tronc de sources qui est sur la version 2.6.20, et réciproquement.

Dans le but d'aider à la compréhension du mécanisme d'utilisation de ConfigFS pour le lecteur, un schéma explicatif est proposé en annexe J.

3.2.4 Linux Test Project et Kerrighed Test Project

Ce projet, qui a été initié par la société SGI est depuis maintenu par IBM. L'objectif est de fournir un framework complet pour réaliser des tests de non régression sur l'ensemble du code source du noyau Linux.

La manière de rédiger les tests est laissée à la convenance du développeur, qui peut soit les écrire en C, soit en exploitant l'API exposée par le projet. Selon la documentation, 94.43% des tests présents dans le framework sont écris en ANSI-C. Les deux autres possibilités offertes concerne l'utilisation de scripts Shell ou le langage Perl. À titre d'information, seuls 4.60% des tests sont écris au moyen de scripts Shell, et les 0.62% restant sont en Perl.

Quelque soit le langage utilisé, l'API reste la même. Les éléments principaux qui sont nécessaires à savoir concernent la remontée des informations, c'est-à-dire l'utilisation des fonctions du noyau à tester, et la comparaison de leurs effets ou retours. La documentation donne l'API utilisée pour notifier au framework le résultat de chaque test :

- `tst_res` – Afficher le message résultat, en incluant le contenu d'un fichier ;
- `tst_resm` – Afficher le message résultat ;
- `tst_brk` – Afficher le message résultat et casser les cas de tests restant ;
- `tst_brkm` – Afficher le message résultat, en incluant le contenu d'un fichier, et casser les cas de tests restant ;
- `tst_brkloop` – Afficher le message résultat, en incluant le contenu d'un fichier, et casser les cas de tests restant dans une boucle ;
- `tst_brkloopm` – Afficher le message résultat et casser les cas de tests restant dans une boucle.

Le premier paramètre de chacune de ces fonctions est le type de résultat. Les valeurs possibles sont les suivantes, et peuvent se combiner avec des opérateurs binaires :

- `TPASS` – Le cas de test a retourné une valeur attendue ;
- `TFAIL` – Le cas de test a retourné une valeur inattendue ;
- `TBROK` – Une ressource nécessaire à l'exécution du test n'a pas pû être obtenue ;
- `TCONF` – Le cas de test n'est pas adapté à la configuration matérielle ou logicielle ;
- `TRETR` – Le cas de test n'est plus valide et a été « retiré » ;
- `TWARN` – Des effets de bord indésirables ont eu lieu, mais le test n'a pas été perturbé ;
- `TINFO` – Affichage d'un message d'information sur le test, qui ne correspond ni à un problème ni à un résultat.

Pour réaliser un test, il suffit donc d'effectuer l'appel qui est à tester, de regarder le résultat puis de renseigner le résultat de ce test à l'aide d'une des fonctions présentées ci-dessus, en indiquant le type de retour adéquat.

Le Kerrighed Test Project est simplement un ensemble de tests développés pour valider les modifications apportées par le projet Kerrighed au noyau Linux, et qui s'appuie sur le Linux Test Project.

4. Réalisation des besoins

Nous allons maintenant nous permettre de reprendre chacun des besoins qui ont été soulevés dans le cahier des charges, disponible en annexe A. Pour chacun de ces besoins, nous commencerons par faire un point sur ce qui a été fait, en détaillant exactement le travail.

4.1 SchedConfig : Ordonnanceur Configurable en espace noyau

L'objet de ce premier point est de permettre de combler le besoin 9, présenté en détails dans l'annexe C.6 : il s'agit de savoir ce qui a changé dans Kerrighed et qui concerne l'ordonnancement des processus. Avant d'être en mesure de commencer les travaux concernant la réalisation des besoins, il a été nécessaire d'identifier les changements dans Kerrighed qui ont eu lieu depuis la version 2.2.0, afin de pouvoir circonscrire le champ des modifications que nous devrons apporter.

Tous nos besoins tournent autour d'un problème principal : la possibilité d'écrire de nouveaux algorithmes d'ordonnancement.

À ce sujet, et c'est ce qui nous a motivé à exploiter la version de développement de Kerrighed, une modification majeure se trouve être SchedConfig, qui a été présentée lors de la réunion Kerrighed de 2008 [RN08].

Son objectif est de fournir un framework complet permettant de subvenir à deux besoins :

- D'un côté, avoir la possibilité d'appliquer des politiques d'ordonnancement différentes à des processus différents, pour permettre de répondre à des problèmes différents.
- De l'autre, abaisser la barrière d'entrée pour l'écriture de ces nouvelles politiques d'ordonnancement.

Ces deux points sont un changement majeur. Le premier permet de répondre déjà en partie à nos besoins. Les travaux de thèse de Matthieu Pérotin[Pér08] ont en effet mis en lumière que les algorithmes ont chacun leurs propriétés, et la question se pose : savoir s'il serait possible d'améliorer les performances du système en utilisant plusieurs algorithmes, pour traiter des cas différents.

Le deuxième point en particulier est lourd de conséquences quant au travail à réaliser : là où les travaux de Mathieu Dabert [Dab08] et de Matthieu Pérotin [Pér08] ont nécessité de directement modifier des structures de données du noyau Linux, et de devoir mettre en place des contournements pour changer l'ordonnanceur utilisé. Ceci est lourd à mettre en place, et nécessite de recompiler tout Kerrighed afin de tester les modifications.

L'arrivée de SchedConfig constitue une avancée considérable à ce niveau, puisque ce sont maintenant des modules noyau qui sont utilisés pour toutes les fonctionnalités : récupération d'informations, ordonnancement. Cela abaisse donc considérablement la barrière à l'entrée pour développer de nouvelles choses.

SchedConfig propose différents « modules », dont nous allons exposer les caractéristiques dans les sections 4.1.1, 4.1.2, 4.1.3 :

- Modules de type *Probe*, qui sont chargés de la remontée d'informations ;
- Modules de type *Filter*, qui sont chargés de filtrer les informations des probes ;
- Modules de type *Policy*, qui sont chargés d'appliquer une politique d'ordonnancement.

Un point important à comprendre pour la suite est que, même pour être en mesure de communiquer entre ces modules en espace noyau, il est nécessaire de lier ensemble leurs composants au niveau de ConfigFS. Cela se fait grâce à l'utilisation de simples . . . liens symboliques et la création de répertoires, pour les filtres.

Comme cela a été indiqué précédemment, les deux composants d'information et de contrôle sont nécessaires. Dans SchedConfig le composant d'information est découpé entre les sources de données et un

ensemble de filtres. Ce sont les modules de Probe et les modules Filter. Le composant de contrôle est quant à lui géré par les modules Policy.

4.1.1 Module *Probe*

Ces modules reprennent le concept de « Composant d'Information », présenté dans [Pér08] – et rappelé en introduction –, et permettent de l'implémenter.

Un module de probe est composé d'un ensemble de *probes sources*. Aucune contrainte technique n'est imposée sur ces probes sources, l'idée étant simplement de regrouper au sein d'une même probe un ensemble de probes sources cohérent. C'est une des raisons qui nous a poussé par la suite, pendant l'implémentation, à réaliser plusieurs modules de probes là où d'aucun aurait pu objecter que toutes ces probes sources auraient pu être regroupées dans un seul module.

Une probe source est donc l'élément de base de remontée d'informations proposé par Kerrighed. Le framework propose d'accéder depuis l'espace noyau – à la fois sur la machine locale ou à distance – aux valeurs de ces probes sources. Pour le permettre, chacune se doit d'implémenter les méthodes adéquates, à savoir la méthode `GET` – ou `GET_WITH_INPUT` si la probe source accepte des paramètres – afin d'effectuer la récupération des données et de les transmettre à qui de droit. Cette transmission est opérée par le framework.

Enfin, le framework offre encore d'autres possibilités comme la capacité à notifier les utilisateurs de notre probe source d'une modification de sa valeur.

4.1.2 Module *Filter*

Les modules de filtrage ont pour but, comme leur nom le stipule, de permettre de « filtrer » des données. Par filtrage, on peut entendre le simple fait de laisser passer ou non une information. Mais de manière plus générale, le rôle de ces modules est d'adapter une probe source à une instance particulière d'une politique d'ordonnancement. Associé à une source qui implémenterait le mécanisme de notification de changement, cela permet de donner la possibilité de réguler la transmission de cette notification.

Une autre utilisation est un peu plus détournée. En effet, nous le verrons par la suite, la récupération des données depuis les probes, en espace noyau, et à distance s'opère de manière asynchrone. Afin de simplifier cette récupération, un module de filtrage est utilisé en tampon, afin de déporter en son sein la complexité liée à la gestion du caractère asynchrone de la récupération des données. Il opère par la même une gestion de cache, afin d'être en mesure de renvoyer une valeur s'il n'est pas en mesure d'en obtenir une.

4.1.3 Module *Policy*

Voici l'élément central du problème d'ordonnancement. Ces modules ont vocation à calculer des politiques, de manière distribuée, c'est à dire sur chaque nœud, sans prise de décision globale. Une politique d'ordonnancement est simplement la réponse à la question de savoir quel processus exécuter sur quel processeur. On retrouve ici la notion de « Composant de Contrôle »introduite dans [Pér08] et rappelée en introduction.

Ceci n'empêche pas que ces modules ont besoin d'informations. Il s'agit des données qui sont remontées par les probe sources, et auxquelles ils peuvent accéder, que ce soit localement ou à distance. L'accès local se fait au moyen de la fonction `scheduler_port_get_value`, et l'accès à distance se fait quant à lui grâce à `scheduler_port_get_remote_value`. Comme précisé dans la section 4.1.2 présentant les modules de filtrage, l'accès à distance est asynchrone.

Le nom de ces fonctions nous amène à présenter un autre concept nécessaire : les ports. Un port est une porte d'entrée du module de politique qui permet de communiquer de manière unidirectionnelle des valeurs depuis des probes, éventuellement au travers d'un filtre. Ces ports sont exposés à l'utilisateur dans le répertoire ConfigFS qui correspond au module de politique, sous la forme d'un répertoire avec un attribut. Pour que la communication se fasse de manière effective, il est nécessaire de « brancher » notre port sur une

source d'alimentation, que ce soit un filtre ou une probe source. Cette opération se fait grâce à ConfigFS : il suffit de faire un lien symbolique entre le répertoire représentant le port du module de politique, et celui représentant le module de filtre ou la probe source.

Une fois que notre module de politique est capable de récupérer des informations, il est en mesure de calculer ce qu'il souhaite. Aucune contrainte n'est imposée sur cette partie, notamment sur la fréquence des calculs. En réalité, on peut se reposer sur les fonctionnalités du framework SchedConfig, qui permettent de réveiller sur occurrence de changement dans une probe source, ou bien tout simplement utiliser un compteur. La première solution est utilisée dans l'ordonnanceur MOSIX [Ril] implémenté dans Kerrighed. Comme nous le verrons par la suite, nous avons opté pour la seconde solution.

4.2 Synthèse sur les avancées proposées par Kerrighed

Nous avons présenté les éléments principaux qui constituent le nouveau framework d'ordonnancement configurable, SchedConfig. Nous proposerons dans la sous-section qui suis un schéma qui résume la hiérarchie entre les différents types de modules, et les flux qui les lient.

La sous-section qui suivra sera chargée de donner les étapes importantes dans l'écriture du code d'un module de type Probe, Filter ou Policy. Les modules écrits et joints en annexe pourront servir de référence pour permettre à l'utilisateur de voir, concrètement, la place occupée par le formalisme de SchedConfig dans l'implémentation réelle.

Un module de probe conseillé pour cette opération est la probe User, présente dans l'annexe F.7. Pour les modules de type filter, le seul exemple disponible est le module RBT Cache, dont le code est disponible dans l'annexe G.1. De même, pour les modules de politique, le code de `rbt_policy` pourra servir de référence, et est présent dans l'annexe G.2.

Enfin, notons par ailleurs qu'il convient de bien saisir les interactions entre SchedConfig et ConfigFS. Le premier est un framework d'ordonnancement en espace noyau, qui a la particularité d'être configurable. C'est dans le but de gérer cette configuration depuis l'espace utilisateur que SchedConfig fait appel à ConfigFS.

4.2.1 Schéma récapitulatif de SchedConfig

Ce schéma, proposé en figure 4.1 se propose de résumer la hiérarchie des modules noyau utilisés pour former SchedConfig. Il est inspiré très fortement d'un schéma contenu dans la présentation de SchedConfig qui a eu lieu pendant la rencontre Kerrighed 2008 [RN08].

4.2.2 Compléments vis-à-vis de la création des modules

L'objectif de cette sous-section est de founir les informations nécessaires pour bien comprendre, dans le code implémenté, ce qui correspond aux déclarations propres à l'intégration dans SchedConfig, et ce qui correspond à notre propre code.

Pour chacun des trois types de modules, il s'agit donc de présenter les étapes et structures principales qu'il convient de déclarer, et comment en notifier SchedConfig.

Création d'un module de type Probe

Pour ce module, nous conseillons de se référer au module de probe *User*, présent en annexe F.7 du fait de sa simplicité. Avant de créer la probe, il est nécessaire de commencer par créer une structure `struct scheduler_probe_type`, via l'appel à la macro `SCHEDULER_PROBE_TYPE`.

Puis, il convient de créer la ou les probe(s) source(s), via l'appel à la fonction `scheduler_probe_source_create()`. La définition des probes sources se fait à l'aide de la paire de macros `BEGIN_SCHEDULER_PROBE_SOURCE_TYPE` et `END_SCHEDULER_PROBE_SOURCE_TYPE` entre lesquels on pourra définir le nom, les méthodes implémentées, le type de la valeur retournée, le type des paramètres s'il y en a.

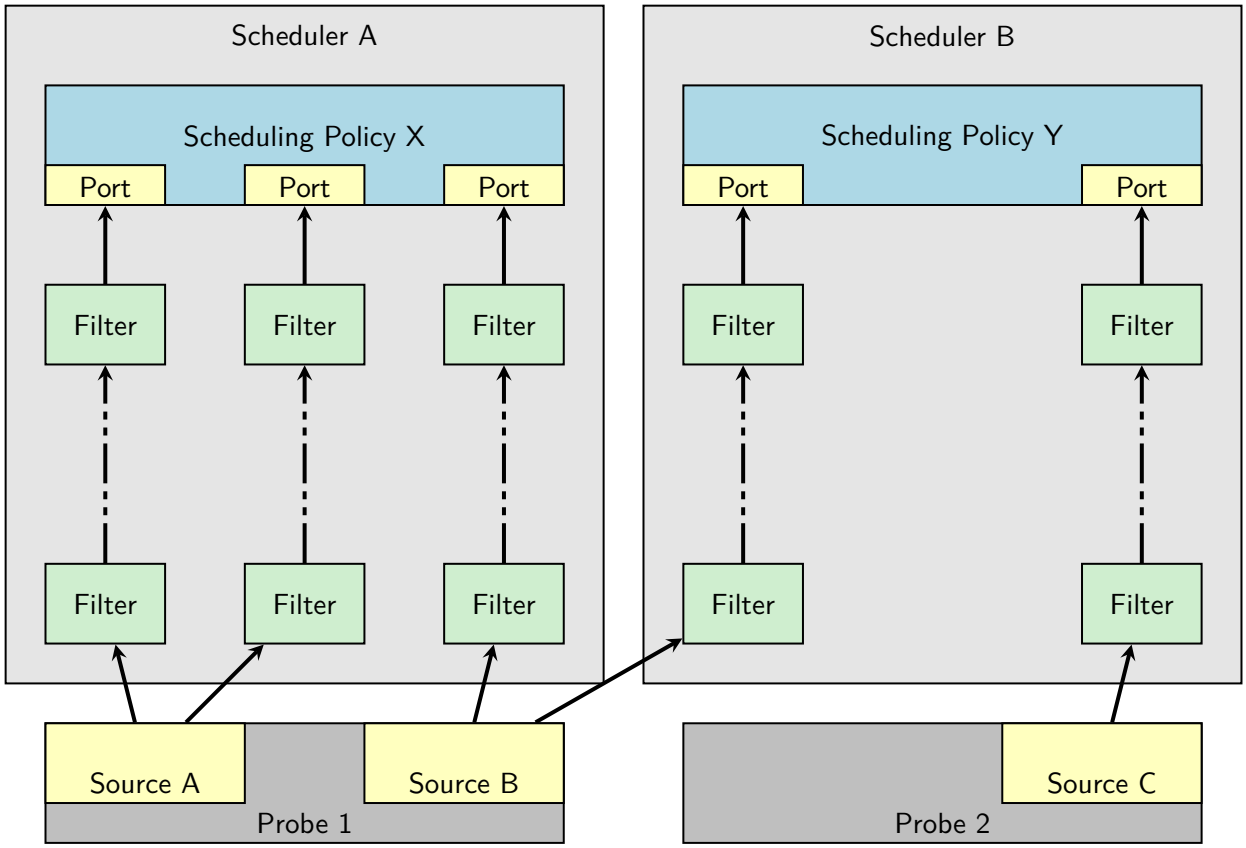


FIG. 4.1 – Résumé de la hiérarchie de SchedConfig. Ce schéma est fortement inspiré de celui présent dans [RN08]

Enfin, la dernière étape consiste en la création effective, grâce à `scheduler_probe_create()`, de la probe, puis à son enregistrement dans le système via l'appel à `scheduler_probe_register()`.

En plus de ces méthodes, il est intéressant d'implémenter aussi la méthode `SHOW`, qui elle sera utilisée sur occurence d'un appel système `read()` dans le ConfigFS. Concrètement, cela permet d'exposer les informations à l'espace utilisateur. Pertinent dans notre cas.

Création d'un module de type Filter

Pour suivre le déroulement de cette explication, il est conseillé de se référer au module de filtre *RBT Cache*, dont le code est donné en annexe G.1.

Il est d'abord nécessaire de déclarer la structure qui défini notre filtre. Ceci est fait au moyen des deux macros `BEGIN_SCHEDULER_FILTER_TYPE` et `END_SCHEDULER_FILTER_TYPE`, comme rappelé dans le code 4.1. De manière analogue à ce qui a été explicité pour le cas des modules de probe, on trouve au sein de cette structure la déclaration des fonctionnalités qui seront implémentées par notre module de filtrage.

C'est également au sein de cette structure que sont déclarés les types de données utilisés ainsi que la liste des attributs que nous utilisons, précisée dans le paramètre `remote_cache_attrs`. Ces attributs seront exposés dans ConfigFS, et dans le cas présent l'attribut nous permet de régler la fréquence de mise à jour du cache.

Attardons nous à la définition de l'attribut. Ceci est rappelé dans le code 4.2. Il s'agit d'exploiter les macros `BEGIN_SCHEDULER_FILTER_ATTRIBUTE` et `END_SCHEDULER_FILTER_ATTRIBUTE` pour créer la structure nécessaire. Les trois points importants à relever ici sont que l'on défini les droits d'accès sur le fichier que va créer ConfigFS – ici, `0666` donc lecture et écriture pour tout le monde – et que nous définissons également les opérations possibles sur cet attribut : `SHOW` et `STORE`. Dès lors, il sera nécessaire d'avoir implémenté ces deux fonctions, en utilisant la macro `DEFINE_SCHEDULER_FILTER_ATTRIBUTE_SHOW` dans le premier cas, et `DEFINE_SCHEDULER_FILTER_ATTRIBUTE_STORE` dans le second.

Pour gérer les opérations de création et de destruction du filtre, et surtout pour être en mesure d'initialiser et de détruire les structures de données que nous utilisons, il convient également d'implémenter deux fonctions grâce aux macros `DEFINE_SCHEDULER_FILTER_NEW` et `DEFINE_SCHEDULER_FILTER_DESTROY`. La première a la tâche d'initialiser le filtre, via l'appel à `scheduler_filter_init()`, et d'initialiser le nécessaire pour que régulièrement le noyau opère une mise à jour des données. La seconde fonction a simplement pour tâche d'arrêter les requêtes encore en cours, et de détruire les structures ayant été allouées.

```
1 static BEGIN_SCHEDULER_FILTER_TYPE(rbt_cache_filter),
2   .SCHEDULER_FILTER_UPDATE_VALUE(rbt_cache_filter),
3   .SCHEDULER_FILTER_GET_REMOTE_VALUE(rbt_cache_filter),
4   .SCHEDULER_FILTER_SOURCE_VALUE_TYPE(rbt_cache_filter, unsigned int),
5   .SCHEDULER_FILTER_PORT_VALUE_TYPE(rbt_cache_filter, unsigned int),
6   .SCHEDULER_FILTER_ATTRIBUTES(rbt_cache_filter, remote_cache_attrs),
7 END_SCHEDULER_FILTER_TYPE(rbt_cache_filter);
```

Code 4.1 – Création d'une structure décrivant un filtre

```
1 static BEGIN_SCHEDULER_FILTER_ATTRIBUTE(polling_period_attr, polling_period, 0666),
2   .SCHEDULER_FILTER_ATTRIBUTE_SHOW(polling_period),
3   .SCHEDULER_FILTER_ATTRIBUTE_STORE(polling_period),
4 END_SCHEDULER_FILTER_ATTRIBUTE(polling_period);
5
6 static struct scheduler_filter_attribute *remote_cache_attrs[] = {
7   &polling_period_attr,
8   NULL
9 };
```

Code 4.2 – Définition des attributs d'un filtre

Création d'un module de type Policy

Dans le cadre de cette explication, le code qu'il convient de suivre est celui du module de politique *RBT*, présent dans l'annexe G.2. Une fois encore, nous retrouvons deux types de structures de données à manipuler afin de pouvoir créer notre module.

La première est le type `struct scheduler_policy_type`. Cette fois, contrairement aux autres structures similaires telles que `struct scheduler_probe_type` ou `struct scheduler_filter_type`, il n'y a pas de couple de macros à disposition pour déclarer et initialiser la structure. Seule la macro `SCHEDULER_POLICY_TYPE` est nécessaire. De la même manière, le mécanisme de création « implicite »des variables suffixées avec `_type` qui était utilisé avec les macros `BEGIN/END` n'est plus présent. Dès lors il convient de faire attention lors de la déclaration de la structure à l'aide ce cette macro : le premier paramètre, qui défini le nom de la variable, ne sera pas modifié. Les autres paramètres renseignent le nom de la politique, tel qu'il sera utilisé pour générer les entrées de ConfigFS, ainsi que deux structures de données. Le dernier de ces paramètres concerne les attributs de la politique.

L'avant-dernier, quant à lui, permet de définir les fonctions qui seront en charge d'implémenter les constructeurs et destructeurs de la politique. La création de ces structures présentée à titre d'exemple dans le code 4.3. Une fois cette variable de type `struct scheduler_policy_type` créée, il convient de l'enregistrer dans le framework SchedConfig, grâce à l'utilisation de la fonction `scheduler_policy_type_register`. Dès cet instant, le framework sera en mesure de créer la politique, en appelant le constructeur.

Dans ce constructeur, il restera à initialiser une variable de type `struct scheduler_policy`, qui aura été préalablement déclarée, au moyen de la fonction `scheduler_policy_init`.

Un dernier point important consiste à retirer le module de politique du framework lors de son déchargement. Ceci a lieu dans le corps du destructeur, et s'opère simplement grâce à la fonction `scheduler_policy_type_unregister`.

```
1 static struct scheduler_policy_operations rbt_policy_ops = {
2   .new = rbt_policy_new,
3   .destroy = rbt_policy_destroy,
4 };
5
6 static SCHEDULER_POLICY_TYPE(rbt_policy_type, "rbt_policy",
7       &rbt_policy_ops, rbt_policy_attrs);
```

Code 4.3 – Nécessaire à la création d'une politique d'ordonnancement

Création d'un Port

Les ports servent, comme cela a déjà été précisé, à permettre de récupérer les données en provenance d'une probe source, éventuellement par l'intermédiaire d'un filtre. Afin de s'assurer que le propos est clair, nous proposons ici de montrer les détails d'implémentation nécessaires pour utiliser les ports. Cette information est commune quelque soit le type de module réalisé : la méthode de création et d'utilisation des ports est identique dans tous les modules de SchedConfig qui peuvent y faire appel.

Le cycle de vie d'un port peut-être résumé aux opérations suivantes. Pour chacune, nous montrerons le code nécessaire.

1. Création d'une structure de données ;
2. Initialisation de la structure du port ;
3. Déclaration du port pour qu'il soit présent dans SchedConfig ;
4. Utilisation du port ;
5. Destruction du port.

Création de la structure de donnée pour un port Deux structures de données sont nécessaires pour utiliser un port. La première, `struct scheduler_port` correspond aux variables qui seront manipulées lorsque l'on souhaite lire une valeur depuis un port. Il suffit de déclarer un symbole de ce type pour l'utiliser.

L'autre élément nécessaire est la structure `struct scheduler_port_type` qui est créée grâce à l'utilisation du couple de macros `BEGIN_SCHEDULER_PORT_TYPE` et `END_SCHEDULER_PORT_TYPE`. Lors de la définition de cette structure, il est nécessaire de définir certaines caractéristiques du port. Notamment, le type de données retournées par le port, grâce à la macro `SCHEDULER_PORT_VALUE_TYPE`.

Si le port concerne une probe source qui admet un paramètre, il est aussi nécessaire de le spécifier à ce moment, en utilisant la macro `SCHEDULER_PORT_PARAM_TYPE`.

Le code 4.4 donne un exemple de la définition d'un port admettant un paramètre, ici de type `pid_t`. La variable créée est de type `struct scheduler_port_type`, et est nommée en utilisant le symbole passé en paramètre aux macros `BEGIN_SCHEDULER_PORT_TYPE` et `END_SCHEDULER_PORT_TYPE` suffixé par `_type`. Dans l'exemple, on obtient donc `port_process_size_type`.

Enfin, la dernière étape nécessaire, et qui doit se dérouler dès le chargement du module dans le noyau, consiste à initialiser les variables de type `struct scheduler_port_type`. Ceci se fait au moyen de la fonction `scheduler_port_type_init`, en utilisant la variable que nous venons donc de créer.

```
1 static BEGIN_SCHEDULER_PORT_TYPE(port_process_size),
2   .SCHEDULER_PORT_VALUE_TYPE(port_process_size, unsigned long),
3   .SCHEDULER_PORT_PARAM_TYPE(port_process_size, pid_t),
4 END_SCHEDULER_PORT_TYPE(port_process_size);
```

Code 4.4 – Définition d'un port paramétré

Initialisation de la structure du port À l'initialisation du module, il est ensuite nécessaire d'initialiser la structure de données de type `struct scheduler_port`. Ceci s'opère au moyen de la fonction `scheduler_port_init()`, à laquelle on doit passer au moins la variable de type `struct scheduler_port` et également celle de type `struct scheduler_port_type` comme créée précédemment. Nous définissons également lors de cet appel le nom qui sera utilisé plus tard lors de l'ajout dans l'arborescence ConfigFS.

Le code 4.5 présente un exemple de l'appel usité.

```
1 scheduler_port_init(&p->port_process_jiffies, "process_jiffies",
2   &port_process_jiffies_type, NULL, NULL)
```

Code 4.5 – Initialisation d'un port

Déclaration du port pour SchedConfig Afin de faire en sorte que les entrées ConfigFS correspondant à nos ports soient créés, puisque ce sont avec ces entrées qu'il va être nécessaire de créer des liens symboliques pour opérer le « branchement »des sources, il convient de renseigner un tableau de `struct config_group`.

Ce tableau doit être initialisé avec les pointeurs retournés par l'appel à une fonction, `scheduler_port_config_group`. Celle-ci prend en unique paramètre la structure de données de type `struct scheduler_port` qui a été initialisée préalablement. L'utilisation de ce tableau dépend ensuite du type de module concerné, mais dans le cas général, il s'agit de le passer en paramètre à une fonction d'initialisation, qui fera ainsi en sorte que ces données soient utilisées pour créer les entrées nécessaires dans le système de fichier virtuel ConfigFS.

Le code 4.6 propose un exemple du renseignement de ce tableau.

```
1 struct config_group *def_groups[6];
2
3 /* initialize default memory groups. */
4 def_groups[0] = scheduler_port_config_group(&p->port_mattload);
5 def_groups[1] = scheduler_port_config_group(&p->port_active_tasks);
6 def_groups[2] = scheduler_port_config_group(&p->port_loadinc);
7 def_groups[3] = scheduler_port_config_group(&p->port_process_jiffies);
```

```
8 def_groups[4] = scheduler_port_config_group(&p->port_process_size);
9 def_groups[5] = NULL;
```

Code 4.6 – Déclaration de ports pour intégration dans ConfigFS

Utilisation du port L'utilisation du port se fait, dans tous les cas, en utilisant la variable de type `struct scheduler_port` qui a été définie et intialisée préalablement.

La lecture effective des données se fait soit au moyen de la fonction `scheduler_port_get_value` si la valeur qui nous intéresse est celle du nœud local, soit avec `scheduler_port_get_remote_value` si nous nous intéressons à la valeur du port sur un nœud distant.

Destruction du port Lors du déchargement du module, il est nécessaire de libérer ce qui a été alloué. Si, comme dans notre cas, aucune allocation de structure de données pour les ports n'est faite via l'utilisation de `kmalloc()` ou dérivés, alors la seule opération nécessaire consiste en l'appel à la fonction `scheduler_port_type_cleanup` sur les variables qui ont été déclarées au moyen des macros `BEGIN_SCHEDULER_PORT_TYPE` et `END_SCHEDULER_PORT_TYPE`.

4.3 Portage de l'API

Afin de répondre au besoin 1 énoncé en annexe C.1 qui consiste à porter l'API réalisée et implémentée dans Kerrighed 2.2.0 lors du travail de Mathieu Dabert [Dab08], il a fallu procéder à un découpage en tenant compte de l'infrastructure SchedConfig à notre disposition. Concrètement, il s'agit de porter les modifications qui avaient été réalisées au cœur des structures de données du noyau, et les Composants d'Informations, de la meilleure manière qu'il soit ; i.e. si possible en évitant de toucher aux sources de Linux, en étant le moins intrusif.

Dans un premier temps, nous rappelons l'API qui a été proposée dans [Dab08]. Nous allons ensuite commenter le découpage qui a été fait, puis nous présenterons en détail chacun des modules implémentés. Le code est disponible en annexe F. Il sera également rappelé pour chaque module l'annexe précise du code en question.

Nous verrons d'ailleurs qu'au cours de la rédaction du cahier des charges, une erreur d'interprétation nous a fait pensé que nous n'avions pas besoin d'une probe source précise, et que nous pouvions ré-exploiter ce qui est exposé par la probe `mosix_probe` inclue dans Kerrighed, modulo l'ajout d'une probe source à cette dernière : c'était pour le cas de la consommation en Jiffies d'un processus.

4.3.1 Rappel sur l'API proposée par Mathieu Dabert

Nous reprenons simplement le récapitulatif de l'API qui a été proposée par Mathieu Dabert, au cours de son Projet de Fin d'Études [Dab08]. Cette API est représentée dans le tableau 4.1.

Il convient de noter que le but de ce portage est d'avoir à disposition les mêmes informations, mais intégrées dans le nouveau framework qu'est SchedConfig. Cela implique que, si une ou plusieurs des informations définies dans cette API sont déjà disponibles dans le code de Kerrighed, nous réutiliserons directement plutôt que de réécrire le code.

4.3.2 Découpage des modules

Dans l'optique d'avoir un code cohérent et facilement réutilisable, le découpage suivant a été opéré :

- Probe *CPUSpeed*, voir l'annexe F.1 ;
- Probe *Load*, voir l'annexe F.2 ;
- Probe *Mattload*, voir l'annexe F.5 ;
- Probe *Local User Presence*, voir l'annexe F.3 ;

Dénomination	Fonction
Charge moyenne d'un nœud	mth_avg_remote_load
Mémoire libre d'un nœud	mth_remote_free_RAM
Nombre de CPU d'un nœud	mth_remote_nbcpu
Vitesse du CPU d'un nœud	mth_remote_speed
Connexion d'un utilisateur	mth_remote_user_logged
Taille d'un processus	mth_task_total_vm
Consommation en jiffes d'un processus	mth_task_jiffies

TAB. 4.1 – API proposée par Mathieu Dabert [Dab08]

- Module d'information de présence *Local User Notifier*, voir l'annexe F.4 ;
- Probe *Processsize*, voir l'annexe F.6 ;
- Probe *User*, voir l'annexe F.7.

4.3.3 Probe *CPUSpeed*

Le code de cette probe est disponible en annexe F.1. Le but de cette probe est d'implémenter deux probes sources :

- La probe source `connected` ;
- La probe source `speed`.

Comme leur nom l'indique, le rôle de la première est de retourner le nombre de processeurs locaux présents, là où le rôle de la seconde se borne à nous retourner la vitesse d'un de ces processeurs.

Probe source *connected*

Cette probe source implémente deux méthodes : `GET` et `SHOW`. Elle exploite une fonctionnalité du noyau Linux, qui nous permet de directement obtenir la réponse que l'on souhaite : `nom_online_cpus()`.

Probe source *speed*

Cette probe source implémente deux méthodes : `GET_WITH_INPUT` et `SHOW`. Les paramètres acceptés par la première sont simplement les numéros des processeurs pour lesquels on souhaite connaître la vitesse d'horloge.

D'ailleurs, cette dernière est retournée en kHz, pour une raison simple : toutes les méthodes permettant d'obtenir cette information retourne une valeur en kHz.

La méthode `SHOW` affiche la vitesse pour tous les processeurs locaux possibles.

Deux manières sont utilisées pour récupérer l'information de fréquence d'horloge : soit via `cpufreq`, soit directement via la variable `cpu_khz`. En effet, l'appel à `cpufreq_get()` peut retourner 0 kHz. Ce qui est difficilement cohérent, ni acceptable. Il semble que ce soit un soucis de support matériel, peut-être lié à la plateforme utilisée pour le développement. Donc, dans le cas où une valeur égale à zéro est retournée, nous lisons la valeur de la variable `cpu_khz`. Cette seconde méthode n'a jamais posée de soucis.

4.3.4 Probe *Load*

Le code de cette probe est disponible en annexe F.2. Le but qui a motivé l'écriture de cette probe était de proposer d'exposer une information précise : le nombre de tâches du système local qui sont dans l'état `ACTIVABLE`.

Ainsi, elle ne proposait au début qu'une seule probe source. Cependant, comme indiqué précédemment en section 4.3, nous avons constaté que ce que nous pensions comme ré-exploitable dans `mosix_probe` ne

l'était pas. Nous avons donc décidé de proposer dans cette probe l'information concernant la consommation en Jiffies d'un processus, puisqu'il nous semblait que c'était cohérent avec le but fixé par la probe.

Nous exposons donc deux probe sources :

– La probe source `active_tasks` ;
– La probe source `process_jiffies`.

Probe source *active_tasks*

Cette probe source n'expose pas de paramètre, et implémente deux méthodes : `GET` et `SHOW`.

Pour obtenir notre information, nous exploitons les structures proposées de base par le noyau Linux. La structure `struct task_struct` propose un membre nommé `state`. Les commentaires associés à sa définition indiquent les valeurs qu'il peut prendre :

– Une valeur $= -1$: processus non activable ;
– Une valeur $= 0$: processus activable ;
– Une valeur > 0 : processus stoppé.

Il nous suffit ensuite de définir un simple compteur, et de parcourir l'ensemble des processus locaux, grâce à la macro définie par le noyau qu'est `for_each_process()`, puis de compter les processus dont l'état est à 0.

Probe source *process_jiffies*

La récupération de cette information est plus complexe. D'abord, il faut savoir que Kerrighed propose des *hooks* sur lesquels il est possible de s'enregistrer, de sorte à pouvoir exécuter du code sur occurrence d'un évènement.

Avant d'aller plus loin, rappelons ce à quoi correspondent les jiffies. Une définition serait[1] :

> jiffies is global variable that holds the number of ticks that have occured after the system has booted. On each timer interrupt the value of this variable is increased by 1.

C'est donc une manière de mesurer le temps qui s'écoule au sein de l'espace noyau.

Deux *hooks* précis vont nous être utiles dans notre cas :

– Le *hook* `kmh_process_on` ;
– Le *hook* `kmh_process_off`.

Ces deux *hooks* sont déclenchés respectivement quand un processus entre dans la queue d'exploitation locale, et quand il en sort. L'idée est donc d'intercepter ces évènements, et de noter quelque part les références de temps – toujours données en jiffies dans le noyau – correspondant à ces entrées et sorties.

Restait à résoudre un autre problème : comment conserver dans le temps l'information obtenue ? Nous sommes en effet en mesure d'intercepter les entrées et sorties de la queue d'exploitation, mais cependant il nous reste impossible d'être informés de la création ou de la destruction d'un processus.

Nous nous sommes donc inspirés du mécanisme déployé dans Kerrighed au sein de la probe `mosix_probe`, et plus précisément, la gestion de la structure `struct mosix_probe_info` [2]. L'idée générale peut se résumer ainsi : Kerrighed propose un autre mécanisme que les *hooks* pour faire ce que l'on veut. Il s'agit d'être appelé lors de la création ou de la destruction d'un processus. Avant d'être en mesure d'intercepter ces évènements, il va être nécessaire de construire plusieurs choses. Dans notre cas, nous avons commencé par définir une structure `struct load_probe_info`, sur le modèle de la structure `struct mosix_probe_info`. Il s'agit surtout d'être en mesure, quand cela sera nécessaire, de faire le lien entre la structure `struct krg_sched_module_info_type` et d'autres informations, que nous référençons également dans `struct load_probe_info`.

En pratique, il faut commencer par déclarer une structure `struct krg_sched_module_info_type`, qui est pourvue de différents membres, qui sont :

[1] http://mail.nl.linux.org/kernelnewbies/2004-05/msg00184.html
[2] http://lxr.kerlabs.com/kerrighed/source/modules/scheduler/probes/mosix_probe.c

- Le membre `copy`, qui défini un pointeur sur une fonction qui sera appelée pour tout appel à `fork()` réalisé par le système local ;
- Le membre `free`, appelé à chaque destruction de processus ;
- Le membre `import`, appelé à chaque fois qu'un processus est migré à destination de notre machine ;
- Le membre `export`, appelé à chaque fois qu'un processus est migré à partir de notre machine.

Seuls les membres un et trois nous intéressent, et pour des questions de simplification du code, la fonction pointée par `import` fait directement appel à celle pointée par `copy`.

Dans les deux cas, parmi les paramètres passés à nos fonctions cibles, on trouve une structure de type `struct task_struct`. Notre fonction `load_probe_info_copy()`, qui est pointée par le membre `copy` se charge donc simplement d'allouer de la mémoire pour une structure `struct load_probe_info`, et d'y copier les informations intéressantes. Grâce au pointeur `module_info` de cette structure de données, et grâce aux fonctions mises à disposition par Kerrighed, on est en mesure d'écrire la fonction `get_load_probe_info()` qui sera essentielle par la suite : elle sait faire le lien entre une structure de type `struct task_struct` et une structure `struct load_probe_info`.

À présent, nous avons tous les éléments pour être en mesure d'intercepter nos processus. Lors de leur création, la structure précédente est créée et initialisée. Par la suite, quand un processus entre ou sort de la queue d'exploitation, le lien est fait grâce à la fonction `get_load_probe_info()` : nous savons donc à quelle date en jiffies il entre, et à quelle date il sort. Ce qui nous permet d'écrire par exemple `count_process_jiffies()`, qui s'alimente directement dans notre structure `struct load_probe_info`.

Deux méthodes sont donc implémentées par cette probe source :

- La méthode `GET_WITH_INPUT` ;
- La méthode `SHOW`.

La première prend en paramètre un identifiant de processus, `pid_t`, et retourne directement la consommation en jiffies. Le noyau propose le lien direct entre `pid_t` et `struct task_struct` grâce à `find_task_by_pid()`. En reprenant les éléments précédents, on remonte rapidement de cette dernière structure aux informations de consommation.

L'implémentation de `SHOW` est presque plus simple : nous ré-exploitons la macro `for_each_process()` qui énumère les tâches locales, via un itérateur de type `struct task_struct *`. Le lien avec la consommation en jiffies est donc immédiat, et l'on est en mesure d'afficher la consommation en jiffies de tous les processus locaux.

4.3.5 Probe *User*

Le code de cette probe est disponible en annexe F.7. Le but de ce module de probe est uniquement d'exploiter l'interface noyau mise à disposition par le module d'information de présence, *Local User Presence*, présenté un peu plus loin en section 4.6.1.

Une seule probe source est définie, et cette dernière n'implémente que les méthodes `GET` et `SHOW`. Il n'y a aucune difficulté particulière sur ce module, qui a été l'un des premiers à être réalisés.

4.3.6 Probe *Mattload*

Le code de cette probe est disponible en annexe F.5. Comme son nom l'indique – ou tout du moins est censé l'indiquer –, cette probe a pour objectif d'implémenter la mesure de charge définie dans [Pér08]. À titre de rappel, la charge est ainsi définie :

$$load_j = (1 + K \times state(m_j)) \frac{load_j^U}{speed(m_j)}$$

La réalisation de ce module a été particulièrement intéressante. Si, dans un premier temps, cette probe n'a eu en son sein qu'une seule probe source, cela n'a pas duré. En effet il n'est pas possible dans l'espace noyau sous Linux d'utiliser autre chose que des nombres entiers. Cela nous oblige donc à quelques petites

adaptations de la méthode de calcul, en incluant un *facteur multiplicatif* dignement nommé `multiply_factor`. À côté de cela, la méthode de calcul intègre un paramètre K.

La présence de ces caractéristiques variables a entraîné la création de fichiers – appelés des attributs au niveau de ConfigFS – permettant d'adapter les valeurs en temps réel. De même, dans l'optique d'implémenter l'algorithme RBT, nous devons connaître l'incrément de charge pour un nœud. Cette valeur va dépendre également de ces paramètres. Donc, pour éviter de dupliquer inutilement du code, nous avons décidé de créer une seconde probe source à cette probe, nommée justement `load_increment`.

Par ailleurs, une partie importante des autres probes écrites sont nécessaires au calcul de la charge dite « Matteo »décrite dans [Pér08]. Il est donc assez logique et pertinent de vouloir réutiliser ce que nous faisons pour cette probe. Là, nous nous sommes heurtés à un problème : si la notion de ports est bien définie, elle n'est à ce stade utilisable que dans le contexte des modules de politique d'ordonnancement.

Étant donné que nous avions besoins des probes sources suivantes `active_tasks`, `speed`, `connected` et `user_connected`, il n'était pas envisageable de recopier du code, ou d'utiliser une technique « sale ».

Cette probe s'articule donc en deux probes sources, qui utilisent *quasiment* le même code (à un paramètre près) :

- La probe source `mattload` ;
- La probe source `load_increment`.

Par ailleurs, elle repose sur quatre ports pour obtenir ses données, que nous avons cité précédemment. Les ports sont nommés de manière explicite :

- Port `active_tasks` ;
- Port `cpu_speed` ;
- Port `cpu_connected` ;
- Port `user_connected`.

Le détail de l'implémentation des probes sources n'est pas particulièrement pertinent. Les principales choses à retenir sont :

- Définition d'un port, grâce au couple `BEGIN_SCHEDULER_PORT_TYPE`/`END_SCHEDULER_PORT_TYPE` ;
- Lecture de la valeur d'un port grâce à `scheduler_port_get_value()`.

D'autres problèmes plus intéressants sont à détailler dans le cadre de l'implémentation de cette probe :

- La gestion d'un attribut ConfigFS afin de paramétrer en temps réel ;
- Comment utiliser des ports depuis une probe ?

Gestion d'un attribut ConfigFS

Il ne faut pas oublier que ConfigFS est un système de fichier virtuel, destiné à configurer des objets noyaux depuis l'espace utilisateur. Une fois que l'on a ce fait en tête, la notion d'attribut prend plus de sens : il s'agit de voir ou changer sa valeur.

En pratique, notre attribut est simplement un fichier attaché à un répertoire. Pour en déclarer un, dans le framework de Kerrighed, il suffit d'exploiter la macro `SCHEDULER_PROBE_ATTRIBUTE`, dans laquelle on donne le nom de la structure à créer, le nom du fichier à créer dans le système de fichier, les droits d'accès, mais surtout deux pointeurs de fonctions. Le premier correspond à la fonction utilisée pour gérer l'appel système `read()`, le second pour gérer `write()`.

Nous utilisons en arrière-plan de simples variables, dans lesquelles ces fonctions viendront écrire en cas d'appel à `write()`, ou qu'elles liront s'il s'agit d'une lecture de la valeur de l'attribut.

Ensuite, une fois ces structures d'attributs déclarées, il convient de les regrouper dans un tableau de `struct scheduler_probe_attribute` – tableau que l'on terminera par un `NULL` – et que nous passerons en second paramètre à la macro `SCHEDULER_PROBE_TYPE`.

À la création des entrées du système de fichier, Kerrighed prendra soin de placer correctement nos attributs, en exploitant en arrière plan ConfigFS.

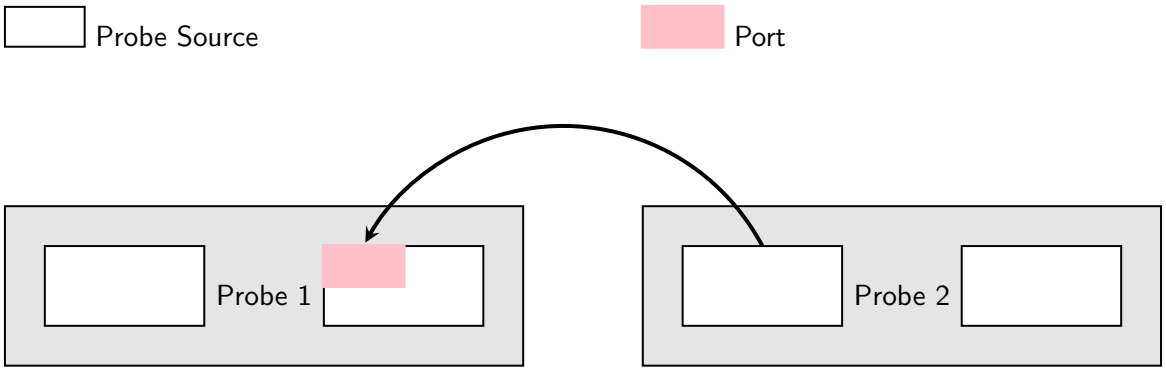


FIG. 4.2 – Création de port dans une probe source : il est ainsi possible de lire la valeur d'une probe source depuis une autre probe source

Utilisation des valeurs d'une probe source dans une autre probe source

Alors que cela peut paraître surprenant, considérant la souplesse du framework SchedConfig, il n'est pas possible de lire la valeur d'une probe source depuis une autre probe source. Plus précisément, nous ne pouvons pas créer les ports qui permettent de réaliser la lecture des données. Heureusement, ConfigFS est un système souple et grâce à sa vision objet, il est possible de facilement l'étendre. De même, Kerrighed exploite plutôt bien ses capacités. Dès lors, nous avons entrepris d'ajouter cette fonctionnalité.

Un petit schéma est proposé en figure 4.2 et résume ce que nos modifications permettent de réaliser.

Dans un premier temps, nous avons regardé comment sont créés les ports dans le cas d'une politique d'ordonnancement :

1. Création des structures de ports, comme cela a été explicité précédemment ;
2. Ensuite, on rempli un tableau de `struct config_group` grâce à `scheduler_port_config_group()` et nos ports créés au point ci-dessus ;
3. Enfin, on passe la main à `scheduler_policy_init()` [3] en lui passant l'information contenue par notre tableau précédent.

Grâce aux lumières de Louis Rilling sur le fonctionnement de ConfigFS et plus particulièrement dans le cadre de cette fonction, nous avons vite compris que la modification était finalement assez simple : il convient, comme dans `scheduler_policy_init()` de recopier dans le membre `default_groups` – contenu dans la structure `struct config_group` présente dans `struct scheduler_policy` – le contenu du tableau que nous avons pu remplir ci-dessus. L'infrastructure de Kerrighed et de ConfigFS s'accommodera du reste.

Nous avons donc modifié la fonction `scheduler_probe_create()`[4] de sorte à opérer cette recopie. Le patch résultant est disponible en annexe F.8.

4.3.7 Probe *Processsize*

Le code de cette probe est disponible en annexe F.6. L'objectif de cette probe, comme son nom l'indique, est d'obtenir l'information de la consommation mémoire totale, en pages mémoires, d'un processus.

C'est une information qui est directement accessible dans la structure noyau `struct task_struct`, dans le membre `mm->total_vm`.

Notre probe source implémente donc deux méthodes, qui sont `GET_WITH_INPUT`, auxquelles on passera un identifiant de processus précis, et `SHOW`, qui nous affichera l'information pour tous les processus du système local.

[3] http://lxr.kerlabs.com/kerrighed/source/modules/scheduler/core/policy.c
[4] http://lxr.kerlabs.com/kerrighed/source/modules/scheduler/core/probe.c

Comme dans les cas précédents, le cœur du travail consiste à récupérer la structure de données correspondant aux processus à examiner. Deux cas sont à considérer :

- La méthode `GET_WITH_INPUT`, où l'on connait déjà l'identifiant du ou des processus pour lesquels on cherche à obtenir l'information ;
- La méthode `SHOW`, où l'on veut l'information pour tous les processus.

Pour le second cas, il s'agit simplement de parcourir la liste de tous les processus locaux, grâce encore une fois à la macro `for_each_process()`.

Dans le premier cas, il convient d'utiliser une fonction qui fait directement la liaison entre une structure de processus et son identifiant : `find_task_by_pid()`.

Il n'y a pas de difficulté autre. À noter tout de même que pendant les tests, il s'est avéré que pour certains processus, le membre `mm` de la structure `struct task_struct` était à `NULL`, empêchant ainsi de le dé-référencer. Il semble, d'après un vieux message posté sur une liste de diffusion du noyau Linux[5] que c'est possible et non choquant. Le fait de rajouter un simple test pour éviter ce cas n'a pas semblé générer de soucis par la suite.

4.4 Utilisation du Kerrighed Test Project

Afin de pouvoir valider le fonctionnement des modules qui implémentent l'API, nous avons entrepris d'écrire des tests de non régression pour le framework de tests Kerrighed Test Project, qui s'appuie lui-même sur le Linux Test Project. Tous les tests ont été écris en langage de script Shell, par commodité, le temps de développement par rapport à un ensemble de programmes en C étant très inférieur, et la nécessité du C n'étant pas justifiée.

L'ensemble des scripts proposés sont disponibles en annexe H. Nous allons, pour chacun des modules testés, expliquer chaque cas de test, et comment nous vérifions le fonctionnement de notre module. Nous présenterons ainsi les tests sur le module Local User Presence, et sur les modules de probe Load, User, Processsize et CPUSpeed.

Enfin, afin de simplifier le développement des premiers scripts, un squelette d'exemple a été réalisé. Son code est présenté dans l'annexe H.1.

4.4.1 Test du module Local User Presence

Présentons le fonctionnement du test de ces modules. À titre de rappel, le premier module est chargé de mettre en place un compteur d'utilisateur en espace noyau, et d'exporter un ensemble de symboles permettant de le manipuler. Il s'agit du module *Local User Presence*. Il est accompagné du module *Local User Notifier*, qui expose une interface dans le système de fichier virtuel *Proc* pour manipuler ce compteur, depuis l'espace utilisateur.

C'est l'interface qui sera utilisée plus tard par un démon de notification, s'exécutant dans la machine virtuelle, et qui recevra les notifications de connexion et de déconnexion en provenance des hôtes physiques, interceptés au moyen des modules PAM et WinLogon. Ce programme de test est chargé de vérifier le fonctionnement de l'ensemble de ces deux modules. Il utilisera donc toutes les fonctions exportées et disponibles via *ProcFS*, même si certaines peuvent sembler inutiles.

Les tests sont exécutés dans l'ordre dans lequel ils sont présentés.

Configuration

L'étape de configuration de ce script de test se charge de :

1. Charger le module *Local User Presence* ;
2. Charger le module *Local User Notifier* ;

[5] http://lkml.indiana.edu/hypermail/linux/kernel/0111.3/1188.html

3. Vérifier que l'arborescence est bien créée dans `/proc/kerrighed/interactive_user`.

Ce dernier point permet également de s'assurer que les modules Kerrighed sont bien chargés et actifs. Ils sont en effet chargés de créer l'entrée `/proc/kerrighed`. Si cette dernière n'existe pas, la création de notre sous-arborescence échouera, et le test qui correspond dans notre script arrêtera tout.

Fonctions de lecture et d'écriture

Afin de simplifier un peu l'écriture de toutes les fonctions de tests, deux fonctions bash ont été définies pour gérer les lectures et les écritures dans les entrées de *ProcFS* qui nous intéressent :

- `read_entry` : chargée de lire une entrée, et de retourner la valeur de retour de la commande `cat` utilisée ;
- `write_entry` : chargée d'écrire dans une entrée, et de retourner la valeur de retour de la commande `echo` utilisée.

Ces deux fonctions prennent deux paramètres :

1. Le premier indique le nom de l'entrée, i.e. le fichier, qu'il faut manipuler ;
2. Le second indique un fichier où écrire le résultat de la sortie standard d'erreur de la commande ;

Cas de test : lecture du nombre d'utilisateurs

Le nom de ce test est `local_user_get`. Son objectif est de vérifier, juste après le chargement des modules, que le compteur est bien initialisé à 0.

Cas de test : la machine est-elle libre ?

Le nom de ce test est `local_user_isfree`. L'objectif est de vérifier que, avec un compteur qui est bien initialisé à 0, les modules remontent bien l'information que la machine est libre.

Cas de test : la machine est-elle utilisée ?

Le nom de ce test est `local_user_isused`. L'objectif est symétrique, et s'assure simplement que les fonctionnalités de base sont correctes. On vérifie donc qu'avec un compteur à 0, la machine est bien reconnue comme non utilisée.

Cas de test : connexion d'un utilisateur

Le nom de ce test est `local_user_simpleconnection`. Nous passons à l'étape supérieure. Nous simulons, par écriture dans le bon fichier de *ProcFS*, la connexion d'un utilisateur interactif sur l'hôte physique qui héberge notre nœud Kerrighed. Nous vérifions ensuite que les modules ont bien pris en compte cette information.

Cas de test : la machine n'est pas libre

Le nom de ce test est `local_user_simplefree`. Notre utilisateur est toujours considéré comme connecté, nous vérifions que les fonctions chargées de notifier que la machine est libre prennent bien en compte sa présence.

Cas de test : la machine est utilisée

Le nom de ce test est `local_user_simpleused`. Symétriquement, nous nous assurons que lorsque l'utilisateur est connecté, la machine est bien reconnue comme utilisée.

Cas de test : déconnexion d'un utilisateur

Le nom de ce test est `local_user_simpledisconnection`. Enfin, nous simulons la déconnexion de l'utilisateur interactif, et nous vérifions que cette information est bien prise en compte par les modules.

Cas de test : connexion de trois utilisateurs

Le nom de ce test est `local_user_complexconnection`. Maintenant que nous avons vérifié le fonctionnement avec un utilisateur interactif, nous nous assurons que le cas de la connexion de plusieurs de ces utilisateurs est bien pris en compte. Nous simulons donc trois connexions, et vérifions que le compteur les a bien compatibilisées.

Cas de test : déconnexion de cinq utilisateurs

Le nom de ce test est `local_user_complexdisconnection`. Enfin, nous nous assurons que la déconnexion de tous nos utilisateurs est bien prise en compte par tous les modules. Nous en profitons pour vérifier que le compteur se comporte correctement, puisque nous simulons cinq déconnexions, alors que seules trois connexions ont eu lieu. Il convient donc de s'assurer que la valeur du compteur reste nulle, et qu'elle ne devient pas négative, ou incohérente.

4.4.2 Test du module de probe User

Maintenant que les modules noyau qui sont exploités par cette probe sont testés, il convient de vérifier que la probe en elle-même fonctionne correctement.

Configuration

La phase de configuration du script de test se charge de vérifier deux points. D'abord, elle s'assure que les modules noyau *Local User Presence* et *Local User Notifier* sont bien présents. Ensuite, elle vérifie que le module de probe `user_probe` est bien chargé.

Cas de test : Connecté

Les modules sur lesquels s'appuie cette probe ayant déjà été testés, un seul cas de test est proposé, pour vérifier la cohérence des données remontées par la probe. Le test consiste donc à comparer ce que retourne la probe source `connected` et à comparer cette valeur à ce qui est retourné par le fichier `get` exposé dans *ProcFS* par le module *Local User Notifier*.

4.4.3 Test du module de probe Load

Ce script est chargé de tester le bon fonctionnement des probes sources exposées par le module de probe *Load*. En réalité, seule la probe source `active_tasks` est testée, en premier lieu parce qu'à l'origine ce module n'exposait que cette probe source, et également parce qu'aucun moyen n'a été trouvé pour accéder à l'information exposée par la probe source `process_jiffies` pour valider son fonctionnement.

Configuration

La phase de configuration du test s'assure que le module `load_probe` est bien chargé.

Cas de test : Tâches actives

Pour valider le fonctionnement de la probe source `active_tasks`, nous comparons la valeur que nous retourne cette dernière avec les informations exposées dans `/proc/loadavg`.

4.4.4 Test du module de probe Processsize

Ce test s'assure que la probe source `process_size` retourne une information correcte.

Configuration

Au cours de cette partie de script, on s'assure que le module de probe `processsize_probe` est bien chargé.

Cas de test : taille de *init*

Nous validons ensuite le fonctionnement de la probe source en comparant la taille en pages mémoires qui nous est retournée par cette dernière, avec la valeur du champ *VmSize* exposé dans `/proc/<PID>/status`. Elles doivent être égales. Attention, le champ *VmSize* doit être converti en pages mémoires : il faut donc le multiplier par 1024, puis le diviser la la taille d'une page mémoire telle que retournée par la commande `getconf PAGESIZE`.

4.4.5 Test du module de probe CPUSpeed

Ce dernier script a pour objectif de valider le fonctionnement de la probe *CPUSeed*, qui expose les probes sources `connected` et `speed`.

Configuration

Afin de pouvoir effectuer ce test, plusieurs points doivent être vérifiés :

1. S'assurer que le module de probe `cpuspeed_probe` est bien chargé ;
2. Vérifier la présence de l'utilitaire `bc` ;
3. Configurer les capacités Kerrighed pour ne voir que les processeurs locaux.

Ce dernier point est critique : en effet, on ne veut valider que le fonctionnement en local. Pour plusieurs raisons. D'abord, au moment de l'écriture des tests, il n'était pas encore possible d'accéder aux valeurs distantes d'une probe source depuis l'espace utilisateur. Ensuite, si le fonctionnement est validé pour les nœuds locaux, alors on sera sûr que chaque nœud retourne des valeurs cohérentes, et donc les valeurs sur l'ensemble du cluster le seront. Enfin, il nous est impossible, dans `/proc/cpuinfo` de savoir à quel nœud appartient un processeur.

Nous forçons donc Kerrighed à ne nous afficher que les processeurs locaux, grâce à la commande `krgcapset -d +SEE_LOCAL_PROC_STAT`.

Cas de test : Nombre de processeurs locaux

Il s'agit ici simplement de vérifier que le nombre de processeurs sur le nœud est correctement rapporté par la probe source `connected`. Nous comparons pour ceci le nombre de fois où apparaît la chaîne `processor` dans la sortie de la commande `cat /proc/cpuinfo`, avec la valeur retournée par la probe source.

Cas de test : Vitesse de chaque processeur local

Après avoir validé que le nombre de processeurs est correct, nous vérifions que les vitesses lues sont les bonnes. Nous opérons également une comparaison, entre la valeur retournée pour chacun des processeurs, par la probe source `speed`, avec l'information présente dans `/proc/cpuinfo`. Nous utilisons `bc` afin de ramener les valeurs en MHz qui sont présentes dans *Proc* à des valeurs en KHz, comme remontées par la probe source. Nous effectuons également une troncature de sorte à ne pas garder de virgule dans nos vitesses.

4.5 Portage des algorithmes d'ordonnancement

À présent, nous avons à notre disposition la totalité des informations nécessaires pour le calcul des ordonnancements, c'est-à-dire pour réaliser le besoin C.2. Cependant, nous ne sommes pas encore en mesure de pouvoir écrire le module se chargeant d'appliquer la politique d'ordonnancement voulue.

En effet, une caractéristique propre du framework Kerrighed à propos des requêtes sur les probes distantes via un port est l'aspect asynchrone. Les appels à la fonction `scheduler_port_get_remote_value()` ne sont pas bloquants. Ceci nous amène donc à nous pencher sur l'utilisation des filtres, avant d'entamer la politique.

Dans la suite de cette section, nous allons donc présenter le travail réalisé pour implémenter les modules suivants :

- Filtre *RBT Cache* ;
- Politique *RBT*.

4.5.1 Module de filtrage *RBT Cache*

Le code de ce module de filtrage est proposé dans l'annexe G.1. Comme cela a déjà été précisé antérieurement, les modules de filtrages ont pour rôle de s'intercaler entre un module de probe, et le port d'un module de politique d'ordonnancement. Classiquement, leur rôle est simplement de ne laisser passer que certaines valeurs de la probe, comme par exemple dans le module inclus de base avec Kerrighed `threshold`. Avant d'aller plus loin, il convient de se souvenir que filtres, probes et politiques sont « connectés »ensembles grâce à de simples liens symboliques, qui permettent le passage de l'information.

Mais une autre utilisation des filtres est possible. Comme cela a été dit auparavant, le mécanisme d'obtention des valeurs d'une probe distante connectée sur un port du module de politique est asynchrone. Dès lors, cela implique que pour arriver à *obtenir* effectivement notre valeur, plusieurs tentatives peuvent être nécessaires. Le problème commence à se profiler à l'horizon : cela nécessite une logique supplémentaire. Logique qui n'est pas complexe, mais qui rajoute une complexité inutile, et potentiellement mutualisable.

C'est pour répondre à ce problème que l'utilisation des modules de filtrage en tant que mécanisme d'interfaçage et de mise en cache s'est développée au sein du projet Kerrighed. C'est particulièrement le cas avec le module `remote_cache_filter`. Nous avons donc proposé le module `rbt_cache_filter`, qui opère de manière similaire à `remote_cache_filter`. En réalité, la seule différence concerne le type des données à récupérer et à mettre en mémoire. Ce type – `unsigned int` – n'était pas pris en charge dans `remote_cache_filter` – qui gère des `unsigned long` –, d'où la décision d'écrire un autre module.

Outre différentes macros définies par Kerrighed pour simplifier la déclaration et la création des structures de données correspondantes pour le filtre, telles que `BEGIN_SCHEDULER_FILTER_TYPE`/`END_SCHEDULER_FILTER_TYPE`, et les méthodes correspondantes à implémenter[6], c'est surtout dans la fonction `try_get_remote_values()` que le vrai travail se joue.

L'optique de cette dernière est de parcourir régulièrement l'ensemble des nœuds du cluster, dans un ordre qui ne nous intéresse pas – on se repose encore une fois sur des fonctions de Kerrighed pour s'occuper de cet aspect des choses – et de lire la valeur de la probe qui est liée à notre filtre. Ces valeurs sont stockées en local, dans un simple tableau indexé sur les identifiants des nœuds.

De cette manière, lorsqu'une requête demandant la valeur arrive en provenance d'un port de politique sur notre filtre, ce dernier peut directement renvoyer la dernière valeur qu'il a en mémoire. Ainsi, toute la « complexité »liée à la gestion de l'aspect asynchrone de la récupération distante des valeurs est centralisé en un point. En sus, on bénéficie d'un mécanisme de cache des valeurs. Et dans tous les cas, la fréquence des mises à jours est paramétrable grâce à l'exposition d'un attribut ConfigFS nommé `polling_period`.

[6] Le lecteur pourra trouver ces informations dans le fichier d'entête `<scheduler/core/filter.h>` du projet Kerrighed

4.5.2 Module de politique *RBT*

Notre périple commence à arriver à son terme, puisque nous attaquons la dernière partie de l'ordonnancement, qui n'est pas nécessairement la plus triviale. Il s'agit à présent d'expliquer l'implémentation réalisée de l'algorithme RBT, disponible en annexe G.2. Par ailleurs, le lecteur pourra se référer aux annexes de [Pér08] où une autre implémentation est disponible. Il s'agit de celle qui avait été réalisée pour Kerrighed 2.2.0.

Le portage qui a été réalisé depuis ce code d'origine a essayé de rester le plus fidèle au premier. Nous retrouvons ainsi sans surprise les quatre fonctions suivantes, dont nous allons préciser le rôle, et le travail réalisé autour :

- `rbt_policy_exec()` ;
- `rbt_find_a_task()` ;
- `rbt_find_a_node()` ;
- `toggle_next_node()`.

Par ailleurs, nous constaterons les aménagements nécessaires par rapport au code réalisé pour Kerrighed 2.2.0.

La création des structures de données nécessaires pour le module de politique n'a rien de précisément excitant. Il s'agit principalement de créer et initialiser les ports sur lesquels viendront se connecter probes et filtres. Nous avons également créé une structure qui référence ces ports, ainsi qu'un autre élément sur lequel nous reviendront : `struct delayed_work`. Ces ports et une référence vers la structure sont regroupés au sein de `struct rbt_policy`, dont le code est disponible dans l'annexe G.2.

Le problème de l'exécution

Avant de pouvoir écrire l'algorithme, une autre question s'est posée : comment l'exécuter régulièrement ? Après avoir compulsé [CRKH05a], et en avoir discuté avec les développeurs de Kerrighed, il nous a semblé le plus pertinent de s'appuyer sur la structure des threads noyaux : `struct work_struct`.

L'utilisation de cette structure se fait en deux étapes : initialisation, et programmation. L'initialisation, au moyen de `INIT_DELAYED_WORK` permet d'associer à la structure de données un pointeur de fonction à exécuter le moment venu. Ensuite, on programme l'exécution grâce à `schedule_delayed_work`, en précisant *quand* on souhaite voir l'action se dérouler.

Dans notre cas, c'est la fonction `rbt_policy_exec()` qui est ciblée.

Fonction d'exécution de la politique : `rbt_policy_exec()`

La première opération importante réalisée dans cette fonction consiste à retrouver la structure de données les `struct rbt_policy`, qui référence notamment tous les ports de données. Il faut en effet savoir que la fonction exécutée par un thread noyau doit avoir comme seul et unique paramètre un pointeur sur `struct work_struct`. C'est pourquoi nous devons le référencer dans notre structure `struct rbt_policy` : nous pouvons utiliser la macro `container_of()` sur cette structure associée à l'exécution de la tâche pour retrouver notre structure de donnée.

Une fois ce concept assimilé, le reste de la fonction est plutôt similaire à ce qui avait déjà été implémenté pour Kerrighed 2.2.0. Seule la manière de récupérer les données change réellement, puisque nous passons par les ports au lieu de faire appel à des symboles exportés.

L'action se déroule donc en trois temps :

1. Initialisation, dans la première boucle ;
2. Calcul d'une solution optimale du problème relaxé, dans la deuxième boucle. Cette solution n'est pas nécessairement réalisable ;
3. Tentative d'atteindre la solution optimale dans la dernière boucle.

Il faut noter que nous ne sommes plus sujets à des limitations telles que l'impossibilité de re-migrer un processus qui a déjà été migré.

Notons la toute dernière étape de la fonction, qui consiste à reprogrammer l'exécution de notre fonction. C'est nécessaire et inhérent à la manière de faire de l'espace noyau : il n'a pas été possible de trouver une structure de données s'occupant d'exécuter régulièrement une fonction.

Fonction de sélection d'une tâche : `rbt_find_a_task()`

L'objectif de cette fonction est de choisir une tâche à migrer. Il y a assez peu de modifications par rapport aux travaux menés sur Kerrighed 2.2.0 : il convient de parcourir l'ensemble des tâches présentes localement, en vérifiant certains cas – capacité de migrer le processus disponible, pas de migration en cours, processus utilisable – puis on choisit simplement le plus « gros »au sens de la consommation mémoire.

Fonction de sélection d'un nœud : `rbt_find_a_node()`

Le rôle de cette fonction est de choisir un nœud où migrer le processus que l'on a sélectionné au point précédent. Là encore, les seules vraies nouveautés sont liées à l'utilisation des ports de Kerrighed, pour lire les valeurs des machines distantes. Notons aussi que certains paramètres inutiles ont été retirés du prototype.

Fonction de changement de nœud : `toggle_next_node()`

Cette fonction a été ré-implémentée par simple mimétisme avec le code d'origine, et pour faciliter la lecture entre les deux versions. En pratique, elle ne fait que se reposer sur une fonction du framework Kerrighed `krgnode_next_possible_in_ring()` qui a exactement le même rôle. Ceci étant, cette abstraction nous laisse la possibilité, si besoin s'en fait sentir, de changer la méthode de déplacement au sein des nœuds pour utiliser une autre façon de faire qu'un simple tour de cercle.

4.6 Signalisation de l'utilisation physique des nœuds

Nous entrons maintenant dans une autre contrée du projet. Il s'avère que la mesure de charge mise au point dans [Pér08] prend en considération la présence ou non d'utilisateurs dit « interactif », i.e. des personnes physiques connectées à une machine.

Or, en pratique, dans les travaux de Mathieu Dabert [Dab08], la prise en compte de ce critère a été effectuée de manière succincte, et même si elle est pleinement fonctionnelle, elle se limite au nœud Kerrighed. Cependant, les déploiements prévus concernent de mettre en place Kerrighed dans une machine virtuelle. Dans ce cadre, c'est la machine physique pour laquelle la présence ou l'absence d'un utilisateur nous intéresse.

Nous allons donc présenter ce qui est nécessaire à la mise en place de ce mécanisme de signalisation *en pratique*, puis nous présenterons comment nous l'avons implémenté.

4.6.1 Interface noyau nécessaire

C'est le premier besoin à combler, présenté dans le cahier des charges en annexe C.3.1. Il s'agit de réfléchir à l'interface noyau nécessaire afin de réaliser ce mécanisme. Dans la sous-section suivante 4.6.1, nous parlerons plus précisément de la manière dont a été implémentée cette interface.

Cette interface n'est pas à confondre avec celle dont nous parlerons dans la section 4.6.2.

Le problème auquel nous faisons face est d'être en mesure de garder en mémoire, en espace noyau, l'information concernant la disponibilité de la machine physique liée à notre machine virtuelle. Dans le cahier des charges, nous avions exposé l'idée d'implémenter un simple compteur exposant une interface en

espace noyau. Après réflexion, il nous a semblé que cette idée était simple et pertinente, nous l'avons donc conservée.

Module d'information de présence *Local User Presence*

Le code de ce module disponible en annexe F.3. Nous avons par ailleurs repris la sémantique exposée dans le cahier des charges C.3.1 lors de l'expression de ce besoin.

Nous définissons donc simplement un entier, qui sera notre compteur. Pour manipuler cet entier, nous définissons un ensemble de fonctions, qui vont réaliser des opérations unitaires :

- `local_user_presence_user_connection` ;
- `local_user_presence_user_disconnection` ;
- `local_user_presence_user_connected` ;
- `local_user_presence_node_free` ;
- `local_user_presence_node_used`.

Toutes ces fonctions sont exportées, ce qui permet à n'importe quel module noyau de s'en servir. Pour éviter les soucis, nous utilisons également un mutex. Il s'agit simplement de protéger les opérations qui modifient la valeur du compteur, pour éviter les problèmes en cas d'accès en écriture concurrents.

4.6.2 Mécanisme de communication avec la machine virtuelle

Il s'agit de répondre au besoin exprimé en annexe C.3.2. Nous avons, dans la section précédente, exposé notre manière de conserver l'information de la présence d'un ou plusieurs utilisateurs sur la machine physique. Maintenant, nous nous concentrons sur le problème de savoir comment permettre à l'espace utilisateur d'exploiter simplement cette interface.

Notons que pour l'instant la seule interface disponible est en espace noyau. Une solution simple pour contourner cette contrainte avait été proposée dans le cahier des charges, et a été considérée comme pertinente. Il s'agit de créer une arborescence dans le système de fichier virtuel `ProcFS`. Nous réutiliserons les chemins proposés : sous l'arborescence `/proc/kerrighed/interactive_user`, nous créons les fichiers suivants :

- `get` ;
- `isfree` ;
- `isused` ;
- `connection` ;
- `disconnection`.

Module pour notification de présence *Local User Notifier*

Le code de ce module disponible en annexe F.4. L'implémentation de ce module n'a posé aucune difficulté particulière. La seule chose qui pourrait amener à penser à des soucis concerne le rattachement à `/proc/kerrgihed`. En effet, ce chemin est créé par le module `Kerrighed`.

Pour créer notre arborescence, nous commençons par créer une racine, qui correspond au chemin `/proc/kerrighed/interactive_user`, grâce à l'appel suivant : `create_proc_entry("kerrighed/interactive_user", S_IFDIR | S_IRUGO | S_IXUGO, NULL)`.

Les paramètres permettent de définir le type – `S_IFDIR` : répertoire –, ainsi que les droits de lecture et d'exécution à tous les utilisateurs. Si jamais le module `kerrighed` n'est pas chargé, la création de cette racine échouera. Si ce n'est pas le cas, nous créons les fils qui correspondent aux actions que l'on veut créer.

Les « pseudos »fichiers `get`, `isfree`, `isused` sont uniquement accessibles en lecture, et créés grâce à la fonction `create_proc_read_entry`. En plus de donner les droits, il nous précise la racine à laquelle les attacher. C'est celle préalablement créée. Enfin, nous spécifions également les fonctions cibles de ces appels.

Pour permettre la communication des évènements de connexion et déconnexion d'un utilisateur, nous avions retenu la possibilité d'écrire – la valeur en elle-même importe peu – sur les entrées idoines du `ProcFS`.

À la différence de la création des entrées en lecture seule, pour celle en écriture il est nécessaire d'utiliser `create_proc_entry`, puis de changer la valeur du membre `write_proc` de la structure `struct proc_dir_entry` qui nous est retournée par le précédent appel, pour indiquer la fonction en charge de la gestion de cette entrée du système de fichier.

Les fonctions cibles réutilisent directement l'interface en espace noyau proposée à la section 4.6.1.

4.6.3 Obtention de l'information de présence

L'idée est assez simple : nous avons tout le nécessaire en espace noyau – et bientôt en espace utilisateur – dans notre machine virtuelle pour être en mesure de notifier la présence d'une – ou plusieurs – personne(s) physique(s) utilisant la ressource.

Il reste cependant à être capable de pouvoir transmettre l'information effective, depuis la machine physique, vers la machine virtuelle. Dans cette optique, on doit nécessairement effectuer une modification assez intrusive sur les systèmes qui hébergent les machines virtuelles, de sorte à pouvoir être informé de la connexion et déconnexion physique des utilisateurs.

Malgré tout, cette modification peut être réalisée sans trop de difficultés : l'ensemble des systèmes d'exploitation possède un système qui gère la connexion et la déconnexion des utilisateurs. De plus, les systèmes déployés dans les salles qui nous intéressent sont restreints, nous ne trouvons que du Linux et du Windows. Pour le premier c'est la bibliothèque PAM[SS, Mor] – Pluggable Authentication Modules – qui s'occupe de gérer tout ceci. Notons au passage que PAM est disponible pour une très large majorité des Unix existants, et a été initié par Sun Microsystems. L'implémentation pour Linux est Linux-PAM[Mor] Sous Windows, c'est le service WinLogon[Win] qui est dévolu à cette tâche, et qui expose le nécessaire pour obtenir les informations que l'on souhaite.

Nous détaillerons ces deux modules par la suite. Notons que, afin de dégager un peu de temps pour pouvoir travailler sereinement sur les modules noyau, l'étude et le développement de ces modules a été externalisé en tant que projet de Système d'Exploitation[PR09].

Module de notification pour PAM

Il s'agit de combler le besoin qui a été décrit dans le cahier des charges, et qui est disponible en annexe C.3.3.

Module de notification pour WinLogon

Nous présentons à présent le module qui est dévolu à répondre au besoin décrit dans l'annexe C.3.4.

4.6.4 Communication PAM/WinLogon vers Machine Virtuelle

C'est le dernier composant dont nous avons besoin pour être en mesure de faire le lien entre machine réelle et machine virtuelle, de sorte à notifier la présence d'utilisateur(s) physique(s).

Nous l'avons spécifié comme un petit service qui tourne en espace utilisateur, et qui ouvre une socket réseau. Sur cette dernière, les modules PAM4.6.3 et WinLogon4.6.3 viendront écrire l'information de connexion ou déconnexion d'un utilisateur, qui sera transmise directement au noyau.

Le protocole de communication entre les modules et ce service n'a pas besoin d'être très évolué. Un mécanisme très simple tel que l'envoie du caractère *C* pour notifier la connexion, et *D* pour notifier la connexion a été retenu.

4.6.5 Schéma récapitulatif

Maintenant que nous avons décris tous les éléments nécessaires pour permettre la signalisation à Kerrighed, et à notre module de calcul de la charge, de la présence d'utilisateur(s) interactif(s), il convient de proposer un schéma 4.3 permettant d'en avoir une vue d'ensemble.

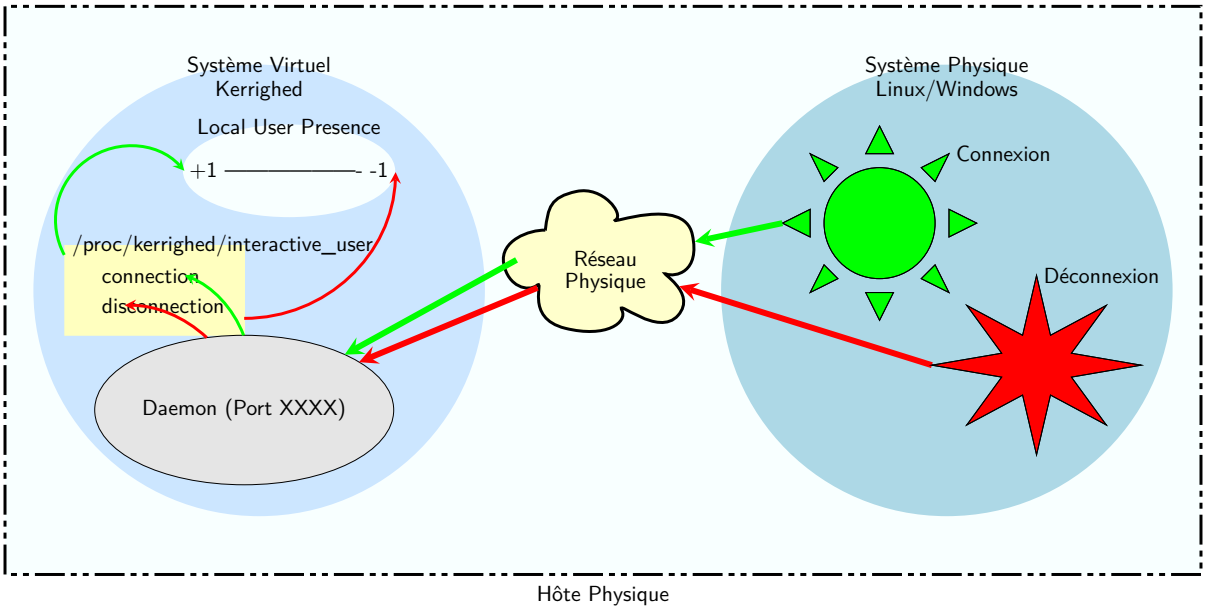


FIG. 4.3 – Schéma récapitulatif de la remontée de l'information de présence d'utilisateur interactif

4.7 Outils nécessaires en espace noyau pour l'ordonnancement en espace utilisateur

Afin de répondre au besoin qui avait été exprimé dans le cahier des charges, et disponible en annexe C.4 étudions les éléments nécessaires, du côté de l'espace noyau, pour permettre le calcul d'un ordonnancement en espace utilisateur.

Nous commencerons par lister tous les éléments nécessaires, puis ferons un point pour chacun afin de voir ce qui doit être implémenté pour combler les manques actuels.

4.7.1 Récapitulatif des outils nécessaires en espace noyau pour l'ordonnancement en espace utilisateur

Afin de prendre des décisions d'ordonnancement depuis l'espace utilisateur, et de les invoquer également dans ce contexte, nous avons besoin des éléments suivants :

- Un appel système permettant de demander la migration d'un processus vers un nœud ;
- Un mécanisme permettant d'accéder aux données des probes depuis l'espace utilisateur.

Nous allons analyser l'état de chacun de ces éléments dans les sections 4.7.2 et 4.7.3.

4.7.2 Migration de processus depuis l'espace utilisateur

Pour être en mesure d'appliquer des décisions d'ordonnancement, il faut absolument que l'on soit en mesure de faire migrer, depuis l'espace utilisateur, un processus vers un autre nœud du cluster.

Il faut donc que les primitives de migrations disponibles en espace noyau soient exportées à l'espace utilisateur, quelque soit la manière de faire.

Actuellement, Kerrighed propose un appel système qui utilise exactement le même code que ce que nous utilisons pour réaliser l'opération de migration en espace noyau. Il s'agit de l'appel système `migrate(2)`, implémenté par la fonction `sys_migrate_process`.

De plus, un programme en espace utilisateur éponyme de l'appel système fait partie des outils fournis par Kerrighed, et que les tests quant à son utilisation ont été couronnés de succès : nous avons réussi à l'utiliser pour migrer des processus.

4.7.3 Accès aux informations des probes depuis l'espace utilisateur

Exposons maintenant l'autre problème que nous devons résoudre. Si l'on veut résoudre notre problème d'ordonnancement pour déplacer des processus entre nœuds du cluster, il nous est nécessaire de pouvoir lire certaines informations. Ces informations proviennent de l'espace noyau. Dans le but de s'intégrer le mieux possible au mécanisme de remontée d'informations présent de Kerrighed, il parait assez logique de vouloir ré-exploiter les probes afin de permettre la lecture des informations de l'espace utilisateur.

Ceci est d'autant plus cohérent que, finalement, grâce à ConfigFS, ces informations sont simplement des répertoires et des fichiers à traverser et à lire. Un bon point donc, puisque l'on peut garder la philosophie propre à Kerrighed : simplifier au maximum la manière d'accéder à l'information. Nous pouvons donc accéder aux données via un schéma simple : lire le fichier `/config/krg_scheduler/probes/<probename>/<probesourcename>/value`.

Mais le lecteur pourrait faire une observation : *quid* de la remontée d'information des machines distantes ? Il est bien évidemment nécessaire de pouvoir accéder aux données des machines distantes pour être en mesure de prendre une décision. Et cela, quelque soit les aspects distribués liés au problème : il nous sera impossible de résoudre le problème sans avoir ces informations. Sauf à utiliser un algorithme travaillant au hasard, ou un simple *round-robin*. Qui, certes, fonctionnent, mais sont loin d'être optimaux.

Plusieurs options se sont présentées à nous pour résoudre ce problème. Deux options principales, que nous avons étudiées :

1. Réaliser un service en espace utilisateur, exécuté sur chaque nœud du cluster, et chargé de deux choses :
   - Permettre l'accès à distance des valeurs des probes sources de ce nœud ;
   - Fournir, localement, une abstraction permettant de lire les valeurs sur les nœuds distants.
2. Modifier la gestion actuelle du ConfigFS dans Kerrighed afin de permettre directement l'accès aux informations des nœuds distants.

Chacune de ces options présente des avantages et inconvénients, présentés en section 4.7.3.

Comparaison des solutions

Il s'agit de montrer les avantages et inconvénients de chacune des deux hypothèses, et finalement d'expliquer pourquoi nous avons choisi la seconde.

Écriture d'un service utilisateur de mise à disposition et requête À titre de rappel, il s'agit de spécifier et d'écrire un service en espace utilisateur, qui fonctionnerait sur un modèle distribué : un agent s'exécute sur chaque nœud du cluster, et donne ainsi accès aux informations des probes source sur ce nœud pour les autres machines.

Il est également nécessaire de spécifier une bibliothèque ou autre mécanisme chargé d'abstraire l'aspect récupération des données. Ceci pourrait être également réalisé par le précédent service, qui aurait ainsi une copie locale de toutes les données de tous les nœuds. Certes, pas nécessairement à jour, mais c'est le jeu des systèmes distribués.

Avantages

1. Aucune modification à faire en espace noyau ;
2. Prototypage très rapide.

Inconvénients

1. Nécessite d'écrire un nouveau service ;
2. Nécessite de spécifier un protocole de communication ;
3. Mécanisme d'utilisation plus complexe.

Modification de ConfigFS pour exposer l'information distante L'idée derrière cette approche est de s'intégrer bien mieux dans le framework actuel. La proposition est de modifier ConfigFS de sorte à créer toute une hiérarchie sous chaque répertoire de probe source, correspondant à chaque nœud présent dans le cluster.

Ainsi, nous aurions en plus de `/config/krg_scheduler/probes/<probename>/<probesourcename>/value` un ensemble de fichiers `/config/krg_scheduler/probes/<probename>/<probesourcename>/<nodeId>/value`, où *nodeID* est un répertoire pour chaque nœud possible du cluster.

Avantages

1. Intégration maximale dans le framework actuel de Kerrighed.

Inconvénients

1. Modification(s) en espace noyau ;
2. Plus long à mettre en place.

Choix retenu : modification de ConfigFS Finalement, après réflexion et consultation des développeurs Kerrighed, nous nous sommes portés sur la seconde option. En effet, même s'il est nécessaire de passer plus de temps pour comprendre comment faire, nous espérons gagner sur d'autres tableaux : nous n'avons pas à spécifier de protocole de communication, puisque nous comptons nous baser sur l'architecture sous-jacente de Kerrighed qui intègre déjà ce concept, le code à écrire, s'il est plus critique – parce qu'en espace noyau – est par contre bien plus succinct.

Enfin, ce choix, de par sa meilleure intégration à l'actuelle vision de Kerrighed facilitera à la fois son inclusion dans le projet, et l'utilisation du résultat : on pourra très facilement lire la valeur des probes sources à distance. Nous expliciterons son implémentation dans la section 4.8.

4.8 Implémentation de la modification de ConfigFS

Maintenant que le problème a été présenté, et que la solution retenue a été exposée dans son principe, ainsi que la raison de sa retenue, nous allons nous intéresser à commenter la réalisation effective de l'implémentation de cette modification.

Nous commencerons par identifier précisément toutes les étapes nécessaires afin de mener à bien l'implémentation de cette modification, dans la section 4.8.1. Une fois les étapes identifiées, nous présenterons les points importants concernant l'implémentation pratique. Ceci sera fait dans les sections 4.8.2, 4.8.3, 4.8.4, 4.8.4, 4.8.4. À l'issue de ces explications, nous proposerons des schémas qui présentent les modifications opérées, dans la section 4.8.5, et plus précisément, les schémas disponibles en sections 4.8.5 et 4.8.5.

4.8.1 Identification des modifications à réaliser

Nous avons pu, précédemment, exposer le but des modifications. Il convient maintenant de réfléchir à comment il va falloir procéder. Auparavant, il a été convenu que les modification s'opèreront dans les répertoires correspondant aux probes sources. Nous devons, dans un premier temps, adapter ConfigFS de sorte à créer des répertoires correspondant aux nœuds du cluster qui sont en ligne, et placer dans ces répertoire l'attribut `value` correspondant à la probe source. Nous présenterons en pratique comment opérer cette modification dans la section 4.8.2. Idéalement, ce changement ne doit pas empêcher le fonctionnement précédent, c'est à dire qu'il doit toujours être possible de lire la valeur locale grâce au chemin `/config/krg_scheduler/probes/<probename>/<probesourcename>/value`.

Une fois que ces répertoires auront été créés, nous devons ensuite procéder à une autre modification : il nous faut diriger les appels en lecture sur le fichier contenu dans le répertoire correspondant au nœud sur une autre fonction que celle habituellement dévolue à la lecture directe de la valeur locale. Le rôle de cette

fonction sera de déduire le nœud concerné, de lire la valeur, et de la retourner. Nous présenterons cette fonction dans la section 4.8.3.

Maintenant que nous saurons comment « détourner »les appels à ces fichiers représentant les nœuds distants, il conviendra bien sûr d'implémenter de manière effective la lecture des valeurs de la probe source concernée. Cette partie sera détaillée dans la section 4.8.4.

4.8.2 Création des répertoires peuplés correspondant aux nœuds

Comme cela a été présenté, nous allons modifier le répertoire correspondant à une probe source. La mise en place de l'infrastructure pour ConfigFS correspondant à ce répertoire a lieu dans la fonction `scheduler_probe_source_create()` présente dans le fichier `modules/scheduler/core/probe.c` du projet Kerrighed. Il convient donc de travailler dans celle-ci, de sorte à ce que n'importe quelle probe source soit pourvue de ces fonctionnalités.

À titre de rappel, l'utilisation qui est faite de ConfigFS par Kerrighed repose beaucoup sur les `config_group`. Il en est un « type »un peu particulier. Une petite observation à cette structure révèle la présence d'un membre de même type, nommé `default_groups`. Renseignements pris, il s'avère qu'à la création d'une arborescence ConfigFS, ce membre est exploité par la fonction `populate_groups` de sorte à peupler automatiquement le répertoire.

Or, un point qui nous intéresse particulièrement est que la finalité de la fonction `scheduler_probe_source_create` que nous examinons est de peupler en attributs la probe source passée en paramètre, puis de faire appel à `scheduler_pipe_init` qui se charge de mettre en place le nécessaire de communication. Et également de définir la propriété `default_groups` de notre probe source.

Cette fonction contient tous les éléments dont nous avons besoin pour continuer notre travail. Nous allons donc continuer sur les points suivants :

- Recopier les structures `config_group` correspondantes aux attributs de la probe source ;
- Modifier le nécessaire de sorte à créer le répertoire ;
- Les injecter dans notre probe source.

En réalité, ces deux premiers points peuvent se faire en une seule opération. L'idée est de recréer de nouvelles structures de données pour nos nouveaux répertoires, de nouveaux *pipes* : `scheduler_pipe_init(&node_pipes[curnode]->pipe, sNodeId, remote_pipe_type, &tmp_ps->source, NULL, NULL)`.

La nouvelle structure cible est `&node_pipes[curnode]->pipe`, et `sNodeId` est tout simplement une chaîne correspondant à un identifiant de nœud. La structure correspondant à `remote_pipe_type` répond à un autre problème. Enfin, le paramètre précédent les deux `NULL` correspond à la source de données. L'appel que nous venons de décrire est exécuté pour chaque nœud en ligne, de sorte à créer une structure de données pour chacun. Nous pouvons ensuite récupérer l'élément `config_group` correspondant, grâce à sa présence dans le *pipe* ainsi créé : `&node_pipes[curnode]->pipe.config`. C'est cette valeur qui sera injectée dans le dernier appel à `scheduler_pipe_init` pour la création de la probe source.

On pourrait s'interroger sur la finalité de la variable `node_pipes`, où nous stockons les *pipes* créés. Il s'agit d'un ensemble de structures de type `struct node_pipe`, présenté en code 4.7, dont l'utilité est en rapport avec la suite des modifications.

```
1 struct node_pipe {
2         struct scheduler_probe_source *probe_source;
3         struct scheduler_pipe pipe;
4         kerrighed_node_t node;
5 };
```

Code 4.7 – Structure node_pipe

4.8.3 Détournement des appels sur les fichiers des répertoires correspondant aux nœuds

Dans la section précédente, nous avons vu comment nous avons été en mesure de créer un nouveau sous-répertoire par nœud en ligne pour chacune des probes sources. Lors de l'explication de la manière de créer ces derniers, nous avons parlé d'une structure passée en paramètre, à savoir `remote_pipe_type`. Son utilité va être détaillée maintenant. Il convient de savoir que le troisième paramètre de la fonction `scheduler_pipe_init` correspond à une structure qui décrit les opérations possibles sur les attributs de l'élément `config_group` que nous nous afférons à créer.

La structure classiquement passée en paramètre est `&type->pipe_type`. Si l'on observe comment elle est créée, comme cela est possible dans le code 4.8, on constate la présence d'une autre structure, `scheduler_probe_source_item_ops`, dont on trouve la déclaration un peu plus haut dans le même fichier source. Cette déclaration est disponible en code 4.9.

C'est cette dernière qui contient les pointeurs pour les opérations *show* et *store*, correspondant respectivement à une lecture et à une écriture de l'attribut ConfigFS. C'est pourquoi nous créons cette variable `remote_pipe_type`, de la même manière que dans le code 4.8, mais en spécifiant une autre structure pour les opérations : `scheduler_probe_source_remote_item_ops` dont le détail est disponible dans le code 4.10. Le code de la fonction chargée de l'exécution des appels en lecture, `scheduler_probe_source_attribute_show_remote` est proposé en annexe I.3.

Cependant, nous n'avons pas encore tous les éléments afin de pouvoir faire notre requête à distance. En effet, nous sommes en mesure d'être appelé sur demande de lecture, mais il reste une question à laquelle nous ne pouvons pas répondre pour le moment : à quel nœud cette requête s'adresse-t-elle ?

C'est là qu'intervient la structure `struct node_pipe` qui est disponible dans le code 4.7. La fonction `scheduler_probe_source_attribute_show_remote` reçoit en paramètre le `config_item` correspondant au fichier sur lequel la requête de lecture a été effectuée. Heureusement pour nous, la structure `struct node_pipe` contient en son sein la structure `struct scheduler_pipe`, qui elle-même fait référence à notre `config_item`. Ainsi, en utilisant l'appel `container_of(to_scheduler_pipe(item), struct node_pipe, pipe)` où *item* est notre paramètre de type `config_item`, nous sommes en mesure de récupérer la structure correspondante.

Une fois cette dernière obtenue, nous pouvons directement lire l'identifiant du nœud cible, information que l'on avait bien pris soin de renseigner à la création de la dite structure.

```
type->pipe_type = (struct scheduler_pipe_type)
  SCHEDULER_PIPE_TYPE_INIT(owner,
      &scheduler_probe_source_item_ops, NULL,
      &type->source_type, NULL);
```

Code 4.8 – Création d'une structure scheduler_pipe_type

```
static struct configfs_item_operations scheduler_probe_source_item_ops = {
  .show_attribute = scheduler_probe_source_attribute_show,
  .store_attribute = scheduler_probe_source_attribute_store,
};
```

Code 4.9 – Création d'une structure scheduler_probe_source_item_ops

```
static struct configfs_item_operations scheduler_probe_source_remote_item_ops = {
  .show_attribute = scheduler_probe_source_attribute_show_remote,
  .store_attribute = NULL,
};
```

Code 4.10 – Création d'une structure configfs_item_operations pour la lecture à distance

4.8.4 Communication RPC pour lecture à distance

Kerrighed propose une infrastructure de communication de type RPC – Remote Procedure Call – déjà utilisé pour propager les modifications locales opérées dans ConfigFS à tous les nœuds. Dans un soucis de

simplicité, et d'intégration, il nous a paru naturel d'exploiter ce potentiel afin d'être en mesure de pouvoir lire à distance les valeurs de nos probes sources.

L'idée qui sous-tend cette communication est plutôt simple. Nous sommes déjà en mesure de connaître le nœud destinataire d'une requête de lecture, ainsi que de savoir quelle probe source est concernée. Il ne reste qu'à transmettre cette demande à la machine concernée, exécuter en local la demande de lecture de l'information, puis retourner cette valeur.

Toujours dans un soucis de non intrusion maximale, nous avons préféré nous appuyer sur les méthodes `SHOW` des probes sources. Le lecteur pourra constater que les fonctions qui implémentent ces méthodes attendent un paramètre correspondant à une page mémoire où écrire les données. Dans le schéma traditionnel, cette page mémoire est allouée localement et destinée à être ensuite retournée à l'outil en espace utilisateur qui en a fait la demande, via l'appel système `read()`.

Notre idée est d'allouer une page mémoire, et de transmettre cette page en paramètre à la méthode qui agira donc comme habituellement, avec les mêmes contraintes. Une fois cet appel effectué, notre page mémoire est remplie et nous pouvons la retransmettre à la machine en attente de l'information.

Pour permettre ces opérations, nous devons implémenter une procédure d'appel à distance. Ceci implique de déclarer, dans les entêtes de Kerrighed – et plus précisément dans le fichier `modules/rpc/rpcid.h` – un nouveau membre à l'énumération `rpcid`. Étant donné qu'il existe déjà un membre `SCHED_PIPE_GET_REMOTE_VALUE`, nous avons de manière très originale nommé le nôtre comme `SCHED_PIPE_SHOW_REMOTE_VALUE`.

Une fois cet identifiant RPC défini, il reste à implémenter deux éléments :

- La partie serveur, qui est exécutée sur le destinataire de la requête ;
- La partie cliente, qui est exécutée sur l'émetteur de la requête.

Client RPC – « côté nœud souhaitant lire »

Le code de la fonction est disponible en annexe I.3. Le client RPC est l'initiateur de la requête RPC. Il n'a besoin d'envoyer qu'une seule information au serveur RPC, i.e. le destinataire : l'élément `config_item` qui correspond à la probe source de laquelle on veut lire la valeur distante.

Pour y parvenir, Kerrighed propose un moyen assez ingénieux et simple : on sait que, de toute façon, notre probe source existe sur tous les nœuds du cluster. On sait de plus que le chemin est unique. Donc, l'idée consiste à ne transmettre que le nécessaire dans la requête RPC pour être en mesure d'identifier, côté serveur, la structure de données qui correspond, et à aller chercher les données dedans.

Reste à obtenir le `config_item` qui correspond à notre probe source, et non pas à l'attribut sur lequel l'utilisateur vient de faire sa requête. Pour cela, il nous suffit de remonter à la structure `struct scheduler_probe_source` telle que stockée dans la structure `struct node_pipe` depuis notre `config_item`. Cette opération se fait plutôt facilement, grâce à l'appel `probe_source = to_node_pipe(item)->probe_source`. Il nous suffit ensuite de lire le `config_item` correspondant au répertoire parent de l'attribut que l'on vient de tenter de lire, grâce à `cible = &probe_source->pipe.config.cg_item`. Tout ceci peut se résumer en une ligne : `cible = &to_node_pipe(item)->probe_source->pipe.config.cg_item`.

Dès cet instant, nous avons tous les éléments nécessaires. Il ne reste qu'à créer une requête RPC, et à l'envoyer, puis à lire la réponse.

Pour créer notre requête RPC, il suffit de remplir une structure `struct rpc_desc` grâce à `rpc_begin()`. Cette fonction a besoin de deux informations :

- L'identifiant de requête RPC : `SCHED_PIPE_SHOW_REMOTE_VALUE` ;
- L'identifiant du nœud destination.

L'opération suivante consiste à *packer*, i.e. transformer en une chaîne et intégrer dans la requête RPC l'élément `config_item` correspondant à la probe source qui nous intéresse. Cette opération se réalise au moyen de l'utilisation de la fonction `global_config_pack_item`.

Enfin, nous attendons deux réponses. La première nous retourne la taille des données de la seconde. L'attente de ces réponses se fait au moyen de deux fonctions :

- `rpc_unpack_type()` pour la première, qui exploite le type des données en paramètre pour déduire la taille de données à lire ;
- `rpc_unpack()` pour laquelle il est nécessaire de spécifier la taille des données.

Ce second appel, `rpc_unpack(desc, 0, page, r)` se charge de lire **r** octets – obtenus grâce à l'utilisation de `rpc_unpack_type(desc, r)` – de la requête RPC désignée par **desc**, et les écrit à l'adresse **page**, qui correspond tout simplement à la page mémoire qui sera renvoyée au programme qui effectue sa lecture sur le fichier.

Nous concluons enfin la requête RPC par l'appel à `rpc_end()`. Notons par ailleurs que les attentes de réponses RPC, telles que `rpc_unpack` et `rpc_unpack_type` sont de type *bloquant*.

Le lecteur aura peut-être remarqué dans le code source de la fonction, présent en annexe I.3, que l'on utilise une petite astuce rappelée dans le code 4.11. Cette astuce est expliquée dans la section suivante, 4.8.4.

Serveur RPC – « côté nœud à lire »

Le code de la fonction est disponible en annexe I.2. Afin d'être en mesure de pouvoir traiter la requête RPC, il convient de signaler au sous-système RPC quelle fonction doit prendre en charge la gestion de la requête que nous avons nommé `SCHED_PIPE_SHOW_REMOTE_VALUE`.

Cette opération se fait tout simplement au moyen de l'appel
`rpc_register_void(SCHED_PIPE_SHOW_REMOTE_VALUE, handle_scheduler_pipe_show_remote_value, 0)`,
que l'on placera judicieusement dans la fonction `scheduler_probe_start`.

Après que cet enregistrement ait eu lieu, le système nous réveillera à chaque réception d'une requête RPC d'identifiant `SCHED_PIPE_SHOW_REMOTE_VALUE`.

Le rôle du serveur RPC est très simple :

1. Identifier la probe source concernée par l'appel ;
2. Allouer une page mémoire ;
3. Faire s'exécuter la méthode `SHOW` de la probe sur cette page mémoire ;
4. Retourner la taille des données écrites par cette méthode ;
5. Retourner la page mémoire au nœud requéreur.

Identification de la probe source Cette opération se déroule en deux temps. D'abord, il nous faut récupérer l'objet `config_item` local qui correspond à la probe source. Ceci se fait simplement à l'aide de la fonction `global_config_unpack_get_item`. Ensuite, nous remontons à la probe source de la même manière que nous avons été en mesure de le faire auparavant, c'est-à-dire, en s'appuyant sur la fonction `to_scheduler_probe_source`.

Arrivés à ce point, nous avons été capables de récupérer la probe source **locale** qui correspond à la probe source distante sur laquelle la requête a été effectuée.

Allocation d'une page et exécution de la méthode SHOW L'allocation d'une page mémoire ne nécessite pas qu'on s'attarde sur ce problème. L'appel à la méthode `SHOW` n'a rien de complexe, il suffit d'exploiter la structure de la probe source, de sorte à tirer l'attribut `show_value` qui correspond à la fonction qui implémente cette méthode.

Il n'y a qu'à lui passer en paramètre, en sus de la probe source, la page mémoire que nous avons allouée et qui sera renvoyée à la machine qui a fait la requête. La seule chose à laquelle il convient de porter attention sur l'allocation mémoire concerne la taille maximale des données : `SCHEDULER_PROBE_SOURCE_ATTR_SIZE`. C'est une limitation de ConfigFS. Cette valeur vaut actuellement 4kO.

Retourner la taille lue et les données Nous utilisons les fonctions inverses du couple `rpc_unpack_type` et `rpc_unpack`, à savoir `rpc_pack` et `rpc_pack_type`, dont les caractéristiques sont similaires. La première étape, le retour de la taille, consiste simplement à envoyer dans la réponse à la requête RPC ce que nous a retourné la méthode `SHOW`, à savoir la quantité de données qu'elle a écrite dans la page mémoire.

Puis, de manière similaire, il est possible de transférer la page mémoire en elle-même.

Note au sujet du comportement de Kerrihed Une petite note sur le comportement par défaut de Kerrighed à la réception des RPC. Nous utilisons le couple de fonctions `global_config_pack_item` et `global_config_unpack_get_item`. Or, lorsque le sous-système RPC de Kerrighed passe la main à notre fonction de gestion de la requête RPC, dans un soucis de simplification, il effectue une première opération de préparation des données.

Or, la première chose que nous envoyons au serveur RPC est une chaîne qui décrit le `config_item` auquel nous nous intéressons. Ainsi, le paramètre `void* msgIn` de la fonction `handle_scheduler_pipe_show_remote_value` pointe déjà sur cette chaîne. Dans notre cas, pour récupérer le `config_item` local correspondant à celui qui nous a été envoyé par le client, il faut passer par la fonction `global_config_unpack_get_item`, laquelle prend en paramètre un descripteur de requête RPC. Ce comportement par défaut nous oligerait à des circonvolutions néfastes à la compréhension du code et inutiles.

C'est ainsi que, sur avis des développeurs Kerrighed, nous avons opté pour la solution d'envoyer, avant notre `config_item`, un élément de requête RPC qui soit de taille nulle. Ceci permet de contourner le problème, de manière triviale, et sans surcoût.

```
rpc_pack(desc, 0, NULL, 0); /* needed as trick */
```

Code 4.11 – Astuce RPC

4.8.5 Schémas récapitulatifs de la modification de ConfigFS

Ces schémas se proposent de résumer les deux modifications principales que nous apportons à ConfigFS :

- Ajout de répertoire pour chaque nœud et détournement des appels ;
- Gestion de la communication via RPC

Schéma de l'ajout et du détournement pour les nœuds

Ce schéma est disponible en figure 4.4. La situation initiale dont il est fait mention doit se comprendre comme « avant nos modifications de code ». Les deux éléments dont l'entourage est un trait noir sont les fonctions utilisées pour réaliser les opérations de lecture lorsqu'un appel système `read()` est émis sur l'attribut concernant la probe source.

Les flèches et arcs représentent les liens entre structures de données ou fonctions. C'est bien sûr une vue simplifiée et à visée synthétique, seuls les éléments pertinents sont indiqués.

Schéma de la communication RPC

Ce schéma est disponible en figure 4.5. Les deux parties que forment le serveur et le client RPC sont bien distinguées. Notons cependant qu'en réalité, puisque nous sommes dans un système distribué, tout nœud est à la fois potentiel serveur et potentiel client. Cependant, à un instant donné et entre deux nœuds donnés, il est tout à fait acceptable de se limiter à l'un étant client et l'autre serveur.

Au centre, nous trouvons ce qui représente la requête RPC. Sa position reflète bien la réalité : c'est dans une seule et même requête RPC que l'on fait la partie « question »et la partie « réponse ». Les deux cadres verts, de part et d'autre du schéma, représentent les fonctions dont le nom est indiqué en haut du cadre. Cette représentation montre bien que tout le travail intéressant est réalisé côté « serveur ».

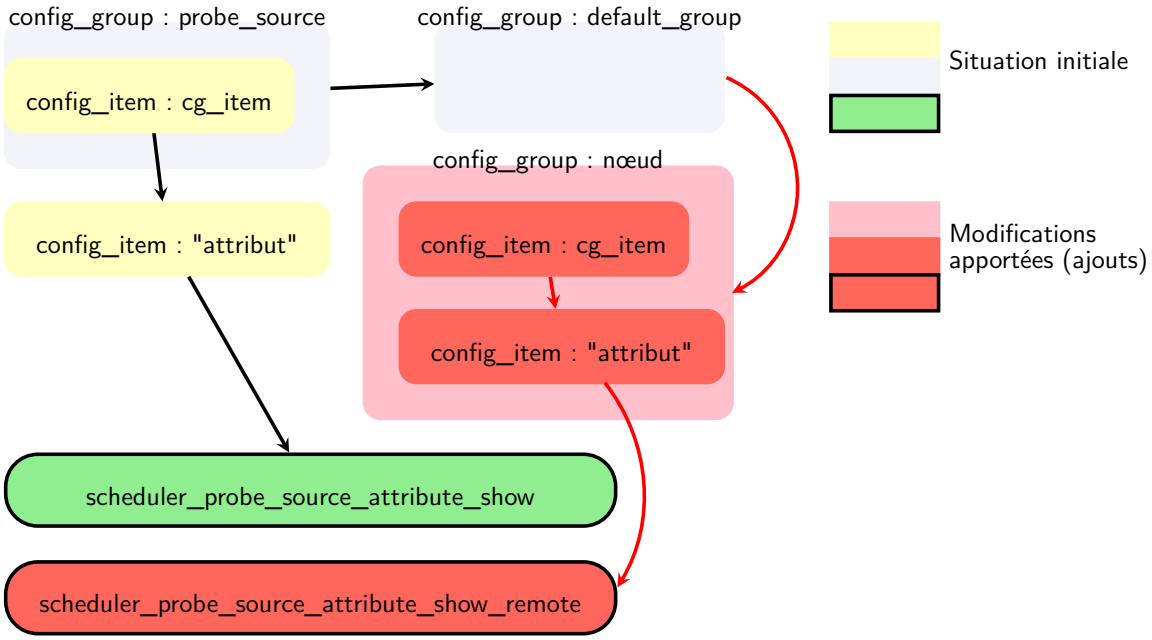


FIG. 4.4 – Ajout de répertoires pour les nœuds et détournement des appels

La couleur utilisée pour les flèches, qui montrent l'ordre de transmission des données, est une inspiration du système que l'on retrouve dans le sang chez l'Homme : le sang neuf, rouge, arrive par les artères, chargé d'oxygène. Puis, une fois qu'il a été utilisé, il repars, chargé – de ce qui représente des déchets pour les cellules – et sans oxygène. Dans notre cas, les données partent vides du client lorsque la requête RPC est émise. Elles sont réceptionnées sur le serveur, et la méthode SHOW locale est exécutée. Une première information remonte dans la requête RPC : c'est la taille des données lues. D'où le changement de couleur. Puis, c'est directement la page mémoire qui est renvoyée, et l'on conserve la nouvelle couleur rouge, comme du sang qui ressortirait des poumons, chargé d'oxygène.

4.9 Outils nécessaires en espace utilisateur pour l'ordonnancement en espace utilisateur

Nous avons décrit tout ce qu'il était nécessaire de mettre en place, dans l'espace noyau, pour être en mesure de réaliser un ordonnancement depuis l'espace utilisateur. À présent, il convient donc de réfléchir quant à la manière de répondre au besoin C.7. Nous nous intéresserons à l'implémentation effective de ce besoin dans la section suivante, 4.10.

Le besoin décrit en annexe C.5 propose l'utilisation de l'écriture d'un démon d'ordonnancement, c'est-à-dire un service qui s'exécute et utilise des plugins qui se chargent de la réalisation effective des calculs d'ordonnancement. C'est une voie de travail intéressante. Une autre manière intéressante pourrait être de simplement proposer une bibliothèque à laquelle viendraient se lier des ordonnanceurs pour lesquels aucune contrainte n'est présente, et dont le seul rôle serait d'abstraire certaines fonctionnalités, telles que la lecture des données depuis les probes.

Nous allons donc explorer un peu plus ces deux manières de répondre au problème, afin de voir les avantages et inconvénients de chacune, et de faire un choix pour l'implémentation effective.

4.9.1 Service d'ordonnancement en espace utilisateur

Intéressons nous un peu plus précisément à l'idée de l'écriture d'un service en espace utilisateur chargé de l'ordonnancement des processus. Plus exactement, du calcul de ce dernier, étant donné que les migrations

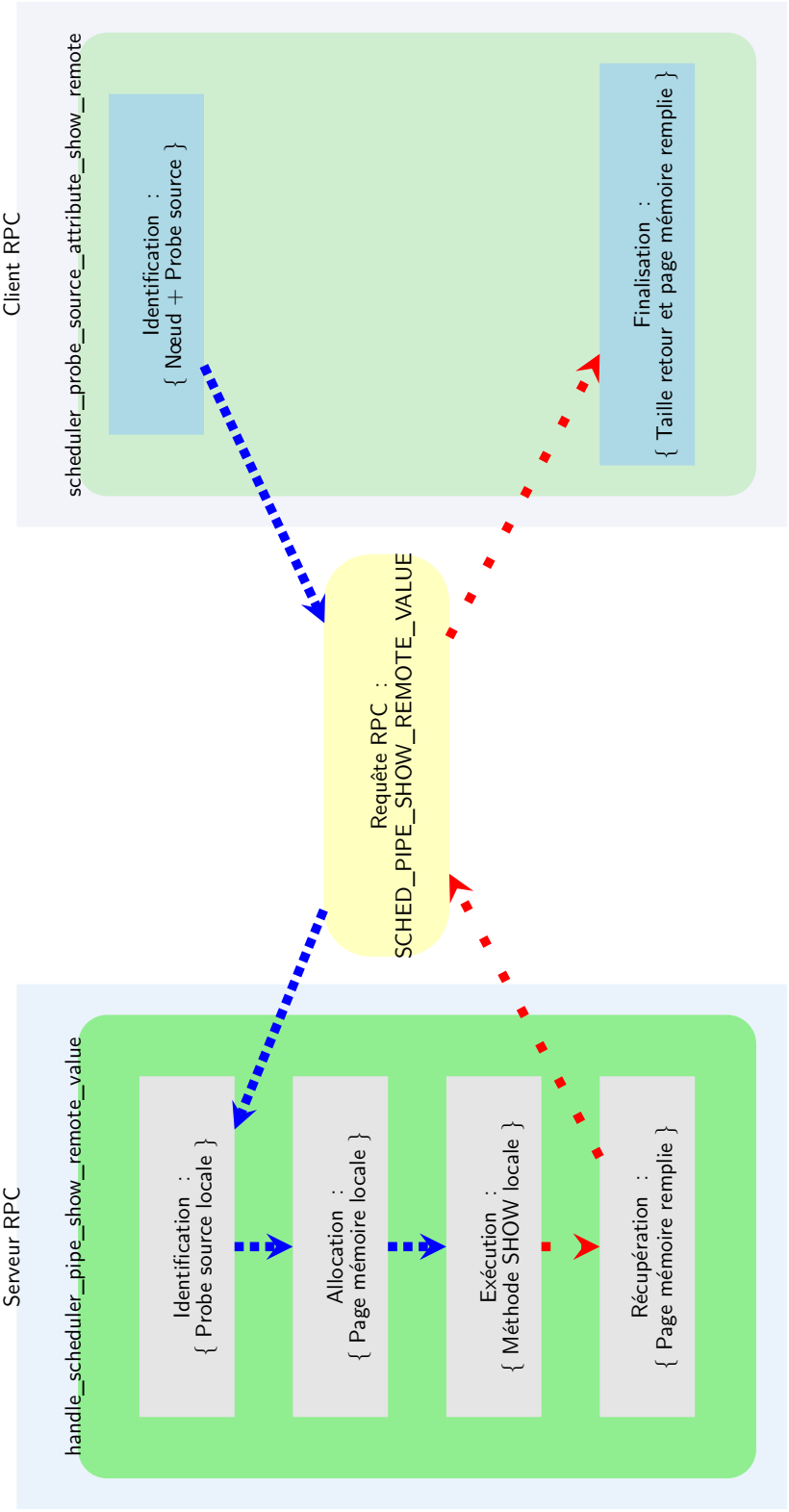


FIG. 4.5 – Communication RPC

seraient bien sûr réalisées à l'aide de l'appel système `migrate(2)`.

Présentation du concept

Nous proposons donc la spécification d'un service. Celui-ci aura pour tâche de fournir plusieurs éléments :

1. Une API pour accéder aux probes, simplifiant l'utilisation de nos modifications ;
2. Une API pour écrire des modules de traitement des données ;
3. Une API pour écrire des modules d'ordonnancement chargeables et déchargeables à chaud ;
4. Une API pour spécifier le calcul de l'ordonnancement, à l'image des fonctions `find_a_node` et `find_a_process` qui ont été implémentées pour RBT.

Le rôle de l'API pour accéder aux données du cluster vise principalement deux buts :
- Abstraire le côté non pas complexe, mais répétitif de l'ouverture de répertoires et de fichiers ;
- Mettre en place un mécanisme de cache, si le besoin s'en fait sentir.

Le rôle de l'API pour écrire des modules de traitements des données est simple. Il convient de se souvenir que les données lisibles à distance sur les nœuds correspondent à ce que la méthode *SHOW* d'une probe source va écrire. Il est donc nécessaire de récupérer ces données, présentes sous formes de texte, et de les transformer, pour être en mesure de les manipuler par la suite. Et comme les méthodes *SHOW* n'ont aucune contrainte quant au formatage des données qu'elles renvoient, on ne peut faire de méthode générique capable de fonctionner dans tous les cas. Une bonne solution est donc de mettre en place un mécanisme de plugins chargés de traiter ce problème.

Le rôle de l'API pour la gestion des modules d'ordonnancement est évident. Il s'agit de définir le nécessaire pour gérer ces derniers, et être en mesure de les charger et décharger à chaud.

Enfin, la dernière API se concentrerait uniquement à spécifier une interface telle que :
- `find_a_node`
- `find_a_process`

que les modules implémenteront et qui permettront de calculer et ainsi déterminer quel processus migrer, et où il doit être déplacé. D'un point de vue technique, ces deux dernières interfaces seront intimement liées. On peut imaginer, par exemple une structure de description du module, définissant quatre points d'entrée :

1. Fonction appelée au chargement ;
2. Fonction de sélection du processus cible ;
3. Fonction de sélection du nœud cible ;
4. Fonction appelée au déchargement.

Un autre paramètre à prendre en compte, et à définir dans ces API, sera la manière et la fréquence d'appel aux modules de calcul d'ordonnancement.

Avantages

La réalisation et l'implémentation de tout un service chargé de l'ordonnancement présente un certain nombre d'avantages, parmi lesquels nous pouvons citer :
- Un cadre strict, limitant la quantité de code à produire pour étendre les possibilités d'ordonnancement ;
- Une forte mutualisation de code, limitant les redondances – et donc les bugs potentiels – ;
- Par le côté chargement/déchargement à chaud, potentiellement plus simple pour le développement.

Inconvénients

Cependant, il convient de prendre en considération les inconvénients suivants :
- Un cadre qui peut s'avérer trop strict dans certains cas, voire limitant ;
- Un travail de spécification et de réflexion aux APIs nécessaires, déjà partiellement réalisé ;
- Un temps de développement qui sera conséquemment impacté.

4.9.2 Bibliothèque d'abstraction pour l'ordonnancement

Étudions ce que nous pourrions faire, dans le cadre du développement d'une bibliothèque dont le rôle sera équivalent. L'organisation de cette étude est calquée sur la précédente, nous présenterons l'idée du concept et les points clefs, puis nous évaluerons avantages et inconvénients.

Présentation du concept

L'idée est de placer la même offre de services, mais cette fois en la réalisant sous la forme d'une bibliothèque dynamique. Bien sûr, les aspects chargement et déchargement de modules à chaud ainsi que les spécifications qui en découleraient seraient inutiles, et donc évincées de cette idée.

Il resterait ainsi les éléments suivants, cochés :

- ☒ Une API pour accéder aux probes, simplifiant l'utilisation de nos modifications ;
- ☒ Une API pour écrire des modules de traitement des données, i.e. étendre les capacités de la bibliothèque ;
- ☐ Une API pour écrire des modules d'ordonnancement chargeables et déchargeables à chaud ;
- ☐ Une API pour spécifier le calcul de l'ordonnancement, à l'image des fonctions `find_a_node` et `find_a_process` qui ont été implémentées pour RBT.

Et c'est assez cohérent, puisque l'utilisateur disposerait déjà de l'appel système `migrate(2)` ainsi que de la commande en espace utilisateur éponyme. Comme, de plus, il n'est plus nécessaire d'utiliser le formalisme de module, puisqu'il n'a tout simplement plus lieu d'être, il devient inutile de garder l'idée d'API pour les modules d'ordonnancement. De la même manière, la spécification de fonctions chargées de déterminer qui et où envoyer est à la charge de l'utilisateur.

Avantages

Intéressons nous rapidement aux avantages présentés par cette solution :
- Trivialement, on se doute que cela sera plus rapide à développer, et tester ;
- Une plus grande liberté laissée à l'utilisateur-développeur pour choisir la manière d'implémenter la totalité de son mécanisme d'ordonnancement.

Inconvénients

Maintenant, nous nous penchons sur les inconvénients inhérents à cette solution :
- Moins de réutilisabilité de code, avec tous les travers qui en découlent.

4.9.3 Solution retenue

À l'heure actuelle, aucun choix n'a été réalisé quant à la solution retenue.

4.10 Mise en place de l'ordonnancement en espace utilisateur

L'implémentation effective d'un mécanisme d'ordonnancement en espace utilisateur, pour lequel seules quelques idées sont décrites dans le besoin disponible en annexe C.5, n'a pas encore été réalisée, principalement par manque de temps. Le peu de temps disponible a été utilisé afin de réfléchir sur les possibilités qui s'offrent à nous pour l'implémentation de ce mécanisme, et voir les avantages et inconvénients de chacune.

5. Suivi du projet

Nous avons pu présenter en détail le travail qui a été réalisé, pour chacun des besoins qui avaient été identifiés. Nous allons maintenant nous intéresser à faire un point quantitatif quant au travail qui a été réalisé.

Nous pourrons ainsi distinguer les zones pour lesquelles le besoin a été totalement comblé de celles où il reste encore des éléments à implémenter.

5.1 État d'avancement des besoins

Intéressons nous d'abord à l'avancement de chaque besoin. Il convient de noter que les durées estimées dans le planning ne tenaient pas compte du temps pour écrire les tests et vérifier que ces derniers fonctionnent, alors que les durées estimées après la réalisation du besoin prennent en considération l'écriture et le test des tests.

5.1.1 Présentation des changements depuis Kerrighed 2.2.0

Ce besoin a été présenté en détails dans l'annexe C.6. Nous avons montré les changements qui ont été apportés par Kerrighed depuis la version 2.2.0. L'évolution principale, celle qui nous intéresse, c'est bien sûr l'arrivée du framework SchedConfig, qui permet une configuration à chaud de l'ordonnanceur utilisé sur toute le cluster, avec la mise en place de politiques différentes, et qui peuvent s'appliquer simultanément à plusieurs processus.

Nous pouvons donc estimer que ce besoin a été totalement comblé. L'effort nécessaire à la réalisation de ce besoin a, globalement, été plutôt bien estimé, même s'il faut reconnaître que l'étendue du framework est assez difficile à jauger au premier abord.

5.1.2 Portage de l'API

Ce besoin a été présenté en détails dans l'annexe C.1. Sa réalisation a été assez longue, et ce pour plusieurs raisons. D'abord, parce qu'il a été nécessaire d'analyser, dans le code existant de Kerrighed, ce que font les modules déjà présents, et en quoi certains pourraient être déjà réutilisés pour combler nos besoins. Nous avons d'ailleurs pû constater, à nos dépends, qu'une information que nous pensions réutilisable au sein du module de probe `mosix_probe` ne l'était en réalité pas. Dès lors, nous avons été dans l'obligation d'implémenter la probe source `process_jiffies` pour pouvoir obtenir l'information que nous souhaitions.

Ensuite, cela a été long parce qu'il a fallu écrire un certain nombre de modules, et que l'implémentation de chacun n'est pas nécessairement triviale. En effet, puisque nous avons essayé d'être le moins intrusif au maximum, i.e. ne pas directement modifier les structures de base du noyau pour nos besoins comme cela avait été fait lors de précédents travaux, cela implique plus de travail de notre côté, pour trouver où aller chercher l'information, et être en mesure de la conserver.

Enfin, parce que cela a été notre premier contact de développement avec le framework SchedConfig, et qu'il a donc fallu un peu de temps pour y entrer correctement. Tout ceci fait abstraction du recensement qui a dû être réalisé au préalable, pour s'assurer que les algorithmes d'ordonnancement n'avaient pas besoin d'informations autres que celles qui avaient été rappelées dans l'API proposée dans [Dab08].

Il convient donc de reconnaître que l'estimation de la charge de travail prévue pour cette tâche, dans le cahier des charges, est complètement erronée. Alors que six jours de travail, à temps plein, avaient été

considérés, c'est plutôt au moins un mois à temps plein qu'il aurait fallu prévoir. Cependant, le besoin a été comblé totalement, il ne reste rien à faire.

5.1.3 Portage des algorithmes d'ordonnancement

Ce besoin a été présenté en détails dans l'annexe C.2. Son estimation en charge de travail avait été portée à 21 jours, à temps plein. Étant donné que, d'une part, l'implémentation de ces algorithmes a eu lieu après le portage de l'API en espace noyau, et d'autre part, le cadre de SchedConfig est moins lourd pour les modules qui implémentent les politiques d'ordonnancement, l'estimation était assez mauvaise. Le fait d'avoir déjà travaillé avant avec SchedConfig a passablement accéléré le travail, puisque malgré quelques différences entre les modules de probe et ceux de policy, la philosophie reste plutôt proche, et nous avons donc pu capitaliser sur les acquis réalisés pendant le portage de l'API. Ce qui est une bonne chose, puisque nous avons dû implémenter le module de filtrage, qui n'était pas prévu à la base.

Malgré tout, cette durée était censée couvrir le temps nécessaire pour la réalisation du portage des deux algorithmes non naïfs qui avaient été implémentés dans le cadre du travail de Mathieu Dabert [Dab08] : RUDOLPH [RSAU91] et RBT. Seul le second a été implémenté, car nous l'estimons plus simple à écrire.

Nous pourrions donc dire que ce besoin n'a été comblé qu'à 65%, puisque même si un seul des deux algorithmes a été implémenté, il est fonctionnel – même s'il a sûrement encore besoin de tests et de validation – et au cours de son implémentation, nous avons pu nous confronter à plusieurs problèmes qui sont déjà résolus ou au moins que nous pouvons considérer lorsque nous nous occuperons d'implémenter RUDOLPH [RSAU91]. Une douzaine de jours, pour la réalisation totale de ce besoin, correspond à une estimation plus réaliste.

5.1.4 Signalisation de l'utilisation physique des nœuds

Pour des raisons pratiques, malgré ce qui avait été prévu dans le planning originel, ce besoin a été l'un des premiers impliquant le développement de modules noyau a être réalisé. La raison est simple, nous avons estimé que la complexité inhérente au problème de la signalisation était faible, puisqu'il s'agit uniquement d'implémenter un compteur, et dès lors, l'écriture de ce module permettrait surtout de se remettre plus aisément à faire du code en espace noyau, sans avoir à gérer en plus la complexité inhérente à SchedConfig par exemple.

Interface noyau nécessaire

Ce besoin a été présenté en détails dans l'annexe C.3.1. Nous avons donc commencé par la réalisation de ce besoin. Il a constitué en l'écriture du module noyau *Local User Presence*, qui n'implémente qu'un compteur manipulé grâce à des symboles exportés aux autres modules. Le temps qui avait été prévu pour ce module était évalué à environ 10 jours. Il s'avère que cette durée était largement sur-estimée.

Même en comptant l'écriture des tests, et comme ce fût le premier test à être écrit il a été un peu plus long, une estimation à deux jours aurait été plus pertinente. Malgré cette erreur de jugement, le module est pleinement fonctionnel, et le besoin est donc comblé en totalité.

Mécanisme de communication avec la machine virtuelle

Ce besoin a été présenté en détails dans l'annexe C.3.2. Pour les mêmes raisons que précédemment, ce besoin a été l'un des premiers à être comblé, avec le développement du module noyau *Local User Notifier*. C'est lui qui permet de fournir l'interface à *Local User Presence* pour l'espace utilisateur. La « complexité »résidait donc dans la gestion des entrées de *ProcFS*.

La spécification de cette interface, et son implémentation ont été plus rapides que ce qui avait été initialement présumé. Sur les six jours prévus, seule la moitié aurait été largement nécessaire.

Cependant, le besoin spécifiait également l'écriture d'un démon s'exécutant en espace utilisateur, sur la machine virtuelle, et qui jouera le rôle de porte d'entrée pour l'hôte physique. Pour cette application, le temps provisionné était également de six jours. En réalité, à la vue du temps qui allait être nécessaire pour réaliser le portage de l'API et des ordonnanceurs, nous avons préféré déléguer cette tâche à un projet de Systèmes d'Exploitation [PR09].

À l'heure actuelle, ce démon est normalement fonctionnel, mais cela reste à valider. Nous admettrons donc pour le moment que seule la partie module noyau est pleinement implémentée, et qu'il reste environ 25% de travail à réaliser. L'estimation de la durée du développement n'est pas pertinente, étant donné qu'elle avait été réalisée sans penser que ce projet serait confié à d'autres élèves.

Module de notification pour PAM

Ce besoin a été présenté en détails dans l'annexe C.3.3. Comme le démon dont il est fait mention dans la section précédente, l'étude et la réalisation de ce module PAM a été confiée au même binôme dans le cadre des projets de Système d'Exploitation [PR09].

Pour le moment, nous savons donc de quelle manière il convient de procéder pour intercepter les évènements de connexion et de déconnexion. Le module a été écrit et est en cours de test. La plus grosse partie du travail a donc été réalisée, et nous pouvons estimer qu'il reste 35% de la charge totale à finir.

Pour les mêmes raisons que le démon précédent, l'estimation de temps n'est plus pertinente.

Module de notification pour WinLogon

Ce besoin a été présenté en détails dans l'annexe C.3.4. De même, l'étude et la réalisation de ce module ont été confiées au même binôme. Là encore, le mécanisme d'interception des évènements est connu, un module a été écrit et est toujours en tests. Ces deniers faisaient état de problèmes de crash lors du chargement du module, mais selon toute vraisemblance, une fois ces crash résolus, le code serait pleinement fonctionnel.

Nous admettrons que 70% du travail nécessaire a été réalisé. Cependant, toujours pour les mêmes raisons que précédemment, l'estimation du temps nécessaire n'est plus pertinente.

5.1.5 Outils nécessaires en espace noyau pour l'ordonnancement en espace utilisateur

Ce besoin a été présenté en détails dans l'annexe C.4. Alors qu'il nous a semblé de prime abord que la mise en place des outils nécessaires, en espace noyau, pour être en mesure de calculer et appliquer un ordonnancement depuis l'espace utilisateur, serait un problème complexe à résoudre, force est de constater que ce fût finalement assez rapide. Bien sûr, l'aide apportée par les développeurs de Kerrighed sur ce point a été un élément crucial, puisqu'elle nous a permis de gagner énormément de temps, tant en compréhension qu'en développement.

Nous avons donc été en mesure de mettre en place les modifications nécessaires au code de SchedConfig qui gère la mise en place des objets utilisés par ConfigFS pour exposer les informations dont nous avions besoin. Nous avons ensuite été en mesure de développer le mécanisme synchrone de lecture à distance des valeurs souhaitées, et de les afficher localement. La totalité du besoin est donc couverte, et il ne manque normalement rien pour être en mesure de faire de l'ordonnancement totalement depuis l'espace utilisateur. En réalité, il reste un petit peu de travail : il manque la gestion de la libération mémoire de toutes les structures que nous avons implémenté. Mis à part cet oubli volontaire, rien ne manque à l'appel. Si l'on cummule le temps qui avait été prévu pour la totalité de la réalisation de ce besoin dans le planning d'origine, nous arrivons à vingt jours.

Il faut admettre que cette estimation était bien supérieure au temps qu'il nous a fallu pour étudier et mettre en place cette solution. Une prévision de douze jours aurait été suffisante. Cependant, cette estimation ne prenait pas en compte l'aide que nous avons reçu de la part des développeurs de Kerrighed, et il est certain que sans leur soutient, le temps prévu aurait été largement dépassé.

5.1.6 Outils nécessaires en espace utilisateur pour l'ordonnancement en espace utilisateur

Ce besoin a été présenté en détails dans l'annexe C.7. La réflexion sur les outils qui sont nécessaires pour réaliser l'ordonnancement en espace utilisateur n'a pu commencer qu'une fois les outils en espace noyau implémentés et fonctionnels. C'est à dire, assez tard dans le déroulement du projet. Mais bien plus tôt que ce qui avait été prévu dans le planning originel réalisé pendant la rédaction du cahier des charges, qui plaçait cette tâche pendant l'été.

Nous avons évoqué plusieurs pistes intéressantes sur les outils qui seront nécessaires, et même si rien n'a été développé, nous pouvons estimer que le plus difficile a été réalisé à ce niveau. Nous estimons que 75% de la réflexion a été menée à bien. La charge de travail de cette tâche est entremêlée avec celle de la tâche précédente, mais globalement, l'estimation de cinq jours qui avait été prévue à l'origine était correcte.

5.1.7 Mise en place de l'ordonnancement en espace utilisateur

Ce besoin a été présenté en détails dans l'annexe C.5. Son objectif est de mettre en application le résultat des réflexions qui ont été menées dans le cadre du besoin précédent. Étant donné qu'aucune orientation claire sur la méthode n'a été retenue, il est normal que ce besoin ne soit pas comblé. Le développement de cette partie a été estimé à six jours. Si l'on reprend les hypothèses qui ont été émises sur la manière de le faire, c'est une moyenne. La solution avec une simple bibliothèque devrait être en dessous, alors que l'option de développer un service plus complet sera facilement au double de temps de travail.

Dans tous les cas, rien n'est fait.

5.1.8 Nouvelles méthodes d'ordonnancement, en espace utilisateur

Ce besoin a été présenté en détails dans l'annexe C.8. La réalisation de cet aspect du travail peut être considérée comme complète. L'effort qui avait été prévu, de vingt jours, était vraisemblablement sous-estimé. La recherche d'articles, leur lecture, puis leur étude plus poussée prend du temps. Il n'aurait pas été inutile de prévoir une durée supérieure, peut-être qu'une charge de trente jours aurait été plus réaliste.

Malgré tout, nous avons été en mesure de rechercher de nouvelles méthodes à implémenter. À la fois au travers des articles qui ont été compulsés, et malgré les déconvenues sur certains, mais également grâce à l'étude des projets de Programmation par Contraintes réalisés les années précédentes, notamment [CL06, GN07].

5.2 Contributions réalisées au projet Kerrighed

À l'heure actuelle, une seule contribution a effectivement été proposée et intégrée au projet Kerrighed. Il s'agit de la modification qui permet l'utilisation des ports au sein d'une probe source. La réalisation de ce changement était plutôt simple, et bien guidée par Louis Rilling, mais elle a également permis de voir comment se déroule le processus de soumission, d'acceptation et de validation de patchs. De même, elle a été l'occasion de voir l'importance des *Coding Style Guidelines*. Au terme de plusieurs retouches, les modifications sont entrées dans l'arbre officiel, sous le numéro de révision 5331[1].

5.3 Contributions restant à transmettre

Il reste encore beaucoup de modifications que nous devons proposer à l'inclusion au projet Kerrighed. C'était un besoin exprimé dans le cahier des charges que de travailler de sorte à ce que tout ce que nous ferons soit intégré plus tard dans le projet. Ceci pour deux raisons principales : l'inclusion dans l'arbre officiel pérennise le travail que nous avons effectué, puisque cela « garantie »qu'il sera maintenu par la

[1] https://gforge.inria.fr/plugins/scmsvn/viewcvs.php?rev=5331&root=kerrighed&view=rev

suite – et c'est bien la raison des allers-retours nécessaires avant l'acceptation du patch – mais également parce que les développements réalisés intéresseront très certainement d'autres personnes. Un projet comme Kerrighed a le mérite d'exister, et d'être fonctionnel, autant faire profiter de notre travail le reste de la communauté.

Dans cette optique, donc, la **totalité** de ce qui a déjà été écrit va prochainement être proposé pour l'inclusion. Cela concernera donc, du côté des modules qui ont été développés :

- Chacun des modules de probe, accompagnés de leur test de non régression ;
- Le module de filtrage *RBT Cache* ;
- Le module de politique pour appliquer l'algorithme *RBT* ;

Par ailleurs, nos modifications visant à exposer dans ConfigFS la valeur des probes sources distantes seront également soumises. Il reste cependant quelques points de détails à compléter dans l'implémentation à ce niveau là avant de pouvoir le faire, mais rien qui ne soit très long à réaliser.

Bien sûr, les modules noyau chargés de gérer la notification de la présence d'un utilisateur interactif seront également proposés à l'inclusion, avec les outils en espace utilisateur qui les accompagnent. Ces outils se résument à :

- Le démon en espace noyau qui s'exécute au sein de la machine virtuelle ;
- Les modules PAM et WinLogon développés dans le cadre du projet de Système d'Exploitation [PR09].

Enfin, à l'utilisation de Kerrighed, il nous a semblé qu'il serait pertinent de développer un module PAM qui serait chargé de configurer automatiquement la gestion des migrations des processus. L'idée est d'être en mesure, sur le système, de créer un groupe, par exemple `krgusers`, dans lequel les utilisateurs qui ont le « droit »d'utiliser les capacités de migration seraient présents. Il ne s'agit pas à proprement parler d'un droit, mais plutôt d'une activation automatique. Ceci éviterait de forcer les utilisateurs à devoir exécuter la commande `krgcapset -d +CAN_MIGRATE` avant de lancer tout processus susceptible d'avoir besoin de migrer. Le développement d'un tel module ne semble pas très complexe, et son code sera également proposé à l'inclusion dans le projet.

6. Poursuite du développement au cours du stage

Ce chapitre vise à compléter le rapport originel, en proposant un retour sur la totalité du travail qui a été effectué pendant la période de stage.

Nous aborderons donc les points suivants :

1. Débuggage du code : principalement, correction de fuites mémoires ;
2. Mise en place de l'infrastructure de monitoring web (Kerrighed Tracker) développée dans le cadre du projet d'Option Web ;
3. Mise en place et documentation d'une infrastructure de centralisation des données pour aider au debug et au suivi du cluster ;
4. Développement d'outils en espace utilisateur pour réaliser l'ordonnancement ;
5. Détection, mise en exergue et correction d'un bug critique dans la probe `cpu_probe`.

Nous présenterons également d'autres aspect du travail qui ont été rencontrés. Ainsi, un point sera fait sur les outils et la manière d'aborder le débug en espace noyau : nous verrons rapidement l'utilisation des outils `objdump`, `kdb` ainsi que les directives de configuration et compilation qu'il est nécessaire de positionner pour travailler en debug.

Une fois ce point général abordé, nous présenterons la manière de travailler pour corriger des erreurs de type fuite de mémoire dans un module noyau. Les options nécessaires à la compilation pour y parvenir seront présentées, ainsi que la manière de faire pour repérer les fuites. Une partie de ce travail reposera sur des points présentés à la section précédente.

La suite présentera le travail nécessaire à la centralisation de tous les journaux des nœuds Kerrighed. Cette centralisation permet, entre autres, de faciliter énormément le debug noyau en permettant d'avoir un accès durable dans le temps aux traces noyau rapportées lors de crashs. Toutes les étapes nécessaires à la mise en place de cet environnement seront détaillées.

L'implémentation des outils d'ordonnancement en espace utilisateur sera ensuite présentée. Il faut rappeler que cette implémentation était un des objectifs d'origine du projet, dans l'optique de pouvoir décider de l'intérêt de politiques d'ordonnancement hybrides. Les différents modules Perl écris pour l'occasion feront donc l'objet d'explications, ainsi que les outils d'abstraction proposés. Cette partie se terminera sur la présentation du daemon d'ordonnancement en espace utilisateur qui a été écrit ainsi que deux clients à titre d'exemple et pour nos besoins futurs, concrétisant ainsi un besoin proposé au cours du projet.

Enfin, un dernier point important dans le suivi du projet concernera l'identification et la correction d'un bug dans l'un des modules de probes qui a été implémenté. Cette partie retracera la découverte puis l'identification du problème, et proposera une explication de son origine. Une solution de contournement, implémentée, sera donnée.

6.1 Mode noyau et debug

Au sein de cette section, nous allons présenter plusieurs choses. Tout d'abord, les directives de compilation qu'il convient d'activer de sorte à avoir toutes les informations nécessaires pour opérer le travail de debug. Ces options **incluent** celles spécifiques à Kerrighed. Dans le cas où le noyau cible n'est pas un noyau Kerrighed, ces options seront facilement identifiables par la présence de la chaîne `KRG`.

Nous préciserons également quelques points qu'il convient de vérifier, notamment pour le niveau des messages du noyau, de sorte à s'assurer que l'on ai accès aux messages de debug.

6.1.1 Directives de compilation

La liste des options que nous avons besoin d'activer est donnée dans le code 6.2. Lors de l'utilisation de `make menuconfig`, ces options sont présentes dans la section « Kernel Hacking », puis « Kernel Debugging », sauf bien sûr pour les options de Kerrighed, qui elles se trouvent dans la section « Cluster Support ».

CONFIG_KRG_DEBUG Activation du mode Debug pour Kerrighed ;
CONFIG_DEBUG_KERNEL C'est l'option de base, « Kernel Debuuging » ;
CONFIG_DEBUG_SLAB Cette option permet de debugger l'utilisation de l'allocateur mémoire noyau, SLAB ;
CONFIG_DEBUG_SLAB_LEAK Permet de traquer les fuites mémoires. Cette option nécessite la précédente ;
CONFIG_DEBUG_DETECT_SOFTLOCKUP Permet de détecter les erreurs de type « Soft lockup » ;
CONFIG_DEBUG_SPINLOCK Permet de détecter les erreurs concernant l'utilisation de verrous tournant ;
CONFIG_DEBUG_MUTEXES Permet de détecter les problèmes avec les mutex ;
CONFIG_DEBUG_RWSEMS Permet de détecter les problèmes liés aux sémaphores ;
CONFIG_DEBUG_LOCK_ALLOC Détection de la libération mémoire invalide de verrous ;
CONFIG_PROVE_LOCKING Prouver, mathématiquement, que les verrous posés par le noyau sont corrects et ne conduisent pas à un deadlock ;
CONFIG_LOCKDEP Vérification de l'outil de vérification des dépendances des verrous ;
CONFIG_DEBUG_SPINLOCK_SLEEP Vérification des tentative de `sleep()` au sein de verrous ;
CONFIG_STACKTRACE Affichage de stacktrace en cas de crash ;
CONFIG_DEBUG_KOBJECT Informations supplémentaires sur les objets noyau Kobject ;
CONFIG_DEBUG_INFO Compilation avec les informations de debug ;
CONFIG_DEBUG_STACKOVERFLOW Notification du passage d'une limite basse de la pile disponible ;
CONFIG_KDB Activation du debugger noyau intégré ;
CONFIG_KDB_CONTINUE_CATASTROPHIC=0 Continuer KDB après une erreur catastrophique. Une valeur de 0 signifie que l'on ne continuera pas ;

Notons que nous avons là toutes les options nécessaire au debug qui sera pratiqué par la suite, tant pour les fuites mémoires que pour la correction du bug « Soft lockup ». À noter également que toutes ces opérations génèrent un flux d'information important. Dans la configuration finale de Kerrighed, avec ces options, et nos probes activées, lors de l'utilisation des outils en espace utilisateur, les logs noyau générés pour nos quatre nœuds consommaient environ 2Mbits/s sur le réseau. C'est certes négligeable pour un réseau à 100Mbits, mais il est important de l'avoir à l'esprit : cela fait environ 500Kbits/s pour chaque nœud. Sur un cluster plus important, l'impact commencerait à devenir non négligeable.

Un dernier point à vérifier concerne le niveau des messages `printk`. Nous devons configurer le niveau de debug suffisant pour que les messages de type `KERN_DEBUG` s'affichent. C'est en partie eux qui vont d'ailleurs générer un trafic important.

Nous proposons, sur la distribution Debian, de modifier la configuration dans `/etc/sysctl.conf`, et d'ajouter ou de modifier de sorte à obtenir ce qui est présenté dans le code 6.1.

```
kernel.printk = 15 4 1 15
```

Code 6.1 – Configuration du mode debug pour printk

```
CONFIG_KRG_DEBUG=y
CONFIG_DEBUG_FS=y
CONFIG_DEBUG_KERNEL=y
```

```
CONFIG_DETECT_SOFTLOCKUP=y
CONFIG_DEBUG_SLAB=y
CONFIG_DEBUG_SLAB_LEAK=y
CONFIG_DEBUG_SPINLOCK=y
CONFIG_DEBUG_MUTEXES=y
CONFIG_DEBUG_RWSEMS=y
CONFIG_DEBUG_LOCK_ALLOC=y
CONFIG_PROVE_LOCKING=y
CONFIG_LOCKDEP=y
CONFIG_DEBUG_SPINLOCK_SLEEP=y
CONFIG_STACKTRACE=y
CONFIG_DEBUG_KOBJECT=y
CONFIG_DEBUG_INFO=y
CONFIG_DEBUG_STACKOVERFLOW=y
CONFIG_KDB=y
CONFIG_KDB_CONTINUE_CATASTROPHIC=0
```

Code 6.2 – Liste des options de compilation de debug à activer

6.1.2 Manipulation des modules et compréhension des traces d'erreur

Plaçons nous dans un monde imaginaire, où le code que nous écrivons est buggué. En cas de soucis dans un module noyau, vous pourrez observer des erreurs sur la sortie console du noyau. Ces erreurs peuvent ressembler à celle rapportée dans le code 6.3 ou 6.4.

Le type d'erreur remonté ne nous intéresse pas vraiment dans ce cas présent. Ce qui nous intéresse, c'est cette manière de présenter les fonctions. Nous avons là le contenu de la pile d'appel. Donc le cheminement du système avant le crash. Les fonctions sont présentées sous la forme `identifiant+pos/taille`, où **pos** correspond à l'offset, depuis le début de la fonction, où se situe l'appel, et **taille** correspond à l'offset de la dernière instruction de la fonction.

Nous avons donc l'information précise de savoir où s'est révélée l'erreur. Attention à ce dernier point, puisque nous ne savons pas où a eu lieu le problème qui a causé cette erreur, mais nous savons précisément quand le noyau s'est retrouvé dans un état qui n'était plus cohérent.

Le dernier point important consiste à être en mesure de naviguer dans le code qui s'est exécuté. Pour cela, il convient d'utiliser l'outil `objdump`. Il nous permet d'obtenir le code assembleur d'un binaire. Si le binaire a été compilé avec les symboles de debug (ce qui doit être le cas avec les options précédemment énoncées) alors en plus nous pourrons obtenir les portions de code C qui correspondent à l'assembleur.

L'appel à utiliser est `objdump -S /lib/modules/VERSION/.../module.ko -x > module.ko.objdump`, puis il suffit d'ouvrir le fichier dans un éditeur de texte. Il nous faut aller voir la table des symboles, repérée par la chaîne `SYMBOL TABLE`, puis y chercher notre fonction. La première colonne donne l'adresse de la fonction dans le module. Si nous prenons l'exemple de l'appel `mattload_source_show_value+0xe/0x3a`, nous trouvons l'adresse : `00000452 l F .text 0000003a mattload_source_show_value`. Nous savons donc que l'erreur s'est produite à l'adresse $00000452 + 0 \times e$, soit 460. Il suffit de rechercher cette adresse, et nous trouvons l'appel incriminé, présenté dans le code 6.5.

Cet exemple met bien en évidence un point important : nous trouvons le moment où le système n'est plus en mesure de s'exécuter, pas le moment où le bug est apparu. Dans ce cas précis, notez que l'erreur se produit sur l'appel `call c4 <matteo_load_calc>`, puisque l'adresse 460, dernière enregistrée dans la pile, pointe sur le `xor \%eax,\%eax`. Et nous pouvons tout de même faire la supposition qu'un simple XOR sur un registre du processeur fonctionne. Sans quoi, autant arrêter tout debug maintenant, si le matériel n'est pas fiable à ce point.

```
Aug 28 10:10:35 192.168.5.64 kernel: BUG: soft lockup detected on CPU#0!
Aug 28 10:10:35 192.168.5.64 kernel: [<c0143403>] softlockup_tick+0xa7/0xb8
Aug 28 10:10:35 192.168.5.64 kernel: [<c0129ebe>] run_local_timers+0xf/0x10
Aug 28 10:10:35 192.168.5.64 kernel: [<c0129f04>] update_process_times+0x45/0x69
Aug 28 10:10:35 192.168.5.64 kernel: [<c0112e95>] smp_apic_timer_interrupt+0x81/0x8f
```

```
Aug 28 10:10:35 192.168.5.64 kernel: [<c0105d37>] apic_timer_interrupt+0x33/0x38
Aug 28 10:10:35 192.168.5.64 kernel: [<c020893f>] delay_tsc+0xb/0x13
Aug 28 10:10:35 192.168.5.64 kernel: [<c0208970>] __delay+0x6/0x7
Aug 28 10:10:35 192.168.5.64 kernel: [<c020a0d4>] _raw_spin_lock+0x8d/0x128
Aug 28 10:10:35 192.168.5.64 kernel: [<c0390361>] _spin_lock+0x30/0x35
Aug 28 10:10:35 192.168.5.64 kernel: [<f9469232>] scheduler_probe_source_lock+0x4f/0
    x55 [kerrighed]
Aug 28 10:10:35 192.168.5.64 kernel: [<f9469232>] scheduler_probe_source_lock+0x4f/0
    x55 [kerrighed]
Aug 28 10:10:35 192.168.5.64 kernel: [<f8827460>] mattload_source_show_value+0xe/0x3a
     [mattload_probe]
Aug 28 10:10:35 192.168.5.64 kernel: [<f946927e>]
    handle_scheduler_pipe_show_remote_value+0x2c/0xad [kerrighed]
Aug 28 10:10:35 192.168.5.64 kernel: [<f946928e>]
    handle_scheduler_pipe_show_remote_value+0x3c/0xad [kerrighed]
Aug 28 10:10:35 192.168.5.64 kernel: [<f9469252>]
    handle_scheduler_pipe_show_remote_value+0x0/0xad [kerrighed]
Aug 28 10:10:35 192.168.5.64 kernel: [<f94abbc8>] rpc_handler_kthread_void+0x50/0x6e
    [kerrighed]
Aug 28 10:10:35 192.168.5.64 kernel: [<f94ab897>] thread_pool_run+0x2df/0x5c0 [
    kerrighed]
Aug 28 10:10:35 192.168.5.64 kernel: [<f94ab5b8>] thread_pool_run+0x0/0x5c0 [
    kerrighed]
Aug 28 10:10:35 192.168.5.64 kernel: [<c01344f0>] kthread+0xf6/0x12a
Aug 28 10:10:35 192.168.5.64 kernel: [<c01343fa>] kthread+0x0/0x12a
Aug 28 10:10:35 192.168.5.64 kernel: [<c0105f3b>] kernel_thread_helper+0x7/0x10
Aug 28 10:10:35 192.168.5.64 kernel: =======================
```

Code 6.3 – Stacktrace générée par le noyau : Soft Lockup

```
Aug 28 15:14:15 192.168.5.65 kernel: PID 4296058: scheduler_probe_source_lock(
    f73e3c1c); scheduler_probe_lock(f7bd7e74); owner=ffffffff
Aug 28 15:14:15 192.168.5.65 BUG: unable to handle kernel NULL pointer dereference at
     virtual address 00000000
Aug 28 15:14:15 192.168.5.65 printing eip:
Aug 28 15:14:15 192.168.5.65 f946b411
Aug 28 15:14:15 192.168.5.65 *pde = 00000000
Aug 28 15:14:15 192.168.5.65 Oops: 0002 [#1]
Aug 28 15:14:15 192.168.5.65 SMP
Aug 28 15:14:15 192.168.5.65 Modules linked in: remote_cache_filter threshold_filter
    freq_limit_filter mosix_load_balancer migration_probe mosix_probe
    round_robin_balancer kerrighed_migration_monitor kerrighed netconsole
Aug 28 15:14:15 192.168.5.65 CPU:    1
Aug 28 15:14:15 192.168.5.65 EIP:    0060:[<f946b411>]    Not tainted VLI
Aug 28 15:14:15 192.168.5.65 EFLAGS: 00010246   (2.6.20-krg #5)
Aug 28 15:14:15 192.168.5.65 EIP is at __get_alloc_kddm_obj_entry+0x59/0x90 [
    kerrighed]
Aug 28 15:14:15 192.168.5.65 eax: 00000112   ebx: 00000000   ecx: c015ee3d   edx:
    00000001
Aug 28 15:14:15 192.168.5.65 esi: f73cd240   edi: f73cd038   ebp: 0000804d   esp:
    f6cc3e4c
Aug 28 15:14:15 192.168.5.65 ds: 007b   es: 007b   ss: 0068
Aug 28 15:14:15 192.168.5.65 Process stress (pid: 4230734, ti=f6cc2000 task=c234a330
    task.ti=f6cc2000)
Aug 28 15:14:15 192.168.5.65 Stack: c2314c58 00000041 f73cd038 00000000 0000804d
    f94741aa 00000004 00000000
Aug 28 15:14:15 192.168.5.65 00000246 00000000 c234a358 c234a330 00000018 c0780c94
    00000002 c234a330
Aug 28 15:14:15 192.168.5.65 00000006 c0150b55 f786b26c f94e1ff8 f788e0b4 00000000
    f6cc3f60 f94747d6
Aug 28 15:14:15 192.168.5.65 Call Trace:
```

```
Aug 28 15:14:15 192.168.5.65 [<f94741aa>] generic_kddm_grab_object+0xae/0x640 [
    kerrighed]
Aug 28 15:14:15 192.168.5.65 [<c0150b55>] __handle_mm_fault+0x2de/0x8e5
Aug 28 15:14:15 192.168.5.65 [<f94747d6>] _kddm_grab_object_manual_ft+0xa/0xb [
    kerrighed]
Aug 28 15:14:15 192.168.5.65 [<f94972d5>] anon_memory_nopage+0x88/0x28d [kerrighed]
Aug 28 15:14:15 192.168.5.65 [<c01456dc>] find_get_page+0x3a/0x40
Aug 28 15:14:15 192.168.5.65 [<c0391ce6>] do_page_fault+0x16d/0x528
Aug 28 15:14:15 192.168.5.65 [<c0150a35>] __handle_mm_fault+0x1be/0x8e5
Aug 28 15:14:15 192.168.5.65 [<c011b289>] schedule_tail+0x0/0xee
Aug 28 15:14:15 192.168.5.65 [<c0391da4>] do_page_fault+0x22b/0x528
Aug 28 15:14:15 192.168.5.65 [<c0391b79>] do_page_fault+0x0/0x528
Aug 28 15:14:15 192.168.5.65 [<c03907ac>] error_code+0x7c/0x84
Aug 28 15:14:15 192.168.5.65 =======================
Aug 28 15:14:15 192.168.5.65 Code: f2 c6 eb 07 89 f0 e8 75 4f f2 c6 8b 5f 24 89 ea 89
     f8 8b 0c 24 ff 53 0c 3b 04 24 89 c3 74 0c 8b 14 24 89 e9 89 f8 e8 cf fd ff ff <
    f0> 0f ba 2b 0b 19 c0 85 c0 74 24 9c 58 f6 c4 02 75 09 89 f0 e8
Aug 28 15:14:15 192.168.5.65 EIP: [<f946b411>] __get_alloc_kddm_obj_entry+0x59/0x90 [
    kerrighed] SS:ESP 0068:f6cc3e4c
Aug 28 15:14:15 192.168.5.65
Aug 28 15:14:15 192.168.5.65 Entering kdb (current=0xc234a330, pid 4230734) on
    processor 1 Oops: Oops
Aug 28 15:14:15 192.168.5.65 due to oops @ 0xf946b411
Aug 28 15:14:15 192.168.5.65 eax = 0x00000112 ebx = 0x00000000 ecx = 0xc015ee3d edx =
     0x00000001
Aug 28 15:14:15 192.168.5.65 esi = 0xf73cd240 edi = 0xf73cd038 esp = 0xf6cc3e4c eip =
     0xf946b411
Aug 28 15:14:15 192.168.5.65 ebp = 0x0000804d xss = 0xc0310068 xcs = 0x00000060
    eflags = 0x00010246
Aug 28 15:14:15 192.168.5.65 xds = 0xc231007b xes = 0xf73c007b origeax = 0xffffffff &
    regs = 0xf6cc3e14
```

Code 6.4 – Stacktrace générée par le noyau : déréférencement de pointeur NULL

```
DEFINE_SCHEDULER_PROBE_SOURCE_SHOW(mattload, page)
 452:   56                      push   %esi
 453:   89 c6                   mov    %eax,%esi
 455:   53                      push   %ebx
 456:   89 d3                   mov    %edx,%ebx
 458:   83 ec 0c                sub    $0xc,%esp
 45b:   e8 fc ff ff ff          call   45c <mattload_source_show_value+0xa>
                        45c: R_386_PC32 scheduler_probe_source_lock
{
        unsigned int load;
        load = matteo_load_calc(0);
 460:   31 c0                   xor    %eax,%eax
 462:   e8 5d fc ff ff          call   c4 <matteo_load_calc>

        return sprintf(page, "%u\n", load);
 467:   89 1c 24                mov    %ebx,(%esp)
 46a:   89 44 24 08             mov    %eax,0x8(%esp)
 46e:   c7 44 24 04 73 02 00    movl   $0x273,0x4(%esp)
 475:   00
                        472: R_386_32   .rodata.str1.1
 476:   e8 fc ff ff ff          call   477 <mattload_source_show_value+0x25>
                        477: R_386_PC32 sprintf
 47b:   89 c3                   mov    %eax,%ebx
{
        *value_p = matteo_load_calc(0);
        return 1;
}
```

Code 6.5 – Portion du code assembleur décoré de C pour le module mattload_probe

6.1.3 Utilisation de KDB

La présente section est une rapide introduction à l'utilisation de base du débuggueur intégré au noyau, KDB. Elle n'abordera que les points qui ont été nécessaires au debug au cours du projet, principalement pour corriger le problème « Soft Lockup » mis en lumière plus tard. Malgré tout, cette présentation permettra de donner un bon point de départ sur la manière de procéder avec cet outil.

Il faut d'abord savoir que KDB sera accessible dès que le noyau fera une erreur, en tapant sur la touche « Pause » du clavier. Si vous n'avez pas activé l'option USB pour KDB, il faudra impérativement brancher un clavier PS2, sous peine de ne pas pouvoir travailler. Il est fortement recommandé d'avoir mis en place l'architecture décrite en section 6.4. Les sorties de KDB seront ainsi disponibles dans les journaux, et seront bien plus simple à partager si besoin.

Au cours de l'utilisation qui a été faite de KDB, seules trois commandes ont été utilisées :

- `md`, qui permet de voir le contenu de la mémoire à partir de l'adresse indiquée en paramètre ;
- `btp`, qui permet d'avoir la *backtrace* des appels d'un processus, en utilisant son identifiant ;
- `btt`, qui permet de connaître la *backtrace* d'un processus, mais en utilisant l'adresse de sa structure comme référence.

Un dernier point à considérer est d'utiliser GDB sur les modules noyaux compilés en mode debug. Ceci permet, par exemple, de localiser des adresses qu'il est intéressant de visualiser dans KDB.

À noter que la fiabilité des *backtrace* n'est pas totale : des fonctions « fantômes » peuvent s'y loger, perturbant ainsi le debug. Ce fut le cas lors de nos opérations, il est donc nécessaire de vérifier la cohérence des traces avec les appels possible par le code.

6.2 Développement noyau et fuites mémoires

Comme dans n'importe quel logiciel développé dans un langage ne s'occupant pas de nettoyer la mémoire, le développeur peut, par inadvertance, oublier de libérer des éléments qu'il a alloué. Un outil assez souvent utilisé pour traquer et corriger ce type d'erreur est `Valgrind`. Malheureusement, celui-ci ne peut fonctionner qu'en espace utilisateur, puisqu'il détourne les appels systèmes pour opérer son instrumentation.

En espace noyau, tout comme en espace utilisateur au travers de la bibliothèque C standard, une portion du code s'occupe de « gérer » la mémoire, en proposant des fonctions pour l'allouer et la libérer. Il s'agit du couple `kmalloc()`/`kfree()` et de leurs descendants. Ces fonctions reposent sur un allocateur mémoire, le *SLAB Allocator* [Jon]. Cet allocateur est donc en mesure de savoir ce qui a été alloué et qui n'a pas ensuite été libéré.

6.2.1 Outils nécessaires pour traquer les fuites mémoire

Deux options de configuration du noyau doivent être activées si l'on veut être en mesure de tracer les fuites mémoires. Il s'agit des options `CONFIG_DEBUG_SLAB` et `CONFIG_DEBUG_SLAB_LEAK`. Une fois activées, il faut bien sûr recompiler le noyau. Au prochain redémarrage, une nouvelle entrée sera présente dans le système de fichier virtuel `/proc` : `/proc/slab-allocator`.

Ce fichier sera ce que nous utiliserons par la suite. Il contient une référence, qui se comprends de la même manière que celle explicitée en 6.1.2, et qui indique la quantité de mémoire consommée ainsi que la localisation de l'appel pour l'allocation.

Nous savons donc qui (i.e. quelle fonction, et mieux encore à quel endroit) a demandé et obtenue quelle quantité de mémoire.

6.2.2 Localisation de la fuite mémoire

Pour opérer la localisation d'une fuite mémoire, l'opération est finalement plutôt simple. Il suffit de charger votre module, effectuer les opérations consommant de la mémoire, puis capturer la sortie de `/proc/slab-allocator`, et de la conserver dans un fichier. Nous le nommerons `mem1.txt`. Faites ensuite les opérations nécessaires pour libérer la mémoire. Enfin, ce qui est censé la libérer. Par exemple, cela peut-être de décharger le module. Une fois que ceci est effectué, capturer une seconde fois le contenu de la sortie de debug de l'allocateur mémoire. Plaçons cette fois-ci ce contenu dans `mem2.txt`.

Il ne reste plus qu'à simplement faire un ...diff entre les fichiers `mem1.txt` et `mem2.txt`. En effet, s'il n'y a pas de fuite mémoire, les blocs qui ont été alloués et enregistrés dans le premier fichiers auront été libérés lorsque nous enregistrons dans le second fichier. S'il n'ont pas été libérés, ils seront toujours présents, et nous savons où ils ont été alloués.

La correction de la fuite mémoire appartient ensuite au développeur.

6.3 Présentation de Kerrighed Tracker

L'outil Kerrighed Tracker [FL09] est une interface web, écrite en Rails, à l'occasion du projet d'Option Web et Multimédia. Nous allons présenter cet outil dans la présente section. Nous commencerons donc par détailler les objectifs à cours terme et à plus long terme qui sont visés par l'application.

Le temps nous étant compté dans le cadre du projet d'Option Web et Multimédia, il a été nécessaire, au cours du stage, de compléter l'application. Tous les nouveaux éléments écris seront présentés, sans rentrer dans les détails, mais de manière à donner une vue plus globale sur tout le système.

6.3.1 Objectifs de l'outil Kerrighed Tracker

Le principal objectif visé par cette application est de permettre de mettre en place un monitoring avancé sur un cluster Kerrighed. Monitoring avancé, cela regroupe plusieurs points :

- Suivi de l'exécution et de la migration des processus ;
- Suivi de la charge globale du cluster ;
- Suivi de la charge individuelle de chaque nœud du cluster ;
- Mise en place d'outils facilitant l'administration.

Seul le premier point a été abordé, à l'heure actuelle. Un objectif secondaire était en effet d'être en mesure de garder une trace des migrations, pour pouvoir effectuer des comparaisons pertinentes sur les performances des algorithmes de répartition et de rééquilibrage de charge.

Un schéma résumant l'architecture mise en place est proposé dans la figure 6.1. L'idée générale est assez simple. Nous avons, au cours du projet, eu la possibilité de mettre en place un mécanisme de suivi des processus pour les besoins d'une des probes. Il est possible, sur le même schéma, de savoir quand un processus migre. Un module noyau est donc dévolu à cette tâche. Il communique avec l'espace utilisateur la liste des processus qui ont migré, ainsi que la date de migration. L'espace utilisateur génère une trappe SNMP avec ces informations, et envoie ceci au serveur SNMP. Ce dernier déclenche l'insertion dans la base de données de l'application. Cette dernière n'a plus qu'à s'occuper de l'affichage des migrations.

Les objectifs à plus long terme sont le suivi de la charge individuelle de chaque nœud du cluster, ainsi que la charge globale de ce dernier. L'ajout de modules à l'application pour permettre, par exemple, une prise en main à distance pour certaines opérations, est également un point d'intérêt pour la suite.

6.3.2 Développements complémentaires apportés

Nous allons présenter les points importants des développements qui ont été nécessaires pour réaliser l'architecture proposée. Il faut rappeler qu'à la fin du Projet d'Option Web et Multimédia, cette architecture n'était qu'une possibilité, et que seule l'interface graphique web était fonctionnelle. À l'heure actuelle, et

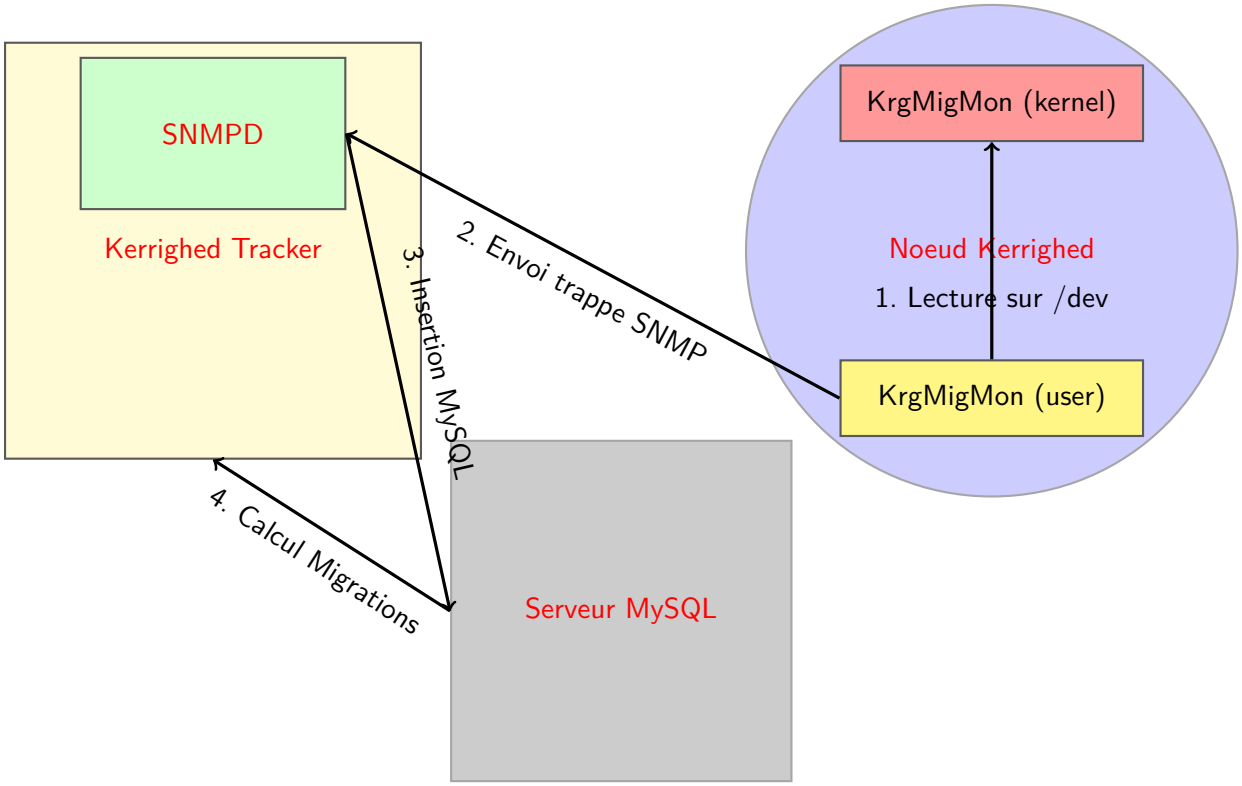


FIG. 6.1 – Architecture de capture des migrations mise en place – Schéma

après debug, l'ensemble de la chaîne fonctionne, et a permis de suivre des migrations réelles sur le cluster de test.

Module noyau de tracking

Le fonctionnement de ce module est très simple. Il écoute les évènements de début de migration, de fin de migration, et de migration avortée. Tout le reste s'articule sur l'utilisation des listes chaînées du noyau. Lorsque l'un des évènements pour lesquels nous souhaitons être tenu au courant se produit, le module récupère différentes informations :

- Le type d'évènement (début, fin ou avortement) ;
- L'identifiant du processus ;
- Le nœud du cluster en question (i.e. notre identifiant) ;
- La date, que nous utiliserons plus tard sous la forme d'un timestamp en nanosecondes.

À l'aide de ces informations, il construit un nouveau membre pour la liste chaînée dont nous parlions avant. Nous avons maintenant en espace noyau toutes les informations sur une (ou plusieurs) migrations en cours. Il reste à être en mesure de les communiquer. Plusieurs essais nous ont indiqués que la seule bonne solution pour opérer cette communication est de passer par un périphérique de type caractère, dans l'arborescence `/dev`. C'est le seul moyen de contrôler proprement et finement la sortie vers l'utilisateur. L'utilisation de `/proc` est donc impossible.

Nous enregistrons donc un nouveau périphérique de type caractère, sous le nom `migration_monitor` . L'élément important de cette opération est la structure `struct file_operations` que nous passons en paramètre : c'est elle qui indique quelle fonction implémentera le nécessaire pour l'appel système `read(2)`. Cette structure et le prototype de la fonction sont donnés dans le code 6.6. L'implémentation de l'appel système est ensuite assez simple. Il suffit de parcourir notre liste, et d'écrire dans un buffer les informations

que chaque élément de la liste contient.

Le formalisme des données communiquées à l'utilisateur est donné par `sprintf(buf + len, "%d :%d :%d :%lld\n", pos->event, pos->pid, pos->node, timespec_to_ns(&pos->date)) ;`. Après avoir été ajouté dans le buffer, chaque élément est retiré de la liste. À la fin, le buffer est copié à l'espace utilisateur : l'appel système `read(2)` réponds donc à l'application qui l'a exécutée, et cette dernière reçoit toute la liste des dernière migrations.

Ce fonctionnement permet d'écrire le mécanisme de récupération des données dans n'importe quel langage, tant que l'on peut lire un fichier. Il est à noter que chaque nœud ne verra qu'une migration : celle correspondant au départ ou à l'arrivée du processus. C'est donc du côté de la base de donnée que nous pourrons faire le lien, grâce aux dates et aux identifiants de processus. Il est donc nécessaire que tous les nœuds du cluster soient synchronisés en terme de date. Cependant, l'utilisation d'un serveur NFS impose déjà cette contrainte.

Un dernier point, anecdotique. L'utilisation d'un périphérique de type caractère dans `/dev` nécessiterait quelques définitions pour SysFS de sorte à ce que le fichier soit physiquement créé automatiquement, et avec les bon droits d'accès. Dans un soucis de concision et de simplicité, nous avons préféré proposer une règle pour `udev` qui créé automatiquement le fichier. Plus précisément, le fichier créé « de base » n'est accessible que par l'utilisateur `root` en lecture et en écriture. La règle proposée permet de le rendre accessible en lecture pour n'importe quel utilisateur.

Ce module doit être chargé manuellement sur chaque nœud du cluster. Pour faciliter l'utilisation, un script de démarrage a été écris et s'occupe de faire ce chargement automatiquement, après que le cluster ait été démarré. Attention, l'utilisation de ce module est incompatible avec l'utilisation de certaines probes fournies par le projet Kerrighed. Le problème est lié au fait qu'il n'est pas possible d'avoir plusieurs modules différents qui espionnent les migrations, simultanément.

```
static struct file_operations migration_monitor_fops = {
        .owner   = THIS_MODULE,
        .read    = migration_monitor_read
}

static ssize_t migration_monitor_read(struct file *, char __user *, size_t, loff_t *)
    ;
```

Code 6.6 – Structure `struct file_operations` et prototype pour `read(2)`

Daemon utilisateur de récupération

Le rôle de ce daemon, dont nous avons parlé précédemment, est de lire le contenu du fichier `/dev/migration_monitor`. La lecture se fait toutes les 500 milisecondes. Nous récupérons donc un nombre inconnu de lignes, chaque ligne correspondant à une information de migration. Ces lignes sont interprétées par l'application, qui en extrait les informations. La principale tâche, à ce niveau, outre la reconnaissance des différents champs, consiste en la conversion de la date en nanosecondes vers une chaîne de caractères normalisées pour l'insertion future dans la base de données MySQL.

À la fois pour des raisons de simplicité, et pour montrer le caractère « universel » et non lié à un langage, nous avons écris ce programme en utilisant Perl. Pas besoin de travailler en C. Perl est d'autant plus intéressant que nous faisons principalement des manipulations de chaînes de caractères, ce dans quoi il excelle.

Le reste du code s'occupe principalement de formater une trappe SNMP puis de l'envoyer. Nous utilisons à cet effet le module Perl `Net : :SNMP`, qui propose la méthode `snmpv2_trap`. Il nous reste à renseigner les champs de notre trappe.

La définition du serveur SNMP se fait lorsque l'on créé la session SNMP via la méthode `Net : :SNMP->session()`. C'est ensuite cette session que nous utilisons pour envoyer la trappe. Le script est à exécuter en

utilisateur sans droit spécifique, étant donné que le fichier à lire a été configuré, via `udev`, pour être lisible par n'importe quel utilisateur.

Ce script est à démarrer sur chacun des nœuds du cluster pour être en mesure de fonctionner. Pour nos tests, un script de démarrage a donc été réalisé et permet d'automatiser le lancement de cet outil.

Script d'insertion MySQL

Il a été présenté, lors de l'explication sur le module noyau, que nous n'avions pas une vue complète sur la migration à cet instant et qu'il serait nécessaire de faire le lien au niveau de la base de donnée. Nous avons choisi d'opérer ce lien au niveau de ce script d'insertion. Comme indiqué sur le schéma, ce script est appelé par le daemon SNMPd, lorsqu'il reçoit une trappe SNMP concernant Kerrighed.

Le script d'insertion a donc pour rôle, dans un premier temps, de récupérer les informations de la trappe. Ceci est fait en lisant l'entrée standard, où SNMPd nous passe toutes les informations. Une fois ces informations obtenues, nous les validons, de sorte à s'assurer qu'elles sont correctes, et semblent cohérentes.

La configuration de la gestion de la trappe est assez simple. Il faut d'abord activer le service `snmptrapd`, puis modifier son fichier de configuration pour qu'il traite les requêtes concernant notre MIB en exécutant le script dont nous parlons.

La partie faisant le lien entre les migrations arrivent ensuite. Lorsque la trappe SNMP est émise, il y est indiqué une information sur le type de migration : début, fin ou avortement. Quand nous recevons la donnée à insérer dans la base de données, nous utilisons cette information :

1. S'il s'agit d'un début de migration, nous sauvegardons toutes les données dans une table temporaire, appelée pour nos essais `tmpMigStart` ;
2. S'il s'agit d'une notification de fin d'une migration, nous recherchons dans cette table temporaire une migration correspondant à l'identifiant de processus. Si nous en trouvons une, et une seule, alors nous insérons la migration dans la table « définitive », celle utilisée par l'interface web. Et bien sûr, nous supprimons les informations de la table temporaire ;
3. Le cas d'une migration avortée n'est pas encore géré, mais il suffit de supprimer l'information de la table temporaire. À titre d'information, il s'avère que dans tous nos essais nous n'avons jamais réussi à générer ce type d'évènement.

À ce stade, notre application web est maintenant en mesure de récupérer et afficher les migrations des processus entre nœuds du cluster. Pour plus d'information sur cet aspect, nous renvoyons le lecteur à la lecture du rapport du projet [FL09].

MIB SNMP pour les trappes

Une MIB a pour objectif de formaliser les objets utilisés au sein d'une arborescence accessible par SNMP. Normalement, la MIB sert de base à l'écriture de modules C pour s'interfacer avec le daemon SNMPd et pouvoir gérer les données qu'il devra renvoyer. Dans notre cas, ce point est inutile, pour le moment. Nous nous plaçons dans la branche réservée aux « Entreprises », sous le nœud `.1.3.6.1.4.1.8192`.

Nous définissons ainsi, simplement, un objet de type Notification, qui contient tous les éléments envoyés par le noyau à l'application utilisateur, et qui doivent parvenir à la base de données. Le chemin d'accès de cet objet est `.1.3.6.1.4.1.8192.1.1.1.1`. Les champs qu'il proposent sont indiqués dans le code 6.7.

```
krgMigrationPid          OBJECT IDENTIFIER  ::= { krgMigration 1 }
krgMigrationStartDate    OBJECT IDENTIFIER  ::= { krgMigration 2 }
krgMigrationEndDate      OBJECT IDENTIFIER  ::= { krgMigration 3 }
krgMigrationStartNode    OBJECT IDENTIFIER  ::= { krgMigration 4 }
krgMigrationEndNode      OBJECT IDENTIFIER  ::= { krgMigration 5 }
krgMigrationType         OBJECT IDENTIFIER  ::= { krgMigration 6 }
```

Code 6.7 – Éléments présents dans l'objet de notification de migration SNMP

6.4 Centralisation des données, aide au debug et au suivi

Cette section est directement reprise du Wiki Kerrighed [Lis]. L'objectif est de mettre en place un serveur de journaux centralisé, qui permettra de suivre facilement la vie des nœuds du cluster, et surtout, qui permettra d'avoir accès très facilement aux stacktraces générées lors des inévitables crashs qui arriveront pendant le développement.

Nous allons dans un premier temps présenter la nécessité d'appliquer un patch au noyau, patch dont nous donnons le code en annexe I.4. Ensuite, nous passerons à l'installation et la configuration du serveur de journaux. Dans la pratique, nous réutilisons le serveur NFS pour fournir ce service, mais rien de l'impose.

La sous-section qui suivra présente l'installation et la configuration des nœuds pour envoyer les journaux vers ce serveur. Enfin, la dernière sous-section vise à mettre en place les outils pour faire en sorte de rediriger directement les messages de la console noyau, en plus des journaux systèmes.

Le trafic réseau généré se fera sur le port udp 6666. Dans la suite, le serveur NFS et le serveur syslog sont confondus, et identifiés par l'adresse `192.168.5.1`.

6.4.1 Préparation du noyau Kerrighed

Il est nécessaire de faire deux modifications dans le noyau Kerrighed. La première consiste à s'assurer que le module NetConsole est configuré pour être compilé en module, et non inclus dans le code du noyau. Pour cela, vérifier que le fichier `.config` contient bien l'option `CONFIG_NETCONSOLE=m`. Par défaut, il contient l'option `CONFIG_NETCONSOLE=y`, il suffit de changer le *y* en `m`.

La seconde consiste à appliquer un patch I.4. Ce patch a été proposé en 2004 sur les listes de diffusion du noyau et a été rejeté. La raison pour cela est simple, c'est que de l'avis des mainteneurs, ce n'est pas à un module dont le reste du noyau est en train de crasher qu'il convient de corriger le formatage des données.

Malgré tout, ce patch nous rends un service bien pratique : il permet de corriger la manière dont sont envoyées les données du noyau au serveur syslog, et permet d'avoir « le même rendu » dans les fichiers journaux qu'à l'écran.

Il serait intéressant de voir quelles seraient les modifications à réaliser dans les services syslog existant pour gérer le cas de NetConsole, et éviter l'utilisation de ce patch.

6.4.2 Installation d'un serveur Syslog centralisé

L'objectif de cette sous-section est d'installer et de configurer un serveur Syslog sur le serveur NFS qui sert actuellement les nœuds Kerrighed.

Nous utiliserons l'outil syslog-ng. Celui-ci nécessite de configurer deux éléments primordiaux :

– Une source de logging ;

– Une destination de logging.

La source et la destination peuvent ensuite être mises en relation au sein d'un élément `log` qui définit, lui, réellement les opérations de journalisation à faire. Optionnellement, des opérations de filtrages peuvent s'effectuer.

Installation de syslog-ng

Cette étape est triviale. Sur une distribution Debian, il suffit de faire ce qui est indiqué dans le code 6.8.

```
# aptitude install syslog-ng
```

Code 6.8 – Installation de Syslog-NG

Définition de la nouvelle source Syslog

Nous définissons une nouvelle source, comme étant tout ce qui arrive sur notre interface, sur le port udp 6666. La configuration correspondante est donnée dans le code 6.9.

```
source s_krg_nodes {
        # Collecter ce qui arrive en UDP
        udp(ip("192.168.5.1"), port(6666));
};
```

Code 6.9 – Configuration de Syslog-NG : définition de la source

Définition de la nouvelle destination syslog

Nous définissons une nouvelle destination. L'objectif est d'obtenir un fichier journal par nœud du cluster. Nous plaçons ces fichiers dans le répertoire `/var/log/krg_nodes/`. Assurez-vous que votre fichier de configuration de syslog-ng contienne `create_dir(yes)` ; de sorte à créer le répertoire automatiquement.

La configuration est donnée dans le code 6.10.

```
destination d_krg_nodes {
        file("/var/log/krg_nodes/$FULLHOST.log", perm(0644));
};
```

Code 6.10 – Configuration de Syslog-NG : définition de la destination

Ajout de la directive de journalisation

Maintenant, nous avons une source et une destination, il ne reste plus qu'à les mettre en relation. Ceci est réalisé au moyen de la directive de configuration `log`, comme présenté dans le code 6.11. Il ne restera plus qu'à redémarrer le service syslog-ng.

```
log {
        source(s_krg_nodes);
        destination(d_krg_nodes);
};
```

Code 6.11 – Configuration de Syslog-NG : ajout de la directive de journalisation

6.4.3 Configurer l'utilisation de Syslog pour les nœuds

Nous allons maintenant installer et configurer le service syslog-ng pour qu'il s'exécute sur chaque nœud du cluster. Son rôle sera de collecter les journaux locaux, grâce à `syslog(3)`, et de les envoyer à notre serveur syslog central.

Toutes ces opérations devront s'effectuer dans le répertoire racine des clients. Vous devrez donc commencer par changer de racine, en utilisant `chroot(3)`.

Installation de syslog-ng

La procédure d'installation est rigoureusement la même que pour le serveur. Elle est rappelée dans le code 6.12.

```
# aptitude install syslog-ng
```

Code 6.12 – Installation de Syslog-NG pour les nœuds

Définition de la nouvelle destination syslog

Il faut que nous définissions une nouvelle destination, correspondant à notre serveur syslog. Ce dernier fonctionne sur le port udp 6666, et a l'adresse `192.168.5.1`. La configuration résultante est donnée dans le code 6.13.

```
# Send all messages to the NFS/syslog server
destination d_nfs_syslog {
  udp("192.168.5.1", port(6666));
};
```

Code 6.13 – Configuration de Syslog-NG : définition d'un serveur distant

Modification du comportement de journalisation par défaut

Maintenant, il faut trouver la directive `log` qui gère le fichier `/var/log/syslog`. Dans la configuration Debian par défaut, il suffit de chercher la chaîne "# *.*;auth,authpriv.none -/var/log/syslog" pour la trouver.

Une fois cette directive trouvée, il reste à ajouter la destination que nous venons de créer, en insérant `destination(df_syslog);` entre les accolades.

Si vous le souhaitez, vous pouvez retirer la directive `destination(df_syslog);` pour ne pas envoyer de journal dans les fichiers « locaux », ceux montés sur le serveur NFS. Il sera peut-être nécessaire de faire la même modification dans toutes les directives `log`.

```
log {
        source(s_all);
        filter(f_syslog);
        destination(df_syslog);
        destination(d_nfs_syslog);
};
```

Code 6.14 – Configuration de Syslog-NG : changement du comportement par défaut pour envoyer vers un serveur distant

6.4.4 Journalisation du noyau avec NetConsole

Voici la dernière étape pour opérer notre centralisation. Il s'agit d'utiliser le module NetConsole pour rediriger les logs du noyau. Principalement, ce sont les erreurs lors des crashs que nous récupèreront de cette manière. Il est impératif, à ce stade, que vous ayiez activé le module NetConsole, et non l'avoir compilé dans le noyau.

Le script de chargement NetConsole.sh

Ce script est à placer dans le répertoire `/etc/init.d`, par exemple sous le nom `netconsole.sh`. Pensez bien à mettre les droits d'exécution dessus. Le code est donné en 6.15. Le script se borne à récupérer l'adresse IPv4 locale de l'interface demandée, puis à charger le module noyau avec les bon paramètres.

```
#!/bin/sh

extract_param()
{
  CMD=$(cat /proc/cmdline)
  DELIM=$1
  VAR="${CMD##*$DELIM}"
  VAR="${VAR%% *}"
  eval "$2=$VAR"
}
```

```
extract_param "logdev=" LOGDEV
extract_param "logdest=" DEST

CMDLINE=$(cat /proc/cmdline)
IPROUTE=$(ip -4 addr show dev ${LOGDEV}|grep inet|sed -e "s/␣//g" -e "s/inet//g")

REMOTEIP="${DEST%% :*}"
REMOTEPORT="${DEST##* :}"
LOCALIP="${IPROUTE%%/*}"
LOCALPORT=${REMOTEPORT}

MODPROBE=modprobe

echo "Loading␣NetConsole␣Kernel␣module␣...␣"
echo "${MODPROBE}␣netconsole␣do_syslog=yes␣netconsole=\"${LOCALPORT}@${LOCALIP}/${
    LOGDEV},${REMOTEPORT}@${REMOTEIP}/\""
${MODPROBE} netconsole do_syslog=yes netconsole="${LOCALPORT}@${LOCALIP}/${LOGDEV},${
    REMOTEPORT}@${REMOTEIP}/"
```

Code 6.15 – Script de chargement NetConsole

Configuration finale du noyau

La toute dernière opération à faire est la suivante. Maintenant que nous avons mis en place les outils pour rediriger la sortie noyau vers un serveur syslog, il reste à indiquer à notre script le serveur de destination. Pour faciliter l'administration, nous avons décidé de définir deux paramètres à passer au noyau, pour spécifier ceci :

- `logdev` : Correspond à l'interface qui sera utilisée pour envoyer les données ;
- `logdest` : Correspond à l'adresse IP et au port du serveur Syslog destinataires.

Il suffit d'ajouter ces deux paramètres à la configuration de démarrage du cluster, PXELinux ou PXEGrub suivant les choix effectués localement.

Dans notre environnement, nous passons donc en paramètre au noyau : `logdest=192.168.5.1 :6666 logdev=eth0`

6.5 Outils d'ordonnancement en espace utilisateur

C'est l'un des objectifs d'origine du Projet de Fin d'Études, que d'être en mesure d'opérer un ordonnancement totalement en espace utilisateur. À le fin de ce dernier, plusieurs hypothèses avaient été émises sur la manière de continuer, toujours dans l'optique de permettre un abaissement maximal de la barrière d'entrée pour développer de nouvelles méthodes.

Le but à terme sera d'intégrer ces développement dans l'arbre officiel de Kerrighed, sous plusieurs formes. Nous allons commencer par présenter les différents éléments nécessaires, et qui ont été implémentés. Il faut retenir que les développements actuels restent à l'état de « prototype », c'est à dire qu'ils fonctionnent, plutôt bien, mais qu'ils ne correspondent pas à l'implémentation finale, tant dans la forme qu'au niveau des fonctionnalités disponibles.

Un point important a toujours été de permettre de pouvoir simplifier la mise en œuvre de nouveaux algorithmes, en abstrayant un maximum de choses, et si possible sans imposer de langage pour développer. Ceci permet de laisser la liberté à la personne qui implémente ses algorithmes, pour le faire avec les outils les plus adaptés. Qui le souhaite pourrait ainsi interfacer Kerrighed avec CPlex.

Nous avons retenu et implémenter le découpage suivant :

- Une « boite à outils » pour Kerrighed, permettant de simplifier certaines opérations redondantes ;
- Une abstraction pour accéder aux données des probes ;

– Une implémentation « générique » pour Kerrighed des algorithmes RBT et Rudolph ;
– Réalisation d'un daemon d'ordonnancement communiquant par socket ;
– Une implémentation des clients pour les algorithmes RBT et Rudolph.

Chacun de ces points sera présenté par la suite. Ces éléments ont été implémentés en Perl, là encore pour montrer la faisabilité d'utiliser un autre langage que le C, et par simplicité pour obtenir rapidement quelque chose de fonctionnel. De même, pour simplifier l'utilisation et les tests par la suite, des scripts de démarrages et de chargement ont été développés. L'ensemble nécessitera donc d'être empaqueter pour fournir un système plus simple à installer.

6.5.1 Boite à outils pour Kerrighed

Ce module Perl vise à fournir tout un ensemble de services « de base », qui sont utiles pour tous les modules d'ordonnancement. Il est implémenté dans le paquet `Kerrighed : :SchedConfig : :Utils`.

À l'heure actuelle, il fournis uniquement deux fonctions intéressantes :
– `get_nodes_list` : retourne un tableau contenant la liste des identifiants des nœuds du cluster ;
– `get_process_list` : retourne un tableau contenant l'ensemble des identifiants de processus qui sont migrables.

Un point est important à faire sur la dernière fonction. Cette dernière est actuellement implémentée « rapidement », en parcourant les entrées de `/proc`, puis en vérifiant les capacités à l'aide de l'outil `krgcapset`. L'exécution est donc potentiellement consommatrice de ressources. Il est par ailleurs nécessaire que l'instance Perl qui utilisera ce module soit lancée avec la capacité Kerrighed `SEE_LOCAL_PROC_STAT` afin de ne retourner **que** les processus qui s'exécutent actuellement sur ce nœud.

6.5.2 Proposition d'une abstraction pour les probes

Ce module Perl, implémenté dans le paquet `Kerrighed : :SchedConfig : :Probes` propose quelques services dans le but de simplifier l'utilisation et la lecture des données depuis le ConfigFS.

Il propose trois grand types de fonctionnalités :
– Lecture de données, depuis une probe locale ou distante ;
– Lecture de données avec filtrage par expression régulière, depuis une probe locale ou distante ;
– Mise en place d'un mécanisme de cache sur les requêtes de lectures.

Les prototypes des fonctions importantes sont donnés dans le code 6.16.

Toutes ces fonctionnalités restent des abstractions simples, dans le sens où elles sont principalement là pour éviter la redondance de code inutile dans les implémentations des algorithmes d'ordonnancement, en regroupant des tâches répétitives. Avec les modifications opérées sur l'interaction ConfigFS et SchedConfig, la lecture des données d'une probe source, qu'elle soit locale ou distance, se résume à lire le fichier correspondant.

Les fonctions `grep_local_probe_source_value()` et `grep_remote_probe_source_value()` sont un peu plus évoluées, en ce sens qu'elles permettent d'utiliser des expressions régulières, appliquées sur chacune des lignes lues dans les fichiers, pour en extraire des données. Ceci réponds à une contrainte assez simple : les méthodes `SHOW` vont afficher toutes les données qu'elles ont lorsqu'elles concernent des probes sources paramétrées. Il faut donc traiter ce flux pour en extraire ce que l'on souhaite.

Enfin, le cache est implémenté de manière triviale, en utilisant une table de hashage, dont les clefs sont les chemins dans ConfigFS. Par défaut, le cache est désactivé, et la durée d'expiration des données est configurée à 30 secondes. Cette valeur est arbitraire, et il faudrait faire des tests pour vérifier sa pertinence. Elle peut de toute façon être changée, via la fonction `set_timeout`.

```
void set_caching(int active)
void set_timeout(int seconds)
$string read_local_probe_source_value(string probe, string probe_source)
$string read_remote_probe_source_value(string probe, string probe_source, int node)
```

```
@string grep_local_probe_source_value(string probe, string probe_source, string
    regexp)
@string grep_remote_probe_source_value(string probe, string probe_source, int node,
    string regexp)
```

Code 6.16 – Prototypes des fonctions d'abstraction Kerrighed en Perl

6.5.3 Implémentation des algorithmes RBT et Rudolph

Ces algorithmes d'ordonnancement ont été implémentés respectivement dans les paquets `Kerrighed : :SchedConfig : :Policies : :RBT` et `Kerrighed : :SchedConfig : :Policies : :Rudolph`. Ils utilisent bien évidemment les modules que nous avons développés précédemment, à savoir `Kerrighed : :SchedConfig : : Probes` et `Kerrighed : :SchedConfig : :Utils`.

Par convention, et pour simplifier les essais pour vérifier la bonne exécution des modules, chaque algorithme implémente et exporte deux fonctions, où `ALG` est à remplacer par le nom de l'algorithme :

- `ALG_calc` : Cette fonction ne prend aucun paramètre, et retourne une table de hashage de migrations ;
- `ALG_exec` : Cette fonction prend en paramètre une table de hashage de migrations, et exécute ou simule l'exécution des migrations.

Les tables de hashages utilisées sont simples. Les clefs qui les composent sont les identifiants des processus à migrer, et les valeurs associées sont, pour chacun des processus, l'identifiant du nœud sur lequel il sera migré. Cette formalisation des tables de hashage n'est pas obligatoire pour la suite, malgré tout elle est simple et efficace, et le code développé pour gérer à un plus haut niveau la question de l'ordonnancement repose fortement dessus.

L'implémentation en elle-même des algorithmes n'a aucune spécificité, elle montre simplement l'utilisation des primitives définies dans les modules précédents.

6.5.4 Implémentation d'un daemon d'ordonnancement en espace utilisateur

Cet outil est la dernière pierre que nous avons à poser pour simplifier l'écriture de politiques en espace utilisateur. Son existence se justifie pour permettre de s'abstraire du langage dans lequel le développeur aura à implémenter son algorithme d'ordonnancement. Malgré tout, certains « services » sont communs, et éviter la redondance de code est toujours une bonne chose.

Plusieurs solutions étaient envisageables, parmi lesquelles l'écriture d'une bibliothèque C, liée avec celle de Kerrighed, offrant ces services, et pour laquelles des passerelles seraient écrites avec d'autres langages. Une autre solution était l'écriture d'un daemon d'ordonnancement, offrant ces services, via une communication par sockets.

La simplicité et rapidité de test de la seconde l'a emportée, même si rien n'empêche de mutualiser le cœur des fonctions utilisées par ce daemon dans une bibliothèque. Nous allons donc présenter les spécifications générales de ce « serveur » d'ordonnancement, et son interaction avec ses futurs clients, puis nous présenterons quelques remarques sur l'implémentation.

Spécifications générales

Afin de permettre de s'abstraire du langage, l'utilisation d'une communication par sockets réseaux classiques a été retenue. Il a donc été nécessaire de spécifier un protocole. Nous n'avons pas de besoins très complexes, et idéalement, il fallait une solution rapide à implémenter. Le choix s'est porté sur l'utilisation de la syntaxe YAML, qui permet de facilement faire communiquer des outils entre eux, à l'instar de XML, en étant très simple à générer et à relire.

L'utilisation de YAML permet donc de construire des objets Perl, de simples tables de hashages, et de les envoyer pour communiquer. Nous proposons donc la spécification d'une table de hashage avec plusieurs clefs, à la convenance de chaque commande :

– `action`, qui défini l'action dont il est question ;
– `value`, qui défini la valeur associée à l'action.

Cette valeur est directement liée à l'action : certaines commandes utiliserons une simple chaîne de caractères, d'autres utiliserons des structures complexes. Nous proposons la liste des actions et les types de valeurs qui correspondent.

1. – Fermer la connexion
   – `action` : `quit`
   – `value` : aucune.
2. – Enregistrer le plugin avec le serveur
   – `action` : `register`
   – `name` : Nom du plugin. Il doit être unique parmi tous les plugins qui se connectent.
3. – Supprimer l'enregistrement du plugin (avant déconnexion)
   – `action` : `unregister`
   – `value` : Nom du plugin utilisé précédemment.
4. – Demande de calcul d'un ordonnancement par le serveur
   – `action` : `schedule`
   – `value` : aucune.
5. – Réponse d'un client avec un ordonnancement
   – `action` : `scheduling`
   – `schedule` : Table de hashage, où les clefs sont les identifiants des processus et les valeurs les nœuds où les migrer.

Remarques sur l'implémentation

L'implémentation des connexions réseaux est faite en utilisant le module Epoll pour Perl, qui permet d'exploiter l'appel système `epoll(2)`, un équivalent de `poll(2)` et `select(2)`, mais basé sur les évènements et plus performant.

La communication est actuellement définie sur le port TCP 4096, arbitrairement. Nous utilisons deux threads, l'un s'occupe de gérer le côté serveur, de faire les requêtes de calcul d'ordonnancement aux clients, et l'autre s'occupe d'effectuer les migrations. Les threads Perl ne partagent pas, par défaut, leurs variables. Il est nécessaire de forcer ce partage lors de la déclaration, en ajoutant `:shared`. Ceci doit être fait pour chaque niveau dans une structure un peu complexe.

L'implémentation actuelle est plus de l'ordre du prototype que d'un véritable outil robuste. Elle fonctionne, correctement. Une réécriture en C, en utilisant des threads POSIX serait plus pertinente.

Enfin, une partie du travail d'abstraction qui est fait dans les modules annexes gagnerait à être accessible au travers de l'interface mise en place par ce daemon, pour faciliter l'écriture des modules. Il faudrait peut-être, dès lors, envisager de passer le mécanisme de cache au niveau de la gestion de ces requêtes.

6.5.5 Implémentation de clients pour le daemon d'ordonnancement : RBT et Rudolph

Nous avons mis en place tout un ensemble d'outils qui visent à faciliter l'écriture de nouveaux algorithmes d'ordonnancement. Il est à présent temps de montrer la simplicité qu'il y a à ajouter un nouveau client à notre système. Attention, il est important de remarquer que le code pour effectuer le calcul en lui-même est déjà implémenté. Il ne reste qu'à interfacer ce code avec notre mécanisme. Cette situation est tout à fait acceptable, puisqu'il n'y a rien de choquant à implémenter l'algorithme d'ordonnancement à part.

Un client est donc réduit à sa plus simple expression. Tout ce qu'il doit implémenter se résume à :
– Ouvrir une connexion vers le serveur ;
– S'enregistrer auprès du serveur ;
– Attendre une demande de calcul ;

– Effectuer le calcul et le retourner au serveur.

Un exemple avec le client pour l'algorithme RBT est proposé dans l'annexe I.5. Un total de 66 lignes, dont la moitié consiste en de simples déclarations de variables et import de modules.

Il faut noter que ce code a réussi à faire migrer des processus avec succès. Nous avons été retardés dans notre élan par les crashs générés à cause du bug « Soft Lockup », dont l'explication est donnée dans la section 6.6. Mais avant que le problème ne se manifeste, il a été possible de faire fonctionner RBT et Rudolph, pour équilibrer la charge sur notre cluster composé de quatre machines. Tout ceci depuis l'espace utilisateur.

6.6 Détection et correction d'un bug de type « Soft lockup »

Lorsque toute la pile logicielle en espace utilisateur a été complète, les premiers tests ont pu être lancés, afin de vérifier que tout fonctionnait comme nous l'attendions. Et le succès était au rendez-vous. Seulement pour quelques minutes, cependant. En effet, le code qui s'exécute imprime une charge certaine sur les modules noyaux qui ont été développés. Même si, dans l'absolu cette charge est assez faible, elle est supérieure à tous les tests que nous avions été en mesure de mener jusqu'à présent. La tenue de notre code pendant quelques minutes est donc déjà une bonne nouvelle. Mais après un certain temps, de l'ordre de cinq à dix minutes, une surprise de taille s'est offerte à nous : le noyau s'est arrêté.

Un message laconique sur la sortie console de ce dernier : "BUG Soft lockup on CPU#0". Parfois, c'était l'autre processeur qui était indiqué. Souvent, après quelques minutes d'attente, nous avions deux messages, un pour chaque processeur. Il a donc fallu admettre l'évidence : le code est buggué. Cela donnait des traces du type de celle présente dans le code 6.17. Cette trace indique également que c'est dans notre code que quelque chose ne semble pas être correct, puisqu'on voit que la dernière fonction en jeu, avant l'entrée dans le `spin_lock` est la méthode `SHOW` de la probe source `mattload`. Pour arriver à corriger ce problème, trois étapes seront nécessaires :

1. Compréhension précise de ce qu'est un bug de type « Soft Lockup » ;
2. Identification de l'origine du problème conduisant à cet état ;
3. Proposition d'une correction et validation.

Dans la suite, nous allons présenter chacune de ces étapes.

```
Aug 26 16:52:40 192.168.5.65 BUG: soft lockup detected on CPU#1!
Aug 26 16:52:40 192.168.5.65 [<c013e1b3>] softlockup_tick+0xa7/0xb4
Aug 26 16:52:40 192.168.5.65 [<c0128fa5>] update_process_times+0x45/0x68
Aug 26 16:52:40 192.168.5.65 [<c01129ac>] smp_apic_timer_interrupt+0x70/0x7f
Aug 26 16:52:40 192.168.5.65 [<c0105c40>] apic_timer_interrupt+0x28/0x30
Aug 26 16:52:40 192.168.5.65 [<c0376021>] _spin_lock+0xa/0xf
Aug 26 16:52:40 192.168.5.65 [<f88612f6>] mattload_source_show_value+0xe/0x3a [
    mattload_probe]
Aug 26 16:52:40 192.168.5.65 [<f93d10a4>] handle_scheduler_pipe_show_remote_value+0
    x2c/0xad [kerrighed]
Aug 26 16:52:40 192.168.5.65 [<f93d10b4>] handle_scheduler_pipe_show_remote_value+0
    x3c/0xad [kerrighed]
Aug 26 16:52:40 192.168.5.65 [<f93d1078>] handle_scheduler_pipe_show_remote_value+0x0
    /0xad [kerrighed]
Aug 26 16:52:40 192.168.5.65 [<f9412f1e>] rpc_handler_kthread_void+0x50/0x6e [
    kerrighed]
Aug 26 16:52:40 192.168.5.65 [<f9412bed>] thread_pool_run+0x2dc/0x5bd [kerrighed]
Aug 26 16:52:40 192.168.5.65 [<f9412911>] thread_pool_run+0x0/0x5bd [kerrighed]
Aug 26 16:52:40 192.168.5.65 [<c0133239>] kthread+0xf6/0x12d
Aug 26 16:52:40 192.168.5.65 [<c0133143>] kthread+0x0/0x12d
Aug 26 16:52:40 192.168.5.65 [<c0105dc3>] kernel_thread_helper+0x7/0x10
Aug 26 16:52:40 192.168.5.65 =======================
```

Code 6.17 – Trace d'un problème Soft Lockup

6.6.1 Soft Lockup : explication

Après une recherche intensive, particulièrement infructueuse, un document est venu répondre à la question préalable : « Qu'est-ce qu'un bug de type Soft Lockup ? À quoi cela peut-il être dû et/ou lié ? ». Ce document, [Den], traite de la question générale du debug dans le noyau linux, et principalement de la correction des problèmes de type *lockup*, *panic* et *oops*.

C'est donc la première catégorie qui nous intéresse. Les erreurs de ce type sont tout simplement liées à une mauvaise utilisation des mécanismes de verrouillage contenus au sein du noyau linux. Une fois que l'on a dit ça, tout est dit. Mais un peu plus d'informations est intéressant. En creusant la question de ces erreurs, on trouvera une donnée importante : le détecteur de « Soft Lockup » se met en fonctionnement lorsqu'il constate qu'une fonction, du noyau, s'exécute sans n'avoir jamais rendu la main au reste du noyau pendant une période de dix secondes.

Cela signifie donc que la trace que nous avons nous indique que la méthode `SHOW()` a été arrêtée par le détecteur de *Soft Lockup*. Deux possibilités à ça : soit la méthode fait une boucle qui se passe mal, soit autre chose se passe mal. Après une rapide vérification, aucune boucle à proximité de cette fonction. Par contre, la lecture des fonctions précédentes, qui appellent notre méthode, donne une indication capitale : la probe source gère un verrou juste avant de nous appeler.

À la lecture de la trace, ce serait donc l'acquisition du verrou qui pose un problème. Nous sommes sur une piste intéressante. À ce niveau, nous avons déjà appris une chose importante : la raison de l'apparition de cette erreur, et des voies d'exploration pour trouver une explication à cet arrêt brutal de l'exécution.

6.6.2 Identification du problème

Maintenant que nous savons ce qu'est cette erreur, se pose la question de « Pourquoi ce verrou n'arrive-t-il pas à être acquis ? ». Ceci rejoint les problèmes de debug dans les programmes utilisant des fils d'exécution multiples. Ce genre de problème est déjà, en espace utilisateur, non trivial à identifier. L'espace noyau ne va pas forcément nous faciliter la tâche. L'aide de Louis Rilling, de Kerlabs, a été un point capital, tant dans la compréhension et l'assistance à apprendre à débugger en espace noyau, que dans l'identification du problème en lui-même.

Avec l'information obtenue, à savoir que le verrou d'une probe source pose problème, nous avons une piste : étudier ce verrou. Pour la suivre, des modifications sont opérées dans le module noyau de kerrighed : la fonction chargée d'appeler la méthode `SHOW` d'une probe source est instrumentée. Ceci nous permet de récupérer le propriétaire du verrou. C'est-à-dire que nous saurons quel processus fait appel, via le noyau, à cette méthode. Nous n'obtenons pas directement l'identifiant du processus, son PID, mais l'adresse de la structure `struct task_struct` en espace noyau, correspondant à ce processus. Cette instrumentation est plutôt simple à réaliser, il s'agit juste d'indiquer la valeur du membre `parent` de la structure `struct scheduler_probe_source`. La modification opérée est donnée dans le code 6.18. La sortie obtenue est donnée dans le code 6.19.

L'objectif de ces modification est donc de savoir précisément dans quel contexte le verrou qui n'arrive pas à être obtenu se trouve. En effet, s'il n'arrive pas à être obtenu, c'est probablement qu'un autre processus l'a toujours, alors qu'il ne devrait plus l'avoir. C'est donc ce que nous cherchons à vérifier, en obtenant le contexte, le processus, dans lequel celui-ci est « bloqué ». Une fois que l'adresse du processus est connue, il ne reste qu'à consulter la backtrace correspondante, grâce à la commande KDB `btt`. Ceci est illustré dans le code 6.20. À ce niveau, nous savons donc dans quel ordre s'est déroulé le problème :

1. Réception d'une requête de lecture de valeur de probe source à distante par notre nœud ;
2. Traitement de la demande par le framework Kerrighed, et plus précisément par nos modifications ;
3. Appel à la méthode `SHOW` correspondante ;
4. Cette méthode a besoin d'informations de probes locales, qu'elle va donc obtenir ;
5. L'une de ces probes est celle qui correspond à la vitesse d'horloge des processeurs ;

6. La méthode GET de la probe source `cpu_speed` s'exécute donc ;
7. Nous faisons appel au module CPUFreq pour obtenir la valeur ;
8. Les appels se transmettent ... ;
9. Premier élément étrange : appel à `default_wake_function` ;
10. Deuxième élément étrange : appel à `wait_for_completion+0x90` ;
11. Mise en exergue du soucis : `schedule+0x987`.

Concrètement, pendant la lecture de la vitesse d'horloge du processeur par le module CPUFreq, nous perdons la main. À la lumière de cet état de fait, Louis Rilling comprends exactement le soucis. Pour des raisons qui ne nous intéressent pas, le framework SchedConfig de Kerrighed empêche de s'endormir. S'endormir, c'est arrêter l'exécution du code en cours pour permettre d'en exécuter un autre. Cela permet d'améliorer la latence [Mol]. Cette opération se fait grâce à la macro `might_sleep()` [Dev]. Or, comme l'indique le commentaire de cette fonction, il y a des contextes où il ne faut pas le faire. SchedConfig en est donc un.

C'est pour cette raison que l'exécution s'arrête. Pendant la lecture de la valeur, ce qui ne devrait pas arriver dans le contexte du framework SchedConfig arrive : nous nous endormissons. Dès lors, le verrou qui avait été acquis sur la probe source ne sera pas relâché. Plus tard, ce verrou attendra indéfiniment pour être acquis. Et cela génèrera une erreur « Soft Lockup ».

```
void scheduler_probe_source_lock(struct scheduler_probe_source *probe_source)
{
        printk(KERN_DEBUG "PID␣%d :␣scheduler_probe_source_lock(%p);␣
            scheduler_probe_lock(%p)\n", current->pid, probe_source, probe_source->
            parent);
        scheduler_probe_lock(probe_source->parent);
}
```

Code 6.18 – Instrumentation de la fonction scheduler_probe_source_lock

```
Aug 28 10:10:26 192.168.5.64 PID 4230667 : scheduler_probe_source_lock(f7bd2508);
    scheduler_probe_lock(f7bd22f0); owner=c22307b0
```

Code 6.19 – Résultat de l'instrumentation de la fonction scheduler_probe_source_lock

```
Aug 28 10:26:26 192.168.5.64 [0]kdb> btt 0xc22307b0
Aug 28 10:26:26 192.168.5.64
Aug 28 10:26:26 192.168.5.64 Stack traceback for pid 4230497
Aug 28 10:26:26 192.168.5.64 0xc22307b0  4230497       11  0    0   R  0xc2230960
    krgrpc(166:125
Aug 28 10:26:26 192.168.5.64 esp        eip        Function (args)
Aug 28 10:26:26 192.168.5.64 0xf7397d90 0xc038debd schedule+0x987
Aug 28 10:26:26 192.168.5.64 0xf7397dbc 0xf881d068 [load_probe]kmcb_process_on+0xa
Aug 28 10:26:26 192.168.5.64 0xf7397dc0 0xc038cc1a module_hook_call+0x59
Aug 28 10:26:26 192.168.5.64 0xf7397e30 0xc038e37d wait_for_completion+0x90
Aug 28 10:26:26 192.168.5.64 0xf7397e3c 0xc011af3d default_wake_function
Aug 28 10:26:26 192.168.5.64 0xf7397e58 0xc011b8ee set_cpus_allowed+0xcd
Aug 28 10:26:26 192.168.5.64 0xf7397eb4 0xc01106f3 cpufreq_p4_get+0x30
Aug 28 10:26:26 192.168.5.64 0xf7397ec0 0xc030b26e cpufreq_get+0x35
Aug 28 10:26:26 192.168.5.64 0xf7397ed4 0xf8824145 [cpuspeed_probe]
    cpu_speed_probe_source_get_value+0x67
Aug 28 10:26:26 192.168.5.64 0xf7397eec 0xf94685d7 [kerrighed]
    scheduler_source_get_value+0x4e
Aug 28 10:26:26 192.168.5.64 0xf7397ef8 0xf88240de [cpuspeed_probe]
    cpu_speed_probe_source_get_value
Aug 28 10:26:26 192.168.5.64 0xf7397f0c 0xf94686c9 [kerrighed]
    scheduler_sink_get_value+0x26
```

```
Aug 28 10:26:26 192.168.5.64 0xf7397f1c 0xf8827197 [mattload_probe]matteo_load_calc+0
    xd3
Aug 28 10:26:26 192.168.5.64 0xf7397f40 0xf88273ff [mattload_probe]
    loadinc_source_show_value+0x18
Aug 28 10:26:26 192.168.5.64 0xf7397f44 0xf946927e [kerrighed]
    handle_scheduler_pipe_show_remote_value+0x2c
Aug 28 10:26:26 192.168.5.64 0xf7397f58 0xf946928e [kerrighed]
    handle_scheduler_pipe_show_remote_value+0x3c
Aug 28 10:26:26 192.168.5.64 0xf7397f6c 0xf9469252 [kerrighed]
    handle_scheduler_pipe_show_remote_value
Aug 28 10:26:26 192.168.5.64 0xf7397f74 0xf94abbc8 [kerrighed]
    rpc_handler_kthread_void+0x50
Aug 28 10:26:39 192.168.5.64 0xf7397f94 0xf94ab897 [kerrighed]thread_pool_run+0x2df
Aug 28 10:26:39 192.168.5.64 0xf7397fc4 0xf94ab5b8 [kerrighed]thread_pool_run
Aug 28 10:26:39 192.168.5.64 0xf7397fcc 0xc01344f0 kthread+0xf6
Aug 28 10:26:39 192.168.5.64 0xf7397fd8 0xc01343fa kthread
Aug 28 10:26:39 192.168.5.64 0xf7397fe4 0xc0105f3b kernel_thread_helper+0x7
```

Code 6.20 – Backtrace d'un processus, avec KDB

6.6.3 Correction proposée

Maintenant que nous savons que c'est la manière de lire la fréquence CPU, plusieurs solutions sont possibles pour corriger le problème :

1. Exploiter la variable `cpu_khz` du noyau, pour lire la valeur. Assez peu de documentation précise disponible à ce sujet ;
2. Utiliser une autre fonction du module CPUFreq : `cpufreq_quick_get()`.

À l'heure actuelle, la seconde solution a été retenue. Mais elle reste temporaire, puisque la lecture de son code et des fonctions auxquelles elle fait appel n'est pas rassurant : le risque de se retrouver avec un bug « Soft Lockup » n'est pas écarté, certaines font appel à `might_slep()`. Contact a été pris avec le mainteneur du module CPUFreq pour avoir plus de précisions sur la meilleure méthode pour notre cas. En dernier recours, l'utilisation de `cpu_khz` restera possible, mais il conviendra de bien étudier les limites qu'elle impose.

Cependant, toute fragile qu'est potentiellement cette correction, le cluster a fonctionné plusieurs heures d'affilées une fois que l'appel à `cpufreq_get()` a été changé par un appel à `cpufreq_quick_get()`. Il faudrait cependant valider de manière plus pertinente cette solution, notamment en ayant l'avis du mainteneur, Dave Jones.

7. Conclusion

L'origine de ce projet se trouvait dans les résultats des travaux de thèse de Matthieu Pérotin, pour qui la question se posait de savoir s'il était intéressant de mettre en place des politiques d'ordonnancement différentes au sein d'un même cluster Kerrighed. Pour être en mesure de répondre à cette question, plusieurs axes ont été explorés. D'abord, il a été nécessaire de s'intéresser aux algorithmes que nous pourrions implémenter pour calculer des ordonnancements dans le cluster.

Ensuite, après avoir étudié les travaux effectués les années passées sur l'utilisation de Kerrighed, et en tenant comptes des algorithmes potentiellement intéressants à implémenter, un nouveau problème s'est dégagé. Il s'agit d'être en mesure de réaliser l'ordonnancement dans l'espace noyau, alors que Kerrighed n'offre pas tous les outils pour le faire.

Il a donc été nécessaire d'analyser précisément ce qu'il manquait encore pour pouvoir faire le travail depuis l'espace utilisateur, puis, une fois que les blocages ont été identifiés, nous avons implémenté les modifications nécessaires pour passer outre.

Parallèlement, les travaux menés par Mathieu Dabert ont été repris de sorte à porter ce qui avait été développé sur la nouvelle version de Kerrighed, dans l'infrastructure SchedConfig. L'utilisation de ce framework nous a poussé à prendre contact avec l'équipe de développement du projet, et plus précisément Louis Rilling. Ce dernier nous a fait part de son intérêt pour nos travaux, tant sur le plan des nouveaux modules de probe que nous développions et qui pourraient intéresser d'autres personnes dans la communauté, que sur le plan de réaliser l'ordonnancement en espace utilisateur.

Cet aspect de collaboration avec Kerlabs est particulièrement intéressant, puisqu'il implique que les résultats de ce projet seront réutilisés par la suite. C'est un vecteur de motivation supplémentaire.

À l'heure actuelle, donc, il reste principalement un axe à développer : il s'agit d'arrêter une solution technique pour la mise en place de l'ordonnancement en espace utilisateur, afin de pouvoir donner un cadre aux développeurs.

Du côté de l'espace noyau, quelques points doivent être complétés, et le plus gros travail restant consistera à préparer les patchs, et à les soumettre. Une fois ces modifications validées, le code que nous avons écrit sera officiellement inclus. Il devrait couvrir à la fois les modules noyaux qui gèrent la présence d'un utilisateur interactif, les modules de probe, de filtre et de politique, les outils annexes que nous avons mis en place, comme les modules PAM et WinLogon, ainsi que le serveur destiné à s'exécuter dans la machine virtuelle. Enfin, nous proposerons également les scripts pour le Kerrighed Test Project destinés à faire les tests de non régression pour tous les modules noyau écrits.

La suite de ce projet constituera principalement en la mise en œuvre technique de l'ordonnancement en espace utilisateur, au moyen de la solution que nous retiendrons. Une fois cet aspect complété, alors nous pourrons implémenter différents algorithmes, et pouvoir étudier les effets d'un ordonnancement hybride au sein du cluster Kerrighed.

À l'issue de ce projet, il me reste donc à remercier tous ceux qui ont permis sa réalisation et sa concrétisation, notamment Matthieu Pérotin, pour l'idée et l'encadrement, et aussi Louis Rilling, de Kerlabs, pour l'aide précieuse qu'il a fournis et ses retours pertinents.

Conclusion suite au stage

À la fin du Projet de Fin d'Études, nous avions été en mesure de proposer les derniers outils en espace noyau qui manquaient pour permettre de réaliser en totalité l'ordonnancement depuis l'espace utilisateur. Le but principal de la période de stage était donc d'arriver à mettre en place tous les outils, au moins à l'état de prototypes fonctionnels, pour faire cet ordonnancement, en espace utilisateur. La mise en place des outils pour suivre l'état du cluster était aussi un besoin critique, dans l'optique de mesurer la pertinence des algorithmes hybridés.

La solution technique qu'il convenait d'arrêter pour permettre la mise en place de cet ordonnancement est retenue : nous proposerons une bibliothèque écrite en C, liée sur la bibliothèque kerrighed de base, qui proposera des services d'abstractions tels qu'une boite à outils, un mécanisme extensible pour simplifier l'accès aux données des probes sources, avec mise en cache des données au besoin. Cette bibliothèque sera complétée par un daemon d'ordonnancement, écrit aussi en C, qui reposera sur cette dernière, et dont l'objectif est de mutualiser un maximum de services, pour arriver au but ultime : l'écriture de nos modules d'ordonnancement comme de petits clients réseaux indépendants, en terme de langage, du serveur précité.

Cette solution a été prototypée au cours du stage, en utilisant Perl, et les premiers tests ont validés son fonctionnement global. Deux clients, un pour RBT et un pour Rudolph sont implémentés et fonctionnels.

Les points qui devaient être complétés dans les modules noyaux l'ont été. Il s'agissait principalement de corriger les fuites mémoires. Cette étape a permis de mettre en place un guide pour assister dans d'autres développements, sur la manière de faire cette correction dans le contexte noyau. Une fois ces corrections effectuées, le code a été intégré, localement, à l'arbre des sources de Kerrighed. C'est-à-dire qu'il était possible de compiler notre code en même temps que l'ensemble du système. Ceci nous a permis de créer nos premiers « vrai » patchs, séparés par « fonctionnalités » pour faciliter leur application sur les sources officielles.

Nous obtenons ainsi la liste des patchs suivants :

- `00_expose_remote_nodes.patch` : Modification de ConfigFS pour permettre l'accès à distance aux probes sources ;
- `10_userspace_ordo_tests.patch` : Modifications apportées à l'arbre des sources pour implémenter nos tests unitaires ;
- `20_cpuspeed_probe.patch` : Ajout de la probe `cpu_probe` ;
- `21_load_probe.patch` : Ajout de la probe `load_probe` ;
- `22_processsize_probe.patch` : Ajout de la probe `processsize_probe` ;
- `23_user_probe.patch` : Ajout de la probe `user_probe` ;
- `24_mattload_probe.patch` : Ajout de la probe `mattload_probe` ;
- `30_rbt_scheduler.patch` : Ajout de l'ordonnanceur `RBT`, politique et filtre ;
- `40_Kconfig.patch` : Mise en place des modifications pour inclure nos outils à la configuration de la compilation ;
- `50_rbt_policy_Kbuild.patch` : Modification sur le système de compilation pour proposer nos options : politique RBT ;
- `51_rbt_cache_filter_Kbuild.patch` : Modification sur le système de compilation pour proposer nos options : filtre RBT ;
- `52_probes_Kbuild.patch` : Modification sur le système de compilation pour proposer nos options : compilation des probes ;

Ces patchs ont été soumis pour une première revue « privée », à Louis Rilling, responsable de la partie Scheduler et architecte de SchedConfig. Les premiers retours sont plutôt positifs, mais quelques modifications

seront apportées, à la fois pour corriger les bugs qui ont été découverts entre temps, et pour se conformer au style de code de tout le projet.

Enfin, le dernier point technique qui restait à régler était la finalisation et l'intégration des systèmes pour interfacer PAM et WinLogon avec notre cluster. Par manque de temps, cet aspect n'a pas pu être traité, mais le code développé dans le cadre du projet de Système d'Exploitation par Quention Proust et Tarik Roumadni a déjà été testé partiellement avec succès. Il reste des problèmes à corriger malgré tout, avant que ces composants ne soient pleinement utilisables.

Par ailleurs, et également par manque de temps à la date de la rédaction du rapport, les tests pour qualifier la pertinence d'un ordonnancement hybride entre RBT et Rudolph n'ont pas pu être menés à bien. Ils le seront une fois que la solution aura été stabilisée. L'intégralité de la plateforme technique est cependant prête.

Dans l'optique de ces tests, le système de suivi des migrations qui avait été proposé dans le cadre du projet d'Option Web et Multimédia, alliant une application Web écrite en Rails, un module noyau pour obtenir l'information de migration, et la remontée de ces données à un serveur centralisé a été développée, et implémentée avec succès. Là encore, les premiers tests ont été concluants, malgré le manque de stabilité de l'ensemble lié aux bugs dans nos modules.

Les dernières touches à apporter à ces travaux consistent donc en la fiabilisation et la stabilisation de ce qui est ne l'est pas à l'heure actuelle, puis à mener une campagne de tests pour répondre à la question de la pertinence d'algorithmes hybrides. Malgré tout, dans l'optique d'une réponse négative, tous ces travaux n'auront pas été effectués en vain. Nous apportons quand même de nouveaux outils pour simplifier le développement de nouvelles politiques d'ordonnancement, ainsi qu'un retour d'expérience sur l'utilisation du framework SchedConfig. Il faut savoir que nous avons été les premiers à développer des modules, hors Kerlabs. Nous avons également initié un projet de système de monitoring web, dont l'utilité n'est pas à démontrer à la vue du nombre de requêtes similaires qui ont été émises sur les groupes de discussions liés à Kerrighed.

A. Cahier des Charges – Présentation du projet

Nous allons commencer par présenter le projet dans sa globalité, en le replaçant dans son contexte, en définissant les objectifs auxquels il concourt. Nous décrirons l'existant, et enfin, nous finirons par donner une date de livraison au projet.

Les critères définissant l'acceptabilité du produit seront par ailleurs donnés à la fin de ce chapitre.

A.1 Contexte

Ce projet s'inscrit dans la suite de la thèse de Matthieu Pérotin, « Calcul Haute Performance sur Matériel Générique » [Pér08]. Le but de ce travail de recherche, maintenant achevé, était d'étudier la mise en place d'une grille de calcul réparti au sein d'une institution – par exemple, au sein de l'École, puis de l'Université entière –, dans le but de résoudre le problème de faisabilité et/ou de performances.

En effet, certains chercheurs ont des besoins tels que : soit leur application est trop lente sur leur poste – problème de performance –, soit celui-ci n'a tout simplement pas les dimensions nécessaires pour envisager l'exécution du programme – problème de faisabilité. Le travail de recherche a conclu sur la mise en place d'une solution technique ainsi que d'algorithmes d'ordonnancement, dont nous reparlerons dans la section A.3.

L'aspect technique a été développé et présenté dans le cadre du projet de fin d'étude de Mathieu Dabert[Dab08]. Les algorithmes d'ordonnancement ont été étudiés au travers de projets de programmation par contraintes [CL06, GN07]. Les résultats ont été présentés lors de conférences, [PME08a, PME08b].

A.1.1 Définitions

Au cours du travail de thèse [Pér08], différents types d'utilisateurs ont pu être identifiés :

- L'utilisateur de Calcul Haute Performance. Typiquement, un chercheur à qui se pose le problème de performance ou de faisabilité ;
- L'utilisateur Interactif. Typiquement, un étudiant sur une machine d'une salle de TP, ou un personnel administratif ;
- Les administrateurs du réseaux.

A.2 Objectifs

Les objectifs qui sont visés par ce projet de fin d'étude sont multiples, certains directement liés aux travaux de thèse, d'autres en sont la continuité :

- Adapter l'existant à la nouvelle version de Kerrighed ;
- Améliorer les modifications réalisées sur Kerrighed ;
- Proposer de nouveaux algorithmes d'ordonnancement ;
- Permettre l'utilisation de plusieurs « familles » d'algorithmes au sein du même cluster.

A.3 Description de l'existant

Les différents travaux, les projets de programmation par contrainte [CL06, GN07], le projet de fin d'étude [Dab08] ou bien l'achèvement de la thèse[Pér08], ont permis de réaliser concrètement un cluster fonctionnel.

Tout d'abord, nous avons un cluster qui utilise la technologie des Systèmes à Image Unique – SSI Single System Image – telle que mise en place avec Kerrighed. Ce système réparti exploite la version 2.2.0 du projet Kerrighed.

Afin de répondre à certaines contraintes soulevées au travers du travail de thèse [Pér08], l'architecture technique exploite les possibilités offertes par la virtualisation : les nœuds de calcul seront des machines virtuelles, exécutées sur les ordinateurs de l'institution. Pour ne pas perturber l'utilisation que font les utilisateurs interactifs, un composant fait le lien entre la machine physique et la machine virtuelle membre du cluster, afin que cette dernière soit informée de la présence – ou non – d'un utilisateur interactif et adapte en conséquence les algorithmes d'ordonnancement. Concrètement, si une personne utilise interactivement l'ordinateur, le membre du cluster associé aura tendance à limiter le nombre de processus qu'il acceptera.

Les machines virtuelles démarrent via le réseau, et chargent leur système depuis un serveur de fichier NFS. Ce dernier contient toute l'arborescence classique d'un système Linux. Kerrighed étant distribué sous la forme d'un ensemble d'outils en espace utilisateur et d'un patch noyau pour la version 2.6.20, tout ceci doit être installé sur le système de fichier exporté par NFS. Toute la procédure de mise en place détaillée d'un cluster Kerrighed est détaillée, pour la version 2.2.0 dans le projet de fin d'étude de Mathieu Dabert[Dab08].

Les projets de programmation par contrainte [CL06, GN07] ont permis de caractériser des algorithmes d'ordonnancement, en plus de ceux qui ont été implémentés dans le cluster. On peut citer pour ces derniers les algorithmes suivants :

- Rudolph[RSAU91] ;
- RBT[Pér08].

Tous ont été présentés à la fois au cours de la thèse[Pér08], mais également dans les articles[PME08a, PME08b]. De ces résultats se dégagent plusieurs points importants :

- Migrer des processus à un intérêt – résultats de l'algorithme RANDOM face à NULL – ;
- Certains algorithmes ont des propriétés différentes :
  - Réponse au problème de faisabilité – RUDOLPH – ;
  - Réponse au problème de performance – RBT –.

L'implémentation de ces algorithmes a montré la nécessité d'avoir de nouvelles informations sur les processus exécutés par le noyau Linux. Dans le cadre du Projet de Fin d'Études de Mathieu Dabert [Dab08], une API a été implémentée à cet effet, permettant d'obtenir toutes les informations nécessaires pour l'écriture de nouveaux algorithmes d'ordonnancement. Ainsi, par exemple, l'algorithme RBT s'écrit en environ 200 lignes de C.

A.4 Date de livraison

Le projet devra-t-être terminé à la fin du stage de Master 2 Recherche, c'est à dire fin septembre 2009.

B. Expression des besoins

B.1 Besoins fonctionnels

Dans le cadre qui nous est donné, nous avons identifié les besoins fonctionnels qui suivent. Ces besoins identifient le travail à réaliser dans le cadre du projet.

1. Porter l'API développée dans le cadre du projet de fin d'étude précédent[Dab08] sur Kerrighed Trunk ;
2. Porter les deux ordonnanceurs implémentés dans le cadre du projet de fin d'étude précédent[Dab08] sur la nouvelle architecture de Kerrighed Trunk ;
3. Développer un module permettant une communication entre l'espace utilisateur et l'espace noyau, pour la notification de présence d'un utilisateur interactif. Ce module se limite à ouvrir une entrée `/proc` à définir, à laquelle on transmettra le nombre d'utilisateurs connectés ;
4. Développer un démon Unix en espace utilisateur, écoutant l'interface réseau virtuelle du noeud de calcul, réalisant le travail de comptage du nombre d'utilisateurs connectés sur la machine physique, via un protocole de communication – à définir – lui transmettant l'information de connexion et de déconnexion d'un utilisateur. Ce démon notifiera par la suite le noyau via l'interface précédemment indiquée ;
5. Développer un module PAM permettant la notification du système virtualisé de la connexion et de la déconnexion d'un utilisateur interactif ;
6. Développer un module Winlogon permettant la notification du système virtualisé de la connexion et de la déconnexion d'un utilisateur interactif ;
7. Implémenter une API noyau sous forme de module permettant la communication avec un service en espace utilisateur réalisant l'ordonnancement ;
8. Spécifier et développer un service en espace utilisateur, dialoguant avec le noyau, et réalisant l'ordonnancement des processus suivant différents algorithmes implémentés sous forme de bibliothèque partagée.

B.2 Besoins non fonctionnels

Nous regroupons ici les besoins du projet qui ne se traduiront pas nécessairement par une production de code.

9. Identifier et documenter les changements entre Kerrighed 2.2.0 et la version Trunk avec SchedConfig ;
10. Étudier la possibilité de faire fonctionner l'ordonnanceur en espace utilisateur ;
11. Proposer et étudier de nouvelles heuristiques d'ordonnancement moins triviales que celles actuellement usitées.

B.3 Hiérarchisation des besoins

Priorité est donnée aux besoins fonctionnels 7, 1, 2, étant donné que pour le moment, les besoins fonctionnels 3, 4, 5 et 6 peuvent être circonvenus. Ils n'empêchent pas le système de fonctionner, même s'ils seront nécessaires à un déploiement grandeur nature.

Nous devons cependant également hiérarchiser les besoins non fonctionnels. Il parait trivial que le besoin 9 est prioritaire vis-à-vis des besoins 10 et 11. Nous pouvons même aller plus loin en affirmant que le besoin 9 est prioritaire sur les besoins 1 et 2 : impossible de réaliser correctement ce travail de portage – il ne s'agira pas uniquement de trivialement sortir les modifications propres aux travaux précédents – sans savoir précisément ce qui a changé entre les deux versions de Kerrighed.

Par ailleurs, deux besoins sont liés, il s'agit des besoins 7 et 8. Leur but est de permettre de réaliser l'ordonnancement des processus en espace utilisateur. Ceci viendra en supplément de la possibilité actuelle d'avoir des ordonnanceurs en espace noyau sous forme de module. Notons que le projet Kerrighed est intéressé par cette possibilité.

Certains besoins sont plus secondaires que d'autres. Le besoin 11 est également plus critique que les besoins 3, 4, 5, et 6 : comme énoncé plus haut, ces besoins ne sont pas actuellement comblés – ou plutôt, ils le sont de manière non pérenne – et n'empêchent pas le système de fonctionner. Par contre, mettre au point des algorithmes plus performants, à savoir le besoin 11 reste plus important.

Nous pouvons donc établir deux ensembles : les besoins fondamentaux, et les besoins secondaires. Dans le premier, on trouvera les besoins suivants : 7, 9, 1, 2, 10 et 11. Dans le second, nous trouvons : 3, 4, 5 et 6.

Pour une vue plus globale, et une meilleure idée de la répartition des tâches, un diagramme de Gantt est disponible à la section E.1.2, en page 114.

C. Présentation des besoins

C.1 Portage de l'API (Besoin 1)

Comme indiqué précédemment, il s'agit de transposer l'API qui a été implémentée au cours du Projet de Fin d'Étude [Dab08] sur la nouvelle infrastructure que propose la branche de développement officielle de Kerrighed : SchedConfig.

Cette API consistant uniquement en une remontée d'informations, il convient en pratique de développer de nouveaux modules de « probe » pour s'intégrer dans cette nouvelle manière de gérer l'ordonnancement.

Après une première étude du fonctionnement de ces mécanismes, nous avons pu identifier les modules de récupération de données que nous devrons développer, et ceux déjà présents dans Kerrighed que nous aurons la possibilité de réutiliser :

- Obtention de la mémoire libre d'un nœud : besoin déjà comblé par le module `mem_probe` de Kerrighed ;
- Charge (au sens Unix) moyenne d'un nœud : besoin déjà comblé par le module `cpu_probe` de Kerrighed ;
- Consommation en Jiffies d'un processus : besoin déjà comblé par le module `mosix_probe` de Kerrighed ;
- Vitesse et nombre de processeurs d'un nœud : inexistant, création du module `cpuspeed_probe` ;
- Présence d'un ou plusieurs utilisateurs sur la machine hôte : inexistant à l'heure actuelle, création du module `user_probe` ;
- Taille d'un processus en nombre de pages : inexistant, création du module `processsize_probe` ;
- Charge d'un nœud en nombre de processus activables (état *runnable*) : inexistant, création du module `load_probe`.

C.2 Portage des ordonnanceurs (Besoin 2)

Dans la continuité de [Dab08] et du point précédent, nous allons également (re)développer sous forme de modules noyau les méthodes d'ordonnancement qui ont été évaluées précédemment, à savoir RBT et Rudolph.

Nous nous appuierons également sur l'infrastructure SchedConfig qui propose un système beaucoup plus souple et modulaire pour réaliser l'ordonnancement en espace noyau. Nous développerons donc deux modules, `rbt_policy` et `rudolph_policy`.

C.3 Signalisation de l'utilisation des machines (Besoins 3, 4, 5, 6)

C.3.1 Interface noyau (Besoin 3)

Il s'agit ici de développer un module en espace noyau, que nous nommerons `local_user_presence` , qui se contentera de définir un simple compteur, sous la forme d'un entier. Autour de ce compteur, on développera une petite API permettant de le manipuler, en exportant les symboles suivants dans le noyau, permettant ainsi d'incrémenter, décrémenter, etc.

Les symboles à exporter sont tous ceux nécessaires à la manipulation du compteur, à savoir :

- Connexion d'un utilisateur : `local_user_presence_user_connection` ;

– Déconnexion d'un utilisateur : `local_user_presence_user_disconnection` ;
– Nombre d'utilisateurs présents : `local_user_presence_user_connected` ;
– Le nœud est-il libre ? (`local_user_presence_node_free`) ;
– Le nœud est-il occupé ? (`local_user_presence_node_used`) ;

De plus, afin de permettre la communication des informations entre l'espace utilisateurs, nous implémenterons un autre petit module noyau, `local_user_notifier`, dont le rôle est de s'appuyer sur le précédent et d'offrir une interface dans le système de fichier `/proc`, à l'adresse `/proc/kerrighed/interactive_user`, et qui permette d'accéder à ces fonctionnalités du compteur.

Nous aurons ainsi une correspondance directe entre les entrées du répertoire cité précédemment et les fonctions exportées depuis le module de comptage :

– `local_user_presence_user_connection` ⇒ `/proc/kerrighed/interactive_user/connecion` ;
– `local_user_presence_user_disconnection` ⇒ `/proc/kerrighed/interactive_user/disconnection` ;
– `local_user_presence_user_connected` ⇒ `/proc/kerrighed/interactive_user/get` ;
– `local_user_presence_node_free` ⇒ `/proc/kerrighed/interactive_user/isfree` ;
– `local_user_presence_node_used` ⇒ `/proc/kerrighed/interactive_user/isused`.

C.3.2 Communication machine virtuelle (Besoin 4)

Nous développerons un simple démon, qui aura un rôle un peu étoffé. Tout d'abord, il réceptionnera les évènements des plugins PAM et WinLogon, et il les relayera au noyau au moyen de l'interface décrite précédemment, et implémentée dans `/proc/kerrighed/interactive_user/`

Ce démon tournera dans la machine virtuelle, et écoutera un port utilisateur. Proposons le port 7919, qui semble libre d'après l'IANA[1].

Un protocole assez simple sera suffisant entre le démon et les modules. Proposons d'envoyer le caractère 'C' à la connexion d'un utilisateur, et le caractère 'D' à sa déconnexion.

C.3.3 Module machine réelle – Linux (Besoin 5)

Ce module PAM va avoir pour rôle d'intercepter les évènements de connexion et de déconnexion d'un utilisateur, et de les transmettre au démon unix précédemment présenté.

Il devra prendre en paramètre une chaîne permettant de décrire l'adresse IP du démon qu'il doit contacter, et ce de manière assez générique. On suppose en effet que les adresses des machines virtuelles sont « prédictibles » en ce sens où elles correspondront aux VLANs.

Par exemple, admettons que l'on ait un hôte en salle Unix B, ayant comme adresse IP *10.172.52.64*, la machine virtuelle qui sera hébergée sur cette machine aura comme adresse *192.168.52.64*.

Dès lors, on passera donc en paramètre le formatage de l'adresse IP cible, par exemple sous la forme suivante `ip=192.168.\$3.\$4`, en s'inspirant de ce que proposent les expressions régulières Perl pour la substitution de données.

C.3.4 Module machine réelle – Windows (Besoin 6)

Le besoin concernant le module pour WinLogon est symétrique.

C.4 Ordonnancement en espace utilisateur : nécessaire en espace noyau (Besoin 7)

Afin de prendre des décisions d'ordonnancement en espace utilisateur, il va être nécessaire de récupérer des informations depuis l'espace noyau. Les informations seront récupérées via le système des probes proposé par Kerrighed.

[1] http://www.iana.org/assignments/port-numbers

Il nous reste cependant à être en mesure de proposer à l'espace utilisateur d'accéder simplement à ces données. Une première approche consisterait à exploiter le système de fichier `/config`. Si cette idée est pertinente pour la récupération des informations de la machine locale, elle se heurte à une limite : impossible de récupérer les données à distance par ce biais.

Nous devons donc mettre en place un mécanisme qui permette à l'espace utilisateur de récupérer tout type de données proposé par toutes les probes. Idéalement, ces dernières ne devraient pas à être modifiées : le système permettant d'y accéder à distance depuis l'espace utilisateur doit être, autant que possible, transparent pour les modules de probe.

Par ailleurs, il serait très intéressant que ce système soit le plus flexible et le plus souple possible : découverte des probes disponibles « à chaud » par exemple.

C.5 Ordonnancement en espace utilisateur : mise en place (Besoin 8)

Pour calculer l'ordonnancement en lui-même, il nous faut un outil, un démon de préférence, qui aura à charge de prendre les décisions et d'effectuer lui-même les migrations, grâce à l'appel système `migrate` mis à disposition par Kerrighed.

Il sera nécessaire de poser les spécifications d'une bibliothèque ou d'une API permettant d'abstraire un minimum la récupération des données depuis les probes (masquer la complexité). Cette API aura en charge non seulement la communication directe avec l'espace noyau, mais également le calcul de la fameuse Mesure de Charge.

Il faudrait également mettre en place un système de plugins, similaire à SchedConfig : chaque module d'ordonnancement sera un plugin du démon.

Les plugins n'auront d'autre but que de :

- Calculer le processus à déplacer ;
- Calculer le nœud où déplacer le processus.

C.6 Changements apportés depuis Kerrighed 2.2.0 (Besoin 9)

Il s'agit, non pas de fournir une documentation complète des changements opérés entre la version 2.2.0 de Kerrighed, sur laquelle étaient précédemment basés les travaux, et la version de développement incluant SchedConfig, mais plutôt d'expliciter :

- Les changements apportés par SchedConfig ;
- En quoi SchedConfig répond déjà en partie à nos besoins ;
- Présenter les éléments importants constitutifs de SchedConfig.

Ce besoin est nécessaire afin de pouvoir identifier le travail de portage qu'il y aura besoin de réaliser.

C.7 Possibilité de réaliser l'ordonnancement en espace utilisateur (Besoin 10)

Il convient ici de s'intéresser à identifier les éléments qui sont nécessaires à permettre de réaliser les décisions d'ordonnancement en espace utilisateur. Il sera intéressant dans un premier temps de regarder ce que propose Kerrighed pour éventuellement déjà effectuer des migrations depuis l'espace utilisateur, et si ce n'est pas le cas, étudier et mettre en place le nécessaire pour le faire.

C.8 Proposition et étude de nouvelles méthodes d'ordonnancement (Besoin 11)

C'est, *in fine*, l'un des buts de ce projet. Actuellement, les algorithmes d'ordonnancement qui sont mis en place, que ce soit RBT ou RUDOLPH restent plutôt triviaux. C'est dû en partie aux contraintes inhérentes à l'espace noyau, où les facilités sont bien moindres.

Puisque nous sortons de l'espace noyau, nous pouvons nous permettre plus de latitude, c'est pourquoi nous devons proposer de nouveaux algorithmes d'ordonnancement, moins triviaux que les actuels.

Une fois l'ordonnancement en espace utilisateur fonctionnel, nous les testerons sur un cluster réel. En attendant que les outils soient prêts pour effectuer ce travail, les algorithmes seront évalués grâce à la plateforme SimGrid, développée par Hamza Benarafa au cours de son projet de fin d'études [Ben09].

D. Contraintes

D.1 Coûts

Considérant les spécificités de Kerrighed, nous pouvons entièrement travailler avec des machines virtuelles. Les outils sont déjà présents au sein de l'école, il n'y a donc rien de spécifique à se procurer.

D.2 Délais

La totalité du projet doit être terminée à la fin du stage de Master 2 Recherche. Cependant, il sera nécessaire que le travail soit suffisamment avancé à la date de la fin des Projets de Fin d'Études. La réalisation des besoins fonctionnels 1, 2 et 7 ainsi que les besoins non fonctionnels 9, 10 et 11 serait souhaitée.

D.3 Contraintes techniques

Puisque nous utilisons le système à image unique Kerrighed, nous sommes nécessairement dans l'obligation de travailler avec le système Linux 2.6.20. Du fait du travail en espace noyau qui sera nécessaire, le langage utilisé sera le langage C. De plus, le choix technique qui a été retenu pour la virtualisation est VMware.

Afin de garder une trace de l'avancement du travail, l'utilisation d'un dépôt Subversion est nécessaire. Nous utiliserons l'infrastructure proposée par l'école – Redmine et Subversion – à cet effet.

D.4 Contraintes organisationnelles

Afin de simplifier la réutilisabilité du travail, il sera mis sous deux formes différentes dans le dépôt Subversion :

- Sous la forme d'un arbre de sources Kerrighed et Linux prêt à l'emploi, incluant toutes les modifications réalisées activées ;
- Sous la forme de *patchs* identifiés et à appliquer sur les sources de Kerrighed officielles. Cet aspect simplifiera le portage des modifications effectuées sur les prochaines versions de Kerrighed. Ces patchs seront idéalement découpés suivant les fonctionnalités qu'ils apportent et les fichiers qu'ils impactent.

On prendra donc soin de s'assurer que les patchs produits s'appliquent correctement à la version de Kerrighed visée.

D.5 Autres contraintes

Il sera nécessaire de respecter les mêmes standards de code que ceux utilisés par Kerrighed. Aussi, on s'efforcera de produire du code qui soit assez générique pour être utilisable par d'autres organisations. Couplées ensembles, ces contraintes maximisent les chances de voir nos modifications incluses dans Kerrighed, limitant ainsi le travail parallèle nécessaire – maintenir une copie locale de tout l'arbre des sources est fastidieux.

E. Déroulement du projet

E.1 Planification

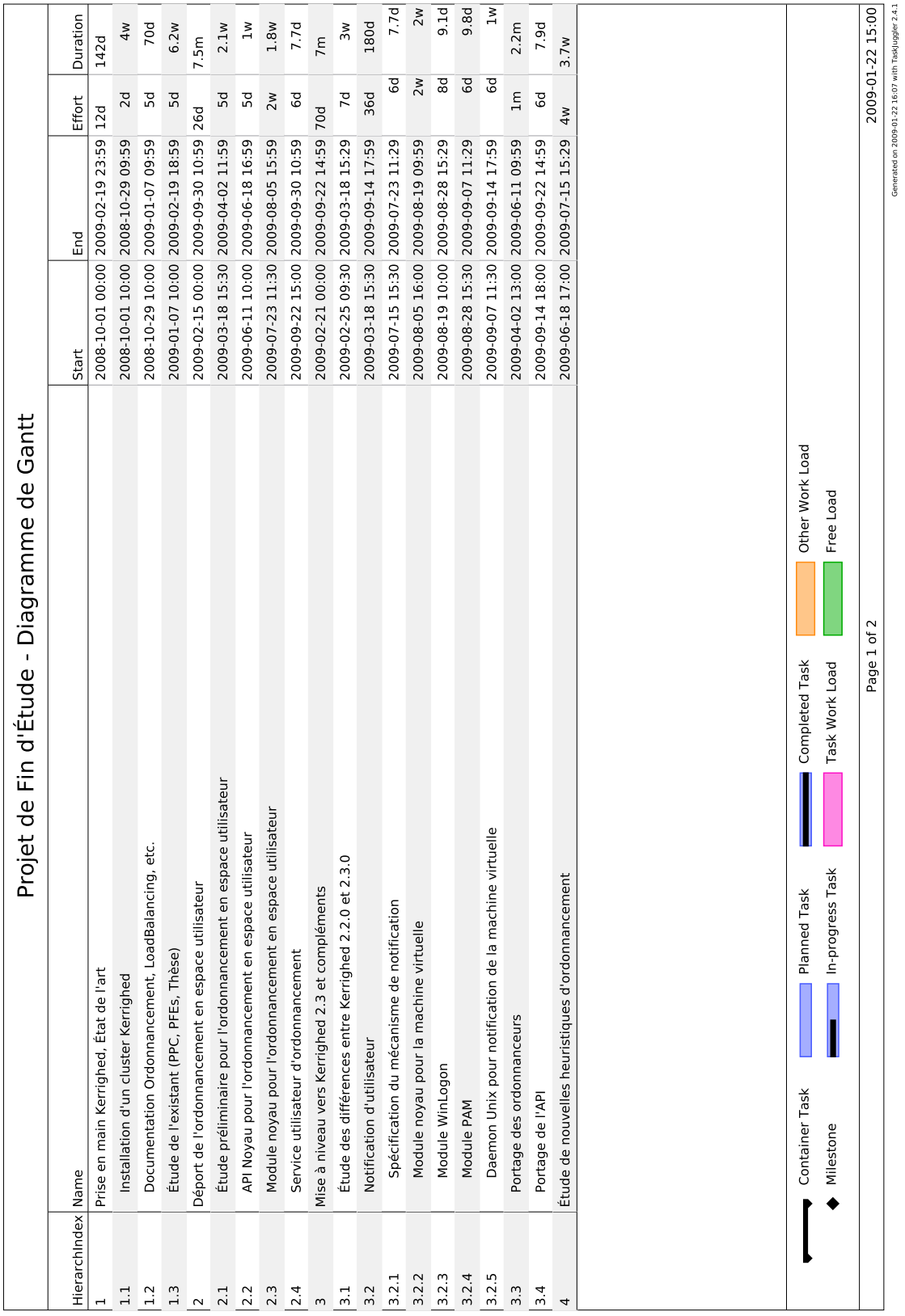
En tenant compte des impératifs liés au Master 2 Recherche, il résulte que jusqu'à la fin février, seuls des travaux préliminaires peuvent se dérouler. Il s'agira donc surtout des études quant aux changements entre les versions de Kerrighed, et au portage de l'existant.

Par la suite, les développements nouveaux pourront se mettre en place. En parallèle peuvent être réalisés de nouveaux algorithmes d'ordonnancement, et la mise en place d'une API permettant la communication entre l'espace noyau et un ordonnanceur en espace utilisateur.

L'utilisation de TaskJuggler permet de créer plusieurs scénarios, qui nous permettrons surtout de gérer la planification, puis le suivi du projet : la première étape sera effectuée au travers du scénario `plan`, et la seconde, au travers du scénario `delayed`.

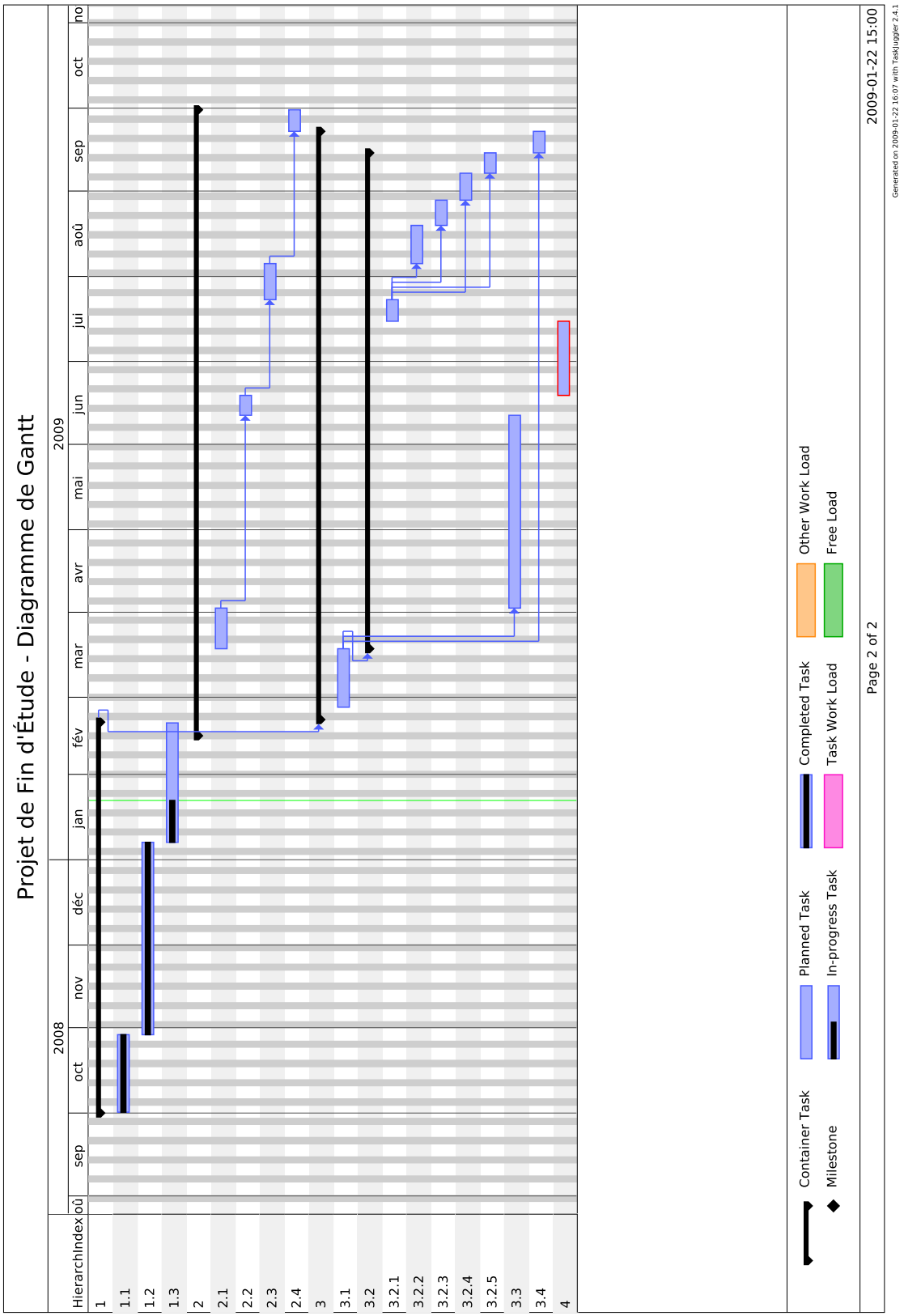
E.1.1 Liste des tâches planifiées

Le projet a été modélisé à l'aide de l'outil TaskJuggler, afin de fournir un ordonnancement sous forme de diagramme de Gantt. Voici la liste des tâches correspondantes. L'ordonnancement est donné en section E.1.2. Cette planification est donnée à titre indicatif. Nous utiliserons les capacités proposées par l'outil pour revenir à la fin du projet sur les différences entre ce qui était prévu et la planification qui aura été effectivement utilisée.



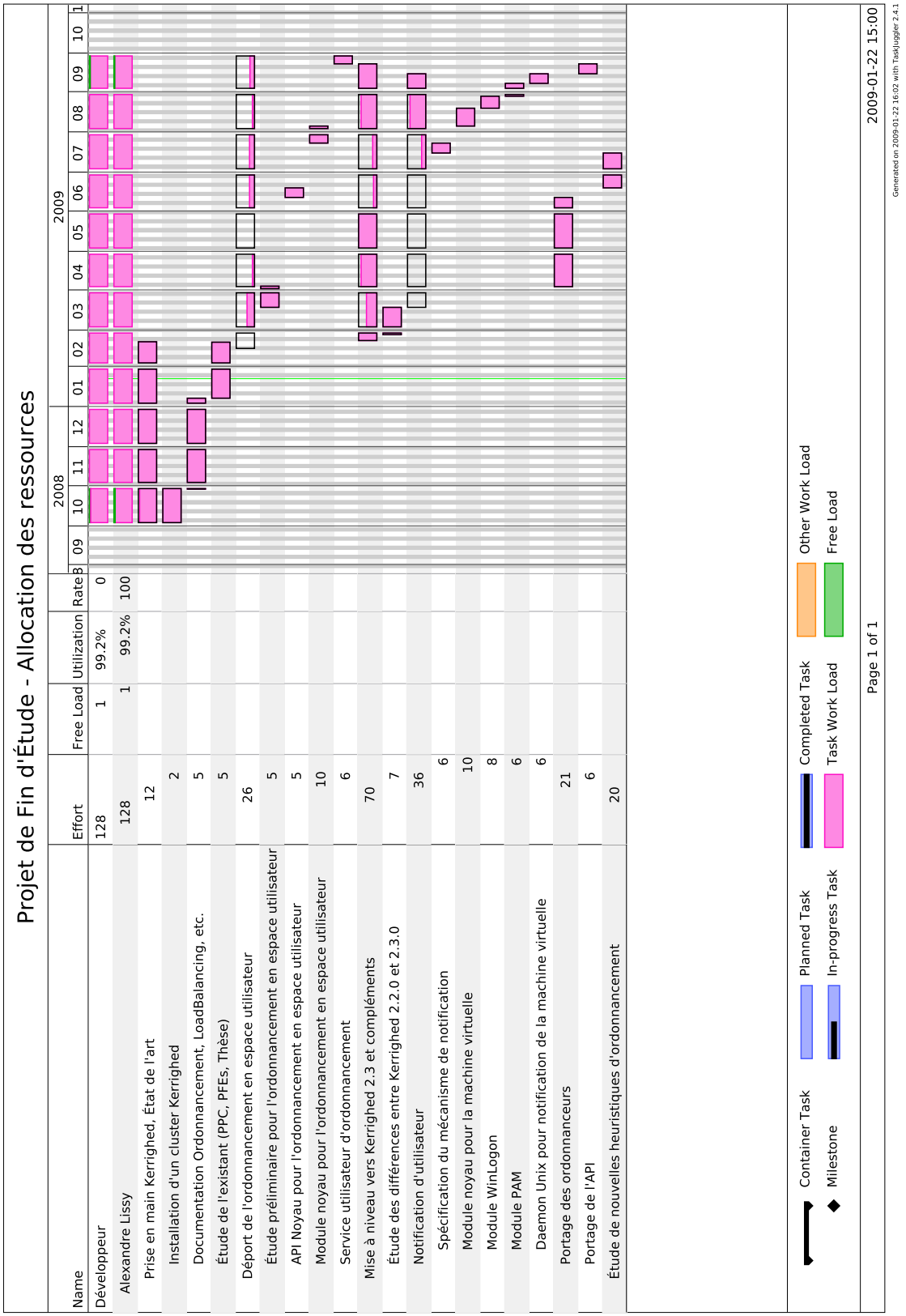
E.1.2 Diagramme de Gantt

Voici un ordonnancement prévisionnel des tâches à réaliser pour mener à bien le projet. L'utilisation des ressources liée à cet ordonnancement est donnée en section E.1.3.



E.1.3 Utilisation des ressources

Voici à présent la répartition de l'utilisation des ressources suivant chacune des tâches programmées.



E.2 Plan d'assurance qualité

Afin de s'assurer de la qualité du code produit, nous allons exploiter l'architecture de tests unitaires utilisée par Kerlabs, à savoir le projet Kerrighed Test Project[K+], qui fût présenté au Kerrighed Summit'07[Ren07].

Il s'agit d'une version adaptée du Linux Test Project, dont le but originel est le test des appels systèmes du noyau.

Le Kerrighed Test Project a donc pour objectif de mettre en place des tests de non régression sur le code du projet, afin de maintenir la qualité. Il nous faudra donc écrire de nouveaux tests pour ce projet, afin de permettre de vérifier et valider le code que nous ajouterons.

Ainsi, au cours de nos développements, nous devrons formaliser les tests pour les éléments suivants :

1. Dans un premier temps, lors du portage des ordonnanceurs actuels, valider leur fonctionnement à l'aide de KTP.
2. Ensuite, valider le fonctionnement des modules permettant la mise à jour des informations dans le noyau, par exemple pour notifier la présence d'un utilisateur.
3. Enfin, valider le fonctionnement pour le côté espace noyau du système de communication nécessaire à un calcul d'ordonnancement en espace utilisateur.

E.3 Responsabilités

E.3.1 Maîtrise d'ouvrage

La maîtrise d'ouvrage est confiée à l'étudiant.

E.3.2 Maîtrise d'œuvre

La maîtrise d'œuvre est confiée à l'encadrant.

F. Code des probes écrites

F.1 Probe CPUSpeed

```
/**
 * Kerrighed Local CPU Informations Probe module
 *
 * Copyright (c) 2009 - Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>
 *
 * Part of Mathieu Dabert's API (Copyright (c) 2008)
 **/

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kernel_stat.h>
#include <linux/threads.h>
#include <linux/cpumask.h>
#include <linux/list.h>
#include <linux/cpufreq.h>

#include <scheduler/core/probe.h>

static struct scheduler_probe *cpuspeed_probe;

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>");
MODULE_DESCRIPTION("Local CPU Informations probe");

DEFINE_SCHEDULER_PROBE_SOURCE_GET(cpu_connected_probe, int, value_p, nr)
{
  *value_p = num_possible_cpus();
  return 1;
}

DEFINE_SCHEDULER_PROBE_SOURCE_SHOW(cpu_connected_probe, page)
{
  return sprintf(page, "%d\n", num_possible_cpus());
}

static BEGIN_SCHEDULER_PROBE_SOURCE_TYPE(cpu_connected_probe),
  .SCHEDULER_PROBE_SOURCE_GET(cpu_connected_probe),
  .SCHEDULER_PROBE_SOURCE_SHOW(cpu_connected_probe),
  .SCHEDULER_PROBE_SOURCE_VALUE_TYPE(cpu_connected_probe, int),
END_SCHEDULER_PROBE_SOURCE_TYPE(cpu_connected_probe);

DEFINE_SCHEDULER_PROBE_SOURCE_GET_WITH_INPUT(cpu_speed_probe, unsigned int, value_p,
    nr, unsigned int, idx_p, in_nr)
{
  int i;
  unsigned int khz;

  if (in_nr) /* Only returns the CPU queried */
    for (i = 0; i < in_nr && i < nr; i++) {
      if (idx_p[i] < num_possible_cpus()) {
```

```
50          khz = cpufreq_quick_get(idx_p[i]);
51          /* same idea than in arch/x86/kernel/cpu/proc.c :108 */
52          if (khz == 0)
53            khz = cpu_khz;
54
55          *value_p++ = khz;
56        } else
57          return -EINVAL;
58      }
59    else  /* Show as many CPUs possible */
60      for (i = 0; i < nr && i < num_possible_cpus(); i++) {
61        khz = cpufreq_quick_get(i);
62        /* same idea than in arch/x86/kernel/cpu/proc.c :108 */
63        if (khz == 0)
64          khz = cpu_khz;
65
66        *value_p++ = khz;
67      }
68    return i;
69  }
70
71  DEFINE_SCHEDULER_PROBE_SOURCE_SHOW(cpu_speed_probe, page)
72  {
73    size_t size = SCHEDULER_PROBE_SOURCE_ATTR_SIZE;
74    ssize_t count;
75    int tmp_count;
76    int i;
77    unsigned int khz;
78
79    printk(KERN_INFO "cpu_speed_probe_show␣function␣called.\n");
80
81    khz = cpufreq_quick_get(0);
82    /* same idea than in arch/x86/kernel/cpu/proc.c :108 */
83    if (khz == 0)
84      khz = cpu_khz;
85
86    tmp_count = snprintf(page, size, "CPU#%d :␣%ukHz\n", 0, khz);
87    count = tmp_count;
88
89    for (i = 1; tmp_count >= 0 && count+1 < size && i < num_possible_cpus(); i++) {
90      khz = cpufreq_quick_get(i);
91      /* same idea than in arch/x86/kernel/cpu/proc.c :108 */
92      if (khz == 0)
93        khz = cpu_khz;
94
95      tmp_count = snprintf(page + count, size - count, "CPU#%d :␣%ukHz\n", i, khz);
96      if (tmp_count >= 0)
97        count += tmp_count;
98    }
99
100   return (tmp_count < 0) ? tmp_count  : min( (size_t)count+1, size );
101 }
102
103 static BEGIN_SCHEDULER_PROBE_SOURCE_TYPE(cpu_speed_probe),
104   .SCHEDULER_PROBE_SOURCE_GET(cpu_speed_probe),
105   .SCHEDULER_PROBE_SOURCE_SHOW(cpu_speed_probe),
106   .SCHEDULER_PROBE_SOURCE_VALUE_TYPE(cpu_speed_probe, unsigned int),
107   .SCHEDULER_PROBE_SOURCE_PARAM_TYPE(cpu_speed_probe, unsigned int),
108 END_SCHEDULER_PROBE_SOURCE_TYPE(cpu_speed_probe);
109
110 static struct scheduler_probe_source *cpuspeed_probe_sources[3];
111
```

```
112 static SCHEDULER_PROBE_TYPE(cpuspeed_probe_type, NULL, NULL);
113
114 static int __init cpuspeed_probe_init(void)
115 {
116   int err = -ENOMEM;
117
118   cpuspeed_probe_sources[0] = scheduler_probe_source_create(&cpu_connected_probe_type
        , "connected");
119   cpuspeed_probe_sources[1] = scheduler_probe_source_create(&cpu_speed_probe_type, "
        speed");
120   cpuspeed_probe_sources[2] = NULL;
121
122   cpuspeed_probe = scheduler_probe_create(&cpuspeed_probe_type, "cpuspeed_probe",
        cpuspeed_probe_sources, NULL);
123
124   if (cpuspeed_probe == NULL) {
125     printk(KERN_DEBUG "error : cpuspeed_probe creation failed !\n");
126     goto out_kmalloc;
127   }
128
129   err = scheduler_probe_register(cpuspeed_probe);
130   if (err)
131     goto err_register;
132
133   printk(KERN_INFO "cpuspeed_probe loaded.\n");
134
135   return 0;
136
137 err_register:
138   scheduler_probe_free(cpuspeed_probe);
139 out_kmalloc:
140   if(cpuspeed_probe_sources[0] != NULL)
141     scheduler_probe_source_free(cpuspeed_probe_sources[0]);
142   if(cpuspeed_probe_sources[1] != NULL)
143     scheduler_probe_source_free(cpuspeed_probe_sources[1]);
144   if(cpuspeed_probe_sources[2] != NULL)
145     scheduler_probe_source_free(cpuspeed_probe_sources[2]);
146
147   return err;
148 }
149
150 static void __exit cpuspeed_probe_exit(void)
151 {
152   int i;
153   printk(KERN_INFO "cpuspeed_probe cleanup function called !\n");
154   scheduler_probe_unregister(cpuspeed_probe);
155   scheduler_probe_free(cpuspeed_probe);
156
157   for(i = 0; cpuspeed_probe_sources[i] != NULL; i++)
158     scheduler_probe_source_free(cpuspeed_probe_sources[i]);
159 }
160
161 module_init(cpuspeed_probe_init);
162 module_exit(cpuspeed_probe_exit);
```

Code F.1 – Module de probe CPUSpeed

F.2 Probe Load

```c
/**
 * Kerrighed Local Unix active tasks probe module
 *
 * Copyright (c) 2009 - Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>
 *
 * Part of Mathieu Dabert's API (Copyright (c) 2008)
 **/

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kernel_stat.h>
#include <linux/threads.h>
#include <linux/cpumask.h>
#include <linux/list.h>
#include <linux/sched.h>
#include <linux/jiffies.h>

#include <kerrighed/sched.h>

#include <scheduler/core/probe.h>
#include <scheduler/core/krg_sched_info.h>
#include <scheduler/core/probe.h>

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>");
MODULE_DESCRIPTION("General load module probe.");

static struct scheduler_probe *load_probe;

struct load_probe_info {
  struct krg_sched_module_info module_info;
  unsigned long entryJiffies;
  unsigned long exitJiffies;
};

#define to_load_probe_info(sched_info) container_of((sched_info), struct
    load_probe_info, module_info)

static void load_probe_init_info(struct task_struct * task, struct load_probe_info *
    info)
{
  info->entryJiffies = 0;
  info->exitJiffies = 0;
}

static struct krg_sched_module_info * load_probe_info_copy(
  struct task_struct * task,
  struct krg_sched_module_info * info)
{
  struct load_probe_info * new_info;

  new_info = kmalloc(sizeof(struct load_probe_info), GFP_KERNEL);
  if (new_info) {
    load_probe_init_info(task, new_info);
    return &new_info->module_info;
  }

  return NULL;
}

static void load_probe_info_free(struct krg_sched_module_info * info)
{
```

```
61    kfree(to_load_probe_info(info));
62  }
63
64  static struct krg_sched_module_info * load_probe_info_import(
65    struct epm_action * action,
66    struct ghost * ghost,
67    struct task_struct * task)
68  {
69    return load_probe_info_copy(task, NULL);
70  }
71
72  static int load_probe_info_export(
73    struct epm_action * action,
74    struct ghost * ghost,
75    struct krg_sched_module_info * info)
76  {
77    return 0;
78  }
79
80  static struct krg_sched_module_info_type load_probe_module_info_type = {
81    .name = "load probe",
82    .owner  = THIS_MODULE,
83    .copy   = load_probe_info_copy, /* Called by framework on fork() */
84    .free = load_probe_info_free,
85    .import = load_probe_info_import,
86    .export = load_probe_info_export
87  };
88
89  static struct load_probe_info * get_load_probe_info(struct task_struct * task)
90  {
91    struct krg_sched_module_info * mod_info;
92
93    mod_info = krg_sched_module_info_get(task, &load_probe_module_info_type);
94    if (mod_info)
95      return to_load_probe_info(mod_info);
96    else  return NULL;
97  }
98
99  static void load_process_on(struct task_struct * task)
100 {
101   struct load_probe_info *p;
102
103   rcu_read_lock();
104   p = get_load_probe_info(task);
105   if (p)
106     p->exitJiffies = jiffies;
107   rcu_read_unlock();
108 }
109
110 static void load_process_off(struct task_struct * task)
111 {
112   struct load_probe_info *p;
113
114   rcu_read_lock();
115   p = get_load_probe_info(task);
116   if (p)
117     p->entryJiffies = jiffies;
118
119   rcu_read_unlock();
120 }
121
122 static void kmcb_process_on(unsigned long arg)
```

```
123 {
124   load_process_on((struct task_struct *)arg);
125 }
126 
127 static void kmcb_process_off(unsigned long arg)
128 {
129   load_process_off((struct task_struct *)arg);
130 }
131 
132 unsigned int count_process_jiffies(struct task_struct * task)
133 {
134   struct load_probe_info *p;
135   unsigned int retval;
136 
137   rcu_read_lock();
138   p = get_load_probe_info(task);
139   if (p) {
140     if (p->exitJiffies == 0) {
141       if (p->entryJiffies == 0)
142         retval = 0;
143       else if (jiffies < p->entryJiffies)
144         retval = 0;
145       else  retval = jiffies - p->entryJiffies;
146     } else { /* p->exitJiffies != 0 */
147       if (p->exitJiffies > p->entryJiffies)
148         retval = p->exitJiffies - p->entryJiffies;
149       else  retval = jiffies - p->entryJiffies; /* p->exitJiffies <= p->entryJiffies
              */
150     }
151   } else  retval = 0;
152 
153 /*
154   if (task && p)
155     printk(KERN_INFO "[%d] entry=%lu, exit=%lu, jiffies=%lu ==> %u\n", task->pid, p->
          entryJiffies, p->exitJiffies, jiffies, retval);*/
156 
157   rcu_read_unlock();
158 
159   return retval;
160 }
161 
162 unsigned int count_active_tasks_on_node(void)
163 {
164   unsigned int count;
165   struct task_struct * p;
166 
167   count = 0;
168 
169   rcu_read_lock();
170 
171   for_each_process(p) {
172     /* *
173      * From include/linux/sched.h :857
174      *
175      *  struct task_struct {
176      *  volatile long state;    // -1 unrunnable, 0 runnable, >0 stopped
177      * */
178     if (p->state == 0)
179       count++;
180   }
181 
182   rcu_read_unlock();
```

```
183
184   return count;
185 }
186
187 DEFINE_SCHEDULER_PROBE_SOURCE_GET(active_tasks, unsigned int, value_p, nr)
188 {
189   *value_p = count_active_tasks_on_node();
190   return 1;
191 }
192
193 DEFINE_SCHEDULER_PROBE_SOURCE_SHOW(active_tasks, page)
194 {
195   return sprintf(page, "%u\n", count_active_tasks_on_node());
196 }
197
198 static BEGIN_SCHEDULER_PROBE_SOURCE_TYPE(active_tasks),
199   .SCHEDULER_PROBE_SOURCE_GET(active_tasks),
200   .SCHEDULER_PROBE_SOURCE_SHOW(active_tasks),
201   .SCHEDULER_PROBE_SOURCE_VALUE_TYPE(active_tasks, unsigned int),
202 END_SCHEDULER_PROBE_SOURCE_TYPE(active_tasks);
203
204 static unsigned int get_process_jiffies_from_task(struct task_struct * task)
205 {
206   return count_process_jiffies(task);
207 }
208
209 static unsigned int get_process_jiffies_from_pid(pid_t process)
210 {
211   unsigned int retval = 0;
212   struct task_struct *task;
213
214   task = find_task_by_pid(process);
215   if (!task)
216     return retval;
217
218   return get_process_jiffies_from_task(task);
219 }
220
221 DEFINE_SCHEDULER_PROBE_SOURCE_SHOW(value_process_jiffies, page)
222 {
223   size_t size = SCHEDULER_PROBE_SOURCE_ATTR_SIZE;
224   ssize_t count = 0;
225   int tmp_count = 0;
226   struct task_struct * tsk;
227
228   printk(KERN_INFO "value_process_jiffies_show␣function␣called.\n");
229
230   rcu_read_lock();
231
232   for_each_process(tsk) {
233     tmp_count = snprintf(page + count, size - count, "Process␣%d␣:␣%u␣jiffies\n", tsk
          ->pid, get_process_jiffies_from_task(tsk));
234
235     if (tmp_count >= 0)
236       count += tmp_count;
237   }
238
239   rcu_read_unlock();
240
241   return (tmp_count < 0) ? tmp_count : min( (size_t) count + 1, size );
242 }
243
```

```
244 DEFINE_SCHEDULER_PROBE_SOURCE_GET_WITH_INPUT(value_process_jiffies, unsigned int,
        value_p, nr, pid_t, in_value_p, in_nr)
245 {
246   int i;
247
248   rcu_read_lock();
249
250   for (i = 0; i < in_nr && i < nr; i++)
251     value_p[i] = get_process_jiffies_from_pid(in_value_p[i]);
252
253   rcu_read_unlock();
254
255   return i;
256 }
257
258 static BEGIN_SCHEDULER_PROBE_SOURCE_TYPE(value_process_jiffies),
259   .SCHEDULER_PROBE_SOURCE_GET(value_process_jiffies),
260   .SCHEDULER_PROBE_SOURCE_SHOW(value_process_jiffies),
261   .SCHEDULER_PROBE_SOURCE_VALUE_TYPE(value_process_jiffies, unsigned int),
262   .SCHEDULER_PROBE_SOURCE_PARAM_TYPE(value_process_jiffies, pid_t),
263 END_SCHEDULER_PROBE_SOURCE_TYPE(value_process_load);
264
265 static struct scheduler_probe_source *load_probe_sources[3];
266
267 static SCHEDULER_PROBE_TYPE(load_probe_type, NULL, NULL);
268
269 static int __init load_probe_init(void)
270 {
271   int err = -ENOMEM;
272
273   load_probe_sources[0] = scheduler_probe_source_create(&active_tasks_type, "
        active_tasks");
274   load_probe_sources[1] = scheduler_probe_source_create(&value_process_jiffies_type,
        "process_jiffies");
275   load_probe_sources[2] = NULL;
276
277   load_probe = scheduler_probe_create(&load_probe_type, "load_probe",
        load_probe_sources, NULL);
278
279   if (load_probe == NULL) {
280     printk(KERN_DEBUG "error : load_probe creation failed !\n");
281     goto out_kmalloc;
282   }
283
284   err = module_hook_register(&kmh_process_on, kmcb_process_on, THIS_MODULE);
285   if (err)
286     goto err_hookon_reg;
287
288   err = module_hook_register(&kmh_process_off, kmcb_process_off, THIS_MODULE);
289   if (err)
290     goto err_hookoff_reg;
291
292   err = krg_sched_module_info_register(&load_probe_module_info_type);
293   if (err)
294     goto err_mod_info;
295
296   printk(KERN_INFO "load_probe : module_info registered.\n");
297
298   err = scheduler_probe_register(load_probe);
299   if (err)
300     goto err_register;
301
```

```
302   printk(KERN_INFO "load_probe␣loaded.\n");
303
304   return 0;
305
306 err_register:
307   printk(KERN_DEBUG "load_probe:␣error␣while␣registering␣probe.\n");
308   scheduler_probe_free(load_probe);
309 err_mod_info:
310   printk(KERN_DEBUG "load_probe:␣error␣while␣registering␣module␣info.\n");
311 err_hookoff_reg:
312 err_hookon_reg:
313 out_kmalloc:
314   if(load_probe_sources[0] != NULL)
315     scheduler_probe_source_free(load_probe_sources[0]);
316   if(load_probe_sources[1] != NULL)
317     scheduler_probe_source_free(load_probe_sources[1]);
318   if(load_probe_sources[2] != NULL)
319     scheduler_probe_source_free(load_probe_sources[2]);
320
321   return err;
322 }
323
324 static void __exit load_probe_exit(void)
325 {
326   int i;
327
328   printk(KERN_INFO "load_probe␣cleanup␣function␣called!\n");
329
330   scheduler_probe_unregister(load_probe);
331
332   krg_sched_module_info_unregister(&load_probe_module_info_type);
333   module_hook_unregister(&kmh_process_off, kmcb_process_off);
334   module_hook_unregister(&kmh_process_on, kmcb_process_on);
335
336   scheduler_probe_free(load_probe);
337
338   for(i = 0; load_probe_sources[i] != NULL; i++)
339     scheduler_probe_source_free(load_probe_sources[i]);
340 }
341
342 module_init(load_probe_init);
343 module_exit(load_probe_exit);
```

Code F.2 – Module de probe Load

F.3 Module Local User Presence

```
1  #include "local_user_presence.h"
2
3  static void local_user_presence_user_connection(void)
4  {
5    mutex_lock(&local_user_presence_lock);
6
7    local_user_presence_value++;
8
9    mutex_unlock(&local_user_presence_lock);
10 }
11
12 static void local_user_presence_user_disconnection(void)
```

```
13 {
14   mutex_lock(&local_user_presence_lock);
15
16   if(local_user_presence_value > 0)
17     local_user_presence_value--;
18
19   mutex_unlock(&local_user_presence_lock);
20 }
21
22 static unsigned int local_user_presence_user_connected(void)
23 {
24   unsigned int val;
25
26   mutex_lock(&local_user_presence_lock);
27
28   val = local_user_presence_value;
29
30   mutex_unlock(&local_user_presence_lock);
31
32   return val;
33 }
34
35 static unsigned int local_user_presence_node_free(void)
36 {
37   unsigned int val;
38
39   mutex_lock(&local_user_presence_lock);
40
41   val = local_user_presence_value;
42
43   mutex_unlock(&local_user_presence_lock);
44
45   return (val == 0);
46 }
47
48 static unsigned int local_user_presence_node_used(void)
49 {
50   unsigned int val;
51
52   mutex_lock(&local_user_presence_lock);
53
54   val = local_user_presence_value;
55
56   mutex_unlock(&local_user_presence_lock);
57
58   return (val > 0);
59 }
60
61 static int __init local_user_presence_init(void)
62 {
63   printk(KERN_INFO "Loading␣Local␣User␣Presence␣Notification␣module␣...\n");
64
65   mutex_lock(&local_user_presence_lock);
66
67   local_user_presence_value = 0;
68
69   mutex_unlock(&local_user_presence_lock);
70
71   printk(KERN_INFO "Local␣User␣Presence␣Notification␣module␣loaded.␣Current␣value␣is␣
       0.\n");
72
73   return 0;
```

```
74 }
75
76 static void __exit local_user_presence_exit(void)
77 {
78   printk(KERN_INFO "Unloading␣Local␣User␣Presence␣Notification␣module.\n");
79 }
```

Code F.3 – Module Local User Presence

```
1  #include <linux/kernel.h>
2  #include <linux/module.h>
3  #include <linux/proc_fs.h>
4  #include <linux/mutex.h>
5
6  MODULE_LICENSE("GPL␣v2");
7  MODULE_AUTHOR("Alexandre␣Lissy␣<alexandre.lissy@etu.univ-tours.fr>");
8  MODULE_DESCRIPTION("Local␣user␣presence␣notification");
9  MODULE_VERSION("1.0");
10
11 static DEFINE_MUTEX(local_user_presence_lock);
12
13 static unsigned int local_user_presence_value;
14
15 static void local_user_presence_user_connection(void);
16 static void local_user_presence_user_disconnection(void);
17 static unsigned int local_user_presence_user_connected(void);
18 static unsigned int local_user_presence_node_free(void);
19 static unsigned int local_user_presence_node_used(void);
20
21 EXPORT_SYMBOL(local_user_presence_user_connection);
22 EXPORT_SYMBOL(local_user_presence_user_disconnection);
23 EXPORT_SYMBOL(local_user_presence_user_connected);
24 EXPORT_SYMBOL(local_user_presence_node_free);
25 EXPORT_SYMBOL(local_user_presence_node_used);
26
27 static int __init local_user_presence_init(void);
28 static void __exit local_user_presence_exit(void);
29
30 module_init(local_user_presence_init);
31 module_exit(local_user_presence_exit);
```

Code F.4 – Module Local User Presence (entête)

F.4 Module Local User Notifier

```
1  #include "local_user_notifier.h"
2
3  static int local_user_notifier_user_connection(struct file * file, const char __user
       * buffer, unsigned long count, void * data)
4  {
5    local_user_presence_user_connection();
6    return count;
7  }
8
9  static int local_user_notifier_user_disconnection(struct file * file, const char
       __user * buffer, unsigned long count, void * data)
10 {
11   local_user_presence_user_disconnection();
12   return count;
```

```
13  }
14
15  static int local_user_notifier_user_connected(char * page, char ** start, off_t
        offset, int count, int * eof, void * data)
16  {
17    int len = 0;
18    len += sprintf(page + len, "%u\n", local_user_presence_user_connected());
19    return len;
20  }
21
22  static int local_user_notifier_node_free(char * page, char ** start, off_t offset,
        int count, int * eof, void * data)
23  {
24    int len = 0;
25    len += sprintf(page + len, "%u\n", local_user_presence_node_free());
26    return len;
27  }
28
29  static int local_user_notifier_node_used(char * page, char ** start, off_t offset,
        int count, int * eof, void * data)
30  {
31    int len = 0;
32    len += sprintf(page + len, "%u\n", local_user_presence_node_used());
33    return len;
34  }
35
36  static int __proc_init(void)
37  {
38    struct proc_dir_entry * p;
39
40    /* /proc/kerrighed/interactive_user  : dr-xr-xr-x */
41    root = create_proc_entry("kerrighed/interactive_user", S_IFDIR | S_IRUGO | S_IXUGO,
          NULL);
42    if(!root) {
43      printk(KERN_DEBUG "Cannot␣create␣proc␣entry␣kerrighed/interactive_user\n");
44      return -EAGAIN;
45    } else {
46      /* /proc/kerrighed/interactive_user/get  : -r--r--r-- */
47      p = create_proc_read_entry("get", S_IFREG | S_IRUGO, root,
            local_user_notifier_user_connected, NULL);
48      if(!p) {
49        printk(KERN_DEBUG "Cannot␣create␣proc␣entry␣kerrighed/interactive_user/get\n");
50        return -EAGAIN;
51      }
52
53      /* /proc/kerrighed/interactive_user/isfree  : -r--r--r-- */
54      p = create_proc_read_entry("isfree", S_IFREG | S_IRUGO, root,
            local_user_notifier_node_free, NULL);
55      if(!p) {
56        printk(KERN_DEBUG "Cannot␣create␣proc␣entry␣kerrighed/interactive_user/isfree\n
              ");
57        return -EAGAIN;
58      }
59
60      /* /proc/kerrighed/interactive_user/isused  : -r--r--r-- */
61      p = create_proc_read_entry("isused", S_IFREG | S_IRUGO, root,
            local_user_notifier_node_used, NULL);
62      if(!p) {
63        printk(KERN_DEBUG "Cannot␣create␣proc␣entry␣kerrighed/interactive_user/isused\n
              ");
64        return -EAGAIN;
65      }
```

```
66
67      /* /proc/kerrighed/interactive_user/connection  : --w--w--w- */
68      p = create_proc_entry("connection", S_IFREG | S_IWUGO, root);
69      if(!p) {
70        printk(KERN_DEBUG "Cannot create proc entry kerrighed/interactive_user/
              connection\n");
71        return -EAGAIN;
72      } else {
73        p->write_proc = local_user_notifier_user_connection;
74      }
75
76      /* /proc/kerrighed/interactive_user/disconnection  : --w--w--w- */
77      p = create_proc_entry("disconnection", S_IFREG | S_IWUGO, root);
78      if(!p) {
79        printk(KERN_DEBUG "Cannot create proc entry kerrighed/interactive_user/
              disconnection\n");
80        return -EAGAIN;
81      } else {
82        p->write_proc = local_user_notifier_user_disconnection;
83      }
84    }
85
86    return 0;
87  }
88
89  static void __proc_exit(void)
90  {
91    remove_proc_entry("get", root);
92    remove_proc_entry("isfree", root);
93    remove_proc_entry("isused", root);
94    remove_proc_entry("connection", root);
95    remove_proc_entry("disconnection", root);
96    remove_proc_entry("kerrighed/interactive_user", NULL);
97  }
98
99  static int __init local_user_notifier_init(void)
100 {
101   int retval;
102
103   printk(KERN_INFO "Loading Local User Presence Notification Interface module ...\n")
          ;
104
105   retval = __proc_init();
106   if(retval >= 0)
107     printk(KERN_INFO "Local User Presence Notification Interface module loaded.\n");
108   else
109     printk(KERN_DEBUG "Error while creating proc tree.\n");
110
111   return retval;
112 }
113
114 static void __exit local_user_notifier_exit(void)
115 {
116   printk(KERN_INFO "Unloading Local User Presence Notification Interface module.\n");
117
118   __proc_exit();
119
120   printk(KERN_INFO "Successfull unload of Local User Presence Notification Interface
          module.\n");
121 }
```

Code F.5 – Module Local User Notifier

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/mutex.h>

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>");
MODULE_DESCRIPTION("Local user presence notification interface for /proc");
MODULE_VERSION("1.0");

static struct proc_dir_entry * root;

extern void local_user_presence_user_connection(void);
extern void local_user_presence_user_disconnection(void);
extern unsigned int local_user_presence_user_connected(void);
extern unsigned int local_user_presence_node_free(void);
extern unsigned int local_user_presence_node_used(void);

static int local_user_notifier_user_connection(struct file * file, const char __user
    * buffer, unsigned long count, void * data);
static int local_user_notifier_user_disconnection(struct file * file, const char
    __user * buffer, unsigned long count, void * data);
static int local_user_notifier_user_connected(char * page, char ** start, off_t
    offset, int count, int * eof, void * data);
static int local_user_notifier_node_free(char * page, char ** start, off_t offset,
    int count, int * eof, void * data);
static int local_user_notifier_node_used(char * page, char ** start, off_t offset,
    int count, int * eof, void * data);

static int __proc_init(void);
static void __proc_exit(void);
static int __init local_user_notifier_init(void);
static void __exit local_user_notifier_exit(void);

module_init(local_user_notifier_init);
module_exit(local_user_notifier_exit);
```

Code F.6 – Module Local User Notifier (entête)

F.5 Probe Mattload

```
/**
 * Kerrighed Local Unix active tasks probe module
 *
 * Copyright (c) 2009 - Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>
 *
 * Matteo's Load Measure is Copyright (c) 2008 - Matthieu Perotin.
 * More informations available at :
 * http://portail.scd.univ-tours.fr/search*frf/X?PEROTIN,%20MATTHIEU&m=t&m=u
 **/

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kernel_stat.h>
#include <linux/threads.h>
#include <linux/cpumask.h>
#include <linux/list.h>
#include <linux/sched.h>

```

```
19 #include <scheduler/core/port.h>
20 #include <scheduler/core/probe.h>
21
22 static struct scheduler_probe *mattload_probe;
23 static struct scheduler_port port_active_tasks;
24 static struct scheduler_port port_cpu_speed;
25 static struct scheduler_port port_cpu_connected;
26 static struct scheduler_port port_user_connected;
27 static int mattload_param_k = 1;
28 static int mattload_multiply_factor = 1000000;
29
30 MODULE_LICENSE("GPL v2");
31 MODULE_AUTHOR("Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>");
32 MODULE_DESCRIPTION("Computing node load using Matteo's formula.");
33
34 static BEGIN_SCHEDULER_PORT_TYPE(port_active_tasks),
35   .SCHEDULER_PORT_VALUE_TYPE(port_active_tasks, unsigned int),
36 END_SCHEDULER_PORT_TYPE(port_active_tasks);
37
38 static BEGIN_SCHEDULER_PORT_TYPE(port_cpu_speed),
39   .SCHEDULER_PORT_VALUE_TYPE(port_cpu_speed, unsigned int),
40   .SCHEDULER_PORT_PARAM_TYPE(port_cpu_speed, unsigned int),
41 END_SCHEDULER_PORT_TYPE(port_cpu_speed);
42
43 static BEGIN_SCHEDULER_PORT_TYPE(port_cpu_connected),
44   .SCHEDULER_PORT_VALUE_TYPE(port_cpu_connected, int),
45 END_SCHEDULER_PORT_TYPE(port_cpu_connected);
46
47 static BEGIN_SCHEDULER_PORT_TYPE(port_user_connected),
48   .SCHEDULER_PORT_VALUE_TYPE(port_user_connected, unsigned int),
49 END_SCHEDULER_PORT_TYPE(port_user_connected);
50
51 static unsigned int matteo_load_calc(int calcLoadIncrement)
52 {
53   /* Matteo's way to compute the load. */
54   int err;
55   unsigned int load;
56   unsigned int state_mj;
57   unsigned int sumXij;
58   unsigned int cpuSpeed;
59   int cpuConnected;
60
61   /* (1 + K*(state(mj)))((sum Xij)/speed(mj)) */
62
63   if (calcLoadIncrement == 0) {
64     printk(KERN_DEBUG "Reading port active_tasks\n");
65     err = scheduler_port_get_value(&port_active_tasks, &sumXij, 1, NULL, 0);
66     if (err < 0)
67       goto err_active_tasks;
68   } else {
69     printk(KERN_DEBUG "calc_load_increment : Fake active_tasks. sumXij = 1\n");
70     sumXij = 1;
71   }
72
73   printk(KERN_DEBUG "Reading port cpu_speed\n");
74   err = scheduler_port_get_value(&port_cpu_speed, &cpuSpeed, 1, NULL, 0);
75   if (err < 0)
76     goto err_cpu_speed;
77
78   printk(KERN_DEBUG "Reading port cpu_connected\n");
79   err = scheduler_port_get_value(&port_cpu_connected, &cpuConnected, 1, NULL, 0);
80   if (err < 0)
```

```
81      goto err_cpu_connected;
82
83    printk(KERN_DEBUG "Reading port user_connected\n");
84    err = scheduler_port_get_value(&port_user_connected, &state_mj, 1, NULL, 0);
85    if (err < 0)
86      goto err_user_connected;
87
88    /* K : parameter
89     * state(mj) => user_probe
90     * sum Xij => load_probe
91     * speed(mj) => cpuspeed_probe
92     * */
93
94    /* Convert to MHz */
95    cpuSpeed /= 1000;
96
97    printk(KERN_DEBUG "active_tasks : %u\n", sumXij);
98    printk(KERN_DEBUG "cpu_speed : %u\n", cpuSpeed);
99    printk(KERN_DEBUG "cpu_connected : %u\n", cpuConnected);
100   printk(KERN_DEBUG "user_connected : %u\n", state_mj);
101   load = ((1 + mattload_param_k*(state_mj))) * ((mattload_multiply_factor*sumXij)/(
          cpuSpeed*cpuConnected));
102
103   return load;
104
105 err_active_tasks:
106   printk(KERN_DEBUG "mattload : error while reading port : active_tasks\n");
107
108 err_cpu_speed:
109   printk(KERN_DEBUG "mattload : error while reading port : cpu_speed\n");
110
111 err_cpu_connected:
112   printk(KERN_DEBUG "mattload : error while reading port : cpu_connected\n");
113
114 err_user_connected:
115   printk(KERN_DEBUG "mattload : error while reading port : user_connected\n");
116
117   return 0;
118 }
119
120 DEFINE_SCHEDULER_PROBE_SOURCE_GET(mattload, unsigned int, value_p, nr)
121 {
122   *value_p = matteo_load_calc(0);
123   return 1;
124 }
125
126 DEFINE_SCHEDULER_PROBE_SOURCE_SHOW(mattload, page)
127 {
128   unsigned int load;
129   load = matteo_load_calc(0);
130
131   return sprintf(page, "%u\n", load);
132 }
133
134 static BEGIN_SCHEDULER_PROBE_SOURCE_TYPE(mattload),
135   .SCHEDULER_PROBE_SOURCE_GET(mattload),
136   .SCHEDULER_PROBE_SOURCE_SHOW(mattload),
137   .SCHEDULER_PROBE_SOURCE_VALUE_TYPE(mattload, unsigned int),
138 END_SCHEDULER_PROBE_SOURCE_TYPE(mattload);
139
140 DEFINE_SCHEDULER_PROBE_SOURCE_GET(loadinc, unsigned int, value_p, nr)
141 {
```

```
142   *value_p = matteo_load_calc(1);
143   return 1;
144 }
145
146 DEFINE_SCHEDULER_PROBE_SOURCE_SHOW(loadinc, page)
147 {
148   unsigned int load;
149   load = matteo_load_calc(1);
150
151   return sprintf(page, "%u\n", load);
152 }
153
154 static BEGIN_SCHEDULER_PROBE_SOURCE_TYPE(loadinc),
155   .SCHEDULER_PROBE_SOURCE_GET(loadinc),
156   .SCHEDULER_PROBE_SOURCE_SHOW(loadinc),
157   .SCHEDULER_PROBE_SOURCE_VALUE_TYPE(loadinc, unsigned int),
158 END_SCHEDULER_PROBE_SOURCE_TYPE(loadinc);
159
160 static struct scheduler_probe_source *mattload_probe_sources[3];
161
162 static ssize_t param_k_attr_show (struct scheduler_probe * probe, char * page)
163 {
164   return sprintf(page, "%d\n", mattload_param_k);
165 }
166
167 static ssize_t param_k_attr_store (struct scheduler_probe * probe, const char * page,
      size_t count)
168 {
169   int new_paramk;
170   char * lastread;
171
172   new_paramk = simple_strtoul(page, &lastread, 0);
173   if (lastread - page + 1 < count
174     || (lastread[1] != '\0' && lastread[1] != '\n'))
175   {
176     return -EINVAL;
177   }
178
179   mattload_param_k = new_paramk;
180
181   return count;
182 }
183
184 static SCHEDULER_PROBE_ATTRIBUTE(
185     param_k_attr,
186     "param_k",
187     S_IRUGO | S_IWUSR,
188     param_k_attr_show,
189     param_k_attr_store
190   );
191
192 static ssize_t multiply_factor_attr_show (struct scheduler_probe * probe, char * page
     )
193 {
194   return sprintf(page, "%d\n", mattload_multiply_factor);
195 }
196
197 static ssize_t multiply_factor_attr_store (struct scheduler_probe * probe, const char
      * page, size_t count)
198 {
199   int new_mfact;
200   char * lastread;
```

```
201
202   new_mfact = simple_strtoul(page, &lastread, 0);
203   if (lastread - page + 1 < count
204     || (lastread[1] != '\0' && lastread[1] != '\n'))
205   {
206     return -EINVAL;
207   }
208
209   mattload_multiply_factor = new_mfact;
210
211   return count;
212 }
213
214 static SCHEDULER_PROBE_ATTRIBUTE(
215     multiply_factor_attr,
216     "multiply_factor",
217     S_IRUGO | S_IWUSR,
218     multiply_factor_attr_show,
219     multiply_factor_attr_store
220   );
221
222 static struct scheduler_probe_attribute * mattload_probe_attributes[] = {
223   &param_k_attr,
224   &multiply_factor_attr,
225   NULL
226 };
227
228 static SCHEDULER_PROBE_TYPE(mattload_probe_type, mattload_probe_attributes, NULL);
229
230 static int mattload_ports_init(void)
231 {
232   int err;
233
234   /* First, initialize ports type */
235   err = scheduler_port_type_init(&port_active_tasks_type, NULL);
236   if (err)
237     goto err_type_active_tasks;
238
239   err = scheduler_port_type_init(&port_cpu_speed_type, NULL);
240   if (err)
241     goto err_type_cpu_speed;
242
243   err = scheduler_port_type_init(&port_cpu_connected_type, NULL);
244   if (err)
245     goto err_type_cpu_connected;
246
247   err = scheduler_port_type_init(&port_user_connected_type, NULL);
248   if (err)
249     goto err_type_user_connected;
250
251   err = scheduler_port_init(&port_active_tasks, "active_tasks", &
       port_active_tasks_type, NULL, NULL);
252   if (err)
253     goto err_active_tasks;
254
255   err = scheduler_port_init(&port_cpu_speed, "cpu_speed", &port_cpu_speed_type, NULL,
        NULL);
256   if (err)
257     goto err_cpu_speed;
258
259   err = scheduler_port_init(&port_cpu_connected, "cpu_connected", &
       port_cpu_connected_type, NULL, NULL);
```

```
260   if (err)
261     goto err_cpu_connected;
262
263   err = scheduler_port_init(&port_user_connected, "user_connected", &
          port_user_connected_type, NULL, NULL);
264   if (err)
265     goto err_user_connected;
266
267   return 0;
268
269 err_user_connected:
270   scheduler_port_cleanup(&port_cpu_connected);
271 err_cpu_connected:
272   scheduler_port_cleanup(&port_cpu_speed);
273 err_cpu_speed:
274   scheduler_port_cleanup(&port_active_tasks);
275 err_active_tasks:
276   scheduler_port_type_cleanup(&port_user_connected_type);
277 err_type_user_connected:
278   scheduler_port_type_cleanup(&port_cpu_connected_type);
279 err_type_cpu_connected:
280   scheduler_port_type_cleanup(&port_cpu_speed_type);
281 err_type_cpu_speed:
282   scheduler_port_type_cleanup(&port_active_tasks_type);
283 err_type_active_tasks:
284   printk(KERN_DEBUG "matteoload_probe: Cannot init ports.\n");
285   return -1;
286 }
287
288 static int mattload_ports_exit(void)
289 {
290   scheduler_port_type_cleanup(&port_active_tasks_type);
291   scheduler_port_type_cleanup(&port_cpu_speed_type);
292   scheduler_port_type_cleanup(&port_cpu_connected_type);
293   scheduler_port_type_cleanup(&port_user_connected_type);
294
295   return 0;
296 }
297
298 static int __init mattload_probe_init(void)
299 {
300   int err = -ENOMEM;
301   struct config_group *def_groups[5];
302
303   err = mattload_ports_init();
304   if (err < 0) {
305     return err;
306   }
307
308         /* Initialize default emory groups */
309   def_groups[0] = scheduler_port_config_group(&port_active_tasks);
310   def_groups[1] = scheduler_port_config_group(&port_cpu_speed);
311   def_groups[2] = scheduler_port_config_group(&port_cpu_connected);
312   def_groups[3] = scheduler_port_config_group(&port_user_connected);
313   def_groups[4] = NULL;
314
315   mattload_probe_sources[0] = scheduler_probe_source_create(&mattload_type, "mattload
          ");
316   mattload_probe_sources[1] = scheduler_probe_source_create(&loadinc_type, "
          load_increment");
317   mattload_probe_sources[2] = NULL;
318
```

```
319   mattload_probe = scheduler_probe_create(&mattload_probe_type, "mattload_probe",
          mattload_probe_sources, def_groups);
320
321   if (mattload_probe == NULL) {
322     printk(KERN_DEBUG "error : mattload_probe creation failed !\n");
323     goto out_kmalloc;
324   }
325
326   err = scheduler_probe_register(mattload_probe);
327   if (err)
328     goto err_register;
329
330   printk(KERN_INFO "mattload_probe loaded.\n");
331
332   return 0;
333
334 err_register:
335   scheduler_probe_free(mattload_probe);
336 out_kmalloc:
337   if(mattload_probe_sources[0] != NULL)
338     scheduler_probe_source_free(mattload_probe_sources[0]);
339   if(mattload_probe_sources[1] != NULL)
340     scheduler_probe_source_free(mattload_probe_sources[1]);
341
342   return err;
343 }
344
345 static void __exit mattload_probe_exit(void)
346 {
347   int i;
348   printk(KERN_INFO "mattload_probe cleanup function called !\n");
349   mattload_ports_exit();
350   scheduler_probe_unregister(mattload_probe);
351   scheduler_probe_free(mattload_probe);
352
353   for(i = 0; mattload_probe_sources[i] != NULL; i++)
354     scheduler_probe_source_free(mattload_probe_sources[i]);
355 }
356
357 module_init(mattload_probe_init);
358 module_exit(mattload_probe_exit);
```

Code F.7 – Module de probe Mattload

F.6 Probe Processsize

```
1  /**
2   * Kerrighed Local CPU Informations Probe module
3   *
4   * Copyright (c) 2009 - Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>
5   *
6   * Part of Mathieu Dabert's API (Copyright (c) 2008)
7   **/
8
9  #include <linux/cpumask.h>
10 #include <linux/kernel.h>
11 #include <linux/kernel_stat.h>
12 #include <linux/list.h>
13 #include <linux/module.h>
```

```
14 #include <linux/sched.h>
15 #include <linux/threads.h>
16
17 #include <kerrighed/pid.h>
18 #include <scheduler/core/probe.h>
19
20 static struct scheduler_probe *processsize_probe;
21
22 MODULE_LICENSE("GPL v2");
23 MODULE_AUTHOR("Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>");
24 MODULE_DESCRIPTION("Process Size module probe");
25
26 DEFINE_SCHEDULER_PROBE_SOURCE_GET_WITH_INPUT(process_total_size, pid_t, value_p, nr,
       unsigned long, idx_p, in_nr)
27 {
28   int i = 0;
29   struct task_struct * ts;
30
31   if (in_nr) { /* Only returns the process queried*/
32     for (i = 0; i < in_nr && i < nr; i++) {
33       ts = find_task_by_pid(idx_p[i]);
34       /* tsk->mm can be NULL
35        * cf. http://lkml.indiana.edu/hypermail/linux/kernel/0111.3/1188.html
36        *
37        * > Hey,
38        * >
39        * > I found in some code checks for task_struct.mm being NULL.
40        * > When can task_struct.mm of a process be NULL except right before the
41        * > process-kill?
42        *
43        * For kernel threads that run in lazy-mm mode. It allows a much cheaper
              context
44        * switch into kernel threads.
45        * */
46       if (ts != NULL && ts->mm != NULL)
47         *value_p++ = ts->mm->total_vm;
48     }
49   }
50
51   return i;
52 }
53
54 DEFINE_SCHEDULER_PROBE_SOURCE_SHOW(process_total_size, page)
55 {
56   struct task_struct * tsk;
57
58   size_t size = SCHEDULER_PROBE_SOURCE_ATTR_SIZE;
59   ssize_t count = 0;
60   int tmp_count = 0;
61
62   printk(KERN_INFO "process_total_size_show function called.\n");
63
64   rcu_read_lock();
65
66   for_each_process(tsk) {
67     /* tsk->mm can be NULL
68      * cf. http://lkml.indiana.edu/hypermail/linux/kernel/0111.3/1188.html
69      *
70      * > Hey,
71      * >
72      * > I found in some code checks for task_struct.mm being NULL.
73      * > When can task_struct.mm of a process be NULL except right before the
```

```
74        * > process-kill?
75        *
76        * For kernel threads that run in lazy-mm mode. It allows a much cheaper context
77        * switch into kernel threads.
78        * */
79       if (tsk != NULL && tsk->mm != NULL) {
80         tmp_count = snprintf(page + count, size - count, "Process␣%d␣:␣%lu␣pages\n", tsk
              ->pid, tsk->mm->total_vm);
81
82         if (tmp_count >= 0)
83           count += tmp_count;
84       }
85     }
86
87     rcu_read_unlock();
88
89     return (tmp_count < 0) ? tmp_count : min( (size_t) count + 1, size );
90   }
91
92   static BEGIN_SCHEDULER_PROBE_SOURCE_TYPE(process_total_size),
93     .SCHEDULER_PROBE_SOURCE_GET(process_total_size),
94     .SCHEDULER_PROBE_SOURCE_SHOW(process_total_size),
95     .SCHEDULER_PROBE_SOURCE_VALUE_TYPE(process_total_size, unsigned long),
96     .SCHEDULER_PROBE_SOURCE_PARAM_TYPE(process_total_size, pid_t),
97   END_SCHEDULER_PROBE_SOURCE_TYPE(process_total_size);
98
99   static struct scheduler_probe_source *processsize_probe_sources[2];
100
101  static SCHEDULER_PROBE_TYPE(processsize_probe_type, NULL, NULL);
102
103  static int __init processsize_probe_init(void)
104  {
105    int err = -ENOMEM;
106
107    processsize_probe_sources[0] = scheduler_probe_source_create(&
         process_total_size_type, "total_vm");
108    processsize_probe_sources[1] = NULL;
109
110    processsize_probe = scheduler_probe_create(&processsize_probe_type, "
         processsize_probe", processsize_probe_sources, NULL);
111
112    if (processsize_probe == NULL) {
113      printk(KERN_ERR "error :␣processsize_probe␣creation␣failed!\n");
114      goto out_kmalloc;
115    }
116
117    err = scheduler_probe_register(processsize_probe);
118    if (err)
119      goto err_register;
120
121    printk(KERN_INFO "processsize_probe␣loaded.\n");
122
123    return 0;
124
125  err_register:
126    scheduler_probe_free(processsize_probe);
127  out_kmalloc:
128    if(processsize_probe_sources[0] != NULL)
129      scheduler_probe_source_free(processsize_probe_sources[0]);
130    if(processsize_probe_sources[1] != NULL)
131      scheduler_probe_source_free(processsize_probe_sources[1]);
132
```

```
133   return err;
134 }
135
136 static void __exit processsize_probe_exit(void)
137 {
138   int i;
139   printk(KERN_INFO "processsize_probe cleanup function called !\n");
140   scheduler_probe_unregister(processsize_probe);
141   scheduler_probe_free(processsize_probe);
142
143   for(i = 0; processsize_probe_sources[i] != NULL; i++)
144     scheduler_probe_source_free(processsize_probe_sources[i]);
145 }
146
147 module_init(processsize_probe_init);
148 module_exit(processsize_probe_exit);
```

Code F.8 – Module de probe Processsize

F.7 Probe User

```
1  /**
2   * Kerrighed User Probe module
3   *
4   * Copyright (c) 2009 - Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>
5   *
6   * Part of Mathieu Dabert's API (Copyright (c) 2008)
7   **/
8
9  #include <linux/kernel.h>
10 #include <linux/module.h>
11 #include <linux/kernel_stat.h>
12 #include <linux/threads.h>
13 #include <linux/cpumask.h>
14 #include <linux/list.h>
15
16 #include <scheduler/core/probe.h>
17 /* #include <local_user_presence/local_user_presence.h> */
18
19 static struct scheduler_probe *user_probe;
20
21 MODULE_LICENSE("GPL v2");
22 MODULE_AUTHOR("Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>");
23 MODULE_DESCRIPTION("Local user presence probe");
24
25 DEFINE_SCHEDULER_PROBE_SOURCE_GET(user_connected_probe, unsigned int, value_p, nr)
26 {
27   *value_p = local_user_presence_user_connected();
28   return 1;
29 }
30
31 DEFINE_SCHEDULER_PROBE_SOURCE_SHOW(user_connected_probe, page)
32 {
33   return sprintf(page, "%u\n", local_user_presence_user_connected());
34 }
35
36 static BEGIN_SCHEDULER_PROBE_SOURCE_TYPE(user_connected_probe),
37   .SCHEDULER_PROBE_SOURCE_GET(user_connected_probe),
38   .SCHEDULER_PROBE_SOURCE_SHOW(user_connected_probe),
```

```
39    .SCHEDULER_PROBE_SOURCE_VALUE_TYPE(user_connected_probe, unsigned int),
40  END_SCHEDULER_PROBE_SOURCE_TYPE(use_connected_probe);
41
42  static struct scheduler_probe_source *user_probe_sources[2];
43
44  static SCHEDULER_PROBE_TYPE(user_probe_type, NULL, NULL);
45
46  static int __init user_probe_init(void)
47  {
48    int err = -ENOMEM;
49
50    user_probe_sources[0] =  scheduler_probe_source_create(&user_connected_probe_type,
        "connected");
51    user_probe_sources[1] = NULL;
52
53    user_probe = scheduler_probe_create(&user_probe_type, "user_probe",
        user_probe_sources, NULL);
54
55    if (user_probe == NULL) {
56      printk(KERN_ERR "error : user_probe creation failed !\n");
57      goto out_kmalloc;
58    }
59
60    err = scheduler_probe_register(user_probe);
61    if (err)
62      goto err_register;
63
64    printk(KERN_INFO "user_probe loaded.\n");
65
66    return 0;
67
68  err_register:
69    scheduler_probe_free(user_probe);
70  out_kmalloc:
71    if(user_probe_sources[0] != NULL)
72      scheduler_probe_source_free(user_probe_sources[0]);
73    if(user_probe_sources[1] != NULL)
74      scheduler_probe_source_free(user_probe_sources[1]);
75
76    return err;
77  }
78
79  static void __exit user_probe_exit(void)
80  {
81    int i;
82    printk(KERN_INFO "user_probe cleanup function called !\n");
83    scheduler_probe_unregister(user_probe);
84    scheduler_probe_free(user_probe);
85
86    for(i = 0; user_probe_sources[i] != NULL; i++)
87      scheduler_probe_source_free(user_probe_sources[i]);
88  }
89
90  module_init(user_probe_init);
91  module_exit(user_probe_exit);
```

Code F.9 – Module de probe User

F.8 Support des Ports depuis les Probes

```
[PATCH] Extend scheduler_probe_create() to allow use of ports from probes.

A probe might need to use port. This patch adds a way to specify ports
to a probe, allowing connection of other probe sources.

A NULL-valued parameter doesn't change the behavior from previous one.

Signed-off-by : Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>
Index : core/policy.c
===================================================================
--- core/policy.c (revision 5256)
+++ core/policy.c (working copy)
@@ -115,15 +115,6 @@
  .store_attribute = scheduler_policy_attribute_store,
 };

-static int nr_def_groups(struct config_group *def_groups[])
-{
- int n = 0;
- if (def_groups)
-  while (def_groups[n])
-   n++;
- return n;
-}
-
 /**
  * This function initializes a new scheduling policy.
  * @author Marko Novak, Louis Rilling
Index : core/probe.c
===================================================================
--- core/probe.c  (revision 5256)
+++ core/probe.c  (working copy)
@@ -531,10 +531,12 @@
 struct scheduler_probe *
 scheduler_probe_create(struct scheduler_probe_type *type,
                        const char *name,
-                       struct scheduler_probe_source **sources)
+                       struct scheduler_probe_source **sources,
+                       struct config_group *def_groups[])
 {
  int num_sources;
  int nr_attrs;
+ int nr_groups;
  int i;
  struct config_group **tmp_def = NULL;
  struct configfs_attribute **tmp_attrs = NULL;
@@ -542,12 +544,13 @@

  num_sources = probe_source_array_length(sources);
  nr_attrs = probe_attribute_array_length(type->attrs);
+ nr_groups = nr_def_groups(def_groups);
  tmp_probe = kmalloc(sizeof(*tmp_probe), GFP_KERNEL);
  /* allocate 2 more elements in array of pointers : one for
   * probe_period attribute and one for NULL element which marks
   * the end of array. */
  tmp_attrs = kmalloc(sizeof(*tmp_attrs) * (nr_attrs + 2), GFP_KERNEL);
- tmp_def = kcalloc(num_sources+1, sizeof(*tmp_def), GFP_KERNEL);
+ tmp_def = kcalloc(num_sources + nr_groups + 1, sizeof(*tmp_def), GFP_KERNEL);

  if (!tmp_probe || !tmp_attrs || !tmp_def)
   goto out_kmalloc;
```

```
@@ -565,8 +568,14 @@
    /* set current probe as parent of scheduler_probe_source. */
    sources[i]->parent = tmp_probe;
  }
- tmp_def[num_sources] = NULL;
+
+ /* append ports to default groups */
+ for (i=0; i<nr_groups; i++) {
+    tmp_def[num_sources + i] = def_groups[i];
+ }

+ tmp_def[num_sources + nr_groups] = NULL;
+
  /* initialize probe type. */
  type->item_type.ct_item_ops = &scheduler_probe_item_ops;
  type->item_type.ct_attrs = tmp_attrs;
Index: core/probe.h
===================================================================
--- core/probe.h    (revision 5256)
+++ core/probe.h    (working copy)
@@ -417,14 +417,18 @@
  *        probe and must match the module file name.
  * @param sources   NULL-terminated array of probe's sources created with
  *      scheduler_probe_source_create().
+ * @param def_groups NULL-terminated array of subdirs of the probe
+ *      directory , or NULL
  *
- * @return              pointer to newly created probe or NULL if probe
+ * @return   pointer to newly created probe or NULL if probe
  *      creation failed.
  */
 struct scheduler_probe *
 scheduler_probe_create(struct scheduler_probe_type *type,
           const char *name,
-          struct scheduler_probe_source **sources);
+          struct scheduler_probe_source **sources,
+          struct config_group *def_groups[]
+          );
 /**
  * This function frees all the memory taken by a probe.
  * @author Marko Novak, Louis Rilling
Index: core/internal.h
===================================================================
--- core/internal.h (revision 5256)
+++ core/internal.h (working copy)
@@ -20,6 +20,15 @@
 struct scheduler_policy * scheduler_policy_new(const char *name);
 void scheduler_policy_drop(struct scheduler_policy *policy);

+static inline int nr_def_groups(struct config_group *def_groups[])
+{
+ int n = 0;
+ if (def_groups)
+    while (def_groups[n])
+       n++;
+ return n;
+}
+
 /* Subsystems initializers / cleaners */

 int krg_sched_info_start(void);
Index: probes/cpu_probe.c
```

```
124 ===================================================================
125 --- probes/cpu_probe.c  (revision 5256)
126 +++ probes/cpu_probe.c  (working copy)
127 @@ -229,7 +229,7 @@
128     cpu_total_prev[i] = 0;
129   }
130   cpu_probe = scheduler_probe_create(&cpu_probe_type, CPU_PROBE_NAME,
131 -             cpu_probe_sources);
132 +             cpu_probe_sources, NULL);
133   if (cpu_probe == NULL){
134     printk(KERN_ERR "error : cpu_probe creation failed !\n");
135     goto out_kmalloc;
136 Index : probes/migration_probe.c
137 ===================================================================
138 --- probes/migration_probe.c  (revision 5256)
139 +++ probes/migration_probe.c  (working copy)
140 @@ -132,7 +132,8 @@
141
142   migration_probe = scheduler_probe_create(&migration_probe_type,
143             "migration_probe",
144 -             migration_probe_sources);
145 +             migration_probe_sources,
146 +             NULL);
147   if (!migration_probe)
148     goto err_probe;
149
150 Index : probes/mem_probe.c
151 ===================================================================
152 --- probes/mem_probe.c  (revision 5256)
153 +++ probes/mem_probe.c  (working copy)
154 @@ -209,7 +209,7 @@
155   }
156
157   mem_probe = scheduler_probe_create(&mem_probe_type, MEM_PROBE_NAME,
158 -             mem_probe_sources);
159 +             mem_probe_sources, NULL);
160   if (mem_probe == NULL) {
161     printk(KERN_ERR "error : mem_probe creation failed !\n");
162     goto out_kmalloc;
163 Index : probes/mosix_probe.c
164 ===================================================================
165 --- probes/mosix_probe.c  (revision 5256)
166 +++ probes/mosix_probe.c  (working copy)
167 @@ -662,7 +735,8 @@
168
169   mosix_probe = scheduler_probe_create(&mosix_probe_type,
170               MOSIX_PROBE_NAME,
171 -             mosix_probe_sources);
172 +             mosix_probe_sources,
173 +             NULL);
174   if (mosix_probe == NULL) {
175     printk(KERN_ERR "error : mosix_probe creation failed\n");
176     goto out_kmalloc;
```

Code F.10 – Support des Ports depuis les Probes

G. Implémentation de RBT

G.1 Filtre RBT Cache

```
/*
 * Mainly inspired from :
 *  Kerrighed/modules/scheduler/filters/rbt_cache_filter.c
 *  Copyright (C) 2007-2008 Louis Rilling - Kerlabs
 *
 * Copyright (c) 2009 - Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>
 *
 * RBT is Copyright (c) 2008 - Matthieu Perotin.
 * More informations available at :
 * http://portail.scd.univ-tours.fr/search*frf/X?PEROTIN,%20MATTHIEU&m=t&m=u
 */

#include <linux/module.h>
#include <linux/workqueue.h>
#include <linux/jiffies.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <kerrighed/sys/types.h>
#include <kerrighed/krgnodemask.h>

#include <scheduler/core/filter.h>

MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>");
MODULE_DESCRIPTION("Filter to proactively cache remote values for RBT policy");

struct rbt_cache_filter {
  struct scheduler_filter filter;
  unsigned int remote_values[KERRIGHED_MAX_NODES];
  krgnodemask_t available_values;
  unsigned int polling_period; /* in jiffies */
  kerrighed_node_t current_node;
  struct delayed_work polling_work;
  int active; /* Is it able to collect values? */
};

static inline
struct rbt_cache_filter *
to_rbt_cache_filter(struct scheduler_filter *filter)
{
  return container_of(filter, struct rbt_cache_filter, filter);
}

static inline void rc_lock(struct rbt_cache_filter *rc_filter)
{
  scheduler_filter_lock(&rc_filter->filter);
}

static inline void rc_unlock(struct rbt_cache_filter *rc_filter)
```

```
51  {
52    scheduler_filter_unlock(&rc_filter->filter);
53  }
54
55  static void schedule_next_poll(struct rbt_cache_filter *rc_filter)
56  {
57    unsigned long delay;
58
59    delay = rc_filter->polling_period;
60    if (rc_filter->current_node != KERRIGHED_NODE_ID_NONE)
61      /* Last polling phase could not finish within period. Schedule
62       * next phase ASAP */
63      delay = 1;
64    else if (!delay)
65      /* We are forced to schedule a poll in order to make
66       * cancel_rearming_delayed_work() do its job.
67       *
68       * Schedule it every hour
69       */
70      delay = msecs_to_jiffies(3600000);
71    schedule_delayed_work(&rc_filter->polling_work, delay);
72  }
73
74  static void reschedule_next_poll(struct rbt_cache_filter *rc_filter)
75  {
76    cancel_delayed_work(&rc_filter->polling_work);
77    schedule_next_poll(rc_filter);
78  }
79
80  DEFINE_SCHEDULER_FILTER_ATTRIBUTE_SHOW(polling_period, filter, attr, page)
81  {
82    struct rbt_cache_filter *f = to_rbt_cache_filter(filter);
83    unsigned long period;
84
85    rc_lock(f);
86    period = f->polling_period;
87    rc_unlock(f);
88    return sprintf(page, "%u", jiffies_to_msecs(period));
89  }
90
91  DEFINE_SCHEDULER_FILTER_ATTRIBUTE_STORE(polling_period, filter, attr, page, count)
92  {
93    struct rbt_cache_filter *f = to_rbt_cache_filter(filter);
94    unsigned long new_period;
95    char *last_read;
96
97    new_period = simple_strtoul(page, &last_read, 0);
98    if (last_read - page + 1 < count
99        || (last_read[1] != '\0' && last_read[1] != '\n'))
100     return -EINVAL;
101   new_period = msecs_to_jiffies(new_period);
102   rc_lock(f);
103   f->polling_period = new_period;
104   reschedule_next_poll(f);
105   rc_unlock(f);
106
107   return count;
108 }
109
110 static BEGIN_SCHEDULER_FILTER_ATTRIBUTE(polling_period_attr, polling_period, 0666),
111   .SCHEDULER_FILTER_ATTRIBUTE_SHOW(polling_period),
112   .SCHEDULER_FILTER_ATTRIBUTE_STORE(polling_period),
```

```
113 END_SCHEDULER_FILTER_ATTRIBUTE(polling_period);
114
115 static struct scheduler_filter_attribute *remote_cache_attrs[] = {
116   &polling_period_attr,
117   NULL
118 };
119
120 /* Gets called:
121  * - from the scheduler framework, but only if it holds a reference,
122  * - or from the polling worker.
123  * So, when the destroy method is called, can only be called from the polling
124  * worker.
125  */
126 static int try_get_remote_values(struct rbt_cache_filter *f)
127 {
128   kerrighed_node_t current_node = f->current_node;
129   int nr = 0;
130   int ret = 0;
131
132   while (current_node != KERRIGHED_NODE_ID_NONE) {
133     ret = scheduler_filter_simple_get_remote_value(
134       &f->filter,
135       current_node,
136       &f->remote_values[current_node], 1,
137       NULL, 0);
138     if (ret == -EAGAIN)
139       break;
140     nr++;
141     if (ret > 0)
142       krgnode_set(current_node, f->available_values);
143     else
144       krgnode_clear(current_node, f->available_values);
145     current_node = krgnode_next_possible(current_node);
146     if (current_node == KERRIGHED_MAX_NODES)
147       current_node = KERRIGHED_NODE_ID_NONE;
148   }
149   f->current_node = current_node;
150
151   if (ret == -EACCES)
152     f->active = 0;
153
154   return nr;
155 }
156
157 static void get_remote_values(struct rbt_cache_filter *rc_filter)
158 {
159   kerrighed_node_t first_node;
160
161   if (rc_filter->current_node == KERRIGHED_NODE_ID_NONE) {
162     first_node = nth_possible_krgnode(0);
163     if (first_node != KERRIGHED_MAX_NODES) {
164       rc_filter->current_node = first_node;
165       try_get_remote_values(rc_filter);
166     }
167   }
168 }
169
170 static void polling_worker(struct work_struct *work)
171 {
172   struct rbt_cache_filter *f =
173     container_of(work,
174           struct rbt_cache_filter, polling_work.work);
```

```
175
176   rc_lock(f);
177   schedule_next_poll(f);
178   get_remote_values(f);
179   rc_unlock(f);
180 }
181
182 DEFINE_SCHEDULER_FILTER_UPDATE_VALUE(rbt_cache_filter, filter)
183 {
184   struct rbt_cache_filter *f = to_rbt_cache_filter(filter);
185   int nr;
186
187   rc_lock(f);
188   nr = try_get_remote_values(f);
189   rc_unlock(f);
190
191   /* Propagate updates from the connected local source.
192    *
193    * We may miss some if incidentally a remote value becomes available at
194    * the same time. Let's hope this is not to bad...
195    */
196   if (!nr)
197     /* Update comes from the connected local source */
198     scheduler_filter_simple_update_value(filter);
199 }
200
201 DEFINE_SCHEDULER_FILTER_GET_REMOTE_VALUE(rbt_cache_filter, filter,
202             node,
203             unsigned int, value_p, nr,
204             unsigned int, param_p, nr_param)
205 {
206   struct rbt_cache_filter *f = to_rbt_cache_filter(filter);
207   int ret = 0;
208
209   rc_lock(f);
210   if (!f->active) {
211     /* Do not wait for the next worker activation to begin reading
212      * remote values */
213     f->active = 1;
214     reschedule_next_poll(f);
215     get_remote_values(f);
216   }
217   if (krgnode_isset(node, f->available_values)) {
218     value_p[0] = f->remote_values[node];
219     ret = 1;
220   }
221   rc_unlock(f);
222
223   return ret;
224 }
225
226 /* Forward declaration */
227 static struct scheduler_filter_type rbt_cache_filter_type;
228
229 DEFINE_SCHEDULER_FILTER_NEW(rbt_cache_filter, name)
230 {
231   struct rbt_cache_filter *f = kmalloc(sizeof(*f), GFP_KERNEL);
232   int err;
233
234   if (!f)
235     goto err_f;
236   err = scheduler_filter_init(&f->filter, name, &rbt_cache_filter_type,
```

```
237             NULL);
238   if (err)
239     goto err_filter;
240   krgnodes_clear(f->available_values);
241   f->polling_period = 0;
242   f->current_node = KERRIGHED_NODE_ID_NONE;
243   INIT_DELAYED_WORK(&f->polling_work, polling_worker);
244   f->active = 0;
245   schedule_next_poll(f);
246
247   return &f->filter;
248
249  err_filter:
250   kfree(f);
251  err_f:
252   return NULL;
253 }
254
255 DEFINE_SCHEDULER_FILTER_DESTROY(rbt_cache_filter, filter)
256 {
257   struct rbt_cache_filter *f = to_rbt_cache_filter(filter);
258   cancel_rearming_delayed_work(&f->polling_work);
259   scheduler_filter_cleanup(filter);
260   kfree(f);
261 }
262
263 static BEGIN_SCHEDULER_FILTER_TYPE(rbt_cache_filter),
264   .SCHEDULER_FILTER_UPDATE_VALUE(rbt_cache_filter),
265   .SCHEDULER_FILTER_GET_REMOTE_VALUE(rbt_cache_filter),
266   .SCHEDULER_FILTER_SOURCE_VALUE_TYPE(rbt_cache_filter, unsigned int),
267   .SCHEDULER_FILTER_PORT_VALUE_TYPE(rbt_cache_filter, unsigned int),
268   .SCHEDULER_FILTER_ATTRIBUTES(rbt_cache_filter, remote_cache_attrs),
269 END_SCHEDULER_FILTER_TYPE(rbt_cache_filter);
270
271 static int remote_cache_start(void)
272 {
273   return scheduler_filter_type_register(&rbt_cache_filter_type);
274 }
275
276 static void remote_cache_exit(void)
277 {
278   scheduler_filter_type_unregister(&rbt_cache_filter_type);
279 }
280
281 module_init(remote_cache_start);
282 module_exit(remote_cache_exit);
```

Code G.1 – Module de filtre RBT Cache

G.2 Politique RBT

```
1 /*
2  *  Copyright (C) 2009 Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>
3  *
4  * RBT is Copyright (c) 2008 - Matthieu Perotin.
5  * More informations available at :
6  * http://portail.scd.univ-tours.fr/search*frf/X?PEROTIN,%20MATTHIEU&m=t&m=u
7  */
8
```

```
9  #include "rbt_policy.h"
10
11 static BEGIN_SCHEDULER_PORT_TYPE(port_mattload),
12   .SCHEDULER_PORT_VALUE_TYPE(port_mattload, unsigned int),
13 END_SCHEDULER_PORT_TYPE(port_mattload);
14
15 static BEGIN_SCHEDULER_PORT_TYPE(port_active_tasks),
16   .SCHEDULER_PORT_VALUE_TYPE(port_active_tasks, unsigned int),
17 END_SCHEDULER_PORT_TYPE(port_active_tasks);
18
19 static BEGIN_SCHEDULER_PORT_TYPE(port_loadinc),
20   .SCHEDULER_PORT_VALUE_TYPE(port_loadinc, unsigned int),
21 END_SCHEDULER_PORT_TYPE(port_loadinc);
22
23 static BEGIN_SCHEDULER_PORT_TYPE(port_process_jiffies),
24   .SCHEDULER_PORT_VALUE_TYPE(port_process_jiffies, unsigned int),
25   .SCHEDULER_PORT_PARAM_TYPE(port_process_jiffies, pid_t),
26 END_SCHEDULER_PORT_TYPE(port_process_jiffies);
27
28 static BEGIN_SCHEDULER_PORT_TYPE(port_process_size),
29   .SCHEDULER_PORT_VALUE_TYPE(port_process_size, unsigned long),
30   .SCHEDULER_PORT_PARAM_TYPE(port_process_size, pid_t),
31 END_SCHEDULER_PORT_TYPE(port_process_size);
32
33 void rbt_policy_exec(struct work_struct * work)
34 {
35   unsigned int loadIncrements[KERRIGHED_MAX_NODES];
36   unsigned int idealLoads[KERRIGHED_MAX_NODES];
37   unsigned long totalActiveTasks = 0;
38   unsigned int curActiveTasks = 0;
39   int totNodes = 0;
40   int err;
41
42   kerrighed_node_t curNode = kerrighed_node_id;
43   kerrighed_node_t migNode;
44   kerrighed_node_t node;
45   pid_t processus;
46
47   unsigned int j, k, idealLoad, bestLoad, iBestLoad = 0;
48   int howMany, upperBound;
49   struct rbt_policy *p = container_of(work, struct rbt_policy, rbt_work.work);
50   struct caller_creds creds;
51   creds.caller_uid = 0;
52   creds.caller_euid = 0;
53
54   for_each_possible_krgnode(node) {
55     /* Compute total load */
56     err = scheduler_port_get_remote_value(
57       &p->port_active_tasks,
58       node,
59       &curActiveTasks,
60       1, NULL, 0);
61     totalActiveTasks += curActiveTasks;
62
63     /* Prepare load increments */
64     err = scheduler_port_get_remote_value(
65       &p->port_loadinc,
66       node,
67       &loadIncrements[node],
68       1, NULL, 0);
69     totNodes++;
70
```

```
71      /* Initialize ideals loads */
72      idealLoads[node] = 0;
73    }
74
75    /* printk(KERN_INFO "rbt_policy[%d] : totalActiveTasks=%lu\n", curNode,
          totalActiveTasks); */
76
77    for (k = 0; k < totalActiveTasks; k++) {
78      /* /include/linux/kernel.h#L25 */
79      bestLoad = UINT_MAX;
80
81      for_each_possible_krgnode(node) {
82        if (idealLoads[node] + loadIncrements[node] < bestLoad) {
83          bestLoad = idealLoads[node] + loadIncrements[node];
84          iBestLoad = node;
85        }
86      }
87
88            idealLoads[iBestLoad] = bestLoad;
89    }
90
91    err = scheduler_port_get_value(
92      &p->port_active_tasks,
93      &curActiveTasks,
94      1, NULL, 0);
95    if (err < 0) {
96      printk(KERN_ERR "rbt_policy : Error getting value (active_tasks) : %d\n", err);
97      return;
98    }
99
100   idealLoad = idealLoads[curNode];
101   howMany   = (curActiveTasks < idealLoad) ? 0  : curActiveTasks - idealLoad;
102   upperBound  = (loadIncrements[curNode] == 0) ? 0  : howMany/loadIncrements[curNode];
103   printk(KERN_INFO "rbt_policy[%d] : looping from 0 to (howMany=%d = (curActiveTasks=%
          u - idealLoad=%u)) / loadIncrements[curNode]=%u = %d\n", curNode, howMany,
          curActiveTasks, idealLoad, loadIncrements[curNode], upperBound);
104
105   for (j = 0; j < upperBound; j++) {
106     processus = rbt_find_a_task(&p->policy);
107     if (processus != 0) {
108       migNode = rbt_find_a_node(&curNode, idealLoads, &p->policy);
109       if (migNode != KERRIGHED_NODE_ID_NONE) {
110         printk(KERN_INFO "rbt_policy : start_migrate :{processus=%d, migNode=%d}\n",
              processus, migNode);
111         migrate_linux_threads(
112           processus,
113           MIGR_GLOBAL_PROCESS,
114           migNode,
115           &creds
116         );
117         printk(KERN_INFO "rbt_policy : end_migrate :{processus=%d, migNode=%d}\n",
              processus, migNode);
118       }
119     }
120   }
121
122   /* Rescheduling the scheduler ;) */
123   schedule_delayed_work(&p->rbt_work, RBT_RATE*HZ);
124 }
125
126 pid_t rbt_find_a_task(struct scheduler_policy * policyPtr) {
127   struct task_struct * p, * max_p;
```

```
128   struct krg_sched_info * info;
129   pid_t task;
130
131   unsigned long total_vm, lowest_vm;
132   unsigned int jiffies;
133   struct rbt_policy *pol = container_of(policyPtr, struct rbt_policy, policy);
134
135   rcu_read_lock();
136
137   for_each_process(p) {
138     /* Check that the process is allowed to migrate */
139     if (likely(!can_use_krg_cap(p, CAP_CAN_MIGRATE))) {
140       continue;
141     }
142
143     /* Check is the process is in a correct state */
144     if (p->state != TASK_RUNNING || p->ptrace != 0) {
145       continue;
146     }
147
148     /* Check if the process is not already migrating */
149     if (p->flags & PF_MIGRATING) {
150       continue;
151     }
152
153     /* Check if the process is a Kerrighed one */
154     if (unlikely(!(info = rcu_dereference(p->krg_sched)))) {
155       continue;
156     }
157
158     scheduler_port_get_value(
159       &pol->port_process_size,
160       &total_vm,
161       1, &p->pid, 1);
162
163     scheduler_port_get_value(
164       &pol->port_process_jiffies,
165       &jiffies,
166       1, &p->pid, 1);
167
168     if ( (lowest_vm == 0) || (total_vm < lowest_vm) ) {
169       lowest_vm = total_vm;
170       max_p = p;
171     }
172   }
173
174   if (max_p) {
175     p = max_p;
176   } else {
177     p = NULL;
178   }
179
180   if (p) {
181     task = p->pid;
182   }
183
184   rcu_read_unlock();
185
186   return task;
187 }
188
```

```
189 kerrighed_node_t rbt_find_a_node(kerrighed_node_t * next, unsigned int * idealLoads,
        struct scheduler_policy * policyPtr) {
190   kerrighed_node_t selected_node = KERRIGHED_NODE_ID_NONE;
191   kerrighed_node_t backup = *next;
192
193   unsigned int mattloadValue;
194   struct rbt_policy *p = container_of(policyPtr, struct rbt_policy, policy);
195
196   scheduler_port_get_remote_value(
197     &p->port_mattload,
198     *next,
199     &mattloadValue,
200     1, NULL, 0);
201
202   if (mattloadValue < idealLoads[*next]) {
203     selected_node = *next;
204     toggle_next_node(next);
205     return selected_node;
206   } else {
207     toggle_next_node(next);
208     scheduler_port_get_remote_value(
209       &p->port_mattload,
210       *next,
211       &mattloadValue,
212       1, NULL, 0);
213     while (*next != backup) {
214       if (mattloadValue < idealLoads[*next]) {
215         selected_node = *next;
216         toggle_next_node(next);
217         return selected_node;
218       } else {
219         toggle_next_node(next);
220       }
221     }
222   }
223
224   if (selected_node == KERRIGHED_NODE_ID_NONE)
225     selected_node = krgnode_next_possible_in_ring(*next);
226
227   return selected_node;
228 }
229
230 void toggle_next_node(kerrighed_node_t * nextNode)
231 {
232   kerrighed_node_t nextSelectedNode, currentNode;
233   currentNode = *nextNode;
234   nextSelectedNode = krgnode_next_possible_in_ring(currentNode);
235   *nextNode = nextSelectedNode;
236 }
237
238 struct scheduler_policy * rbt_policy_new(const char *name)
239 {
240   struct rbt_policy *p;
241   struct config_group *def_groups[6];
242   int err;
243
244   p = kmalloc(sizeof(*p), GFP_KERNEL);
245   if (!p)
246     goto err_rbt_policy;
247
248   err = scheduler_port_init(&p->port_mattload, "mattload",
249           &port_mattload_type, NULL, NULL);
```

```
250   if (err)
251     goto err_mattload;
252
253   err = scheduler_port_init(&p->port_active_tasks, "active_tasks",
254           &port_active_tasks_type, NULL, NULL);
255   if (err)
256     goto err_active_tasks;
257
258   err = scheduler_port_init(&p->port_loadinc, "loadinc",
259           &port_loadinc_type, NULL, NULL);
260   if (err)
261     goto err_loadinc;
262
263   err = scheduler_port_init(&p->port_process_jiffies, "process_jiffies",
264           &port_process_jiffies_type, NULL, NULL);
265   if (err)
266     goto err_process_jiffies;
267
268   err = scheduler_port_init(&p->port_process_size, "process_size",
269           &port_process_size_type, NULL, NULL);
270   if (err)
271     goto err_process_size;
272
273   /* initialize default memory groups. */
274   def_groups[0] = scheduler_port_config_group(&p->port_mattload);
275   def_groups[1] = scheduler_port_config_group(&p->port_active_tasks);
276   def_groups[2] = scheduler_port_config_group(&p->port_loadinc);
277   def_groups[3] = scheduler_port_config_group(&p->port_process_jiffies);
278   def_groups[4] = scheduler_port_config_group(&p->port_process_size);
279   def_groups[5] = NULL;
280
281   err = scheduler_policy_init(&p->policy, name, &rbt_policy_type,
282             def_groups);
283   if (err)
284     goto err_policy;
285
286   /* Initialize work */
287   INIT_DELAYED_WORK(&p->rbt_work, rbt_policy_exec);
288
289   /* Delay work for a few (two) seconds */
290   printk(KERN_INFO "rbt_policy : rbt_work scheduled in %d sec.\n", RBT_RATE);
291   schedule_delayed_work(&p->rbt_work, RBT_RATE*HZ);
292
293   return &p->policy;
294
295 err_policy:
296   scheduler_port_cleanup(&p->port_process_size);
297 err_process_size:
298   scheduler_port_cleanup(&p->port_process_jiffies);
299 err_process_jiffies:
300   scheduler_port_cleanup(&p->port_loadinc);
301 err_loadinc:
302   scheduler_port_cleanup(&p->port_active_tasks);
303 err_active_tasks:
304   scheduler_port_cleanup(&p->port_mattload);
305 err_mattload:
306   kfree(p);
307 err_rbt_policy:
308   printk(KERN_ERR "error : rbt_policy creation failed !\n");
309   return NULL;
310 }
311
```

```
312 void rbt_policy_destroy(struct scheduler_policy *policy)
313 {
314   struct rbt_policy *p =
315     container_of(policy, struct rbt_policy, policy);
316
317   /* Flush then Remove delayed works */
318   flush_scheduled_work();
319   cancel_delayed_work(&p->rbt_work);
320
321   /* Clean memory */
322   scheduler_policy_cleanup(policy);
323   scheduler_port_cleanup(&p->port_mattload);
324   scheduler_port_cleanup(&p->port_active_tasks);
325   scheduler_port_cleanup(&p->port_loadinc);
326   scheduler_port_cleanup(&p->port_process_jiffies);
327   scheduler_port_cleanup(&p->port_process_size);
328   kfree(p);
329 }
330
331 int __init rbt_policy_init(void)
332 {
333   int err;
334
335   err = scheduler_port_type_init(&port_mattload_type, NULL);
336   if (err)
337     goto err_mattload;
338   err = scheduler_port_type_init(&port_active_tasks_type, NULL);
339   if (err)
340     goto err_active_tasks;
341   err = scheduler_port_type_init(&port_loadinc_type, NULL);
342   if (err)
343     goto err_loadinc;
344   err = scheduler_port_type_init(&port_process_jiffies_type, NULL);
345   if (err)
346     goto err_process_jiffies;
347   err = scheduler_port_type_init(&port_process_size_type, NULL);
348   if (err)
349     goto err_process_size;
350   err = scheduler_policy_type_register(&rbt_policy_type);
351   if (err)
352     goto err_register;
353
354  out:
355   return err;
356
357  err_register:
358   scheduler_port_type_cleanup(&port_process_size_type);
359  err_process_size:
360   scheduler_port_type_cleanup(&port_process_jiffies_type);
361  err_process_jiffies:
362   scheduler_port_type_cleanup(&port_loadinc_type);
363  err_loadinc:
364   scheduler_port_type_cleanup(&port_active_tasks_type);
365  err_active_tasks:
366   scheduler_port_type_cleanup(&port_mattload_type);
367  err_mattload:
368   goto out;
369 }
370
371 void __exit rbt_policy_exit(void)
372 {
373   scheduler_policy_type_unregister(&rbt_policy_type);
```

```
374   scheduler_port_type_cleanup(&port_process_size_type);
375   scheduler_port_type_cleanup(&port_process_jiffies_type);
376   scheduler_port_type_cleanup(&port_loadinc_type);
377   scheduler_port_type_cleanup(&port_active_tasks_type);
378   scheduler_port_type_cleanup(&port_mattload_type);
379 }
380
381 module_init(rbt_policy_init);
382 module_exit(rbt_policy_exit);
```

Code G.2 – Module de politique RBT

```
1  /*
2   *  Copyright (C) 2009 Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>
3   */
4
5  #include <linux/kernel.h>
6  #include <linux/module.h>
7  #include <linux/workqueue.h>
8
9  #include <kerrighed/capabilities.h>
10 #include <kerrighed/krginit.h>
11 #include <kerrighed/krgnodemask.h>
12 #include <kerrighed/pid.h>
13 #include <kerrighed/sys/types.h>
14 #include <kerrighed/sched.h>
15
16 #include <tools/syscalls.h>
17 #include <epm/migration.h>
18 #include <epm/migration_api.h>
19
20 #include <scheduler/core/policy.h>
21 #include <scheduler/core/port.h>
22 #include <scheduler/core/scheduler.h>
23 #include <scheduler/core/process_set.h>
24
25 MODULE_LICENSE("GPL v2");
26 MODULE_AUTHOR("Alexandre Lissy <alexandre.lissy@etu.univ-tours.fr>");
27 MODULE_DESCRIPTION("RBT 'Arrosoir' Scheduling policy");
28
29 #define RBT_RATE 5 /* in seconds */
30
31 static struct rbt_policy {
32   struct scheduler_policy policy;
33   struct delayed_work rbt_work;
34   struct scheduler_port port_mattload;
35   struct scheduler_port port_active_tasks;
36   struct scheduler_port port_loadinc;
37   struct scheduler_port port_process_jiffies;
38   struct scheduler_port port_process_size;
39 };
40
41 static ssize_t scheduler_policy_attr_rbt_show(struct scheduler_policy *item,
42   char *page) {
43   ssize_t ret = 0;
44
45   ret = sprintf(page, "calling scheduler_policy_attr_rbt_show !\n");
46
47   return ret;
48 }
49
50 static struct scheduler_policy_attribute rbt_attr = {
```

```
51    .attr = {
52      .ca_owner = THIS_MODULE,
53      .ca_name = "rbt_attr",
54      .ca_mode = S_IRUGO,
55    },
56    .show = scheduler_policy_attr_rbt_show,
57  };
58
59  static struct scheduler_policy_attribute *rbt_policy_attrs[] = {
60    &rbt_attr,
61    NULL,
62  };
63
64  static struct scheduler_policy * rbt_policy_new(const char *name);
65  static void rbt_policy_destroy(struct scheduler_policy *policy);
66
67  static struct scheduler_policy_operations rbt_policy_ops = {
68    .new = rbt_policy_new,
69    .destroy = rbt_policy_destroy,
70  };
71
72  static SCHEDULER_POLICY_TYPE(rbt_policy_type, "rbt_policy",
73              &rbt_policy_ops, rbt_policy_attrs);
74
75  /* Functions */
76  static void rbt_policy_exec(struct work_struct * w);
77  static pid_t rbt_find_a_task(struct scheduler_policy * policyPtr);
78  static kerrighed_node_t rbt_find_a_node(kerrighed_node_t * next, unsigned int *
        idealLoads, struct scheduler_policy * policyPtr);
79  static void toggle_next_node(kerrighed_node_t * nextNode);
80  static struct scheduler_policy * rbt_policy_new(const char *name);
81  static void rbt_policy_destroy(struct scheduler_policy *policy);
82  int __init rbt_policy_init(void);
83  void __exit rbt_policy_exit(void);
```

Code G.3 – Module de politique RBT (entête)

H. Tests écrits pour Kerrighed Test Project

H.1 Squelette de script de test

```
#
# File :        skeletton.sh
#
# Description : Tests skeletton script
#
# Author :      Alexandre Lissy, alexandre.lissy@etu.univ-tours.fr
#
# History :     13 apr 2009 - Created - Alexandre Lissy
#
#! /bin/sh


# Function :    setup
#
# Description : - Setup the tests
#   - Check that /proc/kerrighed/interactive_user exists
#
# Return        - zero on success
#               - non zero on failure. return value from commands ($RC)
setup()
{
  ##  This variable is required by the LTP command line harness APIs
  ##  to keep track of the number of testcases in this file. This
  ##  variable is not local to this file.

  export TST_TOTAL=1  # Total number of test cases in this file.

  # Set up LTPTMP (temporary directory used by the tests).

  ##  Export PATH variable to point to the ltp-yyyymmdd/testcases/bin
  ##  directory before running the tests manually, path is set up
  ##  automatically when you use the runltp script.
  ##  LTPTMP is user defined variables used only in this test case.
  ##  These variables are local to this file.
  ##  The TMP variable is exported by the runltp script.

  export LTPTMP=${TMP}       # Temporary directory to create files, etc.


  ##  The TCID and TST_COUNT variables are required by the LTP
  ##  command line harness APIs, these variables are not local
  ##  to this program.

  export TCID="setup" # Test case identifier
  export TST_COUNT=0  # Set up is initialized as test 0

  # Initialize cleanup function to execute on program exit.
```

```
48    # This function will be called before the test program exits.
49    trap "cleanup" 0
50
51    # Initialize return code to zero.
52
53    ##  RC is used to store the return code from commands that are
54    ##  used in this test. This variable is local to this file.
55
56    RC=0                  # Exit values of system commands used
57
58    #####
59    ## Setup test here
60    #####
61
62    return $RC
63  }
64
65  # Function :     cleanup
66  #
67  # Description   - remove temporary files and directories.
68  #
69  # Return        - zero on success
70  #               - non zero on failure. return value from commands ($RC)
71  cleanup()
72  {
73    ## Clean up code goes here
74  }
75
76  # Function : test01
77  # Description Test number 1
78  #
79  test01()
80  {
81    TCID="test01"
82    TST_COUNT=1
83    RC=0
84
85    # Print test assertion.
86    tst_resm TINFO \
87    "Test #$TST_COUNT : message"
88
89    # exec test
90    RC=$?
91    if [ $RC -ne 0 ]
92    then
93      tst_resm TFAIL $LTPTMP/tst_get.out \
94      "Test #$TST_COUNT : message"
95      return $RC
96    else
97      tst_resm TPASS \
98      "Test #$TST_COUNT : message"
99    fi
100
101   return $RC
102 }
103
104 # Function :     main
105 #
106 # Description :  - Execute all tests, exit with test status.
107 #
108 # Exit :         - zero on success
109 #                - non-zero on failure.
```

```
#
RC=0     # Return value from setup, and test functions.

setup   || exit $RC

test01 || exit $RC
```

Code H.1 – Squelette de script de test

H.2 Test pour Local User Presence

```
#
# File :         local_user_presence.sh
#
# Description :  Test case for the local_user_presence and local_user_notifier
#    kernel modules.
#
# Author :       Alexandre Lissy, alexandre.lissy@etu.univ-tours.fr
#
# History :      21 feb 2009 - Created - Alexandre Lissy
#
#! /bin/sh


# Function :     setup
#
# Description :  - Setup the tests
#    - Check that /proc/kerrighed/interactive_user exists
#
# Return         - zero on success
#                - non zero on failure. return value from commands ($RC)
setup()
{
  ##  This variable is required by the LTP command line harness APIs
  ##  to keep track of the number of testcases in this file. This
  ##  variable is not local to this file.

  export TST_TOTAL=9  # Total number of test cases in this file.

  # Set up LTPTMP (temporary directory used by the tests).

  ##  Export PATH variable to point to the ltp-yyyymmdd/testcases/bin
  ##  directory before running the tests manually, path is set up
  ##  automatically when you use the runltp script.
  ##  LTPTMP is user defined variables used only in this test case.
  ##  These variables are local to this file.
  ##  The TMP variable is exported by the runltp script.

  export LTPTMP=${TMP}        # Temporary directory to create files, etc.


  ##  The TCID and TST_COUNT variables are required by the LTP
  ##  command line harness APIs, these variables are not local
  ##  to this program.

  export TCID="setup" # Test case identifier
  export TST_COUNT=0  # Set up is initialized as test 0
  export MODROOT="/proc/kerrighed/interactive_user"

```

```
49    # Initialize cleanup function to execute on program exit.
50    # This function will be called before the test program exits.
51    trap "cleanup" 0
52
53    # Initialize return code to zero.
54
55    ##  RC is used to store the return code from commands that are
56    ##  used in this test. This variable is local to this file.
57
58    RC=0                  # Exit values of system commands used
59
60    # Load kerrighed-user-local-presence module
61    RC=0
62    tst_resm TINFO "INIT : Loading kerrighed-local-userpresence kernel module"
63    modprobe kerrighed-local-user-presence >$LTPTMP/tst_mod_presence.out; RC=$?
64    if [ $RC -ne 0 ]
65    then
66      tst_brk TBROK $LTPTMP/tst_mod_presence.out NULL "Cannot load kerrighed-local-user
          -presence kernel module. Reason : "
67      return $RC
68    fi
69
70    # Load kerrighed-local-user-notifier module
71    RC=0
72    tst_resm TINFO "INIT : Loading kerrighed-local-user-notifier kernel module"
73    modprobe kerrighed-local-user-notifier >$LTPTMP/tst_mod_notifier.out; RC=$?
74    if [ $RC -ne 0 ]
75    then
76      tst_brk TBROK $LTPTMP/tst_mod_notifier.out NULL "Cannot load kerrighed-local-user
          -notifier kernel module. Reason : "
77      return $RC
78    fi
79
80    # Check if $MODROOT exists
81    RC=0
82    tst_resm TINFO "INIT : Checking for kernel module /proc entry  : $MODROOT"
83    ls $MODROOT >$LTPTMP/tst_ls.out; RC=$?
84    if [ $RC -ne 0 ]
85    then
86      tst_brk TBROK $LTPTMP/tst_ls.out NULL "Are you sure kerrighed-local-user-{
          presence,notifier} are loaded ? Reason : "
87      return $RC
88    fi
89
90    return $RC
91  }
92
93  # Function:     cleanup
94  #
95  # Description   - remove temporary files and directories.
96  #
97  # Return        - zero on success
98  #               - non zero on failure. return value from commands ($RC)
99  cleanup()
100 {
101   ## Clean up code goes here
102   rm \
103     $LTPTMP/tst_mod_presence.out \
104     $LTPTMP/tst_mod_notifier.out \
105     $LTPTMP/tst_ls.out \
106     $LTPTMP/tst_get.out \
107     $LTPTMP/tst_isfree.out \
```

```
108     $LTPTMP/tst_isused.out \
109     $LTPTMP/tst_simpleconnection*.out \
110     $LTPTMP/tst_simpledisconnection*.out \
111     $LTPTMP/tst_complexconnection*.out \
112     $LTPTMP/tst_complexdisconnection*.out
113 }
114
115 # Function : read_entry
116 #
117 # Description - Make a cat on a specified file.
118 #
119 # Return  - What cat returned.
120 read_entry()
121 {
122   ENTRYNAME=$1
123   OUTPUT=$2
124   RC=0
125   RC=`(cat $MODROOT/$ENTRYNAME) 2>$OUTPUT`
126   return $RC
127 }
128
129 # Function : write_entry
130 #
131 # Description - Make a echo on a specified file.
132 #
133 # Return  - What echo returned.
134 write_entry()
135 {
136   ENTRYNAME=$1
137   OUTPUT=$2
138   (echo 1 > $MODROOT/$ENTRYNAME) 2>$OUTPUT
139   return $?
140 }
141
142 # Function : local_user_get
143 # Description - Test if a freshly loaded module returns on /get 0
144 #
145 local_user_get()
146 {
147   TCID="local_user_get"
148   TST_COUNT=1
149   RC=0
150
151   # Print test assertion.
152   tst_resm TINFO \
153   "Test #$TST_COUNT : cat $MODROOT/get = 0"
154
155   read_entry get $LTPTMP/tst_get.out
156   RC=$?
157   if [ $RC -ne 0 ]
158   then
159     tst_resm TFAIL $LTPTMP/tst_get.out \
160     "Test #$TST_COUNT : Cannot get 0 from $MODROOT/get Reason :"
161     return $RC
162   else
163     tst_resm TPASS \
164     "Test #$TST_COUNT : Reading /get works."
165   fi
166   return $RC
167 }
168
169 # Function : local_user_free
```

```
170 # Description - Test if a freshly loaded module returns 1 on /isfree
171 #
172 local_user_isfree()
173 {
174   TCID="local_user_free"
175   TST_COUNT=2
176   RC=0
177
178   # Print test assertion.
179   tst_resm TINFO \
180   "Test #$TST_COUNT : cat $MODROOT/isfree = 1"
181
182   read_entry isfree $LTPTMP/tst_isfree.out
183   RC=$?
184   if [ $RC -ne 1 ]
185   then
186     tst_res TFAIL $LTPTMP/tst_isfree.out \
187     "Test #$TST_COUNT : Cannot get 1 from $MODROOT/isfree Reason :"
188     return $RC
189   else
190     tst_resm TPASS \
191     "Test #$TST_COUNT : Reading /isfree works."
192   fi
193   return 0
194 }
195
196 # Function : local_user_used
197 # Description - Test if a freshly loaded module returns on 0 /isused
198 #
199 local_user_isused()
200 {
201   TCID="local_user_used"
202   TST_COUNT=3
203   RC=0
204
205   # Print test assertion.
206   tst_resm TINFO \
207   "Test #$TST_COUNT : cat $MODROOT/isused = 0"
208
209   read_entry isused $LTPTMP/tst_isused.out
210   RC=$?
211   if [ $RC -ne 0 ]
212   then
213     tst_res TFAIL $LTPTMP/tst_isused.out \
214     "Test #$TST_COUNT : Cannot get 0 from $MODROOT/isused Reason :"
215     return $RC
216   else
217     tst_resm TPASS \
218     "Test #$TST_COUNT : Reading /isused works."
219   fi
220   return 0
221 }
222
223 # Function : local_user_simpleconnection
224 # Description - Write to /connection
225 #   - Check that we have 1 for /get
226 #
227 local_user_simpleconnection()
228 {
229   TCID="local_user_simpleconnection"
230   TST_COUNT=4
231   RC=0
```

```
232
233   write_entry connection $LTPTMP/tst_simpleconnection.out
234
235   # Print test assertion.
236   tst_resm TINFO \
237   "Test #$TST_COUNT : cat $MODROOT/get = 1"
238
239   read_entry get $LTPTMP/tst_simpleconnection2.out
240   RC=$?
241   if [ $RC -ne 1 ]
242   then
243     tst_res TFAIL $LTPTMP/tst_simpleconnection2.out \
244     "Test #$TST_COUNT : Cannot get 1 from $MODROOT/get Reason :"
245     return $RC
246   else
247     tst_resm TPASS \
248     "Test #$TST_COUNT : User is counted."
249   fi
250   return 0
251 }
252
253 # Function : local_user_simplefree
254 # Description - Check that /isfree returns 0
255 #
256 local_user_simplefree()
257 {
258   TCID="local_user_simplefree"
259   TST_COUNT=5
260   RC=0
261
262   # Print test assertion.
263   tst_resm TINFO \
264   "Test #$TST_COUNT : cat $MODROOT/isfree = 0"
265
266   read_entry isfree $LTPTMP/tst_simplefree.out
267   RC=$?
268   if [ $RC -ne 0 ]
269   then
270     tst_res TFAIL $LTPTMP/tst_simplefree.out \
271     "Test #$TST_COUNT : Cannot get 0 from $MODROOT/isfree Reason :"
272     return $RC
273   else
274     tst_resm TPASS \
275     "Test #$TST_COUNT : Node is no more free."
276   fi
277   return 0
278 }
279
280 # Function : local_user_isused
281 # Description - Check that /isused returns 1
282 #
283 local_user_simpleused()
284 {
285   TCID="local_user_isused"
286   TST_COUNT=6
287   RC=0
288
289   # Print test assertion.
290   tst_resm TINFO \
291   "Test #$TST_COUNT : cat $MODROOT/isused = 1"
292
293   read_entry isused $LTPTMP/tst_simpleused.out
```

```
294   RC=$?
295   if [ $RC -ne 1 ]
296   then
297     tst_res TFAIL $LTPTMP/tst_simpleused.out \
298     "Test #$TST_COUNT : Cannot get 1 from $MODROOT/isused Reason :"
299     return $RC
300   else
301     tst_resm TPASS \
302     "Test #$TST_COUNT : Node is used."
303   fi
304   return 0
305 }
306
307 # Function : local_user_simpledisconnection
308 # Description - Write to /disconnection
309 #   - Check that we have 0 for /get
310 #
311 local_user_simpledisconnection()
312 {
313   TCID="local_user_disconnection"
314   TST_COUNT=7
315   RC=0
316
317   write_entry disconnection $LTPTMP/tst_simpledisconnection.out
318
319   # Print test assertion.
320   tst_resm TINFO \
321   "Test #$TST_COUNT : cat $MODROOT/get = 0"
322
323   read_entry get $LTPTMP/tst_simpledisconnection2.out
324   RC=$?
325   if [ $RC -ne 0 ]
326   then
327     tst_res TFAIL $LTPTMP/tst_simpledisconnection2.out \
328     "Test #$TST_COUNT : Cannot get 0 from $MODROOT/get Reason :"
329     return $RC
330   else
331     tst_resm TPASS \
332     "Test #$TST_COUNT : User is disconnected."
333   fi
334   return 0
335 }
336
337 # Function : local_user_complexconnection
338 # Description - Make 3 writes to /connection
339 #   - Check that /get returns 3
340 #
341 local_user_complexconnection()
342 {
343   TCID="local_user_complexconnection"
344   TST_COUNT=8
345   RC=0
346
347   write_entry connection $LTPTMP/tst_complexconnection1.out
348   write_entry connection $LTPTMP/tst_complexconnection2.out
349   write_entry connection $LTPTMP/tst_complexconnection3.out
350
351   # Print test assertion.
352   tst_resm TINFO \
353   "Test #$TST_COUNT : cat $MODROOT/get = 3"
354
355   read_entry get $LTPTMP/tst_complexconnection_get.out
```

```
356   RC=$?
357   if [ $RC -ne 3 ]
358   then
359     tst_res TFAIL $LTPTMP/tst_complexconnection_get.out \
360     "Test #$TST_COUNT : Cannot get 3 from $MODROOT/get Reason :"
361     return $RC
362   else
363     tst_resm TPASS \
364     "Test #$TST_COUNT : We can count 3 connected users."
365   fi
366   return 0
367 }
368
369 # Function : local_user_complexdisconnection
370 # Description - Make 5 writes to /disconnection
371 #   - Check that /get return 0 now
372 #
373 local_user_complexdisconnection()
374 {
375   TCID="local_user_complexdisconnection"
376   TST_COUNT=9
377   RC=0
378
379   write_entry disconnection $LTPTMP/tst_complexdisconnection1.out
380   write_entry disconnection $LTPTMP/tst_complexdisconnection2.out
381   write_entry disconnection $LTPTMP/tst_complexdisconnection3.out
382   write_entry disconnection $LTPTMP/tst_complexdisconnection4.out
383   write_entry disconnection $LTPTMP/tst_complexdisconnection5.out
384
385   # Print test assertion.
386   tst_resm TINFO \
387   "Test #$TST_COUNT : cat $MODROOT/get = 0"
388
389   read_entry get $LTPTMP/tst_complexdisconnection_get.out
390   RC=$?
391   if [ $RC -ne 0 ]
392   then
393     tst_res TFAIL $LTPTMP/tst_complexdisconnection_get.out \
394     "Test #$TST_COUNT : Cannot get 0 from $MODROOT/get Reason :"
395     return $RC
396   else
397     tst_resm TPASS \
398     "Test #$TST_COUNT : We didn't went down zero while more disconnections occured
        than connections."
399   fi
400   return 0
401 }
402
403 # Function :     main
404 #
405 # Description :  - Execute all tests, exit with test status.
406 #
407 # Exit :         - zero on success
408 #                - non-zero on failure.
409 #
410 RC=0     # Return value from setup, and test functions.
411
412 setup  || exit $RC
413
414 local_user_get || exit $RC
415
416 local_user_isfree || exit $RC
```

```
417
418 local_user_isused || exit $RC
419
420 local_user_simpleconnection || exit $RC
421
422 local_user_simplefree || exit $RC
423
424 local_user_simpleused || exit $RC
425
426 local_user_simpledisconnection || exit $RC
427
428 local_user_complexconnection || exit $RC
429
430 local_user_complexdisconnection || exit $RC
```

Code H.2 – Script de test pour la probe Local User Presence

H.3 Test Load

```
1  #
2  # File  :         load_probe.sh
3  #
4  # Description :  Script testing the load probe module
5  #
6  # Author :       Alexandre Lissy, alexandre.lissy@etu.univ-tours.fr
7  #
8  # History :      13 apr 2009 - Created - Alexandre Lissy
9  #
10 #! /bin/sh
11
12
13 # Function :     setup
14 #
15 # Description :  - Setup the tests
16 #    - Check that /config/krg_scheduler/probes/load_probe exists
17 #
18 # Return         - zero on success
19 #                - non zero on failure. return value from commands ($RC)
20 setup()
21 {
22   ##  This variable is required by the LTP command line harness APIs
23   ##  to keep track of the number of testcases in this file. This
24   ##  variable is not local to this file.
25
26   export TST_TOTAL=1  # Total number of test cases in this file.
27
28   # Set up LTPTMP (temporary directory used by the tests).
29
30   ##  Export PATH variable to point to the ltp-yyyymmdd/testcases/bin
31   ##  directory before running the tests manually, path is set up
32   ##  automatically when you use the runltp script.
33   ##  LTPTMP is user defined variables used only in this test case.
34   ##  These variables are local to this file.
35   ##  The TMP variable is exported by the runltp script.
36
37   export LTPTMP=${TMP}        # Temporary directory to create files, etc.
38
39
40   ##  The TCID and TST_COUNT variables are required by the LTP
```

```
41    ##  command line harness APIs, these variables are not local
42    ##  to this program.
43
44    export TCID="setup" # Test case identifier
45    export TST_COUNT=0  # Set up is initialized as test 0
46
47    # Initialize cleanup function to execute on program exit.
48    # This function will be called before the test program exits.
49    trap "cleanup" 0
50
51    # Initialize return code to zero.
52
53    ##  RC is used to store the return code from commands that are
54    ##  used in this test. This variable is local to this file.
55
56    RC=0                  # Exit values of system commands used
57
58    #####
59    ## Setup test here
60    #####
61
62    RC=0
63    ls /config/krg_scheduler/probes/load_probe >$LTPTMP/tst_ls.out; RC=$?
64    if [ $RC -ne 0 ]; then
65      tst_brk TBROK $LTPTMP/tst_ls.out NULL "load_probe␣isn't␣loaded."
66    fi
67
68    return $RC
69  }
70
71  # Function:     cleanup
72  #
73  # Description   - remove temporary files and directories.
74  #
75  # Return        - zero on success
76  #               - non zero on failure. return value from commands ($RC)
77  cleanup()
78  {
79    ## Clean up code goes here
80    rm $LTPTMP/tst_ls.out
81    rm $LTPTMP/tst_cat.out
82  }
83
84  # Function: test_active_tasks
85  # Description Test the number of active tasks.
86  #
87  test_active_tasks()
88  {
89    TCID="active_tasks"
90    TST_COUNT=1
91    RC=0
92
93    # Print test assertion.
94    tst_resm TINFO \
95    "Test␣#$TST_COUNT:␣Test␣load_probe/active_tasks"
96
97    # exec test
98    NBRPROC=$((`cat /proc/loadavg |cut -d' ' -f 4|cut -d'/' -f 1`-2))
99    sleep 1
100   VALUE=`(cat /config/krg_scheduler/probes/load_probe/active_tasks/value) 2>$LTPTMP/
          tst_cat.out`
101   RC=$?
```

```
102   if [ $RC -ne 0 ]
103   then
104     tst_resm TFAIL \
105     "Test #$TST_COUNT : failure to cat"
106     return $RC
107   else
108     if [ $VALUE -ne $NBRPROC ]
109     then
110       tst_resm TFAIL \
111       "Test #$TST_COUNT : different load values  : unix :$NBRPROC probe :$VALUE."
112       return $RC
113     else
114       tst_resm TPASS \
115       "Test #$TST_COUNT : load value correct."
116     fi
117   fi
118
119   return $RC
120 }
121
122 # Function :     main
123 #
124 # Description :  - Execute all tests, exit with test status.
125 #
126 # Exit :         - zero on success
127 #                - non-zero on failure.
128 #
129 RC=0     # Return value from setup, and test functions.
130
131 setup  || exit $RC
132
133 test_active_tasks || exit $RC
```

Code H.3 – Script de test pour la probe Load

H.4 Test User

```
1  #
2  # File  :        user_probe.sh
3  #
4  # Description :  Tests the user probe module
5  #
6  # Author :       Alexandre Lissy, alexandre.lissy@etu.univ-tours.fr
7  #
8  # History :      13 apr 2009 - Created - Alexandre Lissy
9  #
10 #! /bin/sh
11
12
13 # Function :     setup
14 #
15 # Description :  - Setup the tests
16 #    - Check that /proc/kerrighed/interactive_user and /config/krg_scheduler/probes/
       user_probe exists
17 #
18 # Return         - zero on success
19 #                - non zero on failure. return value from commands ($RC)
20 setup()
21 {
```

```
22    ##  This variable is required by the LTP command line harness APIs
23    ##  to keep track of the number of testcases in this file. This
24    ##  variable is not local to this file.
25
26    export TST_TOTAL=1  # Total number of test cases in this file.
27
28    # Set up LTPTMP (temporary directory used by the tests).
29
30    ##  Export PATH variable to point to the ltp-yyyymmdd/testcases/bin
31    ##  directory before running the tests manually, path is set up
32    ##  automatically when you use the runltp script.
33    ##  LTPTMP is user defined variables used only in this test case.
34    ##  These variables are local to this file.
35    ##  The TMP variable is exported by the runltp script.
36
37    export LTPTMP=${TMP}        # Temporary directory to create files, etc.
38
39
40    ##  The TCID and TST_COUNT variables are required by the LTP
41    ##  command line harness APIs, these variables are not local
42    ##  to this program.
43
44    export TCID="setup" # Test case identifier
45    export TST_COUNT=0  # Set up is initialized as test 0
46    export MODROOT="/proc/kerrighed/interactive_user"
47
48    # Initialize cleanup function to execute on program exit.
49    # This function will be called before the test program exits.
50    trap "cleanup" 0
51
52    # Initialize return code to zero.
53
54    ##  RC is used to store the return code from commands that are
55    ##  used in this test. This variable is local to this file.
56
57    RC=0                # Exit values of system commands used
58
59    #####
60    ## Setup test here
61
62          # Check if $MODROOT exists
63          RC=0
64          tst_resm TINFO "INIT : Checking for kernel module /proc entry  : $MODROOT"
65          ls $MODROOT >$LTPTMP/tst_ls.out ; RC=$?
66          if [ $RC -ne 0 ]
67          then
68                  tst_brk TBROK $LTPTMP/tst_ls.out NULL "Are you sure kerrighed-user-
                        local-{presence,notifier} are loaded ? Reason : "
69                  return $RC
70          fi
71
72          RC=0
73          ls /config/krg_scheduler/probes/user_probe >$LTPTMP/tst_ls.out ; RC=$?
74          if [ $RC -ne 0 ] ; then
75                  tst_brk TBROK $LTPTMP/tst_ls.out NULL "user_probe isn't loaded."
76          fi
77
78          return $RC
79    }
80
81    # Function :     cleanup
82    #
```

```
83  # Description   - remove temporary files and directories.
84  #
85  # Return        - zero on success
86  #               - non zero on failure. return value from commands ($RC)
87  cleanup()
88  {
89    ## Clean up code goes here
90    rm $LTPTMP/tst_ls.out
91    rm $LTPTMP/tst_connected.out
92  }
93
94  # Function : test_connected
95  # Description Test that the probe reports the correct amount of users
96  #
97  test_connected()
98  {
99    TCID="test_connected"
100   TST_COUNT=1
101   RC=0
102
103   # Print test assertion.
104   tst_resm TINFO \
105   "Test #$TST_COUNT : Test user_probe/connected"
106
107   # exec test
108   VALUE=$((cat /config/krg_scheduler/probes/user_probe/connected/value) 2>$LTPTMP/
          tst_connected.out)
109   CORRECT=$(cat $MODROOT/get)
110   RC=$?
111   if [ $RC -ne 0 ]
112   then
113     tst_resm TFAIL $LTPTMP/tst_connected.out \
114     "Test #$TST_COUNT : failure to cat"
115     return $RC
116   else
117     if [ $VALUE -ne $CORRECT ]
118     then
119       tst_resm TFAIL $LTPTMP/tst_connected.out \
120       "Test #$TST_COUNT : incoherent values"
121       return $RC
122     else
123       tst_resm TPASS \
124       "Test #$TST_COUNT : coherent values."
125     fi
126   fi
127
128   return $RC
129 }
130
131 # Function :     main
132 #
133 # Description :  - Execute all tests, exit with test status.
134 #
135 # Exit :         - zero on success
136 #                - non-zero on failure.
137 #
138 RC=0     # Return value from setup, and test functions.
139
140 setup   || exit $RC
141
142 test_connected || exit $RC
```

Code H.4 – Script de test pour la probe User

H.5 Test Processsize

```
#
# File :        processsize_probe.sh
#
# Description : Tests skeletton script
#
# Author :      Alexandre Lissy, alexandre.lissy@etu.univ-tours.fr
#
# History :     13 apr 2009 - Created - Alexandre Lissy
#
#! /bin/sh


# Function :    setup
#
# Description : - Setup the tests
#   - Check that /config/krg_scheduler/probes/processsize_probe exists
#
# Return        - zero on success
#               - non zero on failure. return value from commands ($RC)
setup()
{
  ##  This variable is required by the LTP command line harness APIs
  ##  to keep track of the number of testcases in this file. This
  ##  variable is not local to this file.

  export TST_TOTAL=1  # Total number of test cases in this file.

  # Set up LTPTMP (temporary directory used by the tests).

  ##  Export PATH variable to point to the ltp-yyyymmdd/testcases/bin
  ##  directory before running the tests manually, path is set up
  ##  automatically when you use the runltp script.
  ##  LTPTMP is user defined variables used only in this test case.
  ##  These variables are local to this file.
  ##  The TMP variable is exported by the runltp script.

  export LTPTMP=${TMP}       # Temporary directory to create files, etc.


  ##  The TCID and TST_COUNT variables are required by the LTP
  ##  command line harness APIs, these variables are not local
  ##  to this program.

  export TCID="setup" # Test case identifier
  export TST_COUNT=0  # Set up is initialized as test 0

  # Initialize cleanup function to execute on program exit.
  # This function will be called before the test program exits.
  trap "cleanup" 0

  # Initialize return code to zero.

  ##  RC is used to store the return code from commands that are
  ##  used in this test. This variable is local to this file.
```

```
55
56    RC=0                    # Exit values of system commands used
57
58    #####
59    ## Setup test here
60    #####
61
62    RC=0
63    ls /config/krg_scheduler/probes/processsize_probe >$LTPTMP/tst_ls.out; RC=$?
64    if [ $RC -ne 0 ]; then
65      tst_brk TBROK $LTPTMP/tst_ls.out NULL "processsize_probe isn't loaded."
66    fi
67
68    return $RC
69  }
70
71  # Function:     cleanup
72  #
73  # Description   - remove temporary files and directories.
74  #
75  # Return        - zero on success
76  #               - non zero on failure. return value from commands ($RC)
77  cleanup()
78  {
79    ## Clean up code goes here
80    rm $LTPTMP/tst_ls.out
81    rm $LTPTMP/tst_proc.out
82  }
83
84  # Function: test_init_process
85  # Description Test number 1
86  #
87  test_init_process()
88  {
89    TCID="test_init_process"
90    TST_COUNT=1
91    RC=0
92
93    # Print test assertion.
94    tst_resm TINFO \
95    "Test #$TST_COUNT : Checking init process size"
96
97    # exec test
98    PID=1
99    PAGES=$((`grep VmSize /proc/$PID/status |awk '{ print $2 }'`*1024)/(`getconf
          PAGESIZE`))
100   PROBE=`(cat /config/krg_scheduler/probes/processsize_probe/total_vm/value|strings|
          grep "Process $PID :"|awk '{ print $3 }') 2>$LTPTMP/tst_proc.out`
101   RC=$?
102   if [ $RC -ne 0 ]
103   then
104     tst_resm TFAIL "Test #$TST_COUNT : init VmSize unreadable."
105     return $RC
106   else
107     if [ $PAGES -eq $PROBE ]; then
108       tst_resm TPASS "Test #$TST_COUNT : correct init VmSize."
109     else
110       tst_resm TFAIL "Test #$TST_COUNT : error. init VmSize :$PAGES. probe :$PROBE"
111       return $RC
112     fi
113   fi
114
```

```
115   return $RC
116 }
117
118 # Function :      main
119 #
120 # Description :  - Execute all tests, exit with test status.
121 #
122 # Exit :         - zero on success
123 #                - non-zero on failure.
124 #
125 RC=0     # Return value from setup, and test functions.
126
127 setup  || exit $RC
128
129 test_init_process || exit $RC
```

Code H.5 – Script de test pour la probe Processsize

H.6 Test CPUSpeed

```
1  #
2  # File  :        cpuspeed_probe.sh
3  #
4  # Description :  Tests cpuspeed_probe
5  #
6  # Author :       Alexandre Lissy, alexandre.lissy@etu.univ-tours.fr
7  #
8  # History :      13 apr 2009 - Created - Alexandre Lissy
9  #
10 #! /bin/sh
11
12
13 # Function :     setup
14 #
15 # Description :  - Setup the tests
16 #    - Check that /config/krg_scheduler/probes/cpuspeed_probe exists
17 #
18 # Return         - zero on success
19 #                - non zero on failure. return value from commands ($RC)
20 setup()
21 {
22   ##  This variable is required by the LTP command line harness APIs
23   ##  to keep track of the number of testcases in this file. This
24   ##  variable is not local to this file.
25
26   export TST_TOTAL=2  # Total number of test cases in this file.
27
28   # Set up LTPTMP (temporary directory used by the tests).
29
30   ##  Export PATH variable to point to the ltp-yyyymmdd/testcases/bin
31   ##  directory before running the tests manually, path is set up
32   ##  automatically when you use the runltp script.
33   ##  LTPTMP is user defined variables used only in this test case.
34   ##  These variables are local to this file.
35   ##  The TMP variable is exported by the runltp script.
36
37   export LTPTMP=${TMP}        # Temporary directory to create files, etc.
38
39
```

```
40    ##  The TCID and TST_COUNT variables are required by the LTP
41    ##  command line harness APIs, these variables are not local
42    ##  to this program.
43
44    export TCID="setup" # Test case identifier
45    export TST_COUNT=0  # Set up is initialized as test 0
46
47    # Initialize cleanup function to execute on program exit.
48    # This function will be called before the test program exits.
49    trap "cleanup" 0
50
51    # Initialize return code to zero.
52
53    ##  RC is used to store the return code from commands that are
54    ##  used in this test. This variable is local to this file.
55
56    RC=0                # Exit values of system commands used
57
58    #####
59    ## Setup test here
60    #####
61
62    RC=0
63    ls /config/krg_scheduler/probes/cpuspeed_probe >$LTPTMP/tst_ls.out ; RC=$?
64    if [ $RC -ne 0 ] ; then
65      tst_brk TBROK $LTPTMP/tst_ls.out NULL "cpuspeed_probe␣isn't␣loaded."
66    fi
67
68    RC=0
69    echo '1+1'|bc >$LTPTMP/tst_bc.out ; RC=$?
70    if [ $RC -ne 0 ] ; then
71      tst_brk TBROK $LTPTMP/tst_bc.out NULL "bc␣isn't␣installed."
72    fi
73
74    RC=0
75    krgcapset -d +SEE_LOCAL_PROC_STAT >$LTPTMP/tst_locproc.out ; RC=$?
76    if [ $RC -ne 0 ] ; then
77      tst_brk TBROK $LTPTMP/tst_locproc.out NULL "Cannot␣set␣+SEE_LOCAL_PROC_STAT␣
          capability."
78    fi
79
80    return $RC
81  }
82
83  # Function :     cleanup
84  #
85  # Description   - remove temporary files and directories.
86  #
87  # Return        - zero on success
88  #               - non zero on failure. return value from commands ($RC)
89  cleanup()
90  {
91    ## Clean up code goes here
92    rm $LTPTMP/tst_ls.out
93    rm $LTPTMP/tst_bc.out
94    rm $LTPTMP/tst_locproc.out
95    rm $LTPTMP/tst_connected.out
96    rm $LTPTMP/tst_speed.out
97    krgcapset -d -SEE_LOCAL_PROC_STAT
98  }
99
100 # Function : num_cpus
```

```
101 # Description Returns the number of CPUs.
102 num_cpus()
103 {
104   return `grep -c 'processor' /proc/cpuinfo`
105 }
106
107 # Function : test_connected
108 # Description Test the 'connected' probe source of cpuspeed_probe
109 #
110 test_connected()
111 {
112   TCID="connected"
113   TST_COUNT=1
114   RC=0
115
116   # Print test assertion.
117   tst_resm TINFO \
118   "Test #$TST_COUNT : Test cpuspeed_probe/connected"
119
120   # exec test
121   CPUS=`grep -c 'processor' /proc/cpuinfo`
122   VALUE=`(cat /config/krg_scheduler/probes/cpuspeed_probe/connected/value) 2>$LTPTMP/
          tst_connected.out`
123   RC=$?
124
125   if [ $RC -ne 0 ]
126   then
127     tst_resm TFAIL "Test #$TST_COUNT : failure"
128     return $RC
129   else
130     if [ $VALUE -ne $CPUS ]
131     then
132       tst_resm TFAIL "Test #$TST_COUNT : failure : bad numer of CPUs"
133     else
134       tst_resm TPASS "Test #$TST_COUNT : ok"
135     fi
136   fi
137
138   return $RC
139 }
140
141 # Function : test_cpu_speed
142 # Description Test the 'cpu_speed' probe source of cpuspeed_probe
143 #
144 test_cpu_speed()
145 {
146   TCID="cpu_speed"
147   TST_COUNT=2
148   RC=0
149
150   # Print test assertion.
151   tst_resm TINFO \
152   "Test #$TST_COUNT : Testing local CPU speed"
153
154   count=1
155   for line in $(grep "cpu MHz" /proc/cpuinfo|sed -e "s/ //g" -e "s/\t//g")
156   do
157     # exec test
158     CPU=`echo "$line"|cut -d' :' -f 2`
159     CPUKHZ=`(head -n$count /config/krg_scheduler/probes/cpuspeed_probe/speed/value|
            tail -n+$count|cut -d' ' -f 2|cut -d'k' -f 1) 2>$LTPTMP/tst_speed.out`
160     CPUMHZ=`echo "$CPU*1000"|bc|cut -d'.' -f1`
```

```
161
162     if [ "$CPUKHZ" != "$CPUMHZ" ]; then
163       tst_resm TFAIL "Test #$TST_COUNT : CPU ($count) speed doesn't match : probe=
              $CPUKHZ and cpuinfo=$CPUMHZ"
164       return $RC
165     else
166       tst_resm TPASS "Test #$TST_COUNT : CPU ($count) speed match."
167     fi
168     count=$(($count+1))
169   done
170
171   return 0
172 }
173
174 # Function :      main
175 #
176 # Description :  - Execute all tests, exit with test status.
177 #
178 # Exit :         - zero on success
179 #                - non-zero on failure.
180 #
181 RC=0     # Return value from setup, and test functions.
182
183 setup  || exit $RC
184
185 test_connected || exit $RC
186
187 test_cpu_speed || exit $RC
```

Code H.6 – Script de test pour la probe CPUSpeed

I. Autres codes

I.1 Exemple ConfigFS

```
/*
 * vim: noexpandtab ts=8 sts=0 sw=8:
 *
 * configfs_example.c - This file is a demonstration module containing
 *      a number of configfs subsystems.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public
 * License as published by the Free Software Foundation; either
 * version 2 of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public
 * License along with this program; if not, write to the
 * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 021110-1307, USA.
 *
 * Based on sysfs:
 *  sysfs is Copyright (C) 2001, 2002, 2003 Patrick Mochel
 *
 * configfs Copyright (C) 2005 Oracle.  All rights reserved.
 */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>

#include <linux/configfs.h>

/* ---------------------------------------------------------- */

/*
 * 03-group-children
 *
 * This example reuses the simple_children group from above.  However,
 * the simple_children group is not the subsystem itself, it is a
 * child of the subsystem.  Creation of a group in the subsystem creates
 * a new simple_children group.  That group can then have simple_child
 * children of its own.
 */

static struct config_group *group_children_make_group(struct config_group *group,
    const char *name)
{
  struct simple_children *simple_children;

```

```
50    simple_children = kmalloc(sizeof(struct simple_children),
51            GFP_KERNEL);
52    if (!simple_children)
53      return NULL;
54
55    memset(simple_children, 0, sizeof(struct simple_children));
56
57    config_group_init_type_name(&simple_children->group, name,
58              &simple_children_type);
59
60    return &simple_children->group;
61  }
62
63  static struct configfs_attribute group_children_attr_description = {
64    .ca_owner = THIS_MODULE,
65    .ca_name = "description",
66    .ca_mode = S_IRUGO,
67  };
68
69  static struct configfs_attribute *group_children_attrs[] = {
70    &group_children_attr_description,
71    NULL,
72  };
73
74  static ssize_t group_children_attr_show(struct config_item *item,
75              struct configfs_attribute *attr,
76              char *page)
77  {
78    return sprintf(page,
79  "[03-group-children]\n"
80  "\n"
81  "This␣subsystem␣allows␣the␣creation␣of␣child␣config_groups.␣␣These\n"
82  "groups␣are␣like␣the␣subsystem␣simple-children.\n");
83  }
84
85  static struct configfs_item_operations group_children_item_ops = {
86    .show_attribute = group_children_attr_show,
87  };
88
89  /*
90   * Note that, since no extra work is required on ->drop_item(),
91   * no ->drop_item() is provided.
92   */
93  static struct configfs_group_operations group_children_group_ops = {
94    .make_group = group_children_make_group,
95  };
96
97  static struct config_item_type group_children_type = {
98    .ct_item_ops  = &group_children_item_ops,
99    .ct_group_ops = &group_children_group_ops,
100   .ct_attrs = group_children_attrs,
101   .ct_owner = THIS_MODULE,
102 };
103
104 static struct configfs_subsystem group_children_subsys = {
105   .su_group = {
106     .cg_item = {
107       .ci_namebuf = "03-group-children",
108       .ci_type = &group_children_type,
109     },
110   },
111 };
```

```
112
113 /* ------------------------------------------------------------------- */
114
115 /*
116  * We're now done with our subsystem definitions.
117  * For convenience in this module, here's a list of them all.  It
118  * allows the init function to easily register them.  Most modules
119  * will only have one subsystem, and will only call register_subsystem
120  * on it directly.
121  */
122 static struct configfs_subsystem *example_subsys[] = {
123   &group_children_subsys,
124   NULL,
125 };
126
127 static int __init configfs_example_init(void)
128 {
129   int ret;
130   int i;
131   struct configfs_subsystem *subsys;
132
133   for (i = 0; example_subsys[i]; i++) {
134     subsys = example_subsys[i];
135
136     config_group_init(&subsys->su_group);
137     init_MUTEX(&subsys->su_sem);
138     ret = configfs_register_subsystem(subsys);
139     if (ret) {
140       printk(KERN_ERR "Error␣%d␣while␣registering␣subsystem␣%s\n",
141              ret,
142              subsys->su_group.cg_item.ci_namebuf);
143       goto out_unregister;
144     }
145   }
146
147   return 0;
148
149 out_unregister:
150   for (; i >= 0; i--) {
151     configfs_unregister_subsystem(example_subsys[i]);
152   }
153
154   return ret;
155 }
156
157 static void __exit configfs_example_exit(void)
158 {
159   int i;
160
161   for (i = 0; example_subsys[i]; i++) {
162     configfs_unregister_subsystem(example_subsys[i]);
163   }
164 }
165
166 module_init(configfs_example_init);
167 module_exit(configfs_example_exit);
168 MODULE_LICENSE("GPL");
```

Code I.1 – Exemple d'utilisation de ConfigFS

I.2 Fonction implémentant la réception et le traitement d'une requête RPC

```
1  void handle_scheduler_pipe_show_remote_value(struct rpc_desc *desc, void *msgIn,
       size_t size)
2  {
3          struct config_item *item;
4          struct scheduler_probe_source *probe_source;
5          char *page;
6          ssize_t r;
7
8          printk(KERN_INFO "Entering handle_scheduler_pipe_show_remote_value\n");
9          item = global_config_unpack_get_item(desc);
10         probe_source = to_scheduler_probe_source(item);
11
12         page = kmalloc(SCHEDULER_PROBE_SOURCE_ATTR_SIZE, GFP_KERNEL);
13         if (!page)
14                 goto exit;
15
16         r = probe_source->pipe.source->type->show_value(probe_source->pipe.source,
               page);
17         rpc_pack_type(desc, r);
18
19         printk(KERN_INFO "Probe source returned %d bytes.\n", r);
20         if (r > 0) {
21                 rpc_pack(desc, 0, page, r);
22         }
23
24         printk(KERN_INFO "Exiting handle_scheduler_pipe_show_remote_value\n");
25
26         kfree(page);
27         config_item_put(item);
28
29         return;
30
31 exit:
32         printk(KERN_ERR "not enough memory to handle remote pipe show value.\n");
33 }
```

Code I.2 – Fonction implémentant la réception et le traitement d'une requête RPC

I.3 Fonction implémentant la gestion des lectures vers les nœuds distants

```
1  ssize_t scheduler_probe_source_attribute_show_remote(struct config_item *item,
2                                                       struct configfs_attribute * attr
                                                            ,
3                                                       char * page)
4  {
5          kerrighed_node_t node = KERRIGHED_NODE_ID_NONE;
6          struct node_pipe *pipe;
7          struct rpc_desc *desc;
8          struct config_item *cible;
9          struct scheduler_probe_source *probe_source;
10         ssize_t r;
11         int err;
12
13         pipe = to_node_pipe(item);
14         if (!pipe)
```

```
                return -EFAULT;
        node = pipe->node;

        probe_source = to_scheduler_probe_source(item);
        if (!probe_source)
                return -EFAULT;
        cible = &probe_source->pipe.config.cg_item;

        desc = rpc_begin(SCHED_PIPE_SHOW_REMOTE_VALUE, node);
        rpc_pack(desc, 0, NULL, 0); /* needed as trick */
        err = global_config_pack_item(desc, cible);
        if (err)
                printk(KERN_INFO "Error on global_config_pack_item : %d\n", err);

        rpc_unpack_type(desc, r);

        if (r > 0) {
                printk(KERN_INFO "read %d bytes from node %d\n", r, node);
                rpc_unpack(desc, 0, page, r);
        }

        rpc_end(desc, 0);

        return r;
}
```

Code I.3 – Fonction implémentant la gestion des lectures vers les nœuds distants

I.4 Patch NetConsole implémentant le paramètre `do_syslog`

```
--- linux-2.6.6/drivers/net/netconsole.c  2004-06-21 14:06:34.000000000 -0400
+++ linux-2.6.6-netdump/drivers/net/netconsole.c  2004-06-25 13:51:54.000000000 -0400
@@ -54,6 +54,10 @@ static char config[256];
 module_param_string(netconsole, config, 256, 0);
 MODULE_PARM_DESC(netconsole, " netconsole=[src-port]@[src-ip]/[dev],[tgt-port]@<tgt-
    ip>/[tgt-macaddr]\n");

+static int do_syslog;
+module_param(do_syslog, bool, 000);
+MODULE_PARM_DESC(do_syslog, " do_syslog=<yes|no>\n");
+
 static struct netpoll np = {
  .name = "netconsole",
  .dev_name = "eth0",
@@ -64,10 +68,36 @@ static struct netpoll np = {
 static int configured = 0;

 #define MAX_PRINT_CHUNK 1000
+#define SYSLOG_HEADER_LEN 4
+
+static int syslog_chars = SYSLOG_HEADER_LEN;
+static unsigned char syslog_line [MAX_PRINT_CHUNK + 10] = {
+ '<',
+ '5',
+ '>',
+ ' ',
+ [4 ... MAX_PRINT_CHUNK+5] = '\0',
+};
+
```

```
+/*
+ * We feed kernel messages char by char, and send the UDP packet
+ * one linefeed. We buffer all characters received.
+ */
+static inline void feed_syslog_char(const unsigned char c)
+{
+ if (syslog_chars == MAX_PRINT_CHUNK)
+   syslog_chars--;
+ syslog_line[syslog_chars] = c;
+ syslog_chars++;
+ if (c == '\n') {
+   netpoll_send_udp(&np, syslog_line, syslog_chars);
+   syslog_chars = SYSLOG_HEADER_LEN;
+ }
+}

 static void write_msg(struct console *con, const char *msg, unsigned int len)
 {
- int frag, left;
+ int frag, left, i;
  unsigned long flags;

  if (!np.dev)
@@ -75,11 +105,16 @@ static void write_msg(struct console *co

  local_irq_save(flags);

- for(left = len; left; ) {
-   frag = min(left, MAX_PRINT_CHUNK);
-   netpoll_send_udp(&np, msg, frag);
-   msg += frag;
-   left -= frag;
+ if (do_syslog) {
+   for (i = 0; i < len; i++)
+     feed_syslog_char(msg[i]);
+ } else {
+   for(left = len; left; ) {
+     frag = min(left, MAX_PRINT_CHUNK);
+     netpoll_send_udp(&np, msg, frag);
+     msg += frag;
+     left -= frag;
+   }
  }

  local_irq_restore(flags);
-
To unsubscribe from this list: send the line "unsubscribe␣linux-kernel" in
the body of a message to majordomo@vger.kernel.org
More majordomo info at  http://vger.kernel.org/majordomo-info.html
Please read the FAQ at  http://www.tux.org/lkml/
```

Code I.4 – Patch NetConsole

I.5 Implémentation du client RBT pour le serveur d'ordonnancement en espace utilisateur

```
#!/usr/bin/perl -w

use strict;
```

```
4  use warnings;
5
6  use Sys::Syslog;
7  use POSIX;
8  use YAML;
9  use IO::Socket;
10 use Data::Dumper;
11 use Kerrighed::SchedConfig::Policies::RBT;
12
13 use sigtrap 'handler' => \&kill_client, 'INT';
14
15 openlog("RBTScheduler", "ndelay,pid", "info");
16
17 my $client = IO::Socket::INET->new
18   (
19     PeerAddr => "localhost",
20     PeerPort => 4096,
21   ) or die "Can't connect : $!";
22
23 my $register = { action => 'register', name => 'RBT' };
24 my $unregister = { action => 'unregister', name => 'RBT' };
25 my $quit = { action => 'quit' };
26
27 my $continue = 1;
28 my $data;
29 my $bytes_read;
30 my $reply;
31 my %scheduling;
32
33 $client->write(Dump($register));
34 $bytes_read = sysread($client, $data, 65535);
35 $data = Load($data);
36 syslog("info", "Successfully registered with server.") if ($data->{action} eq "result
       ") and ($data->{value} eq "ok");
37
38 while ($continue)
39 {
40   $bytes_read = sysread($client, $data, 1500);
41   $continue = 0 if ($bytes_read eq 0);
42
43   $data = Load($data);
44   if (defined($data->{action}) and $data->{action} eq "schedule")
45   {
46     %scheduling = rbt_calc();
47     $reply->{action} = "scheduling";
48     $reply->{schedule} = \%scheduling;
49     $client->write(Dump($reply));
50   }
51 }
52
53 closelog();
54
55 sub kill_client
56 {
57   syslog("info", "Killing client as requested.");
58   $client->write(Dump($unregister));
59   $bytes_read = sysread($client, $data, 65535);
60   $data = Load($data);
61   syslog("info", "SUCCESS") if ($data) and ($data->{action} eq "result") and ($data
         ->{value} eq "ok");
62
63   $client->write(Dump($quit));
```

```
64    $client->close();
65    closelog();
66  }
```

Code I.5 – Client RBT utilisant le serveur d'ordonnancement

J. Explicitation de ConfigFS par un exemple

Nous avons pu voir précédemment le rôle de ConfigFS, dans le cadre de la section 3.2.3.

Afin d'aider à la compréhension du lecteur, nous allons proposer une illustration des exemples d'utilisation de ce système de fichier virtuel proposés dans les sources de Linux, de sorte à faciliter la compréhension de l'architecture. Cette illustration est disponible en figure J.1.

Le code correspondant à ce schéma est celui proposé en exemple dans les sources du noyau Linux version 2.6.20. Il est disponible en annexe I.1. Ce n'est pas directement le source original, seule la partie concernant l'exemple *Group children* a été conservée, puisque cela correspond à ce qui est fait dans Kerrighed.

Attention à la lecture de ce schéma. Dans ConfigFS, les fichiers ne sont pas représentés par objets dédiés. Leur existence est déduite du répertoire qui les contient, sous la forme d'un tableau d'attributs.

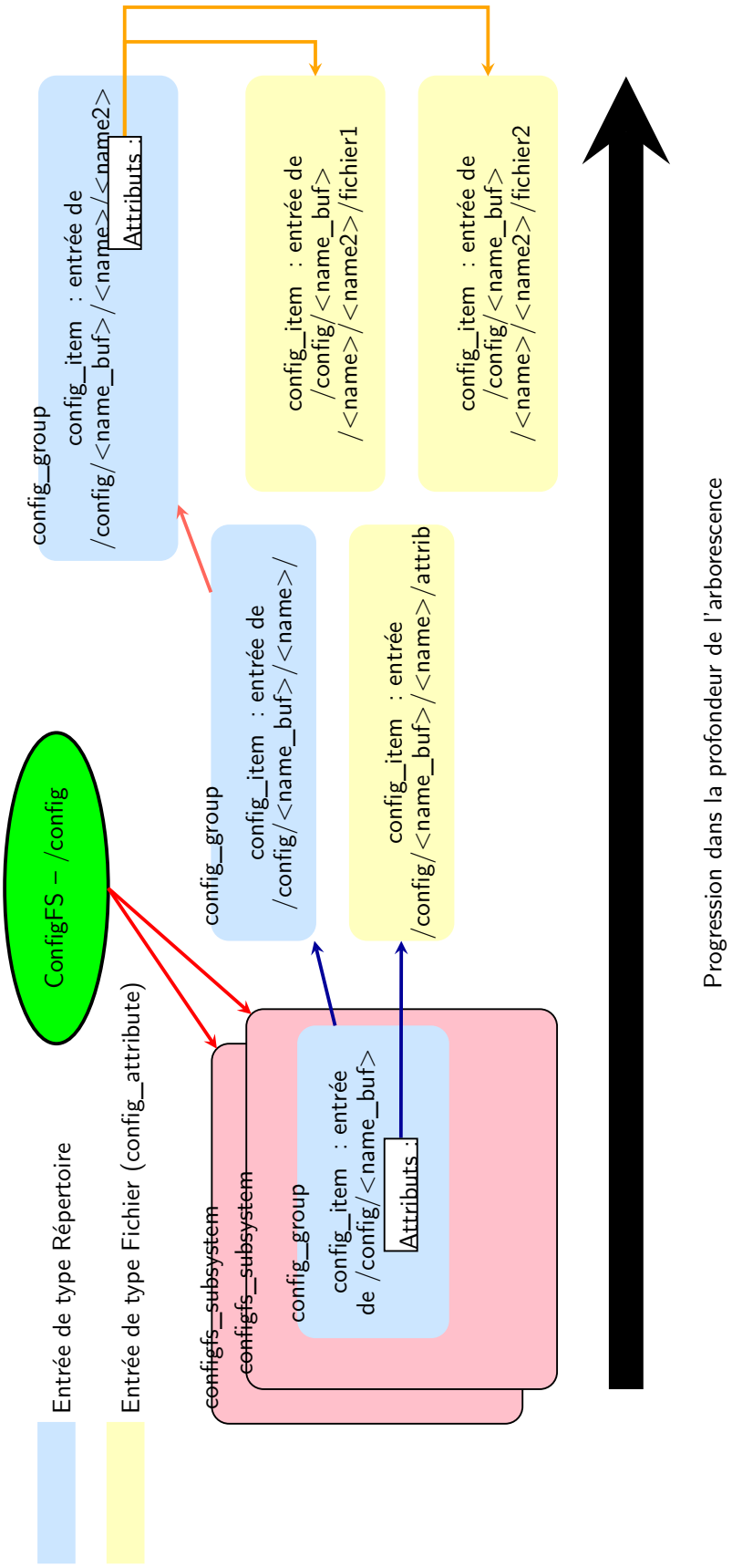


FIG. J.1 – Schéma réprésentant ConfigFS

Bibliographie

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky et Dan Werthimer. « SETI@home : An Experiment in Public-Resource Computing ». *Communications of the ACM*, 45 No. 11, 2002.

[AMZ03] Gagan Aggarwal, Rajeev Motwani et An Zhu. « The Load Rebalancing Problem ». novembre 2003.

[And04] D. Anderson. « BOINC : A System for Public-Resource Computing and Storage ». In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, 2004.

[Bar] Moshe Bar. « The openMosix Project ». Disponible sur : http://openmosix.sourceforge.net/.

[BBN] Terrehon Bowden, Bauer Bodo et Jorge Nerin. « The /proc FileSystem ». Disponible sur : http://lxr.linux.no/linux+v2.6.29/Documentation/filesystems/proc.txt.

[Bec] Joel Becker. « configfs - Userspace-driven kernel object configuration ». Disponible sur : http://lxr.linux.no/linux+v2.6.29/Documentation/filesystems/configfs/configfs.txt.

[Ben09] Hamza Benarafa. « Plateforme de simulation pour l'équilibrage de charge ». Rapport de Projet de Fin d'Études, Département Informatique de l'École Polytechnique de l'Université de Tours, mai 2009.

[BFS99] Petra Berenbrink, Tom Friedetzky et Angelika Steger. « Randomized and adversarial load balancing ». In *SPAA '99 : Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 175–184, New York, NY, USA, 1999. ACM. Disponible sur : http://www14.informatik.tu-muenchen.de/personen/steger/pub/spaa99.ps.gz.

[BS] Amnon Barak et Amnon Shiloh. « MOSIX – Cluster and Multi-Cluster Management ». Disponible sur : http://www.mosix.org.

[Cha96] Steve J. Chapin. « Distributed and Multiprocessor Scheduling ». *ACM Computing Surveys*, 28, 1996. Disponible sur : http://www-users.cs.umn.edu/~jon/papers/handbook.pdf.

[CL06] Nicolas Chenné et Olivier Lesné. « Rééquilibrage de charge ». Rapport technique, Département Informatique de l'École Polytechnique de l'Université de Tours, juin 2006.

[CRKH05a] Jonathan Corbet, Alessandro Rubini et Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly, 3rd edition, feb 2005. Disponible sur : http://lwn.net/Kernel/LDD3/.

[CRKH05b] Jonathan Corbet, Alessandro Rubini et Greg Kroah-Hartman. « *Linux Device Drivers, 3rd Edition* », Code écrit 1, pages 1–12. O'Reilly, 3rd edition, feb 2005. Disponible sur : http://lwn.net/images/pdf/LDD3/ch01.pdf.

[CRKH05c] Jonathan Corbet, Alessandro Rubini et Greg Kroah-Hartman. « *Linux Device Drivers, 3rd Edition* », Code écrit 2, pages 15–39. O'Reilly, 3rd edition, feb 2005. Disponible sur : http://lwn.net/images/pdf/LDD3/ch02.pdf.

[Dab08] Mathieu Dabert. « Outils pour l'Ordonnancement réparti ». Rapport de Projet de Fin d'Études, Département Informatique de l'École Polytechnique de l'Université de Tours, mai 2008.

[DEMT05] Pierre-Francois Dutot, Lionel Eyraud, Grégory Mounié et Denis Trystram. « Bi-criteria Algorithm for Scheduling Jobs on Cluster Platforms ». *ACM Symposium on Parallel Algorithms*

and Architectures (2004) 125-132, January 2005. Disponible sur : http://arxiv.org/abs/cs/0405006.

[Den] John S. Denker. « Debugging Linux Kernel Lockup / Panic / Oops ». Disponible sur : http://www.av8n.com/computer/htm/kernel-lockup.htm.

[Dev] Linux Kernel Developpers. « Macro might_sleep() du noyau Linux ». Disponible sur : http://lxr.linux.no/#linux+v2.6.20/include/linux/kernel.h#L63.

[Dil] Matthew Dillon. « The DragonFlyBSD Project ». Disponible sur : http://www.dragonflybsd.org.

[FL08] Cyrille Faucheux et Alexandre Lissy. « Développement de pilotes USB sous Linux en espace noyau ». Rapport technique, École Polytechnique de l'Université de Tours, 64, avenue Jean Portalis, 37200 Tours, 2008.

[FL09] Cyrille Faucheux et Alexandre Lissy. « Monitoring d'un cluster Kerrighed ». Rapport technique, École Polytechnique de l'Université de Tours, 64, avenue Jean Portalis, 37200 Tours, 2009.

[GN07] Ludovic Gaud et Othmane Nour. « Load Balancing ». Rapport technique, Département Informatique de l'École Polytechnique de l'Université de Tours, juin 2007.

[Jon] M. Tim Jones. « Anatomy of the Linux SLAB Allocator ». Disponible sur : http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/.

[K+] Kerlabs et others. « *Kerrighed regression tests : KTP* ». Disponible sur : http://www.kerrighed.org/wiki/index.php/Ktp.

[Ker] Kerlabs. « Kerrighed – Linux Clusters Made Easy ». Disponible sur : http://www.kerrighed.org.

[Lab] Bell Labs. « Plan 9 from Bell Labs ». Disponible sur : http://cm.bell-labs.com/plan9/.

[Lis] Alexandre Lissy. « Centralizing (Kernel) Logs ». Disponible sur : http://www.kerrighed.org/wiki/index.php/KernelLogs.

[Lon] Julia Longtin. « The LinuxPMI Project ». Disponible sur : http://linuxpmi.org/trac/.

[LR07] Guolong Lin et Rajmohan Rajaraman. « Approximation Algorithms for Multiprocessor Scheduling under Uncertainty », March 2007. Disponible sur : http://arxiv.org/abs/cs/0703100.

[Mac01] David M. Mackie. « Simple and Effective Distributed Computing with a Scheduling Service », June 2001. Disponible sur : http://arxiv.org/abs/cs/0106038.

[Moc] Patrick Mochel. « The sysfs Filesystem ». Disponible sur : http://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf.

[Mol] Ingo Molnar. « Linux : Voluntary Kernel Preemption ». Disponible sur : http://kerneltrap.org/node/3440.

[Mor] Andrew G. Morgan. « Linux Pluggable Authentication Modules ». Disponible sur : http://www.kernel.org/pub/linux/libs/pam/.

[MRRV06] Loris Marchal, Veronika Rehn, Yves Robert et Frédéric Vivien. « Scheduling and data redistribution strategies on star platforms », October 2006. Disponible sur : http://arxiv.org/abs/cs/0610131.

[Pér08] Matthieu Pérotin. « *Calcul Haute Performance sur Matériel Générique* ». Thèse de doctorat, Laboratoire d'Informatique de l'École Polytechnique de l'Université de Tours, décembre 2008.

[PME08a] Matthieu Pérotin, Patrick Martineau et Carl Esswein. « Équilibrage de charge pour la grille ». In *MOSIM'08*, Paris, 31 mars - 2 avril 2008.

[PME08b] Matthieu PÉROTIN, Patrick MARTINEAU et Carl ESSWEIN. « Rééquilibrage de charge en-ligne bi-critère ». In *RenPar'18 / SympA'2008 / CFSE'6*, Fribourg, février 2008.

[PR09] Quentin PROUST et Tarik ROUMADNI. « Développement de modules PAM et WinLogon pour l'interception et la notification de connexion et déconnexion d'utilisateurs à une machine virtuelle ». Rapport technique, École Polytechnique de l'Université de Tours, 64, avenue Jean Portalis, 37200 Tours, 2009.

[Ren07] Lottiaux RENAUD. « Using Linux Test Project for Kerrighed testing ». In *Kerrighed Summit'07*, IBIS Hotel, Paris-La Défense, février 2007.

[Ril] Louis RILLING. « Ordonnanceur MOSIX de Kerrighed ». Kerrighed/modules/scheduler/policies/mosix_load_balancer.c.

[RN08] Louis RILLING et Marko NOVAK. « Run-Time Configurable Scheduler Framework ». In *Kerrighed Summit'08*, avril 2008. Disponible sur : http://www.kerrighed.org/wiki/index.php/SchedConfig.

[RSAU91] Larry RUDOLPH, Miriam SILVKIN-ALLALOUF et Eli UPFAL. « A simple load balancing scheme for task allocation in parallel machines ». *ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[SAL+02] Jahanzeb SHERWANI, Nosheen ALI, Nausheen LOTIA, Zahra HAYAT et Rajkumar BUYYA. « Libra : An Economy driven Job Scheduling System for Clusters ». Rapport technique Technical Report, July 2002, Dept. of Computer Science and Software Engineering, The University of Melbourne, July 2002. Disponible sur : http://arxiv.org/abs/cs/0207077.

[SI] SGI et IBM. « The Linux Test Project ». Disponible sur : http://ltp.sourceforge.net.

[SS] Vipin SAMAR et Roland SCHEMERS. « Pluggable Authentication Modules ». Disponible sur : http://www.sun.com/software/solaris/pam/.

[Uni] Stanford UNIVERSITY. « Folding@Home, Distributed Computing ». Disponible sur : http://folding.stanford.edu/French/Main.

[Wal] Bruce WALKER. « OpenSSI (Single System Image) Clusters for Linux ». Disponible sur : http://www.openssi.org.

[Win] « WinLogon and Credential Providers ». Disponible sur : http://msdn.microsoft.com/en-us/library/bb648647(VS.85).aspx.

Index

Politiques hybrides d'ordonnancement dans un cluster Kerrighed

Département Informatique
5^{e}année
2008-2009

Stage de Fin d'Études

Résumé : Ce travail de fin d'étude a pour objectif l'implémentation de politiques d'équilibrage de charges dans le système à image unique Kerrighed. Afin de le réaliser, une identification précise des informations pertinentes pour la prise de décisions a été menée. Les composants logiciels nécessaires pour faire remonter ces informations ont été implémentés. Ainsi, un framework complet permettant l'implémentation aisée de nouvelles politiques a été spécifié et utilisé afin de fournir à Kerrighed une couche d'ordonnancement convaincante. Par ailleurs, une étude bilbiographique a été effectuée afin d'étudier de nouvelles pistes pour l'implémentation de politiques d'équilibrage de charges innovantes. Enfin, une réflexion visant à permettre l'implémentation d'algorithmes d'équilibrage de charges depuis l'espace utilisateur a été menée.

Mots clefs : ordonnancement en-ligne, équilibrage de charge, système à image unique, kerrighed, calcul haute performance, développement noyau

Abstract : This work's main objective is to implement load-balancing policies for the Kerrighed Single-Image System. In order to achieve this goal, precise identification of needed information for decision-taking was completed. Resulting pieces of software have been implemented to get those informations. Thus, a complete framework easying implementation of new policies has been specified and used to provide Kerrighed with a convincing scheduling layer. Bibliographic study was also lead to study new ways of implementing state-of-the-art scheduling policies. Then, a study was conducted to achieve another goal : allowing the implementation of scheduling policies in userspace.

Keywords : on-line scheduling, load (re)balancing, single-system image, kerrighed, high performance computing, kernel development

---

Encadrant
Matthieu Pérotin
matthieu.perotin@univ-tours.fr

Étudiant
Alexandre LISSY
alexandre.lissy@etu.univ-tours.fr

Université François-Rabelais, Tours

DI5 - Promo 2009