PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Protocole de mise en route et surveillance sur une grille de calcul de noeuds virtuels
Auteur: Author:	Jean-François Richard
Date:	2007
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Richard, J.-F. (2007). Protocole de mise en route et surveillance sur une grille de calcul de noeuds virtuels [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8021/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/8021/
Directeurs de recherche: Advisors:	Robert Roy
Programme: Program:	Unspecified

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

PROTOCOLE DE MISE EN ROUTE ET SURVEILLANCE SUR UNE GRILLE DE CALCUL DE NŒUDS VIRTUELS

JEAN-FRANÇOIS RICHARD
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2007

© Jean-François Richard, 2007.

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-35697-5
Our file Notre référence
ISBN: 978-0-494-35697-5

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

PROTOCOLE DE MISE EN ROUTE ET SURVEILLANCE SUR UNE GRILLE DE CALCUL DE NŒUDS VIRTUELS

présenté par: RICHARD Jean-François
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

M. DAGENAIS Michel, Ph.D., président
M. ROY Robert, Ph.D., membre et directeur de recherche
M. BERTRAND François, Ph.D., membre

Til Helga min yndlings viking

REMERCIEMENTS

Ce travail a été effectué au laboratoire de Design et Réalisation d'Applications Parallèles à l'École Polytechnique de Montréal. En premier lieu, je remercie toute l'équipe de ce laboratoire, particulièrement M. Robert Roy, derrière moi depuis tant d'années et m'ayant poussé dans de nombreuses aventures que je n'aurais jamais envisagé un jour réaliser.

C'est également à ceux qui m'ont supporté tout au long de ces années à l'École Polytechnique que je dis merci.

Un merci aux gens qui, depuis mes premières expériences au CERCA, ont su éveiller en moi une curiosité pour l'informatique appliquée et m'ont amené à découvrir beaucoup sur ce domaine. Merci particulièrement à MM. Olivier Crête, Benoît Morin, Sylvain Fourmanoît.

À tous ceux qui m'ont épaulé et aidé dans divers projets, vous vous reconnaîtrez, un merci.

Enfin, aux membres du jury, un remerciement particulier pour votre aide, vos commentaires et votre patience. C'est grâce à vous que je pourrai poser ma toute petite brique sur l'édifice des sciences appliquées.

RÉSUMÉ

Des recherches récentes, des améliorations au silicone et des implémentations modernes rendent viable l'exploitation de plusieurs machines virtuelles sur un seul ordinateur. Pour plusieurs, cela permet la consolidation de différents systèmes logiques en un seul système physique. Pour d'autres, cette possibilité promet de faciliter la distribution du travail de manière sécuritaire, non-invasive, isolée et encapsulée. On cherche à utiliser la technologie de virtualisation pour déployer des systèmes virtuels qui s'exécuteront dans des environnements entièrement contrôlés sur un parc d'ordinateurs physiques offrant au public de partager une partie de leurs ressources matérielles.

Notre recherche propose un couplage original de la technologie de virtualisation à une approche de démarrage de nœuds à racine unique (*Single Filesystem Image* – SFI) pour faciliter la génération rapide de grilles de calculs. S'appuyant sur des recherches dans le monde des logiciels de gestion de grilles et grappes de calculs, nous voulons permettre à un usager de démarrer des clones de son système actuel en tant que nœuds virtuels. Tout cela sans générer et recopier une image complète du système local : notre approche est légère et ne nécessite que le transfert d'un noyau et système *early-userspace* pour mettre en place un nœud.

Pour réaliser cela, nous développons une méthode de démarrage de systèmes d'exploitation permettant la superposition de systèmes de fichiers par *UnionFS* pour mettre en place la racine du système de fichiers d'un ordinateur. Cette manière de procéder permet de superposer plusieurs systèmes de fichiers contenus sur des dispositifs variés et potentiellement exposés sur un réseau.

Nous montrons ensuite comment exploiter cette possibilité d'union de systèmes de fichiers pour permettre le démarrage en SFI. Une stratégie de différentiation

par génération automatique de couche supérieure de superposition via gabarits et scripts est présentée. Cette superposition permet de modifier précisément un système de fichiers en couche inférieure pour donner un rôle différent au nœud démarrant de la racine résultante de l'union de ces deux couches, sans toutefois interférer avec le système de fichiers de base monté à partir du serveur.

Cette stratégie est subséquemment implémentée en cherchant à minimiser l'effort de configuration et le temps requis pour générer l'ensemble de fichiers utilisés pour démarrer un nœud SFI ayant sa propre configuration.

Pour supporter le démarrage de nœuds virtuels sur des systèmes physiques potentiellement distants, nous développons une approche basée sur le modèle architectural *Representational State Transfer* (REST). Cette approche permet la création, le changement d'état, la suppression et la surveillance de systèmes virtuels à travers une interface HTTP. Un prototype d'infrastructure est présenté dans ce mémoire.

L'arrimage de l'outil de démarrage amélioré, du système de génération de couche de différentiation par superposition ainsi que de l'infrastructure réseau de démarrage et surveillance de nœuds virtuels est ensuite évalué.

Notre évaluation montre que des nœuds virtualisés sur les outils de paravirtualisation *Xen* et démarrés en SFI offrent une performance de calcul brut similaire à des nœuds physiques utilisés comme références. La performance en transfert réseau est aussi très proche du référentiel. On détermine ensuite que la solution est limitée en accès disque, étant basée sur une implémentation de systèmes de fichiers réseau peu performante. Sous *QEMU*, en émulation, on obtient une performance inférieure à *Xen*.

Nous tentons aussi de déterminer l'efficacité des outils pour donner une approximation de l'effort requis pour déployer des nœuds virtuels en SFI sur un parc

informatique. Nous examinons également comment se comporte un nœud lors de son démarrage. Nous corroborons que la création du nécessaire pour démarrer un nœud clone et le déploiement de ce nœud sont des opérations simples et rapides.

Une revue de la sécurité de l'ensemble des outils complète ensuite notre évaluation pour exposer des considérations importantes dont il faut tenir compte au déploiement et lors de l'utilisation de machines virtuelles en SFI.

Enfin, nous avançons des propositions d'améliorations pour pallier aux faiblesses, particulièrement en matière de sécurité et de performance de système de fichiers réseau.

ABSTRACT

Numerous recent studies, enhancements to processor chips and modern implementations make it feasible and efficient to support multiple virtual machines on a single computer. For many, this ushers in a new era of consolidation, where disparate logical systems can be united inside a single physical one. For others, virtualization promises to support distributing workloads in a secure, non-invasive, isolated and encapsulated way. Some want to use virtualization technologies to deploy virtual systems that will run inside controlled virtual environments over a computing farm that offers a part of its physical resources for public consumption.

Our work proposes an original coupling of virtualization technologies and single filesystem image (SFI) node booting approaches to enable users to quickly build computing grids. Based on research works in the field of HPC cluster management software, the proposed technique enables users to boot clones of their systems as virtual nodes. All of this without going through generating and copying an entire operating system : our approach is lightweight in the fact that it simply requires transferring a kernel and an early-userspace module to boot a virtual node.

To be able to do so, we develop an operating system booting method that permits filesystem stacking with *UnionFS* to create the root of the filesystem of a computing entity. By using such a strategy, we are free to stack filesystems contained on various devices, potentially available over a network.

We then show how to exploit this new method to boot systems in a SFI fashion. A technique based on differentiating a running system through the use of an automatically generated upper filesystem layer is presented. This upper layer enables the precise modification of a base layer filesystem to give a different role to a node booting from the stacking of both these layers as its root filesystem. All this without

interfering on the base layer mounted from a server.

The implementation of this strategy is presented. In it, we use several mechanisms to minimize the configuration effort and the time required to generate the set of files to be used as a differentiation layer on SFI nodes.

To support booting virtual nodes on physical systems that are potentially remote, we propose an approach based on the Representational State Transfer (REST) architecture. This approach enables creating, changing the state, deleting and monitoring virtual machines through an HTTP interface. We also provide an implementation of this approach.

This coupling of the enhanced booting tool set, the differentiation layer creation software and the virtual node deployment infrastructure is then evaluated.

Our evaluation shows that virtual SFI nodes on the *Xen* paravirtualization toolkit perform well in raw computational power, almost matching the performance of similarly-configured physical nodes. The network transfer rate also approaches that of the reference, while filesystem access is well below a disk-based counterpart, as it relies on a network filesystem not particularly optimized for performance. Under *QEMU*, by emulation, we get a lower performance than with *Xen*.

We also show the efficiency of the tools we develop, to give an approximation of the efforts required to deploy virtual SFI nodes on a computer farm. We additionally evaluate how such a node behaves when booting up. We assess the easiness and quickness of creating the necessary to boot a clone of the running system and to deploy it.

A security review of the tools implemented exposes vulnerabilities and ways to prevent them so that appropriate measures can be taken when using the tool set

in the so-called "real world".

Finally, we discuss several enhancement possibilities, mainly on the security front and on how to enhance filesystem performance.

TABLE DES MATIÈRES

DÉDICACE iv

REMERCIEMENTS v

RÉSUMÉ vi

ABSTRACT ix

TABLE DES MATIÈRES xii

LISTE DES ANNEXES xvii

LISTE DES FIGURES xviii

LISTE DES TABLEAUX xx

LISTE DES LISTAGES xxii

LISTE DES ACRONYMES ET ABBRÉVIATIONS xxiii

INTRODUCTION 1

CHAPITRE 1 REVUE DES CONCEPTS 6

1.1 Quelques aspects de *Linux* 8

1.2 Technologies de virtualisation 9

1.2.1 Taxonomies 11

1.2.1.1 Virtual Square – T^{VS} 12

1.2.1.2 Robin & Irvine – T^{RI} 13

1.2.1.3 Smith & Nair – T^{SN} 14

1.2.2 Particularité de l'architecture *x86* 16
1.2.3 Implémentations de systèmes de virtualisation 18
1.2.3.1 *Xen* 18
1.2.3.2 *QEMU* 22
1.2.3.3 Autres solutions 25
1.3 Systèmes d'exploitation pour grappes de calculs 25
1.3.1 Approches classiques 25
1.3.2 Systèmes à image unique 27
1.3.3 Systèmes à racine unique 27
1.4 Déploiement réseau 29
1.4.1 Déploiement à configuration minimale 30
1.5 Conclusion de la revue des concepts 31

CHAPITRE 2 COMPOSANTS ARCHITECTURAUX 32
2.1 Système d'exploitation *Gentoo GNU/Linux* 33
2.1.1 Portage 34
2.1.2 Système d'initialisation 37
2.2 Clones à racine unique 38
2.2.1 Superposition de systèmes de fichiers 40
2.2.1.1 Introduction à *UnionFS* 40
2.2.1.2 Couches de différentiation 44
2.2.2 Processus de démarrage 45
2.2.2.1 Modifications nécessaires au système *early-userspace* 49
2.2.3 Granularité et norme de différentiation 52
2.3 Déploiement 53
2.3.1 *Representational State Transfer* 54

2.3.1.1 Primitives de communication 56
2.3.2 Abstraction de la solution de virtualisation 57
2.3.3 Format de transfert de machines virtuelles 58
2.4 Découverte automatique 59
2.5 Surveillance 60
2.6 Conclusion du développement de l'architecture 61

CHAPITRE 3 SYSTÈME DE GÉNÉRATION ET DÉPLOIEMENT DE NŒUDS VIRTUELS LÉGERS 63
3.1 Génération de la couche supérieure – *Clone* 63
3.1.1 Génération par gabarits 65
3.1.2 Utilisation de la couche de différentiation 68
3.2 Modifications au système *early-userspace* de *Linux* 68
3.2.1 Introduction à *Genkernel* 69
3.2.2 Modularisation 70
3.2.3 Dispositifs supportés 72
3.2.4 Support avancé pour l'autoconfiguration réseau 74
3.2.5 Racine *UnionFS* et scripts d'initialisation 75
3.2.6 Décompression de données en mémoire 78
3.3 Implémentation d'infrastructure – *Napalm* 79
3.3.1 Routage des requêtes HTTP 80
3.3.2 Démon de contrôle 83
3.3.3 Contrôle d'état 85
3.3.4 Surveillance des nœuds 86
3.3.5 Publication et découverte 89
3.4 Couplage *Clone* + *Napalm* 89

3.5 Boîte d'outils 89
3.5.1 *Clone* 0.1 89
3.5.1.1 Gabarit pour *Gentoo GNU/Linux* 91
3.5.2 Autres gabarits fournis 94
3.5.3 *Napalm* 0.1 94
3.5.4 Noyaux 95
3.5.5 Installation 97

CHAPITRE 4 ÉVALUATION 100
4.1 Ordinateurs de test 101
4.2 Bancs de test sous Xen 102
4.2.1 Performance générale, compilation du kernel 103
4.2.2 Performance réseau 105
4.2.3 Performance disque, *DBench* 107
4.2.4 Performance en calcul, *FourInARow* 110
4.2.5 Application de rendu d'image 111
4.3 Bancs de test sous *QEMU* 113
4.3.1 Performance générale 114
4.3.2 Performance réseau 114
4.4 Impact d'*UnionFS* 115
4.5 Processus de déploiement 118
4.6 Profilage du démarrage 122
4.7 Sécurité 126
4.7.1 Transferts de données sur HTTP 126
4.7.2 NFS 127
4.7.3 Authentification des serveurs *Napalm* 128

4.7.4 Connexions des clones vers l'extérieur 129
4.7.5 Pollution inter-nœuds 129
4.7.6 Conclusion de l'étude de performance 130

CONCLUSION 133

RÉFÉRENCES 139

ANNEXES 154

LISTE DES ANNEXES

ANNEXE I ANALYSE DE PERFORMANCE 154

ANNEXE II EXEMPLES DE SCRIPTS `FSLOADER` 190

ANNEXE III ACCÈS À LA DOCUMENTATION ET AU CODE 196

ANNEXE IV FICHIERS DE CONFIGURATION 197

ANNEXE V PROPOSITION D'AMÉLIORATION : DÉTERMINATION DES TYPES DE MACHINES VIRTUELLES SUPPORTÉES SUR UN SERVEUR *NAPALM* 199

LISTE DES FIGURES

Figure 1.1	Taxonomie des machines virtuelles de Smith & Nair	15
Figure 1.2	Un VMM *Xen*, un domaine 0 et deux systèmes invités en domaines U	20
Figure 1.3	*QEMU* sur un système hôte, faisant fonctionner deux systèmes invités	23
Figure 1.4	Systèmes de fichiers montés localement sur un nœud avec *Adelie/SSI*	28
Figure 2.1	Union de branches de l'arborescence du système de fichiers .	41
Figure 2.2	Ouverture de fichiers sur un système à deux couches superposées avec *UnionFS*	43
Figure 2.3	Séquence d'opérations lors d'une écriture sur un fichier présent sur une couche inférieure en lecture seulement avec *UnionFS*	43
Figure 2.4	Systèmes de fichiers superposés sur un clone à mémoire d'état	45
Figure 2.5	Systèmes de fichiers superposés sur un clone sans mémoire d'état	45
Figure 2.6	Premières étapes du démarrage du sytème d'exploitation . .	47
Figure 2.7	Déplacement de la racine du système de fichiers à la fin de l'exécution de l'*initramfs*	49
Figure 2.8	Syntaxe de paramètre de sélection de racine proposée	51
Figure 2.9	Survol des composants architecturaux	62
Figure 3.1	Exemple d'arbre pour illustrer le parcours en « cascade » . .	67
Figure 3.2	Schéma UML de la classe WebHandler et des classes héritières	82
Figure 3.3	Aiguillage d'une requête pour *Napalm*	83
Figure 3.4	Schéma des différents éléments reliés au contrôle d'état des machines virtuelles hébergées sous *Xen*	84

Figure 3.5 Schéma des différents éléments reliés au contrôle d'état des machines virtuelles hébergées sous *QEMU* 85
Figure 3.6 Transferts d'information entre les différents composants de *Ganglia* 87
Figure 3.7 Capture d'écran de l'affichage de *Ganglia* 88
Figure 3.8 Exploitation des outils *Clone* et *Napalm* en tandem, survol des composants 90
Figure 3.9 Capture d'écran de l'affichage d'un domaine 96
Figure 4.1 Temps requis pour la compilation du noyau sur N4, clTH@N2 sans mémoire d'état et clTH@N2 avec mémoire d'état . . . 104
Figure 4.2 Performance relative en tests *OSDB* de TH et clTH@N2 avec mémoire d'état 105
Figure 4.3 Bande passante entre TH, N2 et clTH@N2 106
Figure 4.4 Bande passante en accès système de fichiers sur des nœuds natifs et virtualisés 108
Figure 4.5 Temps de calcul nécessaire pour effectuer le test *FourInARow* de *FreeBench* sur des nœuds natifs et virtualisés 112
Figure 4.6 Rendu d'un pot de thé par *Tachyon* 113
Figure 4.7 Temps requis pour la compilation sur N4, clTH@N2 *Xen* sans mémoire d'état et clTH@N2 *QEMU* sans mémoire d'état . . 114
Figure 4.8 Bande passante entre nœuds natifs et nœuds virtualisés sous *Xen* et *QEMU* 115
Figure 4.9 Profilage du démarrage d'un clone sans mémoire d'état . . . 124
Figure 4.10 Profilage du démarrage d'un clone avec mémoire d'état . . . 125

LISTE DES TABLEAUX

Tableau 1.1 Architectures sur lesquelles *QEMU* peut fonctionner 23

Tableau 1.2 Architectures que *QEMU* peut simuler 24

Tableau 4.1 Bande passante moyenne pour les tests de performance avec *DBench* 109

Tableau 4.2 Bande passante selon le nombre de couches *UnionFS* 116

Tableau 4.3 Nombre de requêtes HTTP selon les opérations sous *Napalm* 122

Tableau 4.4 Profilage de la première partie du démarrage d'un clone sous *Xen* 123

Tableau I.1 Mesures du temps de compilation sur N4 158

Tableau I.2 Mesures du temps de compilation sur clTH@N2, sans mémoire d'état 159

Tableau I.3 Mesures du temps de compilation sur clTH@N2, avec mémoire d'état 159

Tableau I.4 Mesures du temps de compilation sur clTH@N2, sans mémoire d'état, sous *QEMU* 159

Tableau I.5 Mesures de la performance de PostgreSQL lors de l'exécution du test *OSDB*, sur N4 160

Tableau I.6 Mesures de la performance de PostgreSQL lors de l'exécution du test *OSDB*, sur clTH@N2, avec mémoire d'état 160

Tableau I.7 Mesures de la bande passante système de fichiers sur TH . . 161

Tableau I.8 Mesures de la bande passante système de fichiers sur N4 ayant monté un système de fichiers à partir de TH par NFS, en lecture-écriture 164

Tableau I.9 Mesures de la bande passante système de fichiers sur clTH@N2, sans mémoire d'état 167

Tableau I.10 Mesures de la bande passante système de fichiers sur clTH@N2, avec mémoire d'état 169

Tableau I.11 Mesures de l'impact sur l'éxecution de *FourInARow* selon les options d'optimisation passées au compilateur GCC, sur N2 173

Tableau I.12 Mesures du temps d'exécution de *FourInARow* sur N4 . . . 174

Tableau I.13 Mesures du temps d'exécution de *FourInARow* sur clTH@N2, sans mémoire d'état 174

Tableau I.14 Mesures du temps d'exécution de *FourInARow* sur clTH@N2, avec mémoire d'état 175

Tableau I.15 Mesures de la bande passante (Mbit/s) entre TH et N2, puis entre TH et clTH@N2 178

Tableau I.16 Mesures de la bande passante (Mbit/s) entre TH et N2, puis entre TH et clTH@N2 sous QEMU 179

Tableau I.17 Mesures du temps de rendu d'un pot de thé par *Tachyon* . . 179

Tableau I.18 Mesures de la bande passante système de fichiers sur `tmpfs` sur N4 180

Tableau I.19 Mesures de la bande passante système de fichiers à une seule couche `tmpfs` dans un système de fichiers *UnionFS* sur N4 . 183

Tableau I.20 Mesures de la bande passante système de fichiers à sept couches `tmpfs` dans un système de fichiers *UnionFS* sur N4 185

Tableau I.21 Mesures du temps d'initialisation du noyau et du temps d'exécution de l'*initramfs* 189

LISTE DES LISTAGES

Listage 2.1 Exemple d'arborescence de *Gentoo Portage* 35
Listage 2.2 Grammaire EBNF du paramètre de sélection de la racine . 50
Listage 2.3 Format XML de *libvirt* 58
Listage 2.4 Fichier ressource de machine virtuelle 59
Listage 3.1 Exemple de fichier YAML 68
Listage 3.2 Contenu de l'*initramfs* sous *Genkernel* 3.3.11 70
Listage 3.3 Contenu de l'*initramfs* après modifications à *Genkernel* . . 71
Listage 3.4 Classe de gestion des requêtes HTTP `/about` 81
Listage 3.5 Schéma XML du fichier de statut 86
Listage 3.6 GCDSE pour un système *Gentoo GNU/Linux* de base . . . 92
Listage 4.1 Mesure de la performance de la création de la couche supérieure 120
Listage I.1 Script de mesure du temps de compilation du kernel 158
Listage I.2 Script de mesure de performance sous *OSDB* 160
Listage I.3 Script de mesure de performance disque avec *DBench* . . . 161
Listage I.4 Script de mesure de performance avec *FourInARow* 173
Listage I.5 Script d'évaluation de la performance TCP/IP avec *Netperf* 176
Listage II.1 Script de montage `nfs` 191
Listage II.2 Script de montage `nfs+loop` 193
Listage IV.1 Fichier de configuration des clones 197
Listage IV.2 Configuration du serveur NFS 198

LISTE DES ACRONYMES ET ABBRÉVIATIONS

API	*Application Programming Interface*
ATA	*Advanced Technology Attachment*
ATAPI	*ATA Packet Interface*
BSD	*Berkeley Software Distribution*
CERCA	Centre de Recherche en Calcul Appliqué
CGI	*Common Gateway Interface*
CRUD	*Create, Read, Update, Delete*
CSIM	*Complete Software Interpreter Machine*
DMA	*Direct Memory Access*
DNS	*Domain Name System*
DNS-SD	*DNS Service Discovery*
dom0	Domaine 0 *Xen* (système hôte)
domU	Domaine U *Xen* (système invité)
GCDSE	Gabarit de couche de différentiation de système d'exploitation
GNU	*GNU is Not Unix* (acronyme de récursivité infinie)
HPC	*High Performance Computing*
i686	Architecture des processeurs *Intel* 686 et descendants
IDE	*Integrated Drive Electronics*
IO	*Input Output*
IP	*Internet Protocol*
ISA	*Instruction Set Architecture*
LDAP	*Lightweight Directory Access Protocol*
LVM	*Logical Volume Manager*
mDNS	*Multicast DNS*
NFS	*Network File System*

NIS	*Network Information Services*
PC	*Personal Computer*
PXE	*Preboot Execution Environment*
RAID	*Redundant Array of Independent Drives*
RAM	*Random Access Memory*
REST	*Representation State Transfer*
RPC	*Remote Procedure Call*
SFI	*Single Filesystem Image*
SMTP	*Simple Mail Transfer Protocol*
SOAP	*Simple Object Access Protocol*
SSI	*Single System Image*
SSL	*Secure Sockets Layer*
TCP	*Transmission Control Protocol*
TCQ	*Tagged Command Queuing*
TFTP	*Trivial File Transfer Protocol*
URL	*Uniform Resource Locator*
VMM	*Virtual Machine Monitor*
VPN	*Virtual Private Network*
WSDL	*Web Services Description Language*
x86	Architecture des processeurs *Intel* 386 et descendants
XML	*Extensible Markup Language*

INTRODUCTION

On a déjà multiplié le nombre de flots de données traités par un processeur, multiplié le nombre de cœurs par processeur, multiplié le nombre de processeurs par machine. On en est maintenant à une étape qui semble être le terme suivant dans cette série : *multiplier le nombre de machines par machine.*

Bien que cette dernière évolution n'apporte pas aussi indubitablement un gain en performance que les étapes la précédant, elle est néanmoins une étape importante sur la voie de l'exploitation optimale des ressources physiques des systèmes. On peut dorénavant *segmenter* des postes en plusieurs ordinateurs virtuels, limitant le gaspillage de ressources physiques.

Des avantages économiques clairs découlent de cette possibilité. Dans le monde de l'informatique commerciale, on nomme « consolidation » la possibilité de limiter les coûts matériels en mandatant une seule machine de faire le travail de plusieurs autres, possiblement d'anciennes générations, en exploitant la segmentation de ressources par systèmes virtualisés sur des ordinateurs modernes et performants.

Or, ce n'est que depuis récemment que ce sujet est, à nouveau, populaire. Des projets tels que *Xen* (BARHAM et al., 2003; PRATT et al., 2005) et des produits logiciels commerciaux (*VMWare*, *Virtual PC*, *Parallels*) ont relancé l'intérêt pour ces approches.

Pourquoi ce regain d'intérêt ? Parce que de nouvelles méthodes innovatrices permettent d'exploiter ces processeurs sur lesquels il était difficile de supporter la virtualisation, et parce que les entreprises derrière cette architecture ont récemment apporté des améliorations au jeu d'instructions *x86* pour faciliter significativement les efforts visant à offrir une virtualisation à faible surcoût en performance.

Le monde scientifique montre aussi un intérêt pour ces technologies, y voyant un substrat idéal pour supporter l'exécution de code et systèmes sur demande. Dans ce contexte, on ne cherche pas à consolider, mais plutôt l'inverse : on veut distribuer de manière sécuritaire, contrôlée et simple des systèmes d'exploitation sur des grilles planétaires (KOTSOVINOS, 2005).

Ayant participé au développement d'un logiciel de gestion de systèmes d'exploitation de nœuds de grappes de calculs, nous sommes à même de voir un point d'intersection entre cette lame de fond dictant la distribution de systèmes virtuels et la problématique de distribution sur grappe.

L'idée de distribuer des machines virtuelles est certes élégante, mais elle amène son lot de problèmes. Comment déployer des machines virtuelles efficacement ? Comment configurer ces machines ? Une réflexion sur les méthodes exploitées sur les grappes de calculs nous amène à proposer une solution à ces problèmes pour certains contextes.

Le projet présenté cherche à développer une solution de déploiement de nœuds virtuels légers par une variation sur le thème des méthodes de déploiement à *image de système de fichiers unique* (SFI).

D'une part, nous cherchons à raffiner les méthodes utilisées pour la mise en place de nœuds de grappes de calculs à racine unique. De longues réflexions sur les travaux que nous avons effectués ont mis en lumière certains défauts que nous voulons corriger et ont suggéré des idées innovatrices qu'il faut mettre à l'épreuve.

D'autre part, nous voulons intégrer ces concepts améliorés dans un schème de distribution qui transcende le matériel.

Notre travail en est un d'intégration. Nous relions deux domaines de l'informatique

appliquée qui semblent, à première vue, éloignés pour développer une architecture, l'implanter et l'évaluer.

Nous nous intéressons au développement et à l'analyse d'une solution articulée selon deux principaux axes :

- développer un mécanisme permettant l'exécution de multiples systèmes d'exploitation partageant une racine de système de fichiers unique ;
- développer une méthode de déploiement de ces systèmes dans des conteneurs virtuels sur un réseau.

Motivation

Le scénario principal motivant cette recherche est la création de « grilles de calcul haute performance *instantanées* », possibilité qu'il serait intéressant de découpler des contraintes physiques reliées à la quantité d'ordinateurs disponibles. Cette approche permettrait de réduire le temps nécessaires et coûts associés pour développer des grilles de tests ou encore pour permettre des tests rapides sur les applications parallèles. À plus grande échelle, elle permettrait d'exploiter des ressources discrètes exportées par un parc institutionnel de stations de travail en tant que grilles de calculs sans pour autant remettre en question l'usage habituel de ce même parc.

Contexte du projet

Ce projet s'inspire du projet *Adelie/SSI* auquel nous avons contribué depuis ses débuts. Ce projet, originaire du CERCA (Centre de Recherche en Calcul Appliqué) puis développé à l'École Polytechnique de Montréal en collaboration avec le secteur privé, visait à développer une trousse d'outils de gestion de systèmes SFI et de gestion de grappes en général. MORIN (2006) explique le développement de ce projet en détail.

Se basant sur l'expérience acquise dans le contexte du développement d'*Adelie/SSI*, nous développons une nouvelle trousse d'outils qui s'attaque cette fois à la création, la gestion et la surveillance de grappes complètement virtuelles.

Présentation du mémoire

Le premier chapitre présente les ensembles logiciels pertinents au développement de cette approche. Les différentes méthodes de virtualisation contemporaines sur architecture *x86* sont présentées. Les paradigmes de déploiement sur réseau sont également étudiés.

Le second chapitre présente l'architecture logicielle du système d'exploitation retenu et la méthode utilisée pour permettre le déploiement en SFI. On y traite de *UnionFS*, qui permet de superposer des systèmes de fichiers. Cet outil est la pierre angulaire de notre révision des mécanismes de démarrage de systèmes d'exploitation. Nous voyons aussi la stratégie utilisée pour déployer un grappe instantanée tant à l'intérieur d'une même machine physique que sur un réseau d'ordinateurs.

Au troisième chapitre, les détails de l'implémentation sont présentés. On y voit les structures et logiciels employés. On y décrit en détail les amélioration apportées

aux outils existants.

Enfin, le dernier chapitre présente une évaluation du système selon plusieurs axes. On cherche à mesurer ses performances dans des cas variés et à évaluer l'utilisabilité et la sécurité de la solution.

Un retour est fait sur le travail accompli et des pistes d'amélioration sont enfin explorées à la conclusion de ce mémoire.

CHAPITRE 1

REVUE DES CONCEPTS

Le déploiement d'applications distribuées est nécessaire pour plusieurs domaines scientifiques qui doivent baser leur travail sur des jeux de données immenses, ou encore éxécuter des algorithmes complexes et coûteux. De nombreuses industries doivent aussi travailler avec des systèmes distribués pour maintenir des bases de données gigantesques, fournir des services de jeux multijoueurs, gérer leur production internationale, etc.

Depuis plusieurs décennies, ces types de calculs distribués et parallèles se font soit sur des machines de haute performance généralement coûteuses, soit, depuis un peu moins longtemps, sur des grappes de calcul de type Beowulf (BECKER et al., 1995). Mais encore, on constate que le calcul s'effectue sur des infrastructures dédiées, qui, malgré la baisse de coût engendrée par la « commodisation » des équipements *off-the-shelf*, sont des investissements encore significatifs.

Dans un autre axe du progrès technologique informatique, qui recoupe parfois le calcul de haute performance et distribué, le paradigme de l'exécution de code comme un service émerge grâce à des applications souvent issues de domaines particuliers qui ont des besoins bien spécifiques (GIMPS, 1996; SETIATHOME, 1999; FOLDINGATHOME, 2000, etc.). On cherche à exploiter les ordinateurs personnels, stations de travail, voire certaines consoles de jeux vidéos[1], en utilisant les « cycles perdus » pour effectuer un travail prédéterminé et codé en dur dans l'application distribuée.

[1]Folding@home tourne par exemple sur la console *Sony Playstation 3* (FOLDINGATHOMEPS3, 2007)

Pour permettre une plus grande généricité dans ce service d'exécution sur demande, des approches distribuées et de grilles larges, voire « planétaires » sont proposées par les projets *Condor* (LITZKOW, 1987; FIELDS, 1993; CONDOR, 2007), *Globus Toolkit* (FOSTER, 2006), *PlanetLab* (PLANETLAB, 2007).

Ces approches sont parfois reliées à une interface de programmation particulière, à des méthodes de déploiement définies et intègrent des modules d'ordonnancement, d'authentification et de paiements pour l'usage de ressources discrètes. Certaines se démarquent en exploitant plutôt la virtualisation de systèmes (*XenoServer Platform*, par KOTSOVINOS (2005) ou *Globus Workspace* (GLOBUSWORKSPACE, 2007)) pour permettre aux usagers de déployer des systèmes d'exploitation complets dans des silos d'exécution, les libérant d'interfaces de programmation particulières puisqu'offrant la possibilité de mettre n'importe quel système dans ces silos où on simule directement un ordinateur physique par virtualisation.

Or, la virtualisation a toujours été un problème présentant une certaine difficulté sur l'architecture *x86*, qui ne fut probablement pas conçue pour la supporter aisément. Bien que des ajouts récents aux jeux d'instructions par INTEL (2005) et AMD (2006) permettent déjà un accroissement de performance en virtualisation sur le matériel qu'on qualifierait de *nec plus ultra* dans le monde du PC, un bassin d'ordinateurs d'une taille extraordinaire reste bien présent, sans cette amélioration matérielle.

L'approche adoptée dans le cadre de notre recherche retient des éléments développés dans le cadre de plusieurs projets cités précédemment et joint les technologies de virtualisation à celles de déploiement de grappes à système de fichiers racine partagé. Nous verrons dans les sections suivantes l'état de l'art quant à ces deux facettes de notre projet. Nous étudierons également les outils et environnements sur lesquels s'appuient ces développements.

1.1 Quelques aspects de *Linux*

Linus Torvalds n'a plus besoin d'introduction. Probablement un des plus importants finlandais du dernier demi-siècle, il a développé un système compatible UNIX pour la plate-forme *x86* en 1991. Son développement a subi un effet boule de neige, amplifié par la possibilité de collaborer plus facilement que jamais avec Internet. Cela en fait aujourd'hui le résultat d'une quantité phénoménale d'hommes-mois.

Linux est un environnement multitâche, permettant à plusieurs programmes de partager des ressources simultanément. Cette simultanéité n'est cependant qu'une illusion, car les processus tournant en « parallèle » sont en fait ordonnancés pour tourner l'un après l'autre sur un processeur, pour une petite tranche de temps. Après avoir été exécuté pendant un certain laps de temps, un processus est remplacé par un autre : le noyau doit alors mémoriser le contexte matériel du processus sortant et charger celui du nouveau processus.

Cette manière de procéder, couplée à la notion de partage des ressources par *usagers* permet de multiplexer de manière relativement équitable les ressources de systèmes physiques.

Linux utilise deux niveaux d'exécution, soit le niveau *noyau* et le niveau *utilisateur* (*userspace*). La plupart des processus fonctionnent en mode utilisateur, mais doivent entrer en mode noyau à travers différents appels systèmes définis selon une interface standardisée. Cela leur permet par exemple d'obtenir ou d'envoyer des données vers un périphérique. Selon ces niveaux d'exécution (souvent appelés *rings*), on peut contrôler quels segments de mémoire sont accessibles, quelles pages de mémoire il est possible de lire ou d'écrire et quelles instructions processeur sont disponibles. Typiquement, le noyau s'attend à tourner au niveau 0, alors que les applications sont au niveau le moins privilégié, le 3 (l'architecture *x86* compte 4

niveaux, numérotés de 0 à 3 (INTEL, 2007)).

Ces détails permettront de mieux comprendre certains aspects de notre travail et des technologies de virtualisation utilisées.

1.2 Technologies de virtualisation

Fondamentalement, les technologies de virtualisation mettent en jeu un logiciel particulier, nommé moniteur de machine virtuelle (*Virtual Machine Monitor* – VMM) ou parfois appelé *hyperviseur*. Ce logiciel exporte une interface matérielle-logicielle simulée, sur laquelle des machines virtuelles (VMs) pourront s'exécuter. Le VMM assure la gestion des « vraies » ressources matérielles d'une machine et permet aux VMs un accès contrôlé à ces dernières.

Selon GOLDBERG (1972), on définit une machine virtuelle ainsi :

> *[...] a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode [...]*

Multiplexer les ressources d'un ordinateur par exécution simultanée de plusieurs machines virtuelles présente certaines caractéristiques. ROSENBLUM (2004) en avance trois :

- les logiciels de virtualisation maintiennent l'*isolation* entre les machines (ou *domaines*) virtuels. L'exécution dans un domaine ne devrait pas avoir d'effets de bord sur l'exécution dans un autre domaine. Aussi, les applications d'un domaine ne peuvent inspecter l'état des applications dans un second domaine ;
- les domaines virtuels sont *encapsulés* au-dessus d'un niveau d'indirection (le

VMM), qui peut précisément manipuler et contrôler l'exécution d'une machine virtuelle ;

- les engins de virtualisation qui maintiennent la *compatibilité* par simulation d'une architecture donnée et constante permettent aux VMs de s'exécuter sur différentes architectures physiques. Des problèmes d'hétérogénéité du matériel sont ainsi évités.

Pour FIGUEIREDO et al. (2003), en divisant des ressources pour des *machines virtuelles* plutôt que, comme traditionnellement sous UNIX et ses variantes, des *usagers* ou *processus*, on obtient

> *[...] support for legacy applications, security against untrusted code and users, and computation deployment independently of site administration.*

Ces auteurs expliquent qu'utiliser des machines virtuelles permet aux usagers les exploitant

- d'être découplés du système d'exploitation sous-jacent des systèmes « fournisseurs » du service d'exécution ;
- d'être complètement découplés des autres usagers de ce système ;

et que globalement, le partage de ressources par des machines virtuelles plutôt que par systèmes multiusagers assure

- une plus grande sécurité par l'isolation forte des usagers (ces derniers doivent briser deux niveaux de sécurité pour accéder à des ressources : le VMM et le système d'exploitation) ;
- une plus grande configurabilité : l'usager prépare sa machine virtuelle selon ses besoins particuliers en termes de logiciels ;
- possiblement un accès privilégié (`root`) aux usagers sur la machine virtuelle

(ajoutant encore à la configurabilité) ;
- une réutilisation du code déjà développé (*legacy code*) qui ne nécessite pas de recompilation ou de modifications pour s'exécuter sur l'architecture virtuelle et le système d'exploitation choisis. On peut même envisager de simuler un environnement physique nécessaire à l'exécution d'une application binaire ou d'un système d'exploitation donné ;
- un contrôle des ressources, puisque chaque VM peut (parfois même dynamiquement) se faire allouer ou retirer des ressources particulières (mémoire, espace disque, etc.) et le VMM peut comptabiliser leur utilisation ;
- enfin, un découplage des particularités locales de sites, puisqu'on peut obtenir une interface d'exploitation consistante exportée par un VMM.

Sachant ce qu'on entend par « virtualisation » et ayant examiné pourquoi il peut être désirable d'effectuer un multiplexage de ressources à l'aide de cette technique, nous verrons maintenant différentes stratégies permettant la virtualisation à travers deux implémentations considérées comme performantes, en plus d'être libres et gratuites, soit *Xen* et *QEMU*. Mais d'abord, nous présentons des taxonomies permettant de mettre en relief les différences entre les implémentations présentées et nous voyons en quoi la virtualisation s'avère être un défi particulier sur l'architecture *x86*.

1.2.1 Taxonomies

Plusieurs auteurs ont avancé des catégorisations des technologies de virtualisation, permettant de différencier les offres techniques les unes des autres. Cette section en présente trois. La notation raccourcie $T^{acronyme}$ sera utilisé pour se référer à ces différentes taxonomies.

1.2.1.1 Virtual Square – T^{VS}

Cette catégorisation est proposée par DAVOLI (2005b, 2007). Elle divise les technologies de virtualisation selon trois critères.

1. Niveau de communications entre la machine virtuelle et le système physique :

   Gestion des appels systèmes (*system calls*) L'exécution du code usager dans la machine virtuelle se produit directement sur le processeur physique, mais les appels systèmes (pagination, IO) sont redirigés à un système de gestion des machines virtuelles. L'exécution directe de la majorité du code usager permet d'obtenir de très bonnes performances. Le code usager dans la machine virtuelle doit alors être compilé pour l'architecture physique ;

   Virtualisation de l'architecture Tout le code usager virtualisé est traduit en instructions pour la machine physique sous-jacente (émulation). Il est possible de virtualiser une architecture différente du système physique sur lequel on exécute l'émulateur. Différentes approches de recompilation dynamique et de mémorisation du code traduit permettent d'améliorer les performances.

2. Complétude de la virtualisation :

   Virtualisation du processeur seulement Seul le processeur est virtualisé. Tous les accès aux ressources passent par le système d'exploitation hôte ;

   Machine virtuelle partielle Les programmes sont exécutés sur la machine physique mais certaines parties du système sont virtualisées, sans toutefois démarrer un système invité complet ;

Virtualisation complète Toute l'architecture, incluant le processeur et ses périphériques, est virtuelle. Un système d'exploitation invité est nécessaire pour faire fonctionner la machine virtuelle.

3. Invasivité sur la machine hôte :

   Niveau utilisateur Peut fonctionner entièrement en utilisant les permissions usagers d'un système ;

   Niveau racine (*root*) La mise en place et le démarrage d'une machine virtuelle requiert le privilège d'accès le plus élevé sur la machine hôte ;

   Rustine noyau nécessaire Le noyau doit être modifié, soit en insérant un module externe ou en lui appliquant une rustine.

1.2.1.2 Robin & Irvine – T^{RI}

ROBIN et IRVINE (2000) avancent que tout système d'exploitation est à la base une masse de code qui doit être interprétée sur un processeur physique. Lorsqu'un système d'exploitation est exécuté, la proportion du code interprété sur le processeur physique peut varier entre 0% et 100% du code original.

Cette proportion détermine en grande partie si on est sur l'un de ces grands types de machines :

Interpréteur logiciel (*complete software interpreter machine* – CSIM)
N'utilise que l'interprétation logicielle du code du système d'exploitation (émulation)[2] ;

[2]ROBIN et IRVINE (2000) notent que les VMs gérés sur des CSIM ne sont donc pas un type de VM au sens de GOLDBERG (1972) puisqu'aucune instruction n'est exécutée directement sur le matériel.

Machine Virtuelle Hybride (*Hybrid Virtual Machine* – HVM) Une portion statistiquement dominante du code est exécutée directement sur le processeur physique. Une interprétation logicielle de toutes les instructions privilégiées et sensibles[3] est effectuée ;

VMM Une portion statistiquement dominante du code est exécutée directement sur le processeur physique. On distingue deux types de VMMs :

Type I Tourne directement sur le matériel. Effectue l'ordonnancement et l'allocation des ressources physiques.

Type II Tourne en tant qu'application « invitée » dans un système d'exploitation « hôte », qui se charge de l'ordonnancement et l'allocation des ressources physiques.

Machine physique Le processeur exécute toutes les instructions directement.

Toujours selon ces auteurs, on doit s'attendre à une performance croissante du CSIM à la machine physique, dans l'ordre de la liste précédente.

1.2.1.3 Smith & Nair – T^{SN}

SMITH et NAIR (2005) proposent également une catégorisation, résumée par la figure 1.1.

Cette taxonomie différencie essentiellement la virtualisation par processus, où un processus unique au sein d'un système a accès à des ressources virtualisées. C'est le cas, notamment, des processus sous *Linux*, qui accèdent à une plage mémoire virtuelle, ou encore des languages interprétés de haut niveau comme Java (GOSLING, 1996).

[3]Selon ces auteurs, une instruction *privilégiée* est une instruction qui peut causer une faute lorsqu'exécutée hors d'un certain *ring*, mais pas dans un autre (typiquement, le *0*) ; une instruction est *sensible* si elle peut interférer avec l'état du système d'exploitation hôte ou le VMM.

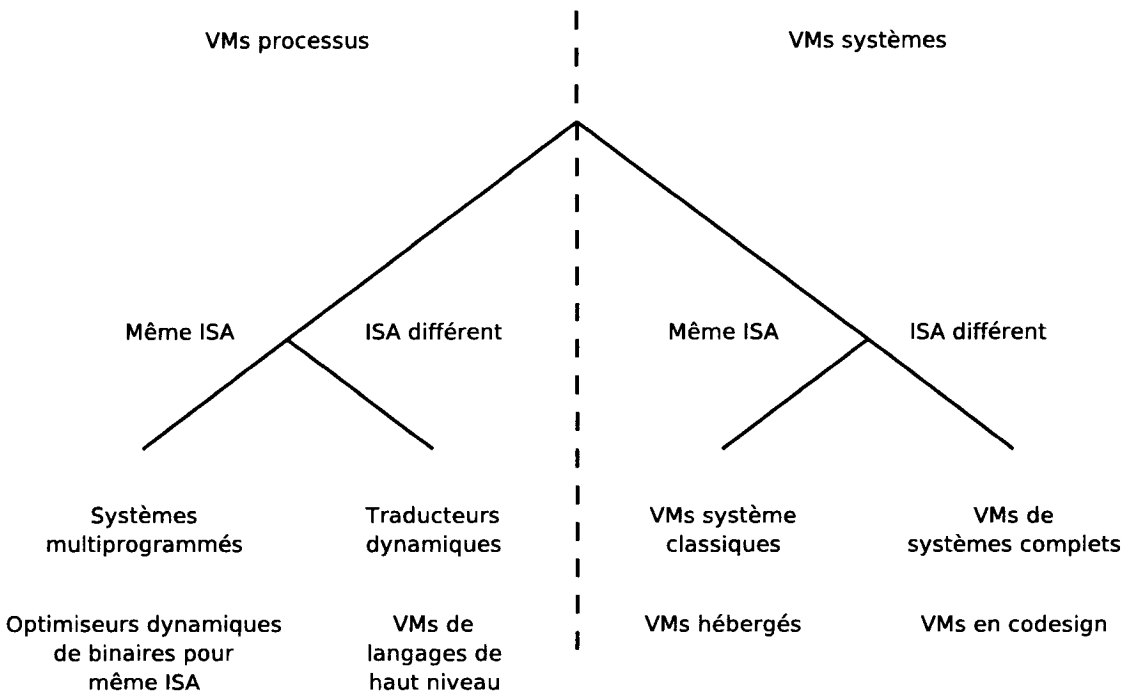

Figure 1.1 Taxonomie des machines virtuelles de Smith & Nair

Dans l'autre cas, on doit plutôt virtualiser un système entier — typiquement, un processeur, sa mémoire et ses périphériques — pour y faire tourner des systèmes d'exploitation et leurs processus simultanément.

On distingue ensuite les types de machines virtuelles en fonction du jeu d'instructions (*Instruction Set Architecture* – ISA) : une machine virtuelle utilisant le même jeu que la machine hôte a des caractéristiques différentes de celle qui doit être traduite d'un jeu A à un jeu B. En général, on obtient une performance supérieure dans le premier cas puisque chaque instruction est directement exécutée sur le processeur hôte.

1.2.2 Particularité de l'architecture *x86*

Pour multiplexer de manière contrôlée et sécuritaire un système *x86* en machines virtuelles, on cherchera à mettre le VMM en plein contrôle des ressources de la machine pour qu'il puisse les partager tout en gérant les accès. Il devra donc s'exécuter au *ring* 0 et lancer des systèmes hébergés à d'autres niveaux d'exécutions plus élevés. Or, ces systèmes hébergés s'attendent à être eux-mêmes en plein contrôle de la machine (donc au *ring* 0 — comme Linux dans le cas général, cf. section 1.1). Idéalement, il faudra donc que le VMM puisse simuler parfaitement le comportement d'une machine physique pour que puissent fonctionner, non-modifiés, ces systèmes d'exploitation.

Or, l'architecture *x86* n'a pas été conçue initialement pour permettre la virtualisation et son jeu d'instruction pose problème. À la base, cette architecture ne satisfait pas le théorème 1 de POPEK et GOLDBERG (1974), énoncé ainsi :

> *For any conventional third generation computer, a VMM may be constructed if the set of sensitive instructions for that computer is a subset of*

the set of privileged instructions.

Cela ne signifie pourtant pas qu'il soit impossible de créer un VMM pour cette architecture, car la condition n'est pas nécessaire et suffisante pour la virtualisation. Il faut alors traiter les cas particuliers. Selon ROBIN et IRVINE (2000), on dénombre 17 instructions *non-privilégiées mais sensibles* dans le jeu d'instruction *x86*[4].

Ces 17 instructions sont problématiques car elles ne satisfont pas les critères 3B et 3C énoncés dans le travail mentionné :

> **Requirement 3** *There must be a way to automatically signal the VMM when a VM attempts to execute a sensitive instruction. It must also be possible for the VMM to simulate the effect of the instruction.*
>
> *Sensitive instructions include :*
>
> *[...]*
>
> **Requirement 3B** *Instructions that read or change sensitive registers and/or memory locations such as a clock register and interrupt registers.*
>
> **Requirement 3C** *Instructions that reference the storage protection system, memory system, or address relocation system. This class includes instructions that would allow the VM to access any location not in its virtual memory.*

Même si l'ISA du processeur pose problème à plusieurs niveaux, nous verrons, en étudiant deux implémentations particulières, qu'il existe certaines techniques

[4] À noter que cette étude portait sur le Intel Pentium de première génération.

permettant tout de même d'offrir des solutions efficaces de virtualisation sur *x86*.

1.2.3 Implémentations de systèmes de virtualisation

Cette section présente différents logiciels permettant la virtualisation, plus particulièrement la *virtualisation complète* (selon T^{VS}), puisque l'objectif de notre travail est de déployer des systèmes d'exploitation entiers, du noyau aux applications.

1.2.3.1 *Xen*

Xen est un système logiciel d'envergure décrit dans les travaux de BARHAM et al. (2003); PRATT et al. (2005, 2006) et est la base de nombreuses recherches récentes sur la virtualisation. Nous survolons certaines de ses caractéristiques dans cette section.

Xen utilise une approche de *paravirtualisation*. Cette approche nécessite l'insertion d'une nouvelle couche logicielle de bas niveau (l'*hyperviseur Xen*) qui deviendra l'interface entre un noyau modifié et le système physique (VMM Type I, selon T^{RI}).

La documentation de *Xen* utilise une terminologie particulière pour distinguer les systèmes virtualisés. On nomme *domaine 0* ou *dom0* le système d'exploitation chargé de contrôler l'accès au matériel. Il y a toujours un domaine zéro par machine physique, qu'on peut utiliser comme système d'exploitation « normal ». Ce domaine est l'aiguilleur des requêtes faites par les autres systèmes d'exploitation en *domaine U* (ou *domU*), s'il y a lieu. Ces derniers sont les systèmes virtualisés lancés sur demande.

Deux noyaux, en plus de l'hyperviseur, sont nécessaires : un premier sera compilé pour gérer le matériel physique et tournera en domaine 0. Un second doit être compilé pour exporter des périphériques virtualisés (dits de *frontend*) à un système invité. Ce noyau « invité » tournera en domaine U. *Xen* se charge de connecter les périphériques virtuels en domaine U aux vrais périphériques gérés par le noyau du domaine 0. Les périphériques de *backend* disponibles sur le domaine 0 sont alors connectés aux différents *frontends* des domaines U et gèrent les requêtes vers les vrais périphériques, à travers les pilotes natifs disponibles dans le noyau *Linux*.

Un noyau invité en domU peut ainsi avoir accès à des périphériques réseau ou par blocs (disques) virtualisés en accédant au matériel physique à travers les interfaces relativement stables et constantes des *backends*. Ce noyau peut donc être très générique. Quant à la manière d'accéder aux périphériques par les applications de domaines U, elle reste aussi constante d'un système invité à un autre, en passant par le noyau et ses périphériques de *frontends* de *Xen*.

La figure 1.2 schématise un système contenant un domaine 0 et deux domaines U et les interconnexions entre les *frontends* et *backends*.

L'utilisation de *Xen* sur *x86* demande à ce que les noyaux de domaine 0 et domaine U s'exécutent au *ring* 1, alors que la couche supplémentaire de paravirtualisation s'exécute au ring 0 pour pouvoir contrôler les requêtes privilégiées des systèmes d'exploitation hébergés et les interruptions et événements matériels. On doit modifier (ou « porter ») les noyaux pour qu'ils fonctionnent sur l'architecture *Xen* exportée par l'hyperviseur. Le noyau utilisera alors des appels à l'hyperviseur pour compléter plusieurs fonctions d'administration. Cela permet à un noyau tournant en *ring* > 0 de compléter des tâches de gestion normalement privilégiées et limitées au *ring* 0, tout en permettant un contrôle par l'hyperviseur des ressources allouées et des accès aux périphériques. Cette manière de procéder permet de contourner

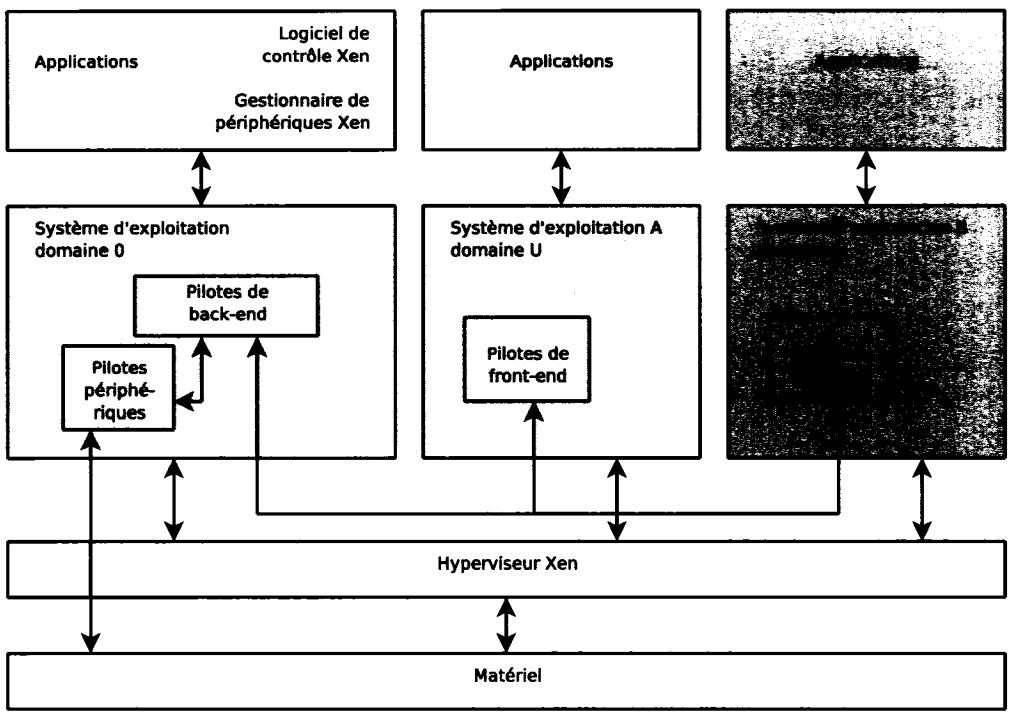

Figure 1.2 Un VMM *Xen*, un domaine 0 et deux systèmes invités en domaines U, schéma inspiré de PRATT (2005)

les problématiques associées à l'architecture *x86* et d'assurer l'intégrité du système. Or, on doit être en mesure de faire cet effort de portage : il faut avoir accès au code source, ce qui peut être problématique dans le cas de certains systèmes d'exploitation propriétaires.

Bien qu'il soit nécessaire de compiler des noyaux particuliers pour utilisation en domaine 0 ou domaine U, *il n'est pas nécessaire de modifier ou recompiler les applications.* L'interface des appels systèmes standards *Linux* reste disponible tant sur les noyaux de domaine 0 que de domaine U.

L'approche préconisée par *Xen*, selon certaines études de performance (BARHAM et al., 2003; CLARK et al., 2004), offre de très bonnes performances en virtualisation sur l'architecture *x86*. Selon ces travaux, on arrive à obtenir des performances quasi-natives pour la plupart des domaines applicatifs (des bancs d'essais de performance en accès disque, en accès mémoire, en utilisation de bases de données et serveurs Web ainsi qu'en calcul brut sont utilisés dans ces travaux). Le partage des ressources sur plusieurs machines virtuelles concurrentes sur une même machine physique est également étudié dans ces travaux. On y montre en pratique une division relativement équitable des ressources.

L'approche est très invasive (T^{VS}), mais offre cependant une efficacité élevée.

Xen offre actuellement, en plus des caractéristiques inhérentes aux systèmes de virtualisations complets, certaines caractéristiques d'intérêt :

- sérialisation d'un système (*snapshotting*) ;
- arrêt et reprise de l'exécution d'une machine virtuelle ;
- migration à chaud d'une VM d'un hôte à un autre avec un minimum de temps d'arrêt (60 ms, (CLARK et al., 2005)).

Xen a d'abord été conçu pour l'architecture *x86*. Le logiciel est maintenant disponible pour *x86_64* et des développements récents promettent un support complet des architectures *IBM Power* et *Intel Itanium*.

1.2.3.2 *QEMU*

QEMU (BELLARD, 2005) est un interpréteur de code natif qui traduit les instructions machines du système invité en routines logicielles à exécuter sur le système hôte et permettant optionnellement d'exécuter sans traduction une partie du code invité directement sur le matériel à travers un module « accélérateur ». Il s'agit d'un système dont l'invasivité peut être minimale (T^{VS}) puisque tout peut être exécuté dans l'espace utilisateur, mais permettant tout de même la virtualisation complète (T^{VS}) et de l'architecture (T^{VS}). *QEMU* permet également faire office de traducteur dynamique pour VM processus (T^{SN}). La figure 1.3 montre *QEMU* fonctionnant dans un système d'exploitation hôte, et émulant deux systèmes complets invités.

QEMU peut fonctionner sur plusieurs architectures (cf. figure 1.1) et, sur chacune d'elles, émuler plusieurs architectures (cf. tableau 1.2).

QEMU utilise une approche de compilation dynamique du code invité. Lorsque *QEMU* lit le texte d'un programme pour une première fois, il le convertit dynamiquement en code pour le jeu d'instructions de la machine hôte et conserve en copie cette traduction pour réutilisation future. Cette conversion utilise des techniques de programmation avancées, décrites dans BELLARD (2005), pour minimiser le travail mis à traduire et pour accélérer les cas communs de traitement. L'approche de *QEMU* lui permet d'être généralement plus performant que les autres CSIM (T^{RI}) élémentaires tout en conservant la possibilité de virtualiser de nombreuses

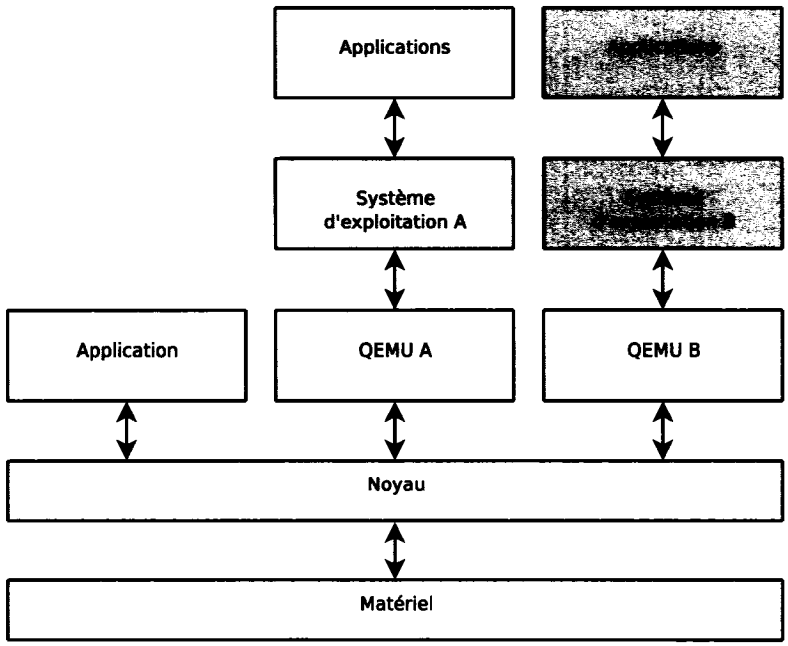

Figure 1.3 *QEMU* sur un système hôte, faisant fonctionner deux systèmes invités

Tableau 1.1 Architectures sur lesquelles *QEMU* peut fonctionner

Processeur hôte	Statut
x86	Supporté
x86_64	Supporté
PowerPC	Supporté
Alpha	En phase de test
Sparc32	En phase de test
ARM	En phase de test
S390	En phase de test
SPARC64	En phase de développement
ia64	En phase de développement
m68k	En phase de développement

Tableau 1.2 Architectures que *QEMU* peut simuler

Processeur invité	Émulation système
x86	Supporté
x86_64	Supporté
ARM	Supporté
SPARC	Supporté
SPARC64	En phase de développement
PowerPC	Supporté
PowerPC64	En phase de développement
MIPS	Supporté
m68k	En phase de développement
SH-4	En phase de développement

architectures. Il faut tout de même procéder à l'analyse et traduction vers le jeu d'instruction hôte, ce qui engendre invariablement une perte de performance. À proprement parler, utiliser seulement cette technique ferait de *QEMU* un *interpréteur logiciel* ou *émulateur*, qu'on distingue des systèmes de *virtualisation* puisque 0% du code invité est exécuté directement sur le matériel (cf. critère de GOLDBERG (1972)).

Un module noyau accélérateur, `kqemu` (BELLARD, 2007), est également disponible pour *Linux* (des versions expérimentales sont disponibles pour *FreeBSD* et pour *Windows*), sur les architectures *x86* et *x86_64*. Ce module s'insère dans le kernel du système hôte et permet d'exécuter du code venant du système hébergé directement sur la plate-forme hôte, en laissant le soin à *QEMU* de gérer les appels systèmes avec une approche hybride (T^{RI}). On revient donc au premier type de niveau de communication de T^{VS}, et, conséquemment, on reste alors contraint à exécuter du code compilé pour l'architecture de la machine hôte. *QEMU* permet donc de virtualiser/émuler 2 types finaux de VMs systèmes dans T^{SN} : VMs hébergés et VMs de systèmes complets.

1.2.3.3 Autres solutions

Plusieurs autres solutions sont également disponibles. Une liste exhaustive est faite par SINGH (2007). WRIGHT (2004) fait également un survol de différentes implémentations fonctionnant sous *Linux*.

1.3 Systèmes d'exploitation pour grappes de calculs

Nous cherchons à développer une méthode de distribution permettant de démarrer des nœuds potentiellement distants. Il convient d'étudier les approches déjà développées, principalement dans le monde des grappes de calculs, premier client de tels systèmes de déploiement.

Plusieurs méthodes sont utilisées quant à la distribution de systèmes d'exploitation sur des grappes de calculs de type Beowulf (BECKER et al., 1995). Deux approches sont couramment employées, soit l'approche complètement distribuée du système où chaque ordinateur a le contrôle complet de ses ressources, soit l'approche visant à représenter un ensemble de systèmes comme un seul système (*Single System Image*, SSI). À une extrémité du spectre, on traite de systèmes indépendants reliés, alors qu'à l'autre on cherche à unifier toutes les ressources d'un parc informatique en une seule entitée directement exploitable par les usagers.

1.3.1 Approches classiques

Dans le cas classique, on exploite plusieurs ordinateurs configurés « normalement ». On les configure de manière traditionnelle ou via des outils automatiques simplifiant la création des configurations et images de système d'exploitation à distribuer sur

les postes faisant partie de la grappe.

Différentes approches ont été développées depuis les débuts du Beowulf. Certaines distributions spécialisées offrent un ensemble d'outils pour simplifier la mise en place de telles grappes. Notamment, *OSCAR* (DES LIGNERIS et al., 2003; OSCAR, 2007) fournit des outils fonctionnant sur différentes distributions *Linux* de la famille *Red Hat*, alors que *ROCKS* (PAPADOPOULOS et al., 2003) est une distribution complète basée sur *Red Hat Linux* et contenant sa trousse d'outils de déploiement et gestion.

Une autre approche cherche à centraliser la gestion des systèmes de fichiers des nœuds sur un seul serveur. On installe sur ce dernier un système complet dans un sous-répertoire isolé, puis on exporte ce répertoire en tant que racine en accès complet pour un nœud. Le nœud démarre en montant sa racine directement du serveur, allégeant ainsi les nœuds, maintenant sans disque (méthode *diskless*, (DES LIGNERIS et al., 2004; ANDREWS et al., 2007)). Cette approche est rendue possible grâce au support *root over nfs* du noyau *Linux* (KUHLMANN et MARES, 1997).

La littérature liste les avantages indéniables découlant de l'approche *diskless* :

- réduction des coûts (pas besoin de disques) ;
- réduction de la probabilité de failles (les disques durs ont un taux de faute important par rapport aux autres composants, étant basés sur des systèmes mécaniques) ;
- administration des systèmes d'exploitation des nœuds en un seul point.

1.3.2 Systèmes à image unique

Cette approche cherche à abstraire le nombre de systèmes physiques dans une grappe en uniformisant l'espace des processus et éventuellement une partie de l'espace de stockage. Un processus sera unique parmi tous les processus de tous les nœuds d'une grappe et sera accessible à partir de chaque ordinateur.

Des boîtes d'outils logiciels permettent ensuite de gérer l'exécution des processus sur la grappe. La distribution *SCYLD* (PENGUINCOMPUTING, 2007), les outils Mosix (BARAK et al., 1999), OpenMosix (OPENMOSIX, 2007), le projet *OpenSSI* (WALKER, 2003), *Kerrhiged* (KERRHIGED, 2007) et *BProc* (BPROC, 2006) en sont les principaux exemples.

1.3.3 Systèmes à racine unique

Sans partager l'espace de processus sur toute une grappe, les systèmes à racine unique partagent un système de fichiers presque sans modifications (un système *purement* SFI étant à toutes fins pratiques impossible, ne serait-ce que parce qu'une configuration réseau doit être différente d'un poste connecté à un autre). On peut évidemment voir un parallèle avec l'approche *diskless*.

En termes pratiques, le SFI se distingue du *diskless* en montant directement par NFS (ou autre système de fichiers réseau) la *racine du serveur comme sa racine.*

Le système *Adelie/SSI*[5] (MORIN, 2006) est une solution de déploiement de nœuds en SFI se basant sur la distribution *Gentoo GNU/Linux*. Elle est utilisée principa-

[5]Il serait plus judicieux de l'appeler *Adelie/SFI*, car il se base sur le partage, à quelques différences près, d'une même racine de système de fichiers entre les membres d'une grappe – l'espace de processus est cependant distinct d'un nœud à un autre, caractéristique d'un SSI.

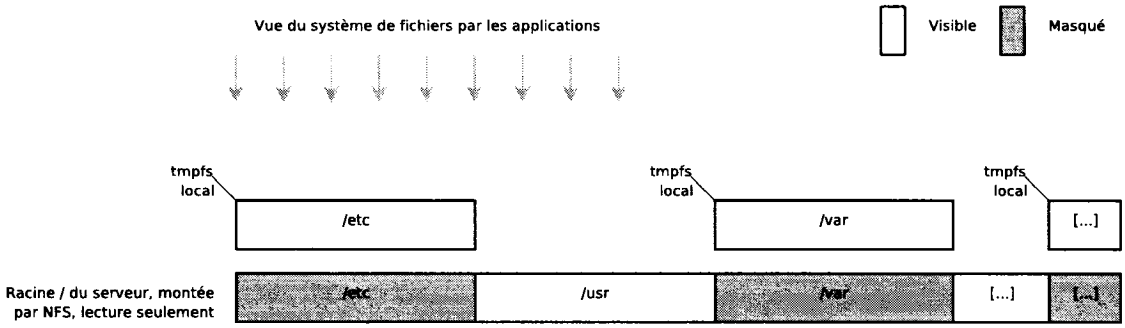

Figure 1.4 *Adelie/SSI* monte des systèmes de fichiers `tmpfs` locaux sur les nœuds, masquant les répertoires du serveur, dont la racine est montée par NFS

lement pour mettre en œuvre de manière simple des réseaux de stations de travail et des grappes de calcul. Son déploiement sur plusieurs sites académiques[6] et industriels ont démontré sa flexibilité et sa performance pour des grappes relativement petites, de l'ordre de 4 à 64 nœuds.

Adelie/SSI découle de l'idée de déplacer, au démarrage, des fichiers de configuration qui servent à différencier les services selon le rôle que joue une machine physique. Au début du processus d'initialisation d'un nœud, on copie en mémoire `tmpfs`[7] le contenu de certains répertoires clefs, tels `/etc`, puis des fichiers de configuration sont remplacés par ceux qui conviennent à la machine courante. Par exemple, on pourra remplacer `/etc/service.conf` par `/etc/service.conf.noeud1` pour la machine qui joue le rôle de nœud 1. Enfin, on déplace ces points de montage par-dessus les répertoires du serveur correspondant. On obtient alors une configuration locale « personnalisée » pour le rôle du nœud. La figure 1.4 illustre ce remplacement des répertoires permettant de distinguer les nœuds les uns des autres, et du serveur.

[6]Notamment au laboratoire de Design et Réalisation d'Applications Parallèles à l'École Polytechnique de Montréal, `http://polymtl.ca/drap`.

[7]`tmpfs` est un système de fichiers où tout est stocké en mémoire vive ou en *swap*. L'espace occupé par ce système de fichiers varie selon la quantité d'information qui y est stockée. Le contenu disparaît lors de la mise hors tension d'un ordinateur.

Cette solution modifie la séquence d'initialisation démarrée par `/sbin/init` : il faut altérer les niveaux d'exécution (*runlevels*) pour y insérer un script qui se chargera très tôt d'effectuer le montage des systèmes de fichiers en mémoire et le remplacement des fichiers de configuration par ceux adaptés au démarrage du système donné.

Les fichiers de configuration doivent être générés à l'avance directement sur la machine faisant office de serveur. Dans l'implémentation la plus récente de *Adelie/SSI*, ils sont générés par traitement programmatique sur les fichiers de configuration du serveur, pour créer, par exemple, le fichier `service.conf.noeud01` en traitant le contenu du fichier de configuration standard `service.conf` par des scripts Bash (FSFBASH, 2007).

1.4 Déploiement réseau

Les grappes de calculs strictement physiques — par opposition à des grappes dont les nœuds sont virtuels — sont soit démarrées comme le sont des ordinateurs traditionnels, c'est-à-dire en amorçant un système d'exploitation résidant sur le disque principal, ou encore via des technologies de démarrage réseau comme *Preboot Execution Environment* (INTEL, 1999) ou Etherboot (ETHERBOOT, 2007) couplées au *Trivial File Transfer Protocol* (SOLLINS, 1992; FINLAYSON, 1984) et à BOOTP ou DHCP (CROFT et GILMORE, 1985; REYNOLDS, 1993), DHCP (DROMS, 1993) pour la transmission de l'image de système d'exploitation et de ses paramètres de démarrage à des ordinateurs nœuds. Le processus démarrage est donc *instancié par le démarrage physique du nœud.*

Dans le contexte de grappes virtualisées ou de services de grilles, le processus de démarrage est *instancié logiciellement par l'usager.* Ce dernier est le point de départ

de requêtes de démarrage de nœuds qui sont envoyées vers un ensemble de services d'exécution qui pourront démarrer des machines virtuelles à la demande.

Pour communiquer avec ces serveurs, on a développé certains protocoles de déploiement particuliers, ou faisant appel à des interfaces et standards de services Web articulés autour de XML-RPC (*XenoServer Platform*) ou une combinaison de SOAP et WSDL (*Globus Toolkit*). D'autres systèmes sont déployés en utilisant une combinaison de technologies plus hétéroclite, comme *Cluster-on-Demand* (CHASE et al., 2003), utilisant DNS, LDAP, PXE, NIS.

1.4.1 Déploiement à configuration minimale

Le travail de GORM HANSEN et JUL (2005) préconise la réutilisation de la configuration logicielle locale dans des machines virtuelles distantes par réplication de systèmes d'exploitation.

Un usager met en place localement, sur son poste, l'ensemble des logiciels nécessaires et enclenche ensuite un processus de migration où son système d'exploitation complet sera envoyé pour traitement sur une grille où le système d'exploitation sera virtualisé.

Ce travail contribue à définir le concept nommé « *Minimal Configuration Grid* ». En premier lieu, nul besoin de configurer de systèmes d'exploitation sur la grille (on se contente de supporter les systèmes d'exploitation envoyés par les usagers) et, en second lieu, on permet à l'usager de réutiliser la configuration locale de son poste pour minimiser la quantité d'effort requise.

Ce concept de configuration minimale, même s'il est appliqué différemment, rejoint une idée fondamentale du SFI : minimiser l'effort de configuration. Dans les deux

cas, on cherche à réutiliser au maximum un système d'exploitation déjà configuré. Ces auteurs montrent en plus que la virtualisation est un substrat idéal pour niveler les différences physiques entre les infrastructures, réduisant encore l'effort de configuration.

1.5 Conclusion de la revue des concepts

Ce chapitre a servi à présenter plusieurs concepts utiles dans le contexte du développement d'un méthode de déploiement de nœuds virtuels légers. Nous avons d'abord étudié la virtualisation sur *x86* et les avantages qu'elle peut procurer, puis nous avons fait une révision de diverses approches quant à la gestion de grappes et au déploiement de systèmes d'exploitation sur des machines physiques ou virtuelles distantes.

En gardant en tête les idées développées dans la littérature, nous proposons au chapitre suivant une architecture reprenant plusieurs éléments présentés.

CHAPITRE 2

COMPOSANTS ARCHITECTURAUX

Le chapitre précédent offrait une revue de différentes approches à la virtualisation et au déploiement de systèmes. Maintenant qu'il est possible de générer des parcs informatiques virtuels, la problématique du déploiement de grappes ou grilles doit être révisée pour inclure la possibilité de démarrer des systèmes entièrement contenus dans des machines virtuelles.

Maintenant plus que jamais, il est envisageable de multiplexer des ensembles d'ordinateurs physiques en ensembles d'ordinateurs virtuels, permettant une philosophie nouvelle du partage des ressources. On peut dorénavant penser à exploiter toutes ces machines utilisées à un faible pourcentage de leurs capacités, sans pour autant les transformer radicalement. La virtualisation promet de laisser en place le rôle primaire de machines physiques, mais de permettre de démarrer en leur sein des systèmes d'exploitation spécialisés et utilisés pour des contextes variés. Pour y distribuer des calculs, pour y lancer des rendus parallèles d'envergure, pour y traiter des problèmes distribués, le réseau de nœuds virtuels permet d'obtenir du temps de calcul supplémentaire à faible coût et à la demande.

Pour réaliser une telle exploitation des parcs informatiques, encore faut-il savoir déployer efficacement des systèmes d'exploitation. Dans ce domaine, on a beaucoup à apprendre des systèmes de gestion de grappes de calculs. C'est d'ailleurs en se basant sur ces systèmes que nous proposons une méthode de génération rapide de *clones* de systèmes déjà fonctionnels que l'on pourra démarrer dans des silos virtuels.

Aux travaux de gestion de grappes et grilles présentés plus tôt, nous joignons une technique de différentiation par superposition de systèmes de fichiers basée sur *UnionFS*. Pourtant résolument dans la lignée des SFI, notre tactique diverge de celle de la solution *Adelie/SSI* en s'appuyant sur une modification architecturale apportée au système *early-userspace* généré par l'outil *Genkernel*. Une proposition d'amélioration de la flexibilité du processus de mise en place de la racine d'un système d'exploitation est fournie.

Ce chapitre présente en outre la méthode de déploiement de nœuds virtuels légers que nous proposons. Il y est question de la stratégie de communication et des problématiques de découverte et utilisation des ressources, ainsi que des considérations visant à assurer une bonne flexibilité et un niveau d'abstraction suffisant pour en faire une solution générique au problème du déploiement de machines virtuelles.

2.1 Système d'exploitation *Gentoo GNU/Linux*

Avant tout, une étude rapide du système d'exploitation sur lequel nous développerons notre implémentation s'impose.

Alors que la plupart des systèmes d'exploitation offrent des paquetages binaires optimisés pour le cas général, *Gentoo GNU/Linux* se démarque en offrant un niveau d'optimisation et de configurabilité élevé sans pour autant sacrifier la flexibilité et la simplicité d'utilisation.

Cette distribution est particulièrement adaptée au système que nous concevons. Puisque tous les paquetages sont installés en compilant leur code source, il est possible de choisir précisémment les optimisations nécessaires pour le matériel choisi et d'installer ou non certaines caractéristiques optionnelles de plusieurs logiciels.

Deux aspects importants de la distribution *Gentoo GNU/Linux* la séparant de la majorité des autres offres actuelles sont le *Portage Tree* et ses outils reliés ainsi que le système d'initialisation. Le survol de ces aspects permettra de mieux comprendre le développement de notre travail. Pour plus d'information au sujet de ces aspects, GAVIN et al. (2007) offre une source d'information compréhensive.

2.1.1 Portage

Gentoo Portage est un gestionnaire de paquetages dont l'objectif est d'abord de permettre d'installer et de maintenir un ensemble de logiciels choisi par l'administrateur d'un système informatique. Il se compare donc aux outils tels le *Red Hat Package Manager* (RPM) (REDHAT, 2007) et la suite de gestion des paquetages `.deb` (DEBIAN, 2007) sur la distribution *Debian* et ses dérivées. Tout comme les outils *Debian*, *Portage* intègre à ses fonctionnalités l'analyse et la résolution de dépendances entre les paquetages à installer.

La notion de paquetage sous *Portage* se distingue cependant de celle définie par les outils mentionnés plus haut. *Portage* contient une arborescence organisée en catégories d'applications. Chacune de ces catégories contient à son tour un ensemble de logiciels et leurs scripts d'installation, appelés *ebuilds*. Ces *ebuilds*, qu'on peut rapprocher de la notion de « paquetages », sont en fait des scripts d'installation qui contiennent les instructions nécessaires pour télécharger le code source d'un logiciel, le configurer et l'installer. Toute cette hiérarchie se nomme le *Portage Tree*, en référence à « *Ports Collection* », ancêtre de *Portage* bien connu des utilisateurs de *FreeBSD*. Par exemple, le listage 2.1 offre une vue partielle de la hiérarchie du *Portage Tree*.

Les fichiers `.ebuild` sont des scripts écrits dans le langage Bash et certaines tech-

Listage 2.1 Quelques catégories et *ebuilds*. On y voit 4 catégories et un exemple de paquetage : OpenOffice. Il est possible d'installer deux versions différentes grâce aux deux *ebuilds*. Différentes rustines sont distribuées avec les *ebuilds*, pour adapter le logiciel à l'environnement *Gentoo GNU/Linux* et apporter des correctifs lors de l'installation

```
/usr/portage/
|-- [...]
|-- sys-devel
|   `-- [...]
|-- x11-base
|   `-- [...]
|-- media-video
|   `-- [...]
`-- app-office
    |-- [...]
    `-- openoffice
        |-- ChangeLog
        |-- Manifest
        |-- files
        |   |-- 2.0.4
        |   |   |-- disable-regcomp-java.diff
        |   |   |-- disable-regcomp-python.diff
        |   |   |-- gentoo-2.0.4.diff
        |   |   `-- regcompapply.diff
        |   |-- 2.1.0
        |   |   |-- disable-regcomp-java.diff
        |   |   |-- disable-regcomp-python.diff
        |   |   |-- gentoo-2.1.0.diff
        |   |   |-- ooo-wrapper.in
        |   |   |-- regcompapply.diff
        |   |   `-- wrapper-readd.diff
        |   |-- digest-openoffice-2.0.4
        |   `-- digest-openoffice-2.1.0
        |-- metadata.xml
        |-- openoffice-2.0.4.ebuild
        `-- openoffice-2.1.0.ebuild
```

niques avancées de scriptage en font des héritiers de différentes classes de *ebuilds*, appelées *eclasses*. Un détour dans `/usr/portage/eclasses` sur un système *Gentoo* listera ces classes.

Il est également possible de modifier les paramètres d'installation d'un paquetage donné en réglant la variable d'environnement `USE`. Les mots-clefs contenus dans cette variable sont appelés *use flags*. Par exemple, si on règle les *use flags* à

```
USE="objc objc++ fortran"
```

alors, lors de l'installation de GCC, le support pour ces langages sera intégré. À l'inverse, si nous avons plutôt

```
USE="-objc -objc++ -fortran"
```

alors GCC ne contiendra pas le support pour ces langages. Le mécanisme des *use flags* rend la distribution très attrayante pour l'administrateur cherchant à configurer un système selon des besoins bien précis.

Plusieurs outils permettent d'obtenir des détails sur les paquetages, les *use flags*, leur installation, leurs dépendances, etc. Le principal outil permettant d'installer ou retirer des paquetages est `emerge`.

L'intérêt de l'approche basée sur la compilation des sources est aussi la possibilité de compiler les paquetages avec des optimisations de compilation variables, permettant de tenir compte d'un maximum de caractéristiques du matériel utilisé sur une machine ou encore de choisir un dénominateur commun lorsqu'on partage une racine (et donc des librairies et binaires exécutables) : on verra alors à optimiser pour le système le moins performant sur lequel un système démarrera. Bien que ce

nivellement par la base puisse être peu attrayant à l'échelle de parcs informatiques, nous croyons, par expérience, que de nombreux parcs offrent justement une base architecturale performante, faisant de *Gentoo GNU/Linux* un choix à considérer pour en tirer le maximum.

2.1.2 Système d'initialisation

La distribution *Gentoo GNU/Linux* offre un système d'initialisation par *runlevels* nommés et par analyse de dépendances entre les scripts d'initialisation (GAVIN et al., 2007) grandement inspiré par la refonte du système d'initialisation de *NetBSD* (MEWBURN, 2001) alors que, typiquement, les systèmes UNIX et *Linux* utilisent le système développé sur *System V*, qui offre un nombre limité de niveaux (en général, 5 ou 6).

Cet aspect de *Gentoo GNU/Linux* offre la flexibilité nécessaire pour l'établissement de rôles variés selon le *runlevel* choisi. Dans un contexte de système à racine unique, une grande partie de la différentiation entre les différents systèmes qui démarrent de la même racine est reliée aux services démarrés. On peut donc donner des rôles distincts à certains nœuds en choisissant le *runlevel* nommé dans lequel ils démarrent.

Par défaut, Gentoo démarre dans le *runlevel* `boot`, puis enchaîne au niveau `default`. Un démarrage de grappe hétérogène en SFI peut se faire par exemple avec un *runlevel* `master-default` et un autre `slave-type1-default`.

2.2 Clones à racine unique

Cette citation de SILVER (2001) décrit assez bien l'étymologie du terme « clone » :

> *Herbert J. Webber coined the word 'clone' in 1903 to describe a colony of organisms derived asexually from a single progenitor.*

Pour ce travail, dont le contexte est éloigné de la biologie, des *clones* sont des systèmes qui partagent la même racine de système de fichiers. Ce sont des systèmes qui démarrent littéralement à partir de la même racine qu'un système de référence : des systèmes SFI.

Démarrer des clones en SFI offre ces propriétés désirables[1] :

- *aucun stockage local* sur les nœuds (pas besoin de disque, ce qui se traduit en coûts réduits) ;
- *réutilisation maximale de la configuration* déjà effectuée sur un serveur (efforts réduits) ;
- *point d'administration central* ;
- approche *pull-based* : seulement les données positivement utiles au nœud seront transférées lorsqu'il leur accédera sur le système de fichiers réseau[2] ;
- *propagation instantanée* : toute modification effectuée sur le système de fichiers racine partagé est instantanémment disponible sur les clones (en dehors des modifications locales au système de fichiers d'un clone, qu'on voudra minimiser) ;
- *déploiement léger* : déployer un nœud demande de le démarrer et de lui faire monter la racine du serveur. Cela se traduit dans le cas général par un transfert initial minimal (chargeur d'amorce, noyau, potentiellement un *initramfs*).

[1] On remarquera certaines similitudes avec l'approche *diskless*, présentée à la section 1.3.1.

[2] Sans mécanisme de cache, ces accès peuvent être répétés parfois plusieurs fois, en faisant aussi une propriété non-désirable.

De manière à déployer des clones d'un système d'exploitation déjà configuré, il faut impérativement pouvoir différencier un clone lors de son démarrage en modifiant le système de fichiers racine à certains emplacements clefs. Typiquement, il faut présenter la racine unique au clone avec des modifications locales au contenu de son système de fichiers, principalement au niveau des fichiers de configuration.

À ce propos, l'approche SFI de *Adelie/SSI* (cf. section 1.3.3) a la faiblesse de limiter la propriété de propagation instantanée, car elle doit rendre opaque les modifications faites sur les répertoires contenant ces configurations personnalisées, qui sont stockés en `tmpfs` (cf. figure 1.4).

De plus, cette approche est éphémère : la configuration locale d'un nœud n'existe que lors de son déploiement et disparaît lorsque la machine déployée est arrêtée : la copie dans un système de fichiers local en mémoire pour les répertoires clefs est en soi volatile puisque stockée en mémoire vive. Cette solution présente aussi une flexibilité restreinte car les répertoires non-clefs (hors ceux qui contiennent des variations de configuration) qui sont partagés par le serveur vers les nœuds sont accessibles en lecture seulement, rendant impossible la modification locale. Plusieurs scénarios d'utilisation ne nécessitent toutefois pas cette flexibilité, on en convient.

Nous amenons une variation en approchant le problème autrement : nous différencions les nœuds par superposition de systèmes de fichiers grâce à *UnionFS*, ce qui amène la mémorisation de l'état des nœuds, l'accès en lecture-écriture sur tout le système de fichier, tout en minimisant le problème d'opacité propre à *Adelie/SSI*. Tout cela, en plus des autres propriétés des SFI mentionnées plus haut.

2.2.1 Superposition de systèmes de fichiers

Cette section traite de la méthode utilisée pour différencier les systèmes clonés du système parent original. On y discute de *UnionFS*, un ensemble logiciel qui permet de superposer des couches de systèmes de fichiers sur *Linux*.

2.2.1.1 Introduction à *UnionFS*

Alors que des systèmes d'exploitation tels que *Plan 9* (PLAN9, 1992) et les variantes de BSD (PENDRY et MCKUSICK, 1995) offrent depuis relativement longtemps un système comparable, *Linux* a accès à la possibilité de superposer ses systèmes de fichiers de façon relativement stable que depuis peu, grâce à *UnionFS* (UNIONFS, 2007), développé par une équipe de chercheurs de l'université Stony Brook (WRIGHT et al., 2006). Depuis le 9 janvier 2007, *UnionFS* fait partie de la version de développement du noyau *Linux*, dans la branche `-mm` de Andrew Morton, développeur clef au sein de la communauté *Linux*. En conséquence, on suppose une intégration prochaine d'*UnionFS* au noyau standard *Linux*.

Ce logiciel fait en sorte que plusieurs systèmes de fichiers puissent être superposés de manière transparente pour l'utilisateur, qui a accès à un arbre de fichiers suivant toujours la sémantique classique UNIX.

Définissons un fichier comme étant une paire $\{\textbf{nom}, \textbf{périphérique}\}$, où **nom** est un nom complet au sens UNIX (c'est-à-dire le chemin d'accès complet à partir de la racine, par exemple `/etc/fstab`) et où **périphérique** dénote un périphérique à partir duquel on peut accéder à ce fichier, prenons par exemple `/dev/sda1`.

En définissant $n(f)$ comme fonction retournant le nom du fichier f, et pour deux systèmes de fichiers A et B, alors une union de deux systèmes de fichiers au sens

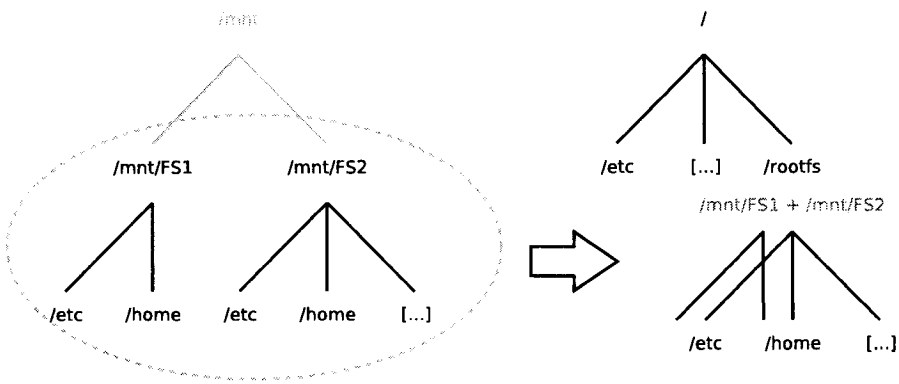

Figure 2.1 Union de branches de l'arborescence du système de fichier : on monte d'abord deux périphériques sur `/mnt`, puis on les insère dans `/rootfs`, un système de fichiers *UnionFS*, qui présentera alors à l'usager le résultat de la superposition

d'*UnionFS* est également un système de fichiers et sera

$$\textbf{union}(A, B) \quad = \quad A \cup \{f_b \in B \mid \forall f_a \in A \quad n(f_b) \neq n(f_a)\}$$

Dans ce cas, on dit que A est la couche supérieure (ou la *plus à gauche* dans certaines publications). La couche A a donc préséance dans les cas où un nom de fichier est disponible dans les deux ensembles en union. L'opération **union** est donc non-commutative, contrairement à son homonyme de la théorie des ensembles.

La figure 2.1 montre un exemple de superposition de branches pour former une union.

Dans l'implémentation d'*UnionFS*, pour simuler la suppression d'un fichier sur la couche supérieure, alors que le fichier est tout de même en pratique disponible sur une couche inférieure et devrait donc apparaître dans l'espace de nommage, on écrit un fichier *whiteout* dans la couche supérieure. Un *whiteout* indique à *UnionFS* de ne pas montrer un fichier donné aux applications.

On peut également utiliser un autre type de *whiteout* pour rendre invisibles tous les fichiers d'un répertoire qui sont disponibles à une couche inférieure (rendre un répertoire *opaque*). Ces fichiers de *whiteout* suivent la convention suivante :

- effacer un fichier `fich` se fait en écrivant un fichier vide `.wh.fich` ;
- rendre un répertoire opaque se fait en écrivant un fichier vide `.wh.__dir_opaque` dans ce répertoire.

UnionFS permet également de limiter les modes d'accès à certaines couches en choisissant l'ordre de superposition et le mode d'accès (*read-only*, *read-write*) des différents systèmes de fichiers participant au système unifié.

Lors d'un accès à un fichier, *UnionFS* retournera un descripteur de fichier pointant en pratique vers le premier fichier rencontré dans le parcours des couches de haut en bas (en tenant compte des fichiers *whiteout*). Les lectures et écritures fonctionnent alors en accédant directement au contenu sur la couche la plus haute. Les écritures, dans le cas où on accède à un fichier en couche inférieure en lecture seulement, se feront en copiant d'abord le fichier de la couche inférieure vers une couche supérieure en lecture-écriture, puis en écrivant dans cette copie. Le descripteur de fichier pointera sur cette copie. On se réfère à cette méthode à l'aide du terme *copyup-on-write* dans la littérature.

Les figures 2.2 et 2.3 montrent comment on accède à des fichiers sur différentes couches et comment s'effectue le *copyup-on-write* lors de l'utilisation de *UnionFS*. Ici, « ro » signifie *read-only*, et « rw » *read-write*, suivant la terminologie UNIX répandue.

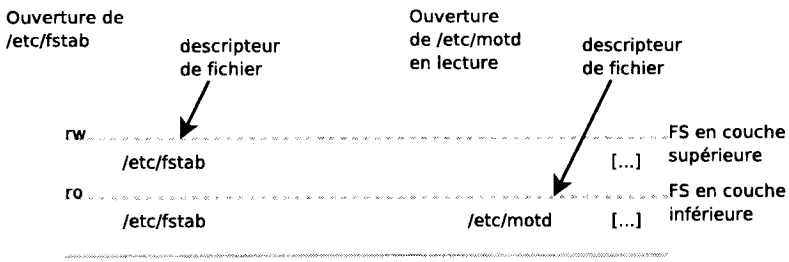

Figure 2.2 Ouverture de fichiers sur un système à deux couches superposées avec *UnionFS*

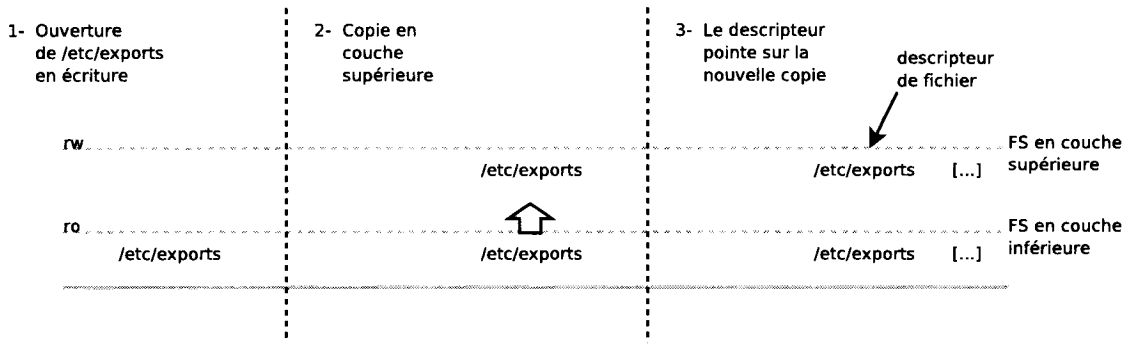

Figure 2.3 Séquence d'opérations lors d'une écriture sur un fichier présent sur une couche inférieure en lecture seulement avec *UnionFS*

2.2.1.2 Couches de différentiation

Sachant qu'il est nécessaire d'avoir des configurations différentes sur chacun des noeuds virtuels par rapport au système cloné de référence, nous développons une technique basée sur *UnionFS*.

Nous différencions les nœuds en leur présentant des fichiers de configuration différents (et toute autre modification locale au système de fichiers étant nécessaire), mais au lieu de remplacer au démarrage de chaque nœud les fichiers dans un système de fichiers en mémoire (`tmpfs`), on peut plutôt exporter deux couches à chacun des nœuds à l'aide d'*UnionFS* : d'abord le système de fichiers de base, puis une surcouche (*overlay*) préparée d'avance qui contient essentiellement les *différences* au niveau système de fichiers entre la configuration recherchée pour le nœud particulier, et celle du système-référence. (Nous verrons à la section 2.2.2 comment notre travail propose de démarrer un système avec une racine *UnionFS* — supposons pour le moment que c'est possible.)

Cette approche permet le déploiement de nœuds clones à mémoire d'état. En effet, il est possible de recouvrir complètement le système de fichiers de base (en lecture seulement) par une couche en accès en lecture et écriture. Lorsque le nœud est modifié, l'état est alors stocké dans cette couche sans toucher au système de base. Comme cette couche supérieure est aussi (potentiellement) montée à partir d'un serveur, il est possible de retrouver la configuration du nœud exactement comme on l'avait laissée à sa mise hors tension lorsqu'on le redémarre.

Nous utiliserons le terme clone ou nœud *à mémoire d'état* (cf. figure 2.4) pour désigner les clones qui sont démarrés sur une union de deux systèmes de fichiers où la couche supérieure est stockée et mémorisée pour usages futurs sur un serveur, par opposition à *sans mémoire d'état* (cf. figure 2.5), qui dénote plutôt un nœud

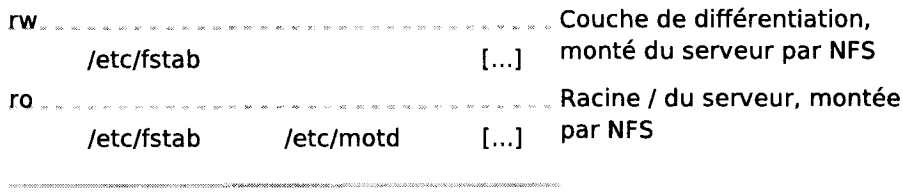

Figure 2.4 Systèmes de fichiers superposés sur un clone à mémoire d'état

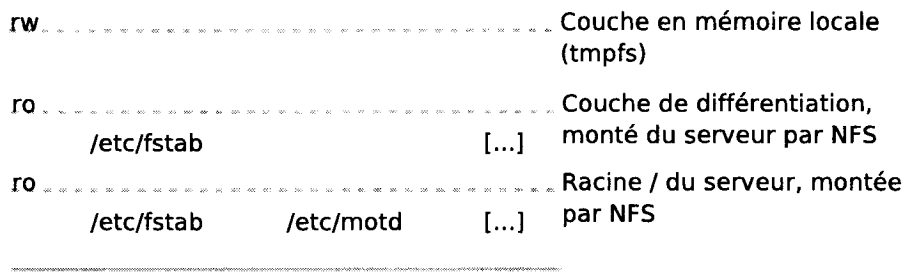

Figure 2.5 Systèmes de fichiers superposés sur un clone sans mémoire d'état

à trois couches superposées pour former la racine : le système de base (sur NFS), la couche des différences (sur NFS) et un système de fichiers en mémoire (`tmpfs`) en lecture-écriture. Lorsque nous parlerons du système qui exporte son système de fichiers racine à des clones, nous utiliserons les termes « serveur » ou système « cloné ».

2.2.2 Processus de démarrage

Cette section explique la mécanique de démarrage d'un système *Linux* 2.6 récent. On y met en contexte le système *early-userspace* et les modifications architecturales proposées pour permettre le démarrage sur une racine-union.

Lorsqu'un ordinateur fonctionnant sous *Linux* démarre, il charge d'abord un amor-

ceur de démarrage, incarné par des logiciels tels que GRUB, LILO, etc. Ce logiciel doit à son tour charger un noyau *Linux* et, possiblement, un fichier contenant un système minimal *early-userspace*.

Ensuite, l'amorceur passe le contrôle au noyau qui est maintenant en mémoire ; ce dernier peut soit démarrer le système en montant directement la racine et en appelant `/sbin/init`, soit décompresser l'image du système minimal en mémoire et lui passer le contrôle via le script `/init` contenu dans cette image. Dans ce dernier cas, le système minimal est chargé de configurer les dispositifs matériels et d'initialiser certains logiciels, puis de passer à son tour le contrôle au système final en ayant pris soin de monter le système de fichiers racine, et en appelant `/sbin/init`.

La figure 2.6 résume la séquence d'événements lors du démarrage jusqu'à la passation du contrôle au système *early-userspace* qui tient sur un fichier au format *initramfs*.

Depuis le noyau *Linux* 2.5.46 (LWN, 2002), il est possible d'utiliser le format d'image mémoire *initramfs* pour stocker un système minimal *early-userspace*. Le terme *initramfs* désigne un *format* de fichier stocké via l'utilitaire `cpio` et compressé par `gzip`. Le terme est également utilisé pour désigner un *fichier* au format *initramfs*. Le nom « *initramfs* » provient du fait qu'il s'agit d'un système de fichiers responsable de l'**init**ialisation qui sera décompressé en **RAM** au démarrage par le noyau. Ce format, étant standard et documenté (VIRO et ANVIN, 2002), permet à différents outils logiciels de créer des systèmes minimaux. Le format *initramfs* remplace l'ancien format de système minimal de démarrage, *initrd* (ALMESBERGER et LERMER, 1996).

Les systèmes minimaux de démarrage ont été créés principalement pour les raisons

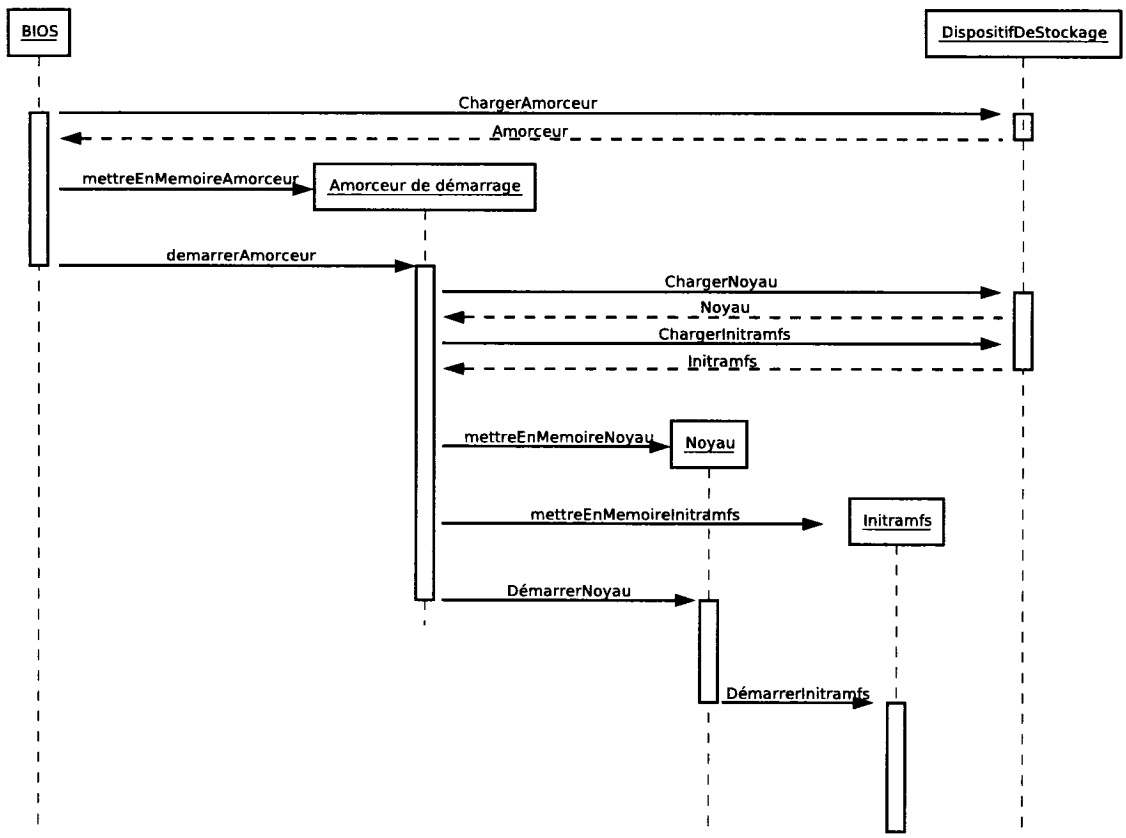

Figure 2.6 Étapes principales du démarrage du système d'exploitation jusqu'à la prise de contrôle par l'*initramfs*. Le chargement de l'*initramfs* peut être fait par le noyau si ce dernier est inclus directement dans le binaire du noyau. Dans un contexte virtuel, certains logiciels de virtualisation se chargeront de mettre le noyau et l'*initramfs* en mémoire et de démarrer le système à partir de ce point

suivantes :

- donner une plus grande flexibilité au processus en permettant de modifier l'environnement de démarrage par des outils externes au noyau *Linux* ;
- sortir les routines d'intialisation du noyau et les placer dans la zone utilisateur (d'où le terme *early-userspace*), permettant de simplifier le code du noyau et ainsi de minimiser le surcoût de maintenance, tout en bénéficiant d'une meilleure sécurité et fiabilité vu les mécanismes de protection inter-processus dans le *userspace*.

D'un point de vue fonctionnel, ils ont pour but de mettre en place la racine du système final : pour ce faire, ils doivent donc monter un ou plusieurs systèmes de fichiers adéquatement pour ensuite passer la main au système final qui se situe sur ces derniers[3].

La figure 2.7 montre un arbre mis en place par le système *early-userspace*, puis le déplacement de la racine vers `/rootfs`, qui, typiquement, contient un système GNU/*Linux* complet. La tâche du système *early-userspace* est justement de mettre en place `/rootfs` à partir des paramètres (par exemple, `root=/dev/hda6`) qui lui sont donnés par l'armorceur de démarrage.

Dans le cas général, un *initramfs* contient une librairie C minimale et des outils de gestion de systèmes et de dispositifs matériels. En utilisant des versions simplifiées de la librairie C (comme *klibc*[4] ou *uCLibC*[5]) et des outils standards UNIX (comme *Busybox*[6]), on obtient des *initramfs* compacts.

---

[3]Exception faite de certaines distributions *Linux*, comme *Minimax*, qui sont entièrement contenues sur un *initramfs*.

[4]`http://www.kernel.org/pub/linux/libs/klibc/`

[5]`http://uclibc.org/`

[6]`http://www.busybox.net/`

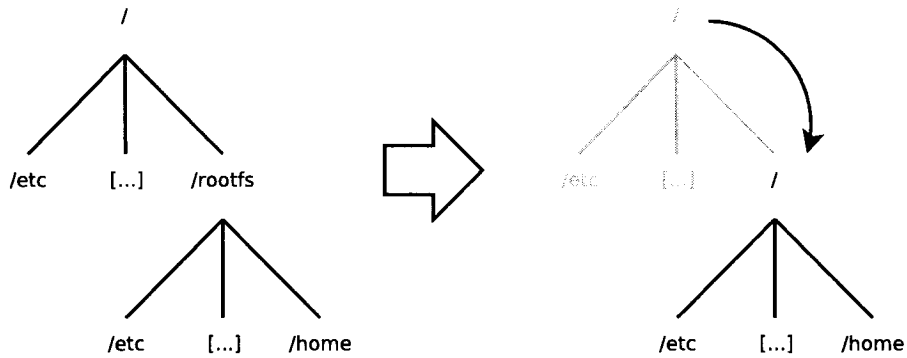

Figure 2.7 Déplacement de la racine du système de fichiers à la fin de l'exécution de l'*initramfs*. Après ce déplacement, il ne sera plus possible de référencer l'ancienne racine et ses autres branches dans la hiérarchie du système de fichiers

Lorsque le noyau passe le contrôle à un *initramfs*, il appelle un point d'entrée fixe de la routine d'initialisation : le fichier exécutable `/init`. Ce fichier doit alors se charger de la tâche d'initialisation.

Sous la distribution *Gentoo GNU/Linux*, l'outil *Genkernel* permet de générer un fichier au format *initramfs* et d'y placer les outils nécessaires pour initialiser un système. C'est cet outil qui est modifié dans le cadre de ce projet. Nous verrons dans les sections suivantes comment un *initramfs* est conçu et son rôle dans le processus de démarrage. Nous survolerons ensuite les modifications que nous proposons pour permettre l'union de systèmes de fichiers générant la racine du système qui démarrera.

2.2.2.1 Modifications nécessaires au système *early-userspace*

Comme nous cherchons à démarrer un système d'exploitation dont la racine est stockée sur un ordinateur distant, il faut impérativement embarquer les outils nécessaires pour permettre le démarrage réseau. Nous proposons donc en premier

Listage 2.2 Grammaire en EBNF (ISO/IEC 14977-1996) du paramètre de sélection de la racine proposé

```
Syntaxe          = "root=" Dispositif { ";" Dispositif } [ ";" ] ;
Dispositif       = TypeDispositif { Params } ;
Params           = ":" Option ;
Option           = Symbole { Symbole } ;
TypeDispositif   = ( "/" | lettre ) { Symbole } ;
Symbole          = lettre | chiffre | "_" | "." | "+" | "/" | "\" ;
```

lieu d'améliorer le support du démarrage réseau dans l'*initramfs* produit par *Genkernel*.

En second lieu, nous proposons de redéfinir la syntaxe du paramètre `root=`, qui donne à l'*initramfs* le dispositif à partir duquel monter la racine du système final, pour qu'il puisse exprimer la mise en place de systèmes de fichiers en union. En arrière-plan, ce changement de syntaxe doit être supporté par une nouvelle méthode de mise en place de la racine, nécessitant une révision des mécanismes internes de *Genkernel*.

La syntaxe actuelle mime la syntaxe supportée par le noyau, soit

`root=périphérique`

où `périphérique` est un identifiant tel que ceux utilisés en standard dans `fstab` ou par la commande `mount` d'un système UNIX.

Nous proposons de transformer la syntaxe. Le listage 2.2 présente notre proposition. De façon graphique, on obtient le diagramme en chemin de fer présenté à la figure 2.8.

Le nombre de paramètres (après les « : ») spécifiés pour un dispositif donné est variable, offrant la flexibilité nécessaire pour supporter un grand nombre de possibilité de configuration de dispositifs et systèmes de fichiers.

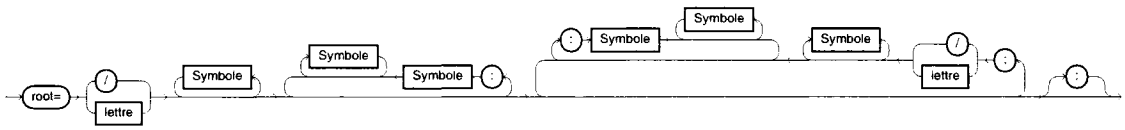

Figure 2.8 Diagramme en chemin de fer de la syntaxe proposée du paramètre `root=` à passer à l'*initramfs* au démarrage du système

De même pour le nombre de dispositifs à monter : un nombre variable de systèmes de fichiers peuvent être superposés, tous séparés par des « ; ».

Cette syntaxe permet de rester compatible avec le schéma standard pour les cas généraux de périphériques par blocs (précisément, ceux débutant par le caractère « / ») et permettra de représenter les unions.

Par exemple, pour superposer deux périphériques par blocs :

```
root=/dev/sda1;/dev/sda2
```

Mais surtout, pour superposer deux périphériques distants, nous envisageons un schéma semblable à :

```
root=nfs:10.0.0.1:/:ro;nfs:10.0.0.1:/var/state/systems/↩
    clone01:rw
```

Nous verrons à la section 3.2 le détail des modifications apportées pour supporter cette syntaxe. Nous verrons aussi comment nous nous y prenons pour rendre superposables pratiquement tous les types de dispositifs de stockage.

2.2.3 Granularité et norme de différentiation

Pour comparer deux solutions de SFI, il est utile de se doter de quelques définitions de caractéristiques. Cette section en propose deux et compare la solution développée avec *Adelie/SSI.*

En définissant la *granularité de différentiation locale d'un SFI* comme étant la taille minimale des blocs d'informations locaux à mettre en place pour différencier un nœud fils d'un serveur-référence en mode SFI, alors

- *Adelie/SSI* a une *granularité répertoire.* Des répertoires entiers doivent être copiés en mémoire locale pour permettre à un nœud d'être différent du système de référence ;
- la solution proposée par superposition de couches a une *granularité fichier.* Les différences sont seulement sur certains fichiers particuliers, présents dans la couche de différentiation.

En définissant également la *norme de différentiation locale d'un SFI* comme étant le nombre total de blocs d'information (de taille définie en granularité) stockés localement, on s'aperçoit alors que

- *Adelie/SSI* a une norme *fixe* : un nombre donné de répertoires sont montés à partir de la mémoire locale lors du démarrage ; on ne peut le changer de façon transparente[7] ;
- la solution proposée a une norme *variable* : un nombre donné de fichiers de différentiations sont fournis au démarrage. On peut cependant modifier localement le système de fichiers en n'importe quel emplacement de façon transparente,

[7] Bien sûr, un administrateur agile saura mettre en place un autre répertoire en mémoire, copier le contenu approprié, déplacer le nouveau répertoire sur l'ancien, mais cela n'est pas transparent et n'est pas directement supporté par la solution.

en accédant aux fichiers normalement.

En multipliant la norme par la taille de grain, on constate également que *Adelie/SSI* consomme plus de ressources pour stocker la différentiation : l'élément de différentiation étant le répertoire, on doit y copier à la fois les fichiers différents et les autres fichiers nécessaires, même s'ils ne sont pas différents des fichiers-références sur le serveur. En comparaison, l'architecture proposée se limite précisément aux éléments différents, et que ceux-là.

En offrant une granularité plus fine et la possibilité de varier la norme de la différentiation, la solution proposée minimise la répétition d'information tout en maximisant la propriété de propagation instantanée vue plus tôt (cf. section 2.2).

2.3 Déploiement

Nous cherchons à développer un système où chaque machine peut exposer une partie de ses ressources pour « consommation » par le public. Cette consommation doit se faire en respectant l'intégrité du rôle primaire de la machine en ne laissant aucune trace locale après l'exécution du code public. Nous proposons par conséquent une architecture qui permet de démarrer des machines virtuelles sur les hôtes pour ensuite les retirer proprement. La virtualisation est en effet garante de l'isolation des systèmes fonctionnant sur une machine physique et le contrôle des ressources utilisées, deux caractéristiques vitales au partage de parcs informatiques.

Nous développons une architecture que nous voulons pouvoir utiliser à l'échelle d'un réseau local. Nous visons également à minimiser le travail qui doit être effectué par l'administrateur d'un parc lors de l'installation et de l'exploitation de cette architecture.

Pour permettre de démarrer des clones virtuels sur des postes d'un réseau local, il faut être en mesure de communiquer avec un serveur, lui décrire la machine virtuelle nécessaire (sa mémoire, son espace disque, son nombre de processeurs, etc.), et il faut être en mesure de fournir les fichiers permettant de la démarrer (son noyau, son image disque, etc.). Nous traiterons de ces aspects dans cette section.

2.3.1 *Representational State Transfer*

S'éloignant du paradigme des appels de procédures distantes, le modèle architectural *Representational State Transfer* (REST) (FIELDING, 2000, 2002) se base sur ces principes de conception :

- les états et fonctionnalités d'une application sont divisés en ressources ;
- toute ressource a une adresse unique qui suit une syntaxe universelle pour les liens hypermédias ;
- toutes les ressources partagent une interface uniforme pour le transfert d'état entre un client et une ressource. On doit avoir :
  - un groupe restreint d'opérations bien définies ;
  - un groupe restreint de types de contenus
- le protocole de communication doit :
  - permettre des accès selon le modèle *client-serveur* ;
  - être *sans état* ;
  - être *cachable* ;
  - être en *couches.*

Selon Roy Fielding, derrière l'acronyme :

> *REST's client-server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves*

> *the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries — proxies, gateways, and firewalls — to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching.*

De plus :

> *REST enables intermediate processing by constraining messages to be self-descriptive : interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cacheability.*

Ces caractéristiques sont probablement familières car descriptives du *World Wide Web* basé sur HTTP, qui s'appuie sur un modèle REST tel que décrit par Fielding.

De ces citations, on retient, entre autres, la possibilité de mise en cache de ressources de façon transparente. Cela peut permettre de rendre plus efficace le transfert de fichiers volumineux, pensons notamment à des images de système d'exploitation. On observe cette caractéristique en pratique dans la mise en place de services de proxy à différents niveaux des infrastructures Internet.

On retient aussi qu'il est simple d'implémenter un frontal qui fera l'abstraction d'un groupe d'hôtes. Ce serveur peut offrir exactement la même interface qu'un serveur régulier, mais maintenir une « liste d'aiguillage » pour savoir à quel nœud interne correspond une ressource addressée donnée. L'équilibrage de charges peut se faire de façon transparente.

Pour ces raisons, et pour la grande simplicité d'implémentation amenant une bonne maintenabilité, notre système de déploiement propose d'utiliser REST. Ce modèle architectural se couple bien avec *libvirt* (nous étudions cette librairie à la section 2.3.2) pour offrir des représentations d'état dans le format XML de cette librairie à des adresses (*Uniform Resource Locator* – URL) définies.

REST se distingue donc des approches de type RPC basées sur SOAP/WSDL utilisées dans certaines solutions de déploiement (cf. section 1.4) à plusieurs égards. Il s'appuie sur une méthode d'interaction simple dictée par les primitives d'accès du protocole de communication, sur des types de représentation de données suivant des formats définis et sur un format d'adressage uniforme permettant les liens hypermedias. Le lecteur intéressé pourra se référer au travail de MUEHLEN et al. (2005), qui étudie les différences entre REST et SOAP dans le détail.

2.3.1.1 Primitives de communication

Pour le déploiement des machines virtuelles, nous proposons donc une architecture REST, basée sur HTTP, pour effectuer les opérations réseau : nous avons ainsi accès à un groupe restreint d'opérations de base **CRUD** (*create*, *read*, *update*, *delete*) fourni par ce protocole (il s'agit de notre *groupe d'opérations* REST).

Primitive CRUD	Méthode HTTP
Create	PUT
Read	GET
Update	POST
Delete	DELETE

Ces opérations sont utilisées par un client pour communiquer vers (à partir de) un serveur les informations reliées au transfert et au contrôle d'état de machines

virtuelles.

Pour cela, il est nécessaire de définir une représentation de la ressource à transférer entre les entités. Les sections suivantes s'y attardent.

2.3.2 Abstraction de la solution de virtualisation

La librairie *libvirt* (VEILLARD et ZAK, 2007) cherche à offrir une interface de programmation stable[8] qui abstrait l'implémentation du système de virtualisation sous-jacent. Comme le monde de la virtualisation est en ébullition, il est nécessaire de contrôler la variabilité des interfaces à l'aide d'une telle couche d'abstraction. *libvirt* a été développée d'abord pour permettre d'exploiter *Xen*, mais des développements récents permettent, avec la même API, de démarrer une machine virtuelle sous *QEMU*, voire même sous *KVM*[9].

L'interface de programmation est basée sur la lecture de fichiers de descriptions de machines virtuelles au format XML. Par exemple, le listage 2.3 décrit une machine virtuelle.

libvirt est proposée dans ce projet comme couche d'abstraction pour développer notre outil de déploiement de VMs.

Dans le modèle REST, le descriptif XML *libvirt* agit comme représentation d'état. Évidemment, chacune des représentations XML n'est pas suffisante pour faire fonctionner une machine virtuelle, il faut mettre en place les entités externes qui y sont

[8]Étant encore, au moment de l'écriture de ce texte, en développement intense, des modifications sont tout de même à prévoir fréquemment.

[9]Ensemble logiciel récemment inclus dans le noyau *Linux* standard permettant de tirer profit des extensions Intel/AMD pour la virtualisation de systèmes. L'application usager est très similaire à celle de *QEMU* et son fonctionnement est calqué sur cette dernière (cf. section 1.2.3.2). Voir `http://kvm.sourceforge.net`.

Listage 2.3 Exemple de déclaration de machine virtuelle au format XML de *libvirt*

```
<?xml version='1.0'?>
<domain type='xen'>
  <name>la_machine</name>
  <os>
    <type>linux</type>
    <kernel>linux-2.6.16.28-xen-domU</kernel>
    <cmdline>root=block:/dev/sda1|block:/dev/sdb2</cmdline>
  </os>
  <memory>32000</memory>
  <vcpu>1</vcpu>
  <devices>
   <disk type='file'>
      <source file='disk.img'/>
      <target dev='sda1'/>
    </disk>
    <interface type='bridge'>
      <source bridge='xenbr0'/>
      <mac address='de:ad:00:be:ef::11'/>
      <script path='/etc/xen/scripts/vif-bridge'/>
    </interface>
  </devices>
</domain>
```

décrites : noyaux, images disques, etc. Pour simplifier le transfert de ces groupements liés, nous proposons un format d'archive de machine virtuelle.

2.3.3 Format de transfert de machines virtuelles

Les ressources nécessaires au fonctionnement d'une machine virtuelle peuvent être transférées dans un ensemble encapsulé dans un fichier d'archive. Nous nous inspirons vaguement du format *OpenDocument* (ISO/IEC 26300-2006) : l'archive contient un fichier principal `config.xml` contenant les directives de mise en place de la machine virtuelle, au format XML de *libvirt*. Les fichiers images, noyaux et *initramfs* sont inclus dans l'archive et déclarés dans le fichier de configuration principal.

Nous proposons le format `tar` avec la compression `gzip` ou `bzip2` pour obtenir un maximum de compression. Si l'on se fie aux expériences de KNOPPER (2000) à propos de la compression de systèmes de fichiers pour créer Knoppix, on peut

Listage 2.4 Contenu d'un fichier ressource représentant une machine virtuelle

```
/config.xml <-- format specifié par libvirt
/kernel.fichier    <--| le reste des fichiers est référencé
/initramfs.fichier <--| par le fichier config.xml et est
/disks/hdd1.image  <--| placé dans une hiérarchie
/disks/hdd2.image  <--|
```

envisager une réduction de l'espace nécessaire pour stocker un système d'exploitation d'environ 50%. Si l'on ajoute à cela l'espace inutilisé, « vide », d'une *image* (au sens d'un dispositif à taille fixée, un fichier `loop`) de disque typique, on peut envisager des ratios de compressions plus élevés.

Cette manière de représenter les ressources permet :

- la réutilisation d'outils standards, libres et disponibles (compression et archivage) ;
- l'extensibilité par ajout de fichiers à noms réservés, tels que `config.xml` ;
- l'extensibilité du format XML pour décrire les machines virtuelles, tout en restant compatible avec les éléments des schémas précédents ;
- l'encapsulation de données en relations les unes avec les autres dans un objet unique ;
- la compression des données.

Le listage 2.4 montre un exemple de hiérarchie à l'intérieur d'une archive de machine virtuelle au format proposé.

2.4 Découverte automatique

La problématique de la découverte de systèmes prêts à offrir du temps de traitement CPU et autres ressources pour participer à une grille instantanée se résume à une question : comment les différents systèmes reliés en réseau peuvent-ils communiquer

les uns avec les autres de manière à publier les services qu'ils offrent ?

Différentes approches sont possibles, soit d'abord créer un catalogue centralisé des systèmes qui peuvent participer ainsi que des ressources dont ils disposent. Cette approche est proposée par Kotsovinos pour le système *XenoServer*.

On peut aussi envisager des solutions décentralisées impliquant des algorithmes de catalogage distribués en paire-à-paire (P2P) ou des méthodes de multicasting.

Notre projet se tourne vers DNS. Il est possible de baser la découverte de services sur un amalgame d'innovation apportées récemment au vénérable et ubiquitaire protocole : *DNS Service Discovery* (DNS-SD) (CHESIRE et KROCHMAL, 2006) et *Multicast DNS* (mDNS) (CHESHIRE et KROCHMAL, 2006). Mieux connues sous les appellations « Rendez-Vous » ou « Bonjour » associées à l'implémentation réalisée par *Apple Inc.*, ces technologies permettent à un système d'intégrer les services qu'il offre à l'infrastructure DNS et à simplifier la découverte de ces derniers via différents mécanisme de communication (*unicast* DNS classique ou *multicast*).

Cette démarche permet de réutiliser un système de propagation d'informations déjà existant en ajoutant simplement un champ `SRV` qui associe un service à une machine, un peu comme un champ `MX` associe un ou des serveurs SMTP à un domaine.

2.5 Surveillance

Nous proposons d'intégrer un système de surveillance des nœuds déployés à notre architecture.

La surveillance d'état de nœuds nécessite une mise à jour constante de métriques de

performance et d'état local pour chacune des machines. Il faut ensuite communiquer ces données à un point de contrôle qui en fera un rendu à l'administrateur pour l'informer.

Ce module est d'une grande importance puisque nous voulons déployer des nœuds sur des réseaux de stations qui peuvent avoir d'autres utilisations parallèles : une variabilité dans les performances et l'état physique des machines est à prévoir, et sans surveillance, il est difficile de réagir correctement à ces variations.

Nous proposons l'utilisation du logiciel *Ganglia* (MASSIE et al., 2004; GANGLIA, 2007) pour assurer la surveillance.

2.6 Conclusion du développement de l'architecture

Ce chapitre a présenté plusieurs composants architecturaux utilisés développer une méthode de déploiement de nœuds virtuels légers. Nous avons vu trois éléments à modifier ou développer pour rendre possible le démarrage de nœuds virtuels en SFI, soit le système *early-userspace*, une méthode de génération de couche de différentiation et un service exposant et rendant facile la découverte de ressources consommables sur un ou plusieurs systèmes physiques.

Le couplage de ces trois aspects permet de rejoindre le concept de *Minimal Configuration Grid* présenté à la section 1.4.1, mais en offrant une variation importante sur le thème en préconisant une approche *pull-based* (cf. section 2.2), rendue possible par l'utilisation de techniques de SFI. Il n'est pas nécessaire de transférer une image entière (approche *push-based*) de système d'exploitation pour le démarrer, on sauve donc en temps de transfert initial ou, de manière équivalente, on gagne en rapidité de déploiement.

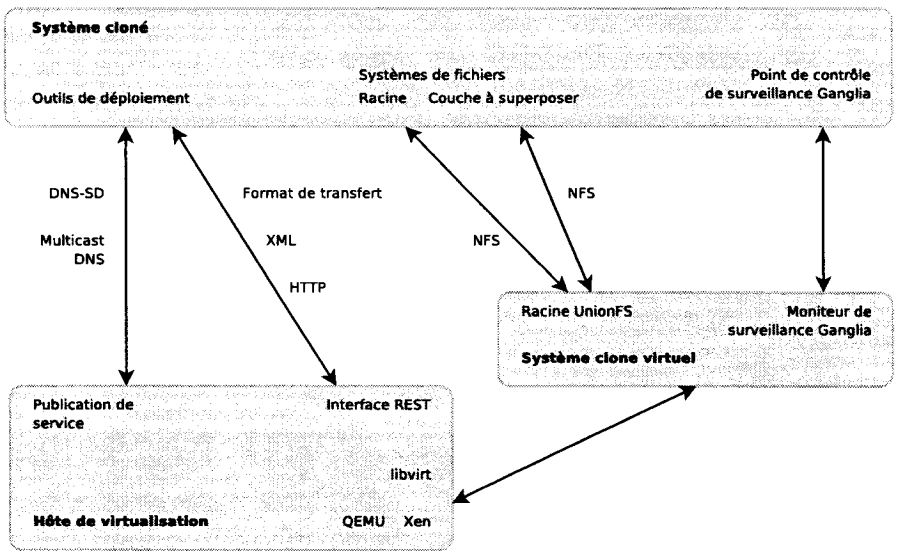

Figure 2.9 Survol des composants architecturaux

La figure 2.9 offre une revue des différents composants architecturaux arrimés pour obtenir un système de déploiement de nœuds virtuels légers.

Cette architecture se distingue de celle des systèmes de gestion de grappes de calculs strictement physiques exposés au chapitre précédent pour la raison fondamentale que le déploiement de nœuds est *instancié logiciellement par l'usager*, alors que dans l'autre paradigme, le démarrage est *instancié par le démarrage physique du nœud* (cf. section 1.4). [10]

Nous verrons au chapitre suivant comment l'implémentation de cette architecture a été réalisée.

[10]Nous avons cependant fourni une documentation au projet *Genkernel* expliquant une méthode pour réaliser le démarrage de nœuds physiques en s'appuyant sur notre architecture.

CHAPITRE 3

SYSTÈME DE GÉNÉRATION ET DÉPLOIEMENT DE NŒUDS VIRTUELS LÉGERS

Ce chapitre explore la mise en place et l'intégration des différentes parties de l'architecture définie au chapitre 2. Nous voyons d'abord comment on crée les couches *UnionFS* qui sont superposées pour permettre à un nœud de démarrer dans sa configuration personnalisée. Ensuite, nous décrirons en détails notre refonte du système d'initialisation *early-userspace* généré par *Genkernel*. Enfin, nous examinerons comment nous avons développé notre infrastructure de déploiement de nœuds. Une section finale décrira ensuite les outils usagers développés.

Nous introduisons dans ce chapitre de nouveaux outils entièrement développés dans le cadre de ce mémoire, *Clone* et *Napalm*.

3.1 Génération de la couche supérieure – *Clone*

Nous avons développé un logiciel, appelé simplement « *Clone* », permettant de créer la couche supérieure initiale du système de fichiers en union qui sera montée sur les nœuds. Cela se traduit par la génération de fichiers de configuration adaptés à chacun des nœuds

Pour ce faire, deux stratégies sont employées :

Utilisation de gabarits Nous utilisons un système de génération de fichiers par gabarits, *Cheetah* (CHEETAH, 2007), pour générer la plupart des fichiers de-

vant différer d'un clone à un autre. Des variables sont remplacées à l'intérieur de ces fichiers gabarits pour générer en sortie un fichier de configuration ;

Approche programmatique Dans les cas où une plus grande flexibilité est nécessaire pour préparer certains aspects de la configuration, nous laissons la possibilité à l'utilisateur de programmer lui-même avec une grande liberté ses scripts : il est possible d'hériter d'une classe `SetupScript` très simple pour écrire des classes *Python* qui permettent de modifier les environnements des futurs clones.

Nous croyons que de travailler, dans la majorité des cas, sur des gabarits mimant le format du fichier de configuration original est plus intuitif et se compare favorablement aux autres approches plus complexes, comme l'écriture de scripts de configuration. Dans un souci de flexibilité, cette alternative est tout de même supportée.

Nous appellerons GCDSE, pour *gabarit de couche de différentiation de système d'exploitation*, chacun des ensembles de fichiers gabarits nécessaires à la configuration d'une couche de différentiation de système d'exploitation clone (se référer à la section 3.5.1.1 pour un exemple). On peut créer différents GCDSE contenant des fichiers gabarits différents pour le même ensemble de fichiers de configuration. De cette façon, il est possible de sélectionner quels GCDSE on veut appliquer pour générer un clone donné. Cela permet de supporter la création de différents rôles pour les clones.

Qui plus est, fort du concept de superposition utilisé avec *UnionFS*, nous permettons également de partir d'un GCDSE A donné, puis de remplacer certains fichiers gabarits de cet ensemble par ceux venant d'un second ensemble B. On superpose B sur A au sens d'*UnionFS*.

Ainsi, on peut se baser sur un GCDSE basique qui traite le cas général et y superposer un ensemble qui modifiera certains aspects seulement de ce GCDSE. Par ce mécanisme, nous souhaitons éviter les répétitions d'information inutiles.

Par exemple, on peut seulement vouloir modifier la configuration standard de SSH pour un certain groupe de nœuds. On superpose alors le GCDSE `X` sur le basique, `A`. Le gabarit résultant étant celui contenant, pour `sshd_config`, le résultat de l'interprétation du gabarit de la couche `X`.

```
Vue de l'usager

Gabarit X:  /etc/ssh/sshd_config
Gabarit A:  /etc/ssh/sshd_config /etc/serv_a.conf [...]
```

Nous fournissons un jeu standard de gabarits pour les services communs. Il est donc simple de générer les configurations de nœuds « standards », et, pour des besoins plus avancés, notre solution permet, à notre avis, une grande facilité d'adaptation.

3.1.1 Génération par gabarits

L'engin *Cheetah* fournit une syntaxe basée sur des variables préfixées de sigils (« `$` ») qui seront remplacées par leur valeur dans les fichiers de sortie. L'engin fournit également des constructions syntaxiques à l'intérieur des gabarits, permettant le branchement, les boucles, les tests.

Par exemple, on pourrait avoir un tel gabarit pour `fstab` :

```
# /etc/fstab

${os.root_device}    ${os.root_device.fstype}   noatime   0   0
```

Cet engin permet aussi des cas d'utilisation plus complexes, sans toutefois alourdir l'écriture de gabarits. Par exemple :

```
# /etc/modules.autoload.d/kernel-2.6

#if $varExists('modules_autoload')
#for $module in ${modules_autoload}
${module}
#end for
#end if
```

Pour « remplir » les variables présentes dans les gabarits, un fichier de configuration est nécessaire. Les paramètres de configuration des clones sont écrits selon la syntaxe *Yet Another Markup Language* (YAML), définie formellement (BEN-KIKI et al., 2005). De nombreuses librairies supportent ce format, tant pour *Python* (le langage utilisé pour notre système) que pour différents autres langages de programmation (*Objective-C*, *.NET Framework*, *Perl*, *PHP*, *Ruby*, *Java*, *Haskell*). Comme toujours, nous favorisons l'utilisation d'un format offrant une conformité à un standard, il s'agit d'une bonne pratique de génie logiciel.

Les valeurs placées dans les variables proviennent d'un parcours en « cascade » d'une hiérarchie YAML de valeurs associées à des hôtes clones. À chaque niveau, on peut redéfinir une variable ou conserver la valeur fixée dans le nœud parent.

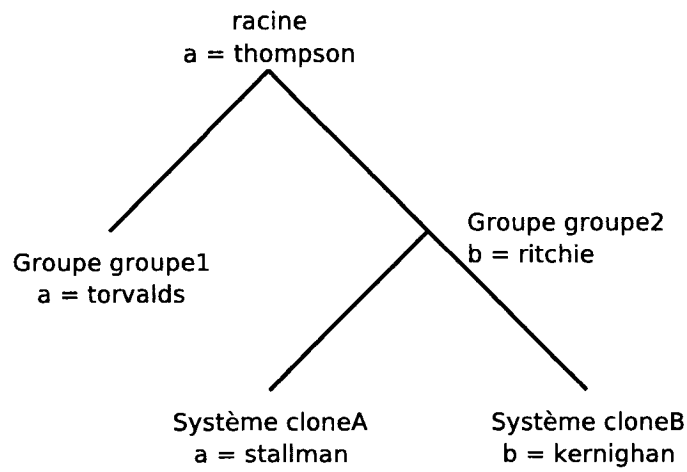

Figure 3.1 Exemple d'arbre pour illustrer le parcours en « cascade »

Pour exemple, en considérant l'arbre illustré à la figure 3.1, pour `cloneA`, nous aurons `a = stallman`, pour *cloneB*, nous conservons la valeur fixée dans son grand-père et transmise en cascade, donc `a = thompson`, mais on définiera localement (et pour d'éventuels nœuds enfants) `b = kernighan`, remplaçant la valeur définie au groupe `groupe2`. On peut donc définir un groupe de valeurs pour un groupe de machines et seulement redéfinir certains aspects plus bas dans la hiérarchie. Encore une fois, cela minimise les répétitions inutiles d'information.

L'arbre de la figure 3.1 se représente dans un fichier YAML ayant l'allure de celui présenté au listage 3.1. À titre d'exemple plus concret, l'annexe IV montre le fichier de configuration YAML que nous avons utilisé pour notre grappe de test.

Une classe relativement simple cachant les détails d'implémentation reliés à l'extraction des variables de configuration pour les gabarits et scripts est fournie. Cette dernière abstrait la lecture et l'interprétation du fichier au format YAML. Cette abstraction permet d'utiliser un autre format de stockage sans devoir revoir le reste du logiciel d'interprétation des gabarits.

Listage 3.1 Exemple de fichier YAML

```
a:thompson
Groupes:
  groupe1:
    a:torvalds
  groupe2:
    b:ritchie
    systèmes:
      cloneA:
        a:stallman
      cloneB:
        b:kernighan
```

3.1.2 Utilisation de la couche de différentiation

Le résultat de l'interprétation des gabarits et de l'exécution des scripts de configuration donne une hiérarchie de fichiers à superposer sur la racine du serveur pour démarrer un clone configuré selon les besoins.

Cette hiérarchie sera stockée sur le serveur dans `/var/state/systems/`*`nomduclone`* et sera exportée par NFS, soit en lecture seulement ou en lecture-écriture pour permettre la mémoire d'état. Il faudra (évidemment — nous sommes en SFI) aussi exporter la racine même du serveur, en lecture seulement.

3.2 Modifications au système *early-userspace* de *Linux*

L'outil *Genkernel* permet de créer un système minimal *early-userspace* de manière simple, mais rend difficile l'ajout de modules et outils supplémentaires pour ajouter des fonctions et un support pour de nouveaux dispositifs et systèmes de fichiers.

L'incarnation stable la plus récente avant nos modifications était la version 3.3.11. L'architecture de *Genkernel* était monolithique et le code présenté était difficile-

ment maintenable. Depuis plusieurs années, ce dernier n'avait subi que des corrections de bogues et des ajouts minimes. Il est resté difficile d'ajouter du support pour de nouveaux systèmes matériels et logiciels. Nous avions déjà fourni un effort dans le cadre du projet *Adelie/SSI* pour ajouter le support du démarrage réseau, mais il est resté embryonnaire, non-documenté et fragile.

En gardant en tête la modification proposée à la section 2.2.2.1, et après étude du code de *Genkernel*, les objectifs au niveau de la modification de cet outil ont donc été de :

- concevoir une nouvelle architecture permettant d'ajouter de manière simple de nouveaux modules pour monter différents types de systèmes de fichiers, à partir de différents dispositifs matériels locaux et éloignés (réseau) ;
- améliorer le support réseau ;
- rendre possible l'union de différents systèmes de fichiers contenus sur des dispositifs variés pour former la racine du système final.

Ces modifications ont été faites pour être intégrées à la version 4 de *Genkernel*, version de développement au moment de la réalisation de notre travail.

3.2.1 Introduction à *Genkernel*

Dans son essence, l'*initramfs* généré par *Genkernel* est un petit système UNIX minimaliste contenant plusieurs outils standards. Un fichier `/init` est présent et constitue le point clef de son fonctionnement.

Ce fichier est une séquence de code Bash qui met en place les dispositifs et systèmes de fichiers pour démarrer le système final. Dans la version 3.3.11 de *Genkernel*, ce fichier fait appel à quelques variables prédéfinies dans un fichier externe

Listage 3.2 Contenu de l'*initramfs* sous *Genkernel* 3.3.11

```
1  /
2  |-- init <-- script principal d'initialisation
3  |-- initrd.scripts <-- bibliothèque de fonctions
4  |-- etc
5  |   `-- initrd.defaults <-- paramètres de configuration
6  |-- lib
7  |   `-- [...]
8  |-- usr
9  |   `-- [...]
10 `-- [...]
```

`initrd.defaults` et à une librairie de fonctions dans le fichier `initrd.scripts`.

Le listage 3.2 présente la hiérarchie du contenu d'un *initramfs* généré par *Genkernel* 3.3.11. Nous y omettons les outils standards UNIX pour mettre l'accent sur l'infrastructure particulière nécessaire au fonctionnement de `/sbin/init` :

3.2.2 Modularisation

Alors qu'auparavant l'unique librairie de fonctions `/initrd.scripts` contenait toutes les directives utilisées pour le montage de tous les systèmes et dispositifs, nous avons effectué des modifications permettant d'obtenir une architecture modulaire. Notre travail permet maintenant d'encapsuler les fonctionnalités reliées à chacun des systèmes de fichiers dans des scripts séparés et spécialisés.

La structure du système *early-userspace* a maintenant l'aspect présenté au listage 3.3.

Chacun des scripts de montage de système de fichiers racine (dans `fsloaders`, ligne 11) est directement appelé sur une ligne de commande, prenant différentes directives en arguments. Cette stratégie permet d'isoler les espaces de nommage des scripts. Auparavant, tous les scripts étaient chargés dans un seul processus via une commande d'inclusion (`source` en langage Bash).

Listage 3.3 Contenu de l'*initramfs* après modifications à *Genkernel*

```
/
|-- init <-- script principal d'initialisation
|-- etc
|   `-- initrd.defaults <-- paramètres de configuration
|-- lib
|   `-- [...]
|-- usr
|   |-- [...]
|   `-- lib
|       `-- genkernel <-- librairie de scripts et fonctions
|           |-- fsloaders <-- scripts de montage
|           |   |-- block
|           |   |-- dmraid
|           |   |-- evms2
|           |   |-- livecd
|           |   |-- livecd+nfs
|           |   |-- livecd+url
|           |   |-- loop
|           |   |-- loop+nfs
|           |   |-- luks+block
|           |   |-- lvm2
|           |   |-- lvm2+md
|           |   |-- md
|           |   |-- nfs
|           |   |-- quickinstallcd
|           |   `-- tmpfs
|           |-- libgmi.sh
|           |-- libgmi_core.sh   <-- | librairies
|           |-- libgmi_kernel.sh <-- | spécalisées
|           |-- libgmi_net.sh    <-- |
|           `-- udhcp.sh <-- autoconfigurateur réseau
`-- [...]
```

On utilise les scripts `fsloaders` spécialisés comme des commandes traditionnelles :

```
$ fsloaders/md --setup-device /dev/md0
$ fsloaders/lvm2 --do-mount /mnt/abc /dev/vg/volume
```

Parmi les fonctionnalités améliorées, notons que ces scripts de base peuvent être combinés pour offrir des services plus avancés. À titre d'exemple, si un usager veut obtenir une racine de système final qui tient sur un LVM qui à son tour tient sur un RAID logiciel (`md`), il suffit d'utiliser

```
root=lvm2+md:/dev/vg/volume:/dev/md0,/dev/md1
```

Cette déclaration fera un appel à un script englobant, `lvm2+md`, qui utilisera en fait `md`, qui mettra en place `/dev/md0` et `/dev/md1`, puis `lvm2`, qui détectera le dispositif LVM contenu sur ces deux disques RAID logiciels et le mettra en fonction comme racine finale.

On peut faire de même avec, par exemple, `loop+nfs:[...]` pour monter la racine à partir d'une image stockée en fichier *loopback* stockée sur NFS et ainsi démarrer un système virtuel avec une image de *LiveCD* compressée. De nombreuses autres possibilités excitantes existent et nous avons conçu cette architecture pour en supporter facilement d'autres. On pourra se référer à l'annexe II pour voir le contenu de deux de ces scripts, soit `nfs` et `loop+nfs`.

3.2.3 Dispositifs supportés

Le support des dispositifs qui étaient déjà supportés est assuré :

block Montage à partir de dispositifs matériels par accès en blocs tels les disques durs, CD/DVD-ROM, clefs USB. (Un paramètre `root=` débutant par un « / », par exemple `root=/dev/sda1`, sera traité comme un appel à `block`. Ici, `root=block:/dev/sda1`) ;

md Montage à partir de dispositifs RAID logiciels *Linux* ;

dmraid Montage à partir de dispositifs RAID matériels-logiciels ;

evms2 Montage à partir de dispositifs EVMS2 – *Enterprise Volume Management System 2* ;

livecd Montage à partir d'un CD/DVD d'installation *Gentoo GNU/Linux* ;

lvm2 Montage à partir d'un dispositif LVM2 – *Logical Volume Manager 2* ;

lvm2+md Montage à partir d'un dispositif LVM2 qui lui-même tient sur un RAID logiciel *Linux* ;

nfs Montage à partir d'un système de fichiers réseau.

Dans l'optique de rendre possible le montage réseau des racines de postes de travail ou de nœuds de grappe de calculs, et pour faciliter l'installation de *Gentoo GNU/Linux*, le support d'autres dispositifs et système de fichiers à été ajouté :

livecd+url Montage à partir de l'image ISO9660 d'un CD/DVD d'installation *Gentoo* stocké sur un serveur FTP ou HTTP[1] ;

livecd+nfs Montage à partir de l'image ISO9660 d'un CD/DVD d'installation *Gentoo* stocké sur un serveur NFS ;

[1]Notre solution s'approche de celle de SUZAKY et al. (2006), mais n'est pas conçue pour télécharger à la demande des fichiers-blocs constituant, rassemblés, un système de fichiers complet. Notre approche est développée pour fonctionner dans un système d'initialisation minimal et se limite à télécharger l'image complète – on la destine donc à servir pour démarrer des systèmes de petites tailles, tels les images d'installation « minimales » de *Gentoo GNU/Linux*, qui peuvent tenir en mémoire.

loop Montage à partir d'un système de fichiers en boucle (*loopback*) contenant un système de fichiers racine (formats SquashFS, `loop`, ISO9660) ;

loop+nfs Montage à partir d'un système de fichiers loopback contenu sur un lecteur réseau NFS ;

luks+block Montage à partir d'un dispositif par blocs dont le contenu est encrypté à l'aide de LUKS – *Linux Unified Key Setup* ;

quickinstallcd Détermination automatique de l'architecture du système à démarrer et téléchargement sur un miroir optimal de l'image ISO9660 d'un installateur *Gentoo*. Montage à partir de cette image ;

tmpfs Mise en place d'un système de fichiers (vide) contenu en mémoire. Utile comme couche supérieure d'un *UnionFS*, pour obtenir un accès en lecture et écriture.

Ces scripts de montage ont tous été conçus pour permettre de les superposer avec *UnionFS*. Bien que la plupart de ces dispositifs et fonctionnalités associées ne soient pas exploités directement dans le cadre de notre projet, nous croyons que d'offrir cette flexibilité ouvre la porte à diverses innovations dans le processus de démarrage. Nous expliquons comment ces dispositifs peuvent être superposés à la section 3.2.5.

3.2.4 Support avancé pour l'autoconfiguration réseau

Parmi les modifications proposées, un support plus complet pour l'autoconfiguration réseau a été mis en place.

La résolution de noms à l'intérieur du système *early-userspace* est maintenant disponible par l'ajout d'une librairie C minimale le supportant (cette modification a été apportée au même moment que notre travail par les développeurs de *Genkernel*).

Pour la configuration TCP/IP, nous avons suivi la syntaxe de ligne de commande noyau proposée par les développeurs du mécanisme de montage de la racine par NFS dans le noyau *Linux* (KUHLMANN et MARES, 1997) :

```
ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<↩
   device>:<autoconf>
```

Nous avons ajouté les scripts nécessaires pour le support de DHCP en espace usager plutôt que par autoconfiguration du noyau. Cela permet d'accéder plus facilement aux données reçues par un serveur DHCP, y compris les extensions au protocole (REYNOLDS, 1993) pour des configurations plus avancées.

Une rustine permettant la configuration réseau automatique sur une interface sans-fil de type IEEE 802.11a/b/g a également été proposée, mais n'est pas encore inclue dans *Genkernel* 4, les développeurs étant peu enclins à ajouter les outils de configuration pour réseaux sans-fil dans l'*initramfs*.

3.2.5 Racine *UnionFS* et scripts d'initialisation

L'architecture présentée permet de monter plusieurs systèmes de fichiers en superposition en donnant une directive `root=` qui contient plus d'un système à monter. On peut superposer tous les dispositifs disponibles (cf. les scripts de montage développés présentés à la section 3.2.3), dans l'ordre qui nous convient.

Une difficulté importante est celle du référencement des systèmes de fichiers qui sont montés pour être fusionnés en la racine finale, car l'espace de nommage vu dans le système *early-userspace* disparaît en partie lors du déplacement de la racine de l'arbre vers un de ses nœuds (l'opération `switch_root`).

Par exemple, pour fusionner deux systèmes de fichiers, il faut d'abord monter le

premier sur `/mnt/FS1`, le second sur `/mnt/FS2`, puis on les fusionne dans l'*UnionFS* `/rootfs` (cf. figure 2.1). Si on pivote `/` sur `/rootfs`, alors les deux `/mnt/FS{1,2}` ne sont plus visibles dans l'arborescence (cf. figure 2.7).

Nous utilisons une stratégie qui vise à assurer, après le déplacement de la racine, la possibilité de référencer tous les systèmes de fichiers montés et fusionnés à l'aide d'*UnionFS*, permettant à l'usager du système final de modifier les paramètres des systèmes de fichiers fusionnés en la racine finale à sa guise.

Pour ce faire, chaque système de fichiers devant être mis en superposition avec d'autres pour former la racine finale est d'abord monté à l'intérieur d'un répertoire `tmpfs`, appelons-le `/mnt/unions`. Lorsque tous les systèmes sont fusionnés dans la future racine *UnionFS* qui tient alors dans le répertoire `/rootfs`, les « participants » à l'union sont déplacés simplement en bougeant le point de montage `/mnt/unions` vers, ici, `/rootfs/.unions`, pour peu qu'il soit possible de créer ce répertoire `.unions` (donc qu'on ait un système accessible en lecture-écriture), ou que `.unions` soit déjà existant dans `/rootfs`. On peut alors déplacer la racine sur `/rootfs`.

Cette stratégie permet aux scripts d'initialisation et à l'usager du système final de modifier l'état des systèmes participants à l'union en pointant les outils comme `mount` vers `/.unions/xyz`, où `xyz` est l'un des participants à la superposition constituant le système de fichiers racine.

Tout cela peut évidemment, au premier coup d'œil, heurter le sens de l'élégance, mais la fonctionnalité que cela apporte contrecarre toute considération noble.

À titre d'exemple d'utilisation, lors du démarrage avec un paramètre tel que

```
root=nfs:10.0.0.1:/:ro;nfs:10.0.0.1:/var/state/systems/clone01:rw
```

on obtient :

```
[messages noyau]

   GMI starting ...

>> Setting up networking on eth0 (dhcp)
>> Enabling UnionFS support
>> Mounting filesystems
>>      nfs:10.0.0.1:/:ro
>>      nfs:10.0.0.1:/var/state/systems/clone01:rw
>> Adding unions to the root filesystem
>>      10.0.0.1:/ (ro)
>>      10.0.0.1:/var/state/systems/clone01 (rw)
>> Transferring control to /sbin/init ...

[prise de contrôle par /sbin/init]
```

À noter que notre implémentation utilise un ordre inversé des unions, par rapport à fonction « union » décrite à la section 2.2.1.1 : le système le plus à *gauche* sur la ligne sera monté en *premier*, et les systèmes subséquents seront ajoutés par-dessus, l'un après l'autre. Le plus à droite a donc priorité.

Les scripts d'initialisation ne sont pas touchés par cette façon de procéder, minimisant ainsi le travail à faire au niveau de la distribution qui veut supporter une racine en union. Une exception cependant : les scripts se chargeant, lors de la mise hors tension d'un ordinateur, de la gestion du démontage de systèmes de fichiers ont dû être modifiés. Nous avons proposé au projet *Gentoo GNU/Linux* les mécanismes nécessaires pour démonter proprement, et dans l'ordre correct, les systèmes de

fichiers participant à l'union du système de fichiers racine. Dans le contexte de systèmes clones, on peut évidemment distribuer ce correctif dans le GCDSE si on ne veut pas modifier l'OS : on reste agnostique quant à la distribution clonée.

3.2.6 Décompression de données en mémoire

Lors de l'utilisation de clones sans mémoire d'état, on pourrait vouloir installer certains logiciels ou jeux de donnés particuliers sur la couche supérieure, donc en mémoire. Nous avons implementé un mécanisme permettant de le faire.

Tirant profit du système *Gentoo Portage*, qui peut créer des archives compressées contenant des logiciels donnés, il est possible de bâtir une bibliothèque de paquetages pour installation directe par décompression sur un ensemble de nœuds clones ayant des besoins logiciels particuliers. On doit simplement créer les paquetages à l'aide de

```
$ emerge --buildpkg <logiciel>
```

On pourra ensuite les placer dans un répertoire du serveur visible par les clones. Au démarrage, il suffira de spécifier sur la ligne de commande du noyau où sont les archives à décompresser.

Par exemple, considérons que `/var/tmpfs-overlays` contient l'arborescence suivante :

```
/var
'-- tmpfs-overlay/
    |-- pkg1.tar.gz
```

```
|-- pkg2.tar.bz2
|-- pkg3.tbz2
|-- other-dataset1.tbz2
`-- [...]
```

On peut alors démarrer un clone en mode sans mémoire d'état et installer sur le système de fichiers en mémoire certains paquetages :

```
root=nfs:10.0.0.1:/:ro;nfs:10.0.0.1:/var/state/systems/↩
    clone01:ro;tmpfs:rw unpack=/var/tmpfs-overlay
```

L'*initramfs* se chargera d'itérer sur le contenu du répertoire spécifié par `unpack=` et de décompresser le contenu dans la racine du système de fichier du clone. On peut spécifier plusieurs fichiers d'archives ou répertoires en les séparant par des « ; ».

Cette méthode de stockage de certains ensembles logiciels (ou de données) en mémoire permet de maximiser la bande passante en lecture/écriture pour les applications s'exécutant sur le clone, caractéristique qui peut être désirable selon le contexte. Cela permet de remédier, dans une certaine mesure, à la perte de performance reliée à l'utilisation de systèmes de fichiers réseaux comme couches *UnionFS*. Il faudra évidemment sacrifier une partie de la mémoire disponible sur le clone pour stocker ces informations.

3.3 Implémentation d'infrastructure – *Napalm*

Cette section présente le travail réalisé pour permettre le démarrage *instancié par l'usager* de nœuds virtuels. Bien que d'abord créée pour le déploiement de nœuds clones, la boîte d'outils permet tout autant le déploiement de machines virtuelles à

partir d'images disques : nous avons pris soin de garder ce cas en tête lors de notre réflexion sur l'architecture à proposer.

L'implémentation est réalisée avec le langage *Python* pour un maximum de portabilité et de maintenabilité. Les contraintes de performance à l'exécution ne sont pas très importantes pour cet aspect du travail : on traite essentiellement des requêtes HTTP et des transferts de fichiers.

Le logiciel développé est nommé « *Napalm* », de *Networked And Parallel Applications on Lazy Machines.*

Pour faire fonctionner le système dont l'architecture a été décrite à la section 2.3, il est nécessaire de

- traiter les requêtes HTTP PUT, GET, POST, et DELETE (nos primitives CRUD) ;
- traiter les différents types de fichiers (types MIME) qui peuvent être reçus et envoyés ;
- communiquer avec la librairie *libvirt* pour configurer et contrôler les machines virtuelles.

Les sections suivantes montrent comment nous avons traité ces trois axes.

3.3.1 Routage des requêtes HTTP

De manière à accéder aux ressources sur un serveur donné, nous utilisons une convention hiérarchique qui suit le format classique des systèmes de fichiers UNIX.

En partant d'un préfixe P, égal à `http://serveur/cgi-bin/napalm.cgi`, voici cette convention :

Listage 3.4 Classe de gestion des requêtes HTTP `/about` reçues par un serveur *Napalm*

```
1  class AboutWebHandler(WebHandler):
2      """Simple WebHandler that shows information about the software."""
3      path = r''/about$'
4
5      def get(self):
6          """Answers GET requests."""
7          http.ok(napalm.__doc__+"\n"+
8                  napalm.__name__ +
9                  napalm.__copyright__+"\n"+
10                 napalm.__license__)
```

P/**about** Retourne des informations sur le logiciel *Napalm* ;

P/**domains.xml** Retourne la liste des domaines et offre des hyperliens vers les fichiers de configuration de chacun d'eux ;

P/**domain**/*nom*/**config.xml** Configuration d'un domaine, au format XML *libvirt*. Contient des éléments faisant références à des fichiers associés au paquetage de cette machine virtuelle ;

P/**domain**/*nom*/**status.xml** Contient l'état de la machine virtuelle, suivant un format défini ;

P/**domain**/*nom*/*fichier* Tout fichier référencé (et seulement eux) dans le fichier `config.xml` est accessible à partir de la racine `/domain/nom/` dans une arborescence.

Cette hiérarchie est gérée par des classes dynamiquement instanciées et invoquées par le script `napalm.cgi` par sélection basée sur des expressions régulières. Un aiguilleur se charge de trouver la bonne classe à partir de la chaîne de caractères placée après *P*. Le listage 3.4 montre par exemple une de ces classes, qui traite le cas le plus simple, `/about`.

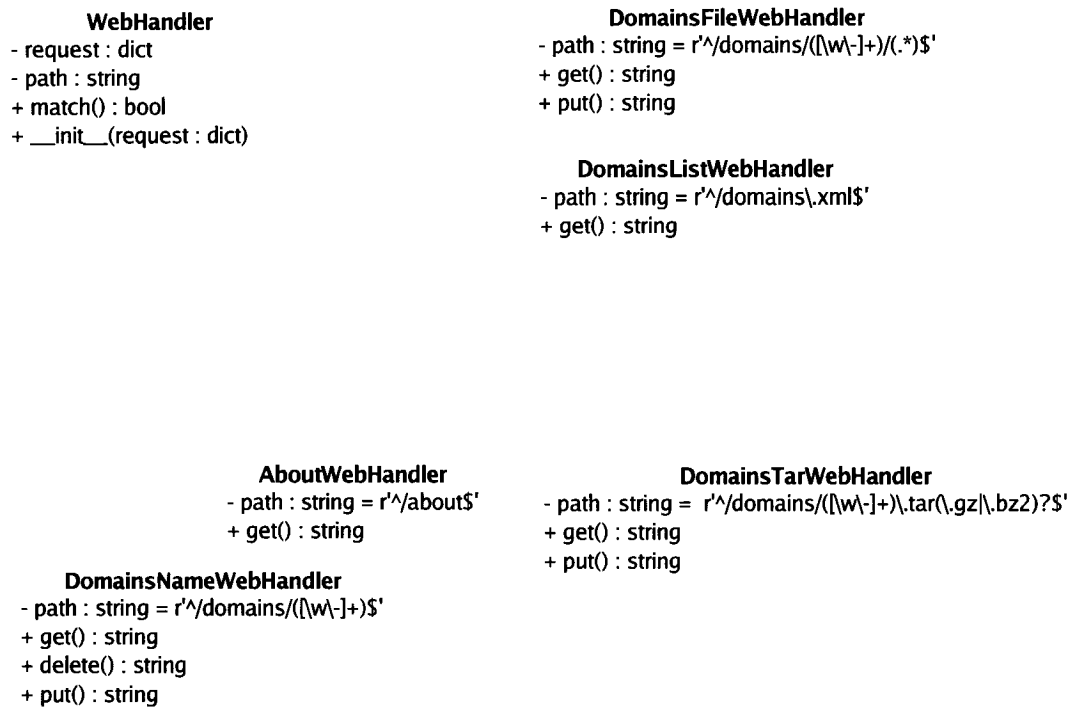

Figure 3.2 Schéma UML de la classe WebHandler et des classes héritières

Tous ces classes héritent de `WebHandler`, classe très simple, que le schéma UML 3.2 représente.

Globalement, une requête HTTP est donc d'abord reçue par le serveur *Apache*, qui la passe à `napalm.cgi`. Ce script CGI se charge ensuite de déterminer si une instance de classe est en mesure de répondre à la requête en comparant le chemin d'accès de cette dernière avec les expressions régulières (cf. variable `path`) contenues dans ces classes. La figure 3.3 résume les différentes étapes.

Si une instance d'une des classes dérivées est en mesure de répondre à une requête, on doit alors invoquer la méthode associée à la primitive HTTP utilisée et selon le type MIME. Le relation entre la requête et la méthode se fait en normalisant

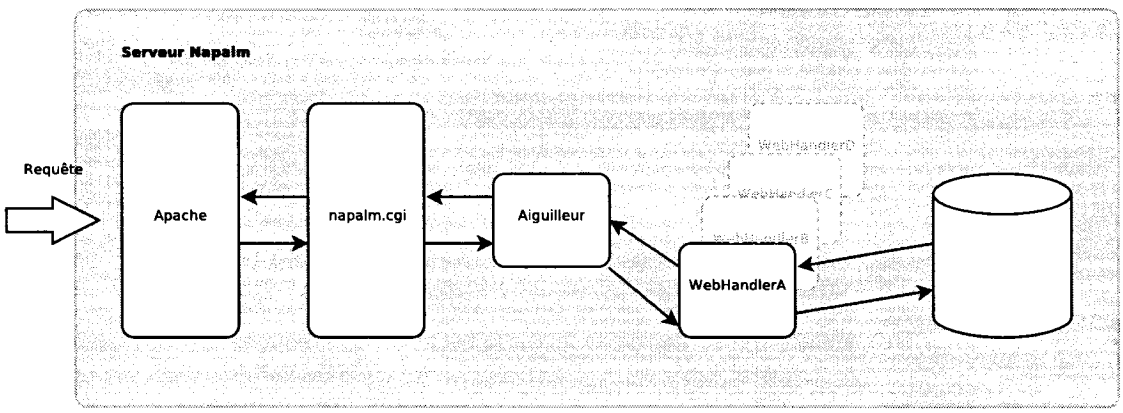

Figure 3.3 Une requête traitée par *Napalm* sera aiguillée à la bonne instance de classe et les informations reçues seront stockées sur le disque, pour consommation éventuelle par le démon *Napalm*, qui mettra l'état des machines virtuelles hébergées sur l'ordinateur à jour.

d'abord la requête selon son « verbe » (primitives CRUD PUT, GET, POST, DELETE) puis son type MIME. Par exemple, à la réception d'un `GET` de type MIME `text/html`, on essaiera d'abord d'invoquer la fonction `get_text_html()`, puis, si cette dernière n'existe pas, la méthode `get()`. Si celle-ci n'existe pas non plus, on retourne une erreur HTTP au client ayant fait la requête.

Les informations qui transitent à travers l'interface HTTP de *Napalm* sont traitées, nettoyées et finalement stockées sur disque dans un répertoire de travail. Un démon se charge ensuite des changements d'états des machines virtuelles.

3.3.2 Démon de contrôle

Un service constant d'arrière-plan (démon) a été développé pour traiter les requêtes demandant le démarrage et l'arrêt des machines virtuelles. Ce démon s'exécute en boucle en arrière-plan et surveille par lecture répétitive les différents fichiers posés

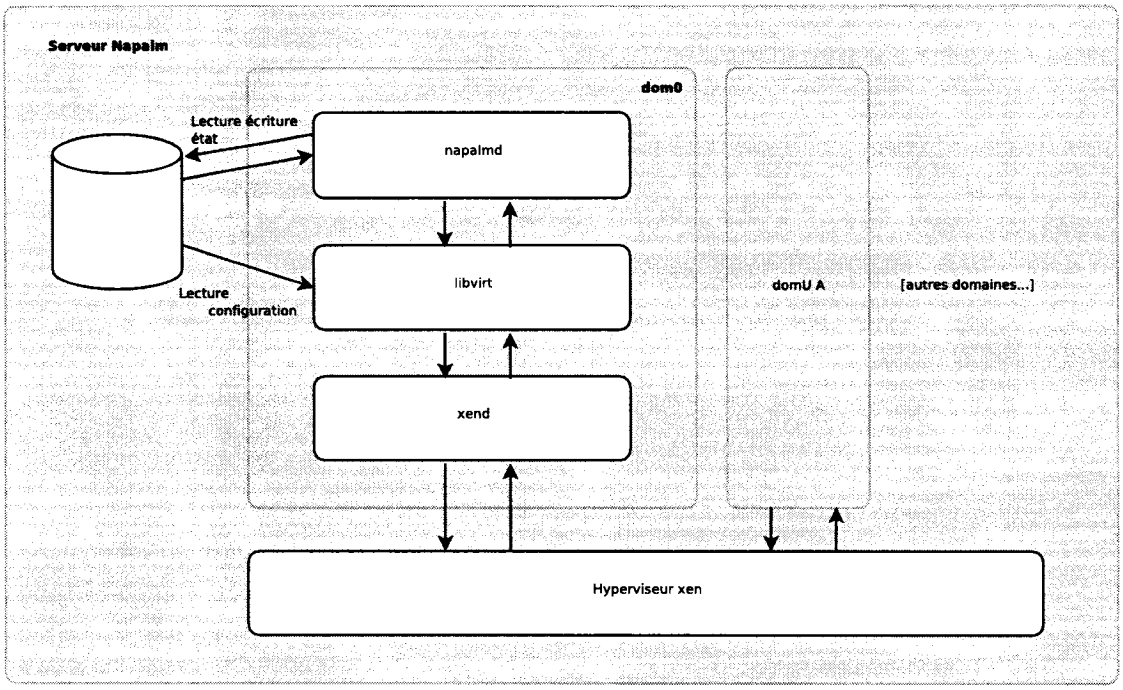

Figure 3.4 Schéma des différents éléments reliés au contrôle d'état des machines virtuelles hébergées sous *Xen*

dans le répertoire de travail de *Napalm* par l'interface HTTP.

Ce démon réagit aux changements d'états des machines virtuelles en envoyant à *libvirt* les commandes nécessaires pour démarrer ou arrêter les domaines. Cette librairie fera à son tour la lecture de la configuration du domaine donné et se chargera de paramétrer les outils de virtualisation.

La figure 3.4 montre les différents éléments en jeu en ce qui a trait au système de contrôle d'état des machines virtuelles. Dans un contexte où *QEMU* serait plutôt utilisé, on aurait alors un schéma tel que celui présenté dans la figure 3.5.

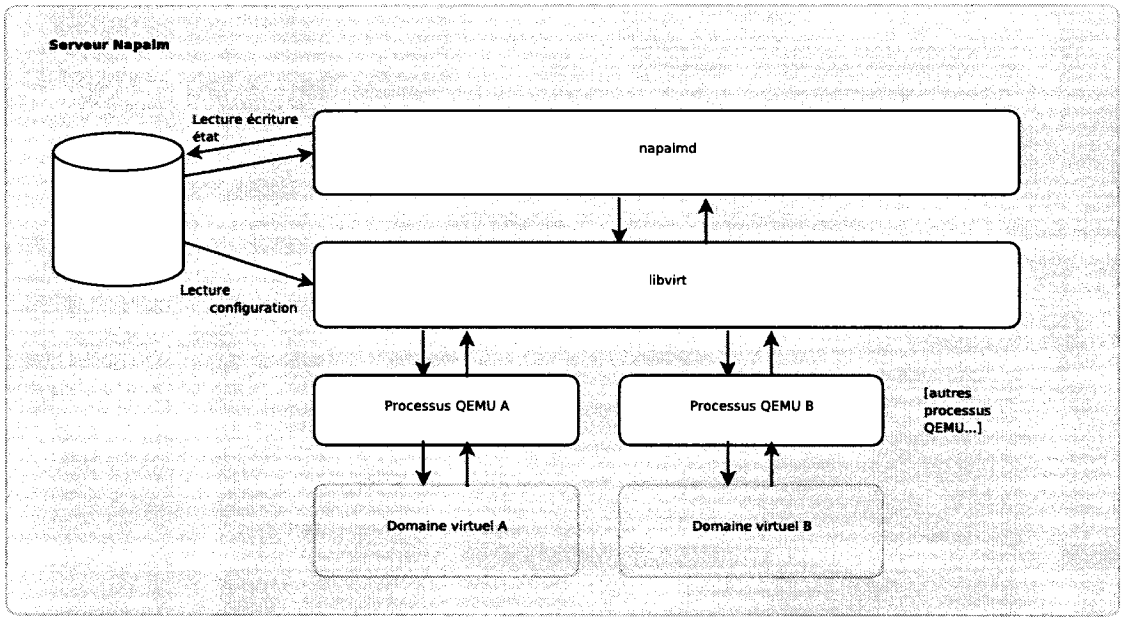

Figure 3.5 Schéma des différents éléments reliés au contrôle d'état des machines virtuelles hébergées sous *QEMU*

3.3.3 Contrôle d'état

Ayant défini une architecture REST et les schémas XML associés, le contrôle d'état se fait de manière très simple, soit en accédant en lecture au fichier représentant l'état actuel de la machine virtuelle, à l'aide de la primitive CRUD *read*, ou HTTP GET dans notre implémentation.

Ce fichier d'état suit le schéma simple présenté au listage 3.5.

À l'aide de la primitive HTTP PUT, il est possible de modifier l'état d'une machine virtuelle donnée : on envoie un nouveau fichier d'état suivant le bon schéma au serveur. Celui-ci change en conséquence l'état de la machine virtuelle.

Listage 3.5 Schéma XML du fichier de statut

```
1  <?xml version='1.0'?>
2  <xs:schema
3    xmlns:xs="http://www.w3.org/2001/XMLSchema">
4    <xs:element name="status" type="Status"/>
5    <xs:complexType name="Status">
6      <xs:sequence>
7        <xs:element name="state" type="xs:string"/>
8      </xs:sequence>
9    </xs:complexType>
10 </xs:schema>
```

3.3.4 Surveillance des nœuds

Pour la surveillance de l'état des clones déployés, nous utilisons le logiciel *Ganglia* avec des communications point-à-point. *Ganglia* mesure l'état d'un système et retourne une quarantaine de métriques permettant de connaître la charge du processeur, l'utilisation de la mémoire, du réseau, du disque, en plus de servir comme moniteur de « battements cardiaques », permettant de distinguer quels clones sont toujours accessibles de ceux qui sont tombés.

Selon le modèle de *Ganglia*, chacun des nœuds surveillé contient un observateur d'état, `gmond`, qui effectue des mesures sur l'état du nœud à intervalles réguliers.

Dans notre modèle serveur-clones, il est nécessaire de transmettre les informations mesurées sur chacun des clones au moniteur `gmond` du serveur. Ce dernier `gmond` transmet quant à lui les informations rassemblées à un second démon, `gmetad`, qui se charge d'interpréter les informations reçues et de les fournir en format XML à l'usager en mode console, ou encore à une application Web qui génèrera un résumé de l'état de toutes les machines impliquées. La figure 3.6 résume ces échanges d'informations et la figure 3.7 donne une exemple de page Web généré, donnant l'état des nœuds d'une grappe virtuelle de clones.

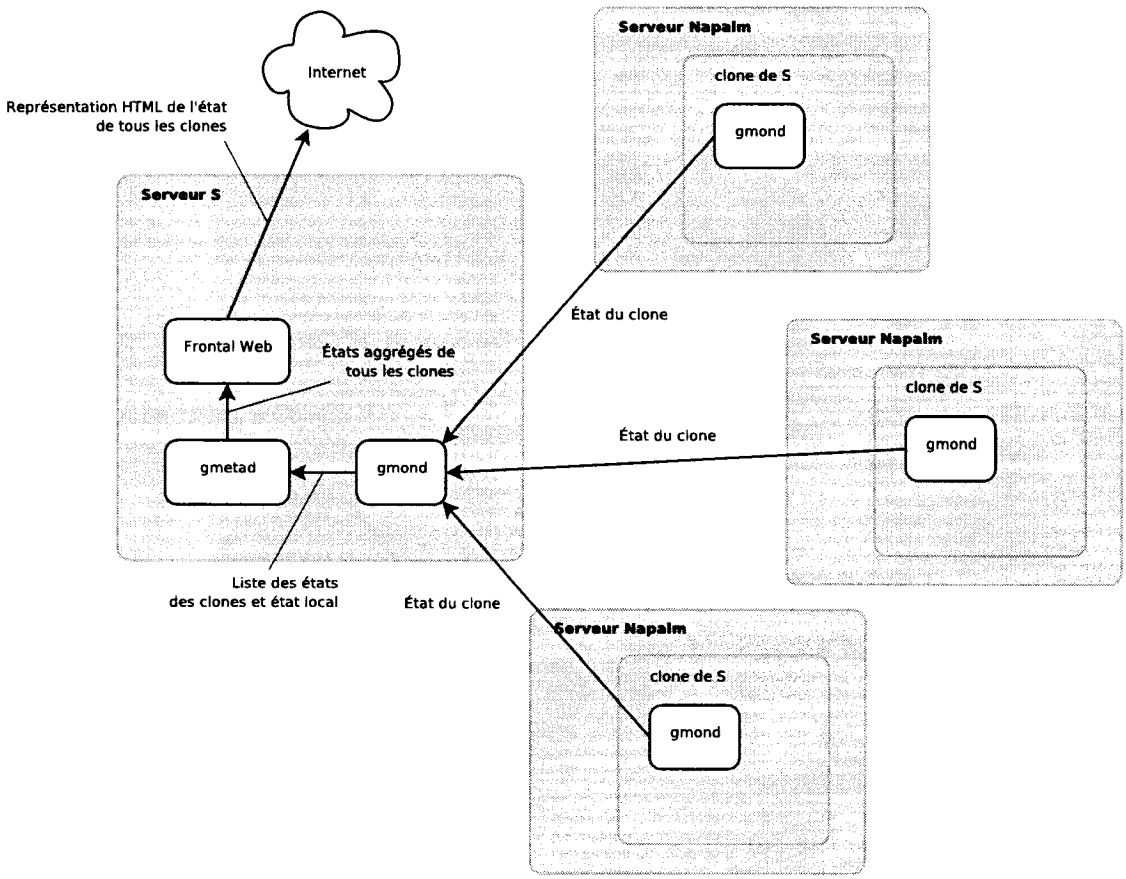

Figure 3.6 Transferts d'information entre les différents composants de *Ganglia*

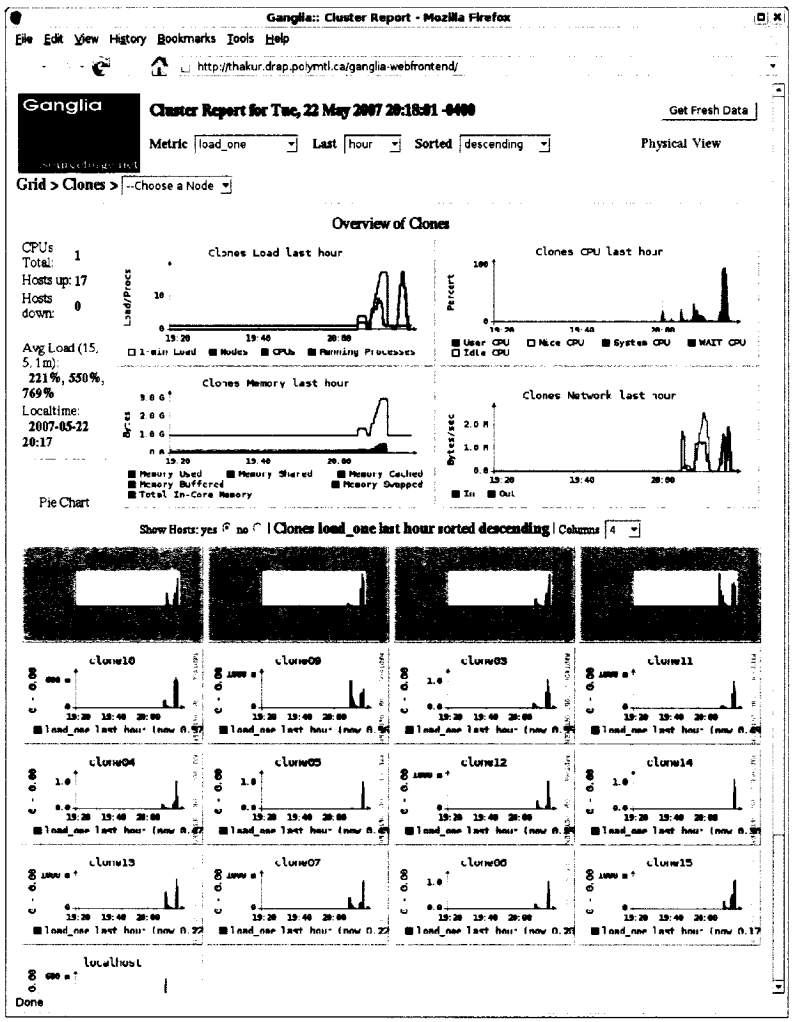

Figure 3.7 Capture d'écran de la page générée par le frontal Web de *Ganglia*

3.3.5 Publication et découverte

Nous avons défini un type de service DNS-SD, « `_napalm-http._tcp` », et publions ses instances via l'implémentation *Avahi* (AVAHI, 2007). *Avahi* observe les protocoles DNS-SD et mDNS. À l'aide des interfaces de programmation que *Avahi* rend publiques, nous pouvons d'abord publier à l'échelle d'un réseau qu'une machine donnée offre le service d'exécution de machines virtuelles via *Napalm*. En second lieu, nous pouvons également utiliser Avahi pour découvrir ces mêmes machines : l'implémentation nous permet d'accéder à une liste des services offerts sur le réseau en maintenant une base de données interne contenant les associations service-hôte qui ont été publiées.

3.4 Couplage *Clone* + *Napalm*

Le couplage des deux outils développés permet de déployer des grappes de clones virtuels légers. La figure 3.8 offre un survol de l'intégration de ces deux logiciels et des composants en jeu.

3.5 Boîte d'outils

Nous avons implémenté des outils console pour permettre de gérer la création et la mise à jour de clones de systèmes d'exploitation. Le déploiement de systèmes et leur découverte est aussi simplifiée par un groupe d'outils console.

3.5.1 *Clone* 0.1

Cette section présente un survol des outils usagers du logiciel *Clone* :

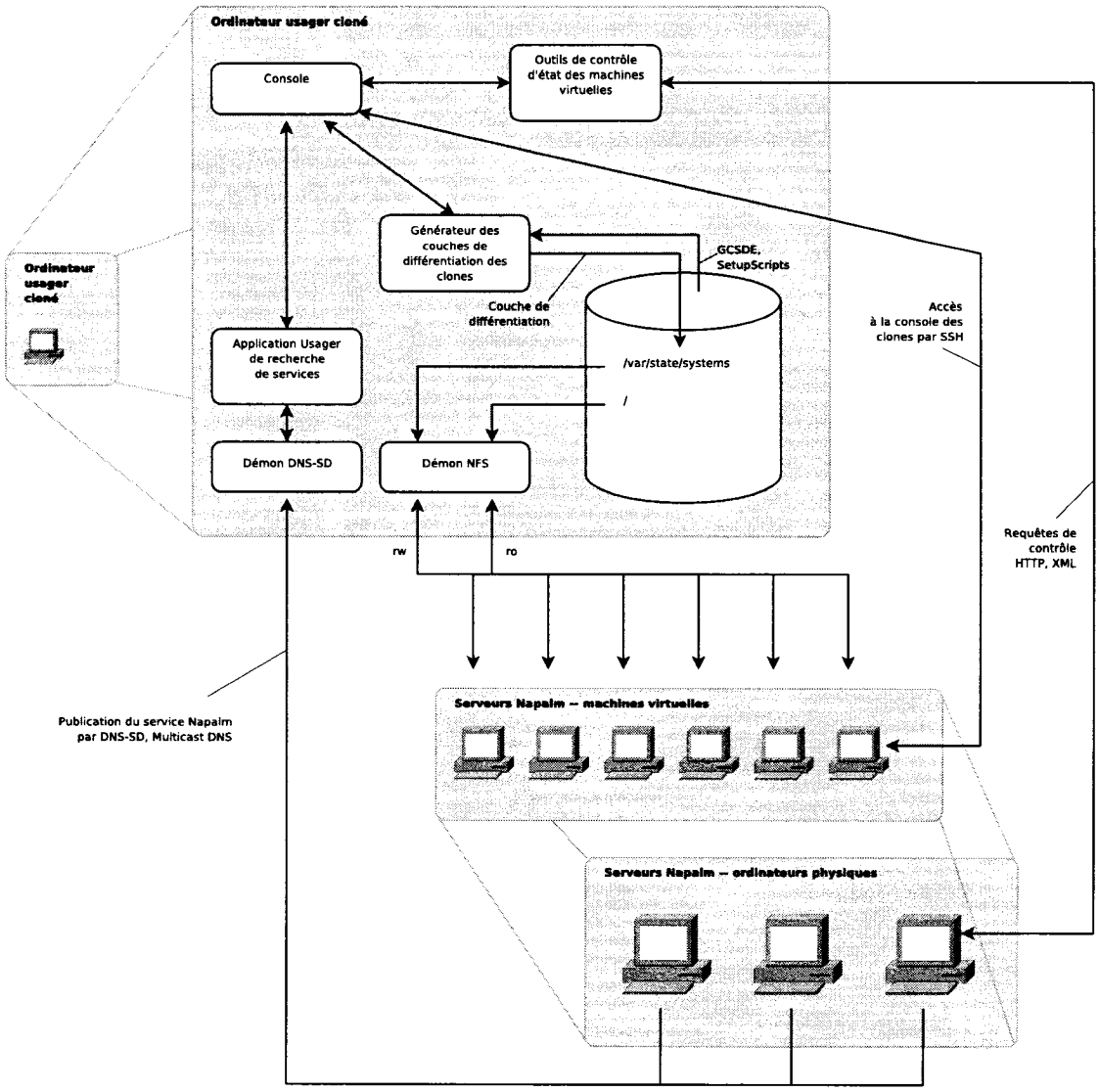

Figure 3.8 Exploitation des outils *Clone* et *Napalm* en tandem : composants de découverte, de création de couches de différentiation, de déploiement réseau et de partage de systèmes de fichiers.

clone Génère la couche supérieure à superposer sur la racine du système d'exploitation du nœud référence à partir de données de configuration fournies dans `/etc/clones.conf` ;

clone-qemu Démarre un clone donné dans une machine virtuelle sur le système de l'usager. Utile pour déverminer le processus de démarrage d'un clone donné ;

clone-list Retourne la liste des clones qui sont enregistrés, sous différents formats, dont le format `/etc/hosts`, ou encore au format du fichier *hostfile* de *OpenMPI*. Utile pour fournir les informations à `mpirun` en l'invoquant ainsi :

```
$ mpirun --hostfile <(clone-list --mpi-hostfile) -np 8 ...
```

clone-update-hosts-file Se charge de mettre à jour le fichier `/etc/hosts` à partir des noms et adresses IP des clones enregistrés ;

clone-update-known-hosts Se charge de mettre à jour le fichier des signatures des hôtes distants SSH (`/etc/ssh/ssh_known_hosts`) à partir des noms et adresses IP des clones enregistrés.

Des jeux de gabarits pour générer une couche de différentiation sont fournis.

3.5.1.1 Gabarit pour *Gentoo GNU/Linux*

Pour différencier un clone sous la distribution *Gentoo GNU/Linux*, un GCDSE est fournie. Le listage 3.6 présente l'arborescence de ce gabarit.

Voici la description de certains de ces fichiers :

dump-clone Permet de copier intégralement le contenu du système clone, tel que vu par l'usager (c'est-à-dire tel que vu d'un observateur au-dessus de la couche supérieure des unions) vers un répertoire donné. Permet entre autres de copier des systèmes clones sur disques durs locaux ;

Listage 3.6 GCDSE pour un système *Gentoo GNU/Linux* de base

```
clone-extension-gentoo-defaults-0.1
|-- setup_scripts
|   `-- LoggerSetupScript.py
`-- template
    |-- etc
    |   |-- conf.d
    |   |   |-- hostname.tmpl
    |   |   `-- net
    |   |-- fstab.tmpl
    |   |-- hosts.tmpl
    |   |-- init.d
    |   |   `-- register-clone
    |   |-- modules.autoload.d
    |   |   `-- kernel-2.6.tmpl
    |   `-- runlevels
    |       |-- boot -> clone-boot
    |       |-- clone-boot
    |       |-- clone-default
    |       |   `-- register-clone -> /etc/init.d/register-clone
    |       `-- default -> clone-default
    |-- sbin
    |   |-- dump-clone
    |   |-- register-clone
    |   `-- unregister-clone
    |-- tmp
    |   `-- .wh.__dir_opaque
    `-- var
        |-- log
        |   `-- .wh.__dir_opaque
        |-- run
        |   `-- .wh.__dir_opaque
        `-- tmp
            `-- .wh.__dir_opaque
```

register-clone Stocke des informations sur le clone dans un répertoire accessible par le système de référence. Cela permet, sur le serveur, de lister les clones et d'obtenir de l'information sur leur configuration ;

unregister-clone Remet à zéro les informations stockés à propos du clone ;

***.tmpl** Fichiers gabarits qui seront interprétés par *Cheetah*. Le résultat de l'interprétation sera écrit dans un fichier portant le même nom, sans l'extension `.tmpl`.

Les liens symboliques dans `runlevel` nous permettent de configurer deux *runlevels* nommés, `clone-boot` et `clone-default` directement sur le système de fichiers du serveur. Ils sont traités comme n'importe quel *runlevel* par les outils de gestion, mais seront seulement appelés au démarrage d'un nœud clone. Grâce aux liens symboliques, au démarrage du clone, le système d'initialisation se trouvera en fait à exécuter les scripts du niveau `clone-boot` lorsqu'il lancera le niveau `boot`, et `clone-default` lorsqu'il lancera le niveau `default`.

Nous tirons donc profit du sytème d'initialisation de *Gentoo GNU/Linux* pour simplifier la gestion des rôles des clones et les services qu'ils démarrent. Les ajouts et retraits de services à exécuter sur les clones se font alors directement avec

```
serveur $ rc-update add clone-default mon_service
serveur $ rc-update del clone-default service_a_retirer
```

Dans une distribution utilisant des scripts d'initialisation à la *System V*, il faudrait plutôt redéfinir les *runlevels* numérotés par superposition à l'aide d'un GCDSE. En pratique, il faudrait superposer une couche contenant une redéfinition des répertoires `rc*.d`.

3.5.2 Autres gabarits fournis

Des gabarits ont été développés pour permettre la différentiation appropriée de clones pour supporter quelques autre services :

clone-extension-dummy-0.1 Permet de déverminer la génération de clones ;

clone-extension-napalm-0.1 Contient un script héritant de `SetupScript` pour créer automatiquement les fichiers nécessaires au démarrage d'un clone par les outils *Napalm* ;

clone-extension-postgresql-0.1 Contient les informations nécessaires pour générer une couche supérieure permettant l'exécution de PostgreSQL (pour nos tests *OSDB*, cf. section 4.2.1).

3.5.3 *Napalm* 0.1

Cette section présente un survol des outils usagers du logiciel `napalm` :

napalm Script faisant l'abstraction des primitives de communication avec un serveur *Napalm* pour permettre à un usager d'interagir facilement avec un tel serveur. Permet de démarrer, arrêter, lister, modifier des domaines distants ;

napalmd Démon mettant à jour le statut des machines virtuelles selon les requêtes des usagers ;

napalm-browse Retourne des informations sur les serveurs *Napalm* présents sur un réseau local ;

napalm-deploy Permet de déployer un groupe de domaines sur un groupe de serveurs *Napalm*. Ces déploiements peuvent être nommés ;

napalm-list-deployments Retourne la liste des déploiements préalablement effectués ;

napalm-undeploy Permet d'arrêter et supprimer les machines virtuelles d'un déploiement préalablement effectué.

Ces scripts s'appuient à leur tour sur des modules *Python* fournis.

La configuration d'un domaine est représentée comme à la figure 3.9 lorsqu'on accède à un serveur *Napalm* à l'aide d'un fureteur Web. Cet aspect visuel est le résultat d'une transformation par *Extensible Stylesheet Language* (XSL) des fichiers XML *libvirt* de base (un ensemble de fichiers de support à placer sur la racine Web accessible grâce à *Apache* est fourni).

3.5.4 Noyaux

Étant donnée la caractéristique de *compatibilité*, exposée à la section 1.2, on peut aussi pallier à la nécessité de configurer et compiler un noyau sur chaque site en fournissant, avec les outils développés, un kernel pour l'architecture *i686*[2] en domaine U, contenant les pilotes pour les différents périphériques de *frontend Xen*. Ce noyau peut être utilisé partout où un hyperviseur *Xen* et un noyau en domaine 0 contenant le support de *backend* sont présents.

Le même principe peut s'appliquer pour *QEMU* quant à l'aspect *compatibilité* puisque les systèmes hébergés sous ce logiciel ont accès a un jeu de périphérique fixe et constant (cf. page de manuel de *QEMU*). Nous fournissons un noyau *i686* pour utilisation avec *QEMU*, mais il est envisageable de simplement réutiliser le noyau local servant à gérer une machine physique donnée.

Ces deux noyaux fournis contiennent également en leur sein un *initramfs* produit

[2]Jeu d'instruction des processeurs *Intel Pentium Pro* ou *AMD Athlon*, supporté sur tous leurs contemporains.

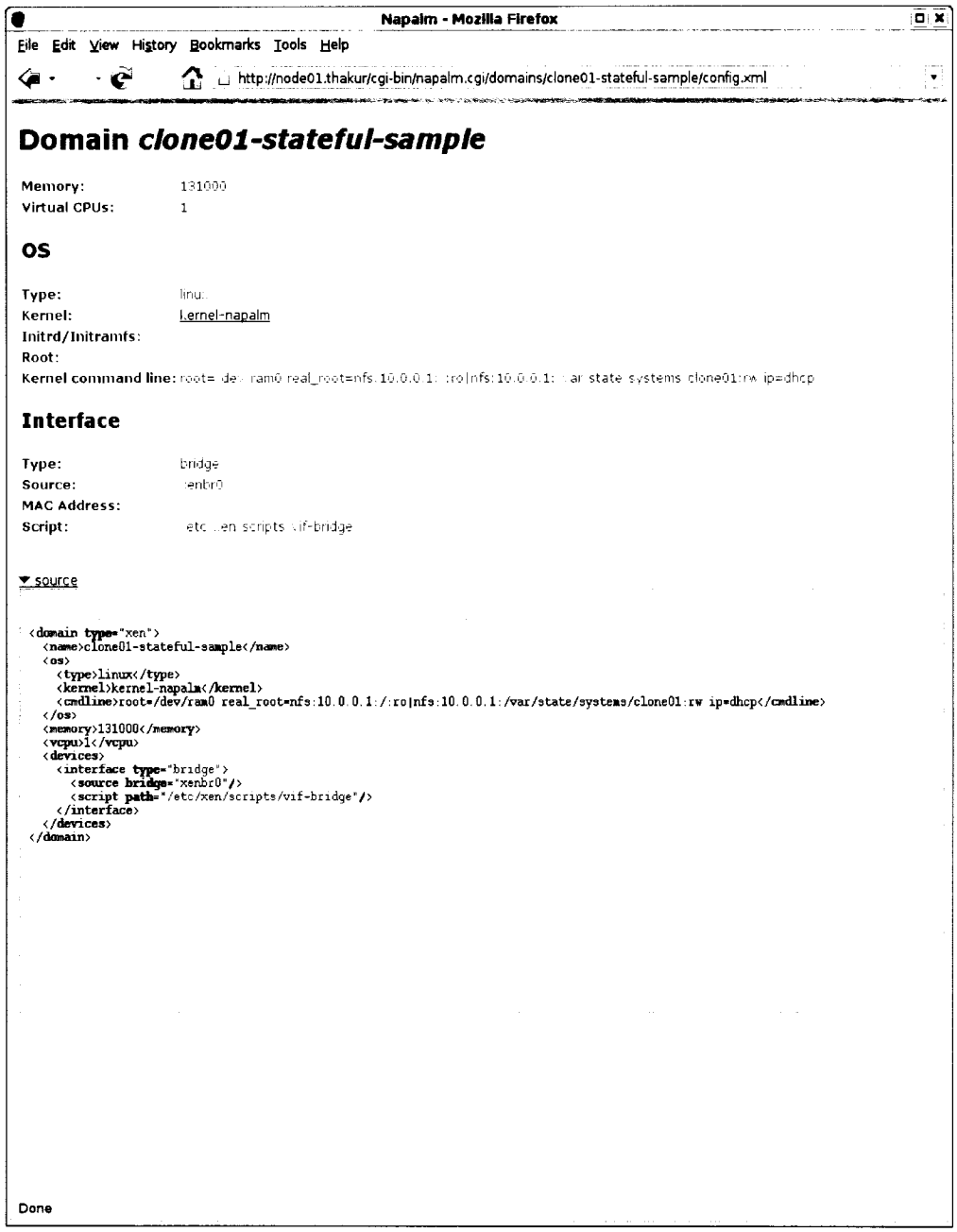

Figure 3.9 Capture d'écran de `/domains/clone01-stateful-sample/config.xml` tel que vu par un usager pointant son fureteur sur un serveur *Napalm*. Représentation du domaine `clone01-stateful-sample`. Le XML source est affiché dans l'encadré au bas de la figure.

par le *Genkernel* modifié par nos travaux.

3.5.5 Installation

Nous avons développé plusieurs *ebuilds* pour *Gentoo GNU/Linux* qui permettent d'installer les paquetages *Napalm* et *Clone* et de télécharger toutes les dépendances nécessaires.

En suivant le format préconisé dans les fichiers *ebuilds*, la liste de dépendances pour *Clone* est :

- `>=dev-lang/python-2.4`
- `>=dev-python/pyyaml-3.0`
- `>=dev-python/cheetah-0.9`
- `>=net-fs/nfs-utils-1.0`
- `virtual/dhcpc`

La liste de dépendances directes pour *Napalm* est :

- `>=app-text/xmlstarlet-1.0`
- `>=net-misc/curl-7.0`
- `>=net-dns/avahi-0.6.16`
- `>=dev-lang/python-2.4`

S'ajoute à cette liste, dans le cas où l'on veut installer la portion serveur de *Napalm* :

- `>=app-emulation/xen-tools-3.0`
- `>=app-emulation/libvirt-0.1.3`
- `>=net-www/apache-2.0`

Le *use flag* « `server` » contrôle si cette portion est installée ou non. Elle contient `napalmd` et tous les fichiers nécessaires au fonctionnement du script `napalm.cgi`.

L'administrateur peut donc installer tous les paquetages simplement en faisant :

```
$ emerge clone napalm
```

Cette commande affichera à la fin les quelques étapes restantes pour compléter l'installation. Nous les montrons ici pour permettre de juger de l'effort requis lors de l'installation des logiciels développés.

Pour *Clone* :

```
 * Please review /etc/clones.conf.example and create your own
 * /etc/clones.conf configuration file

 * Make sure to export these directories with NFS:
 *  - / as read-only
 *  - /var/state/systems as read-write
 *  - /home as read-write (optional)

 * Two new runlevels contain the services to start on clone startup:
 *  - clone-default
 *  - clone-boot
```

Pour *Napalm*, avec la portion serveur :

```
 * Please add the following directives in your Apache configuration:
 *
```

```
* <Location "/cgi-bin/napalm.cgi">
*     SetEnv NAPALM_ROOT "/var/napalm"
* </Location>
*
* The typical file in which to add this is
*     /etc/apache2/vhosts.d/00_default_vhost.conf
* in the DEFAULT_VHOST declaration.

* Make sure to set the user id and group id of /var/napalm to match
* your web server's user id and group id.

* Make sure to add napalmd in your runlevel scripts:
*     $ rc-update add napalmd default
* and to set the proper permissions in /etc/conf.d/napalmd
* so that napalmd has read/write access to /var/napalm

* Make sure Xen is configured for access through a UNIX server.
* In /etc/xen/xend-config.sxp:    (xend-unix-server yes)
```

Globalement, nous croyons que la liste relativement courte de dépendance et la simplicité du processus d'installation sont garants d'une adoption facile et rapide. Le plus important problème à ce niveau est peut-être la nécessité d'avoir un hyperviseur *Xen* et un noyau de domaine 0 pour obtenir une bonne performance. On peut aussi utiliser *QEMU*. Nous évaluerons au prochain chapitre les aspects de performance reliés à ces deux ensembles logiciels.

CHAPITRE 4

ÉVALUATION

Ce chapitre analyse les performances des systèmes virtuels déployés et les compare avec des systèmes natifs fonctionnant directement sur le matériel. On quantifie en outre les efforts requis pour déployer, gérer et surveiller une grappe de nœuds virtuels avec les outils présentés précédemment.

Nous débuterons par une analyse des performances générales de systèmes clonés. Nous déterminerons où sont les goulots d'étranglement et discuterons des améliorations possibles. Il ne s'agit en rien d'un profilage complet et exhaustif de la solution : le but est d'obtenir un comparatif de performances pour des cas communs et déterminer si elle est viable et où se trouvent grossièrement les faiblesses. Des éléments de recherche plus poussés sur certains aspects de la performance sont également décrits.

Nous chercherons ensuite à déterminer les failles de sécurité possibles pour le système développé plus tôt et proposerons des méthodes permettant de colmater ces brèches.

Nous terminerons par une évaluation des outils développés pour déployer et configurer les clones sur les grilles/grappes. Cette évaluation témoignera du degré d'utilisabilité de la suite d'outils.

4.1 Ordinateurs de test

Les tests sont effectués sur un groupe de cinq ordinateurs possédant chacun un *AMD Athlon XP 2500+* (cadencé à 1800MHz) et 1 Go de mémoire vive. Ils sont connectés entre eux derrière un frontal, `thakur`, sur un réseau *Fast Ethernet* privé. On peut se référer à l'annexe I pour plus de détails.

Les systèmes physiques et virtuels utilisent des noyaux en version 2.6.16.28 compilés sans rustines ou avec les rustines *Xen*[1] en domaine 0 ou domaine U. Nous utilisons une librairie C (*Glibc*) en version 2.5 et les outils et le module *UnionFS* en version 1.2. Les développeurs de *Gentoo* recommandent d'utiliser cette version avec tout noyau 2.6.16.

L'ensemble du système de test est installé avec *Gentoo Portage*. Les paquetages sont compilés avec GCC 4.1.1. GCC, lors de la compilation des paquetages, est paramétré avec `-march=athlon-xp -O3 -pipe -mno-tls-direct-seg-refs`. Le dernier paramètre étant recommandé par GENTOOWIKI (2007) pour optimiser la performance sous *Xen*, particulièrement pour la librairie C (XENWIKI, 2007).

Nous utilisons les abbréviations suivantes pour les résultats présentés dans ce chapitre et dans les annexes :

TH	`thakur`
N2	`node02`, domaine 0, 1 Go de mémoire
N4	`node04`, limité à 512 Mo de mémoire
clTH@N2	clone de `thakur`, sur `node02`, 512 Mo de mémoire

Pour comparer la performance d'un clone de `thakur`, nous prenons, à moins d'indi-

[1] *ebuild* `sys-kernel/xen-sources-2.6.16.28`, le plus récent des noyaux *Xen* disponible sous Gentoo lors de nos travaux.

cations contraires, l'exécution des bancs sur N4 comme référence de la performance d'un système non-virtualisé ayant les mêmes caractéristiques que le clone. Nous utilisons alors un noyau sans rustine sur N4.

4.2 Bancs de test sous Xen

Les mesures de performance sont calquées sur les mesures effectuées dans la littérature. S'inspirant de BARHAM et al. (2003); CLARK et al. (2004), nous utilisons ces bancs d'essais, tous disponibles et sous des licences libres :

Compilation du kernel Comme étalon de performance générale, nous mesurons le temps requis pour compiler le kernel 2.6.16.36 dans sa configuration par défaut ;

DBench Nous mesurons la performance du système de fichiers à l'aide de *DBench* (TRIDGELL, 2007) en version 2.0 ;

FreeBench*, *FourInARow Utilisé pour tester la puissance de calcul brute. En version 1.03 ;

OSDB Une suite de tests de performance de base de données, utilisée par BARHAM et al. (2003). En version 0.21, sur la base de données PostgreSQL 8.0.8 ;

Netperf Suggéré par MORIN (2006), il permet de quantifier la performance en transferts de données sur TCP/IP entre deux machines. Utilisé en version 2.3 (HP-NETWORK-DIVISION, 1995) ;

Tachyon Logiciel de rendu de scène par tracé de rayons, par STONE (2007). Nous mesurons le temps requis pour la génération distribuée d'une image. Utilisé en version 20070319, fonctionnant sur la librairie *OpenMPI* version 1.2.1.

Tous les bancs de tests sont effectués sur des systèmes sans charge autre que les services normaux du système d'exploitation. Les systèmes sont configurés pour ne pas démarrer de tâches programmées en arrière-plan. À moins d'indications contraires, au moins six exécutions sont effectuées pour chacun des bancs de test. La première exécution n'est pas tenue en compte puisqu'effectuée en « cache froide ».

Le lecteur interessé trouvera les tableaux de résultats bruts sous *Xen* en annexe I.

4.2.1 Performance générale, compilation du kernel

Le temps requis pour compiler un noyau est une mesure employée pour déterminer la performance générale d'un système. La compilation du noyau demande de la puissance de calcul ainsi que de nombreux accès mémoires et système de fichiers.

Nous avons compilé le kernel *Linux* en version 2.6.16.36, sans rustine et dans sa configuration par défaut.

La figure 4.1 montre les résultats obtenus lors de la compilation du kernel. Globalement, les noeuds clones sont moins performants qu'un noeud physique équivalent, le noeud sans mémoire d'état offrant une meilleure performance que le noeud avec mémoire d'état. Le temps de traitement système est significativement plus grand sur les clones, illustrant le fait que le passage des données à travers la combinaison *UnionFS* et NFS implique plus de traitement noyau qu'un accès direct à un disque.

Nous avons également utilisé les mesures de performances fournies par le logiciel *OSDB* pour l'évaluation de bases de données. Nous retenons les mesures *OSDB Information Retrieval* (IR) et *OSDB* On-Line Transaction Processing (OLTP), comme l'ont fait BARHAM et al. (2003) et CLARK et al. (2004).

Ces deux tests sont, selon SUCHOMSKI (2002) :

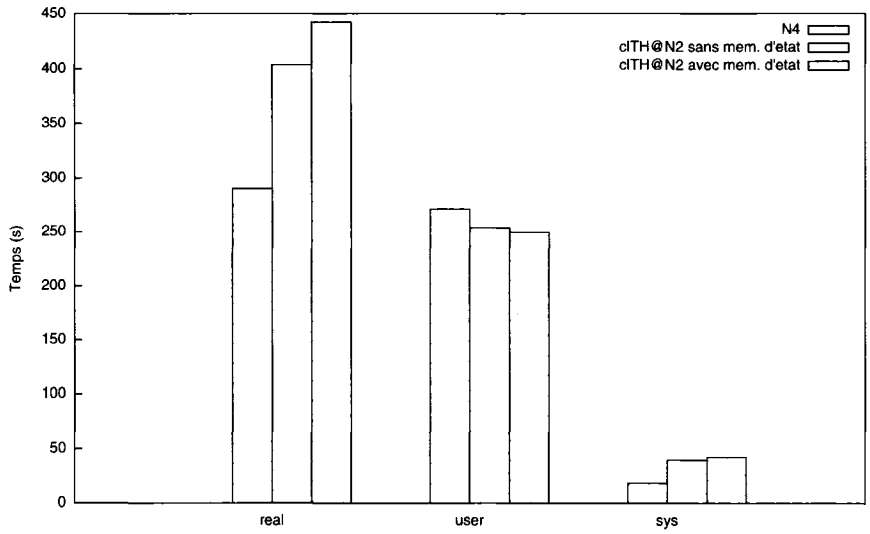

Figure 4.1 Temps requis pour la compilation du noyau sur N4, clTH@N2 sans mémoire d'état et clTH@N2 avec mémoire d'état

> *Mixed IR - selects a tuple from a table with a given condition that is randomly generated from all possible values stored in the table.*

et

> *Mixed OLTP (On-Line Transaction Processing) - updates a tuple which fulfills a condition randomly selected, in a table.*

La figure 4.2 présente une comparaison des performances. Les résultats obtenus sont le nombre de tuples traités par secondes. Notons qu'il a été impossible de compléter *OSDB-IR* et *OSDB-OLTP* sur un nœud sans mémoire d'état : les tests nécessitaient trop de mémoire (les fichiers de tests et de la base de données sont stockés, rappellons-le, en mémoire sur `tmpfs`).

La performance *OSDB-OLTP* est mauvaise pour les clones à mémoire d'état, comparativement à un test effectué sur une machine physique avec accès direct à un

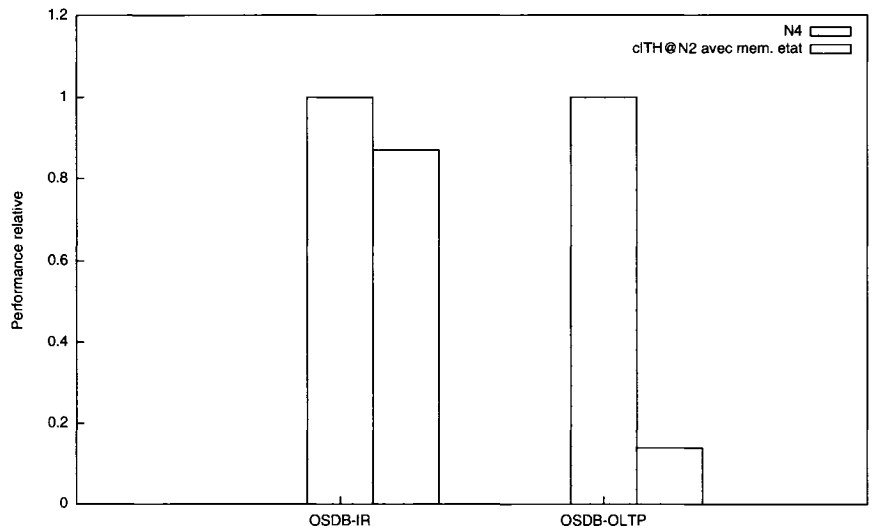

Figure 4.2 Performance relative en tests *OSDB* de TH et clTH@N2 avec mémoire d'état

disque. Selon BARHAM et al. (2003), « *OSDB-OLTP requires many synchronous disk operations, resulting in many protection domain transitions.* » Or, toujours selon l'auteur, il s'agit là de l'explication des piètres performances sur *VMWare* et *User Mode Linux*, et non sur *Xen*, qui affiche une performance relative à un système *Linux* natif très respectable (environ 95% de la performance native).

Nous verrons à la section 4.4 que l'utilisation d'*UnionFS* peut expliquer une partie de cette perte de performance. La piètre performance en IO sur NFS est également un facteur important dans cette diminution de performance, comme on le voit à la section 4.2.3.

4.2.2 Performance réseau

Nous avons configuré *Xen* pour que soient connectées les interfaces réseau virtuelles au réseau physique via un pont (*bridge*) logiciel mis en place dans le domaine 0. On

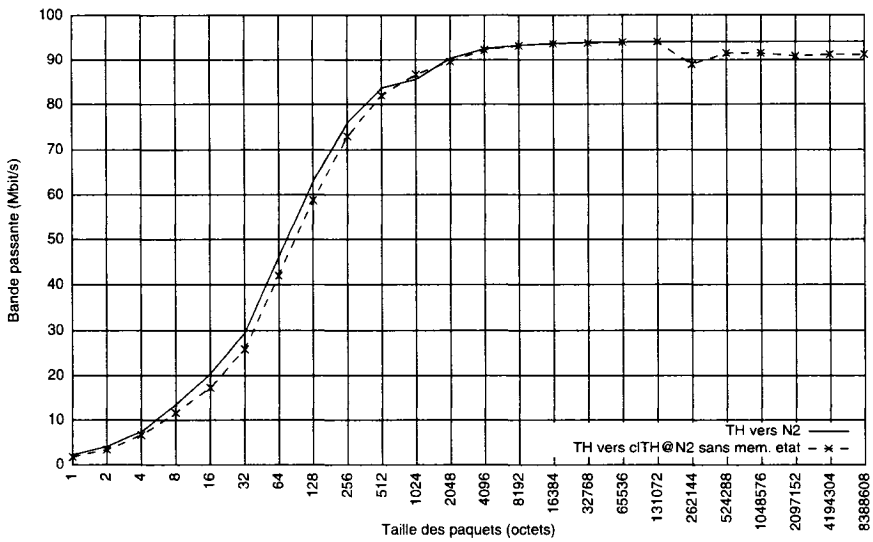

Figure 4.3 Bande passante entre TH, N2 et clTH@N2

peut se référer à YU (2004) pour une étude de performance des ponts sous *Linux*.

La bande passante est calculée sur différentes tailles de messages. Chaque message est envoyé 50 fois et le temps moyen du transfert est retenu. Pour ces tests, les nœuds virtuels sont démarrés en mode sans mémoire d'état.

La figure 4.3 présente un graphique de la bande passante entre différentes paires de systèmes en fonction de la taille des paquets envoyés. On atteint pratiquement une performance native.

Il faut bien sûr mettre ces résultats en perspective. Le matériel *Fast Ethernet* utilisé est relativement limité par rapport aux offres les plus performantes sur le marché actuel.

Différents profilages détaillés effectués par MENON et al. (2005, 2006) montrent des faiblesses importantes dans les performances réseau, en comparaison avec des systèmes natifs.

À considérer également : dans le contexte de domaines virtuels, il est difficile d'utiliser à leur pleine efficacité certains matériels récents qui contiennent des mécanismes permettant des accès directs entre les applications usagers, sur des systèmes non-virtualisés, vers le matériel. À ce propos, dans le but d'améliorer les performances des applications gérées dans une machine virtuelle, le travail de LIU et al. (2006) propose un mécanisme de « *VMM-bypass* ». Il permet de transférer certaines données directement d'un domaine virtualisé à un périphérique physique sur la machine hôte. *Xen-IB* est développé dans le travail cité pour permettre à une application virtualisée d'obtenir une performance quasi-native sur *InfiniBand* (INFINIBAND-TA, 2007). De plus, l'approche *VMM-bypass* se veut suffisamment générique pour supporter différents types d'interconnexions de haute performance, offrant une piste de solution prometteuse au problème de la performance d'accès réseau.

4.2.3 Performance disque, *DBench*

Le logiciel *DBench* a été créé par Andrew Tridgell dans le cadre d'une réimplémentation libre de l'application d'étalonnage *NetBench*, utilisée dans l'industrie pour mesurer les performances de serveurs de fichiers. Nous avons mesuré la bande passante obtenue par un ou plusieurs clients parallèles, effectuant chacun 90000 opérations sur le système de fichiers. Nous avons fait 10 mesures pour chaque essai. Les essais ont été faits avec 1, 2, 3, 4, 5, 6, 9, 12, 15, 18, 21 ou 24 clients *DBench* simultanés, comme dans le travail de CLARK et al. (2004). La figure 4.4 montre l'écart entre les performances d'accès aux fichiers.

L'accès direct au disque donne au système TH un avantage sur l'accès via NFS en mode *avec mémoire d'état*. Le mode *sans mémoire d'état* bénéficie du fait que les fichiers créés par *DBench* sont stockés dans la mémoire vive de l'ordinateur

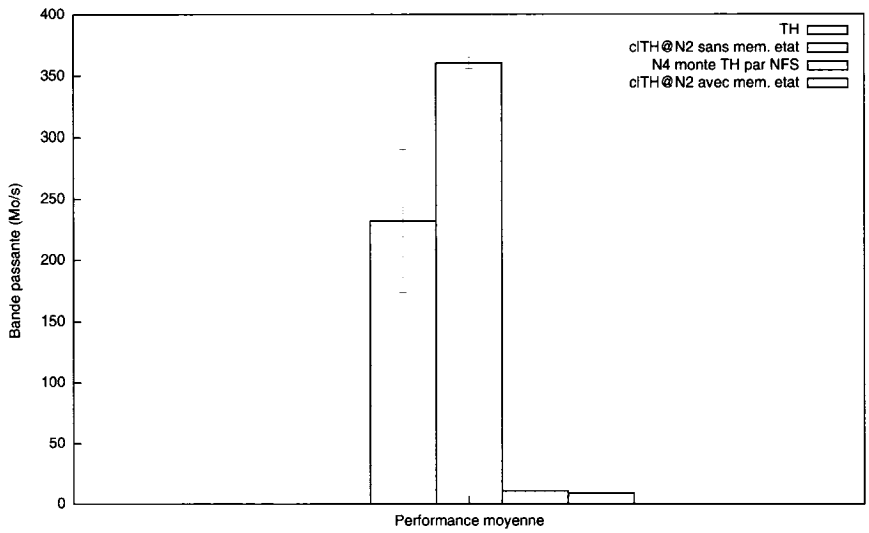

Figure 4.4 Bande passante en accès système de fichiers sur des nœuds natifs et virtualisés

(sur la couche supérieure `tmpfs`). Cela permet une accélération importante par rapport à l'utilisation d'un disque, *dans le cas où l'accès aux données est fait sur la couche supérieure en* `tmpfs` — les lectures sur les couches inférieures devraient se comporter comme pour le nœud à mémoire d'état. L'envers de la médaille de ce gain en performance, dans ce cas-ci, est évidemment qu'on consomme la mémoire utile du nœud virtuel avec le contenu des modifications locales au système de fichiers.

Afin justement de pallier à cette faiblesse, il a été nécessaire d'accroître la quantité de mémoire du clone pour mener à bien les bancs d'essais en mode sans mémoire d'état. Nous avons donc dû allouer 750 Mo pour permettre de terminer les tests *sans mémoire d'état.*

L'étude de ces résultats révèle un écart-type important quant aux données sur TH, comme le tableau 4.1 le montre. Ces données sont loin des résultats de CLARK et al. (2004) : « *Dbench has a standard deviation of 14% and 18% for native*

Tableau 4.1 Bande passante moyenne pour les tests de performance avec *DBench*

Système	B. pass. moy. (Mo/s)	Écart-type (% de la moy.)
TH	231.93	25.13
N4	10.94	2.91
clTH@N2 sans mémoire d'état	360.59	1.28
clTH@N2 avec mémoire d'état	9.00	3.19

Linux and XenoLinux [*Linux running in Xen*] *respectively.* » Les différences au niveau du matériel pourraient expliquer notre écart-type différent de celui de cette équipe. Notamment, ils ont utilisé des disques et un contrôleur SCSI, bénéficiant donc d'un système matériel de mise en queue des requêtes (TCQ) améliorant les performances, alors que nous étions sur un système IDE (ATA/ATAPI-5), n'offrant pas d'optimisation particulière sur l'ordonnancement des lectures et écritures selon l'état interne du disque dur.

L'étude des performances réseau, à la section 4.2.2, permet d'obtenir la bande passante *maximale* qu'on peut atteindre sur le matériel physique. Il s'agit de 94.10 Mbit/s, en mesurant avec *Netperf* les communications entre N4 et TH. En ce qui a trait à la bande passante d'accès à un système de fichiers monté par-dessus ce même réseau, on obtient, pour N4 montant un système de fichiers NFS à partir de TH, une mesure de 10.94 Mo/s (87.52 Mbit/s) à l'aide de *DBench*. La différence est de 7% du maximum atteint. En revanche, en mode *avec mémoire d'état*, *DBench* mesure 9.00 Mo/s (72.00 Mbit/s). En comparaison avec le maximum atteint sur *Netperf*, on a une différence de 22.10 Mbit/s, soit une différence de près de 23% par rapport au maximum. Nous étudierons plus loin l'impact de l'utilisation de couches *UnionFS*, qui peut expliquer cet écart entre l'accès direct par NFS et l'accès en mode *avec mémoire d'état*.

Des tests sur du matériel plus performant restent à être faits pour déterminer

si la performance du système de fichiers réseau est bornée par la performance des interfaces *Fast Ethernet* que nous avons utilisées. La différence entre les accès disques et la performance des systèmes de fichiers NFS exportant le contenu de ces mêmes disques indique une marge de manœuvre de près de 220 Mo/s en comparant la performance disque sur TH et NFS sur N4. Cela porte raisonnablement à croire qu'une amélioration du tissu réseau aurait un impact positif sur les performances en IO des nœuds clones.

Sur notre configuration, on peut cependant tirer de ces résultats que l'utilisation de nœuds virtuels peut signifier une perte importante en bande passante en accès au système de fichiers par rapport à des équivalents accédant directement à des disques. Les applications gourmandes en accès au système de fichiers seront ralenties, à moins de tirer profit du mode sans mémoire d'état, possiblement en utilisant le stratagème de décompression exposé à la section 3.2.6 ou simplement en mettant en place elles-mêmes un jeu de données fréquemment accédées sur la couche supérieure en `tmpfs`.

4.2.4 Performance en calcul, *FourInARow*

FreeBench FourInARow est un banc d'essais proposé par CLARK et al. (2004) pour calculer la performance de calcul brute sur des entiers. Ce logiciel joue une partie de *Four In A Row* contre lui-même. Selon les auteurs de FreeBench :

> *[FourInARow] uses a min-max method with alpha-beta pruning as a search algorithm. [...] This program is integer only, but is not limited by the memory system. Some arithmetics is done using 64 bit integers, so 64 bit machines should fare well. The memory footprint is small and the execution time is spent in small recursive loops. Most of the execution fits in on-chip caches.*

Ce logiciel n'effectue presque pas d'IO sur système de fichiers ou en console et n'a pratiquement pas d'interactions directes avec le système d'exploitation.

Nous avons selectionné la meilleure optimisation d'architecture CPU (cf. annexe I.2.4 pour nos tests à ce propos), s'avérant être `-O3 -march=athlon-xp`, d'ailleurs la même que celle utilisée pour compiler le système d'exploitation complet du groupe d'ordinateurs de tests.

La figure 4.5 présente les temps de calcul nécessaires pour trois configurations différentes. On constate que l'exécution sur N4 est légèrement plus rapide que sur les deux variétés de nœuds virtuels. Le nœud sans mémoire d'état nécessite en moyenne 1.65 s (3.0% du temps sur N4) de plus pour exécuter le test, alors que le nœud avec mémoire d'état a besoin de 1.54 s (2.8% du temps sur N4) de plus. Ces différences sont minimes et probablement attribuables à des surcoûts lors de l'ordonnancement des processus et des machines virtuelles. Des profilages plus avancés permettraient de cerner les causes des ces très faibles ralentissements. Nous retenons cependant que la performance de calcul brute s'approche de la performance native, confirmant ainsi les résultats des recherches précédentes à ce sujet.

4.2.5 Application de rendu d'image

Nous avons testé une application de rendu d'image, *Tachyon*. Nous croyons qu'un rendu parallèle est une application qui se distribue très bien sur une grappe de nœuds virtuels, où les conditions de performance peuvent varier grandement d'un nœud à l'autre. Ce logiciel est essentiellement borné par le processeur, et une lenteur locale ne compromet pas le rendu global : chaque nœud se charge d'effectuer le rendu d'une *tuile* de l'image globale. Une fois une tuile complétée, un nœud passe à la suivante, sans dépendre du travail des autres nœuds.

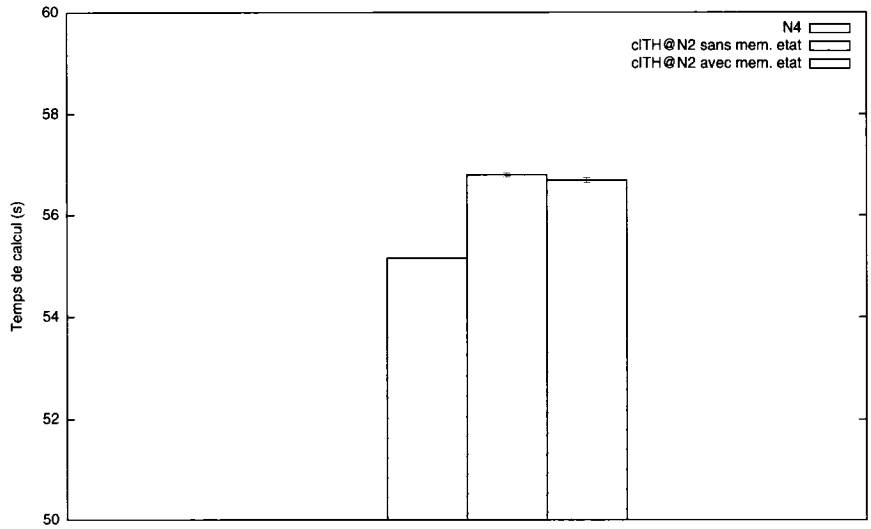

Figure 4.5 Temps de calcul nécessaire pour effectuer le test *FourInARow* de *FreeBench* sur des nœuds natifs et virtualisés

De plus, selon l'auteur, « *Tachyon* has been selected for inclusion in the SPEC MPI2007 benchmark suite » , en faisant un référentiel d'une valeur reconnue.

Nous avons testé le rendu d'un classique *Utah teapot* (BAKER, 2007), en 3200 par 2400 pixels, avec l'option `-fullshade` pour obtenir la meilleure qualité d'image possible. La figure 4.6 montre le résultat.

Nous avons compilé le logiciel avec l'option d'optimisation `-O3 -march=athlon-xp`, comme pour *FreeBench*.

En comparant le temps d'exécution sur 4 nœuds natifs à 512 Mo de mémoire et 4 nœuds virtuels à 512 Mo virtualisés sur ces 4 nœuds natifs, nous obtenons un temps d'exécution moyen de 18.37 s pour 4 nœuds natifs et 18.88 s pour 4 nœuds virtuels sans mémoire d'état. Passer à des nœuds virtuels entraîne une perte moyenne de performance de 3%, similaire à la perte de performance observée avec le test *FourInARow*.

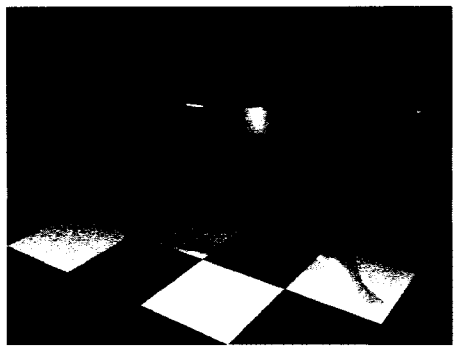

Figure 4.6 Rendu d'un pot de thé par *Tachyon*

4.3 Bancs de test sous *QEMU*

QEMU peut être, en terme de fonctionnalité, une solution de rechange à l'utilisation de *Xen*. *QEMU* ne nécessite pas de modifications invasives sur le serveur qui héberge des machines virtuelles, tout en permettant l'émulation de différentes architectures de systèmes complets (ce que *Xen* ne permet pas de faire, étant limité à l'architecture physique sous-jacente). Cependant, tel que vu dans l'état de l'art (cf. sections 1.2.1 et 1.2.3.2), *QEMU* reste un interpréteur logiciel[2] et on s'attend donc à des performances beaucoup moins intéressantes que celles de *Xen*. Nous verrons dans cette section les résultats obtenus pour quelques bancs d'essais.

Le lecteur interessé trouvera les tableaux de résultats bruts des évaluations sous *QEMU* en annexe I.

[2]Nous n'utilisons pas le module noyau `kqemu` : nous cherchons à évaluer la viabilité d'une solution qui pourrait remplacer *Xen* sur des sites où on doit minimiser l'*invasivité*.

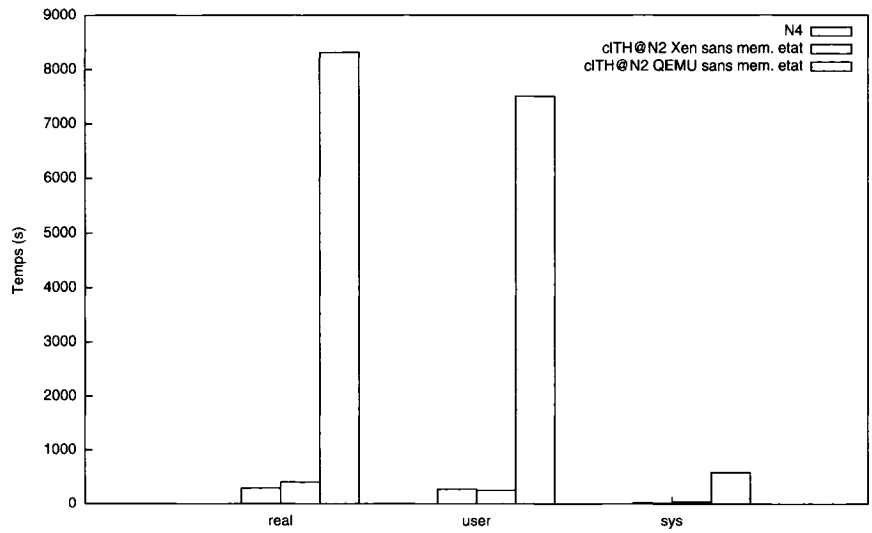

Figure 4.7 Temps requis pour la compilation sur N4, clTH@N2 *Xen* sans mémoire d'état et clTH@N2 *QEMU* sans mémoire d'état

4.3.1 Performance générale

Nous avons utilisé le même test de compilation du noyau que sous *Xen* (section 4.2.1). Les résultats sont présentés à la figure 4.7.

La performance de *QEMU* est pire, par plusieurs ordres de grandeur. Si l'on compare le temps *user*, on a besoin de près de 30 fois plus de temps sur *QEMU* que sur *Xen* pour effectuer la même tâche. Si l'on compare le temps *sys*, on a besoin de près de 15 fois plus de temps sur *QEMU*.

4.3.2 Performance réseau

Tout comme dans la configuration de *Xen* utilisée, nous lions les interfaces virtuelles de *QEMU* avec le réseau physique en les connectant sur un pont logiciel mis en

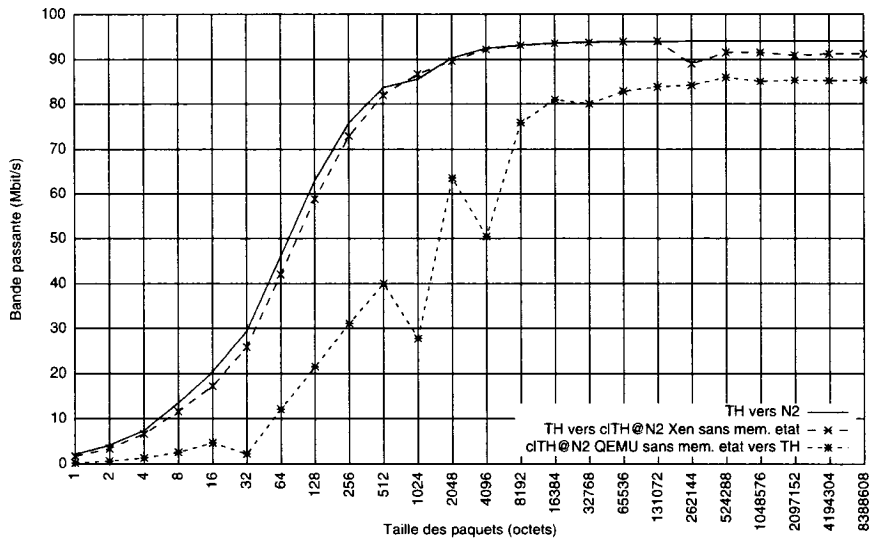

Figure 4.8 Bande passante entre nœuds natifs et nœuds virtualisés sous *Xen* et *QEMU*

place par le système natif hôte. Nous effectuons le même test que sous *Xen* avec *Netperf*.

Les performances d'un clone sans mémoire d'état sous *QEMU* sont plus faibles, tel que la figure 4.8 en témoigne. La bande passante maximale sous *QEMU* s'approche cependant de la bande passante maximale atteinte entre deux nœuds natifs, pour de grandes tailles de paquets : une différence d'en-deçà de 10% de la bande passante maximale sépare alors *QEMU* des nœuds natifs.

4.4 Impact d'*UnionFS*

La performance dans différents scénarios d'utilisation et avec un nombre varié de couches en unions est étudiée en profondeur par WRIGHT et al. (2006). Nous avons effectué des essais pour quantifier les pertes de performance liées à l'utilisation

Tableau 4.2 Bande passante moyenne et écart-type pour les tests de performance avec *DBench* sur `tmpfs` et plusieurs `tmpfs` en couches avec *UnionFS*

Test	B. pass. moy. (Mo/s)	Écart-type (% de la moy.)
`tmpfs` rw direct	424.03	1.11
un seul `tmpfs` rw en union	332.77	1.24
un `tmpfs` rw sur 6 `tmpfs` ro	330.43	1.58

d'*UnionFS* sur notre système de test.

Nous avons comparé les résultats du banc d'essai *DBench* sur un système de fichiers en RAM (`tmpfs`) auquel on accède directement avec soit un seul ou un ensemble de sept `tmpfs` insérés dans une union (la couche supérieure en lecture-écriture, les couches inférieures en lecture seulement). Dans notre test, aucune donnée n'était stockée sur les couches inférieures : il s'agissait de système de fichiers ne contenant aucun fichier ou répertoire. Le test *DBench* n'écrivant et ne lisant que sur la couche supérieure, nous pouvons isoler le temps de traitement relié au simple fait d'avoir un système en couches.

Nous obtenons les résultats présentés au tableau 4.2. L'utilisation d'*UnionFS* engendre une perte d'environ 22% par rapport à la performance dans un cas non-*UnionFS*. Ajouter d'autres branches ne change pas les performances significativement. (On s'attend bien sûr à ce qu'accéder à leur contenu puisse faire varier les performances — ici les branches inférieures ne contenaient rien et il était inutile de les accéder.)

La figure 4.4 présentée plus tôt va également dans ce sens, indiquant une perte de performance reliée à *UnionFS* : N4, accédant à un répertoire monté par NFS (de TH) avait une performance moyenne de 10.94 Mo/s, alors que le nœud sans mémoire d'état clTH@N2, qui accédait également par NFS à deux couches en union, avait une performance moyenne de 9.00 Mo/s. Cette différence correspond à 17.73% de

la performance observée sur N4.

À la lumière de nos résultats, *UnionFS* semble donc être un facteur important de perte de performance en IO système de fichiers.

4.5 Processus de déploiement

Le processus de déploiement d'un clone se subdivise en plusieurs étapes :

1. création d'un noyau et *initramfs* pour les clones (noyau de domaine U sous *Xen*, noyau standard avec *QEMU*) ;
2. Création de la couche à superposer, contenant les particularités du clone ;
3. création d'un fichier de configuration décrivant le système à simuler sur les machines hôtes ;
4. découverte des machines pouvant potentiellement être hôtes de clones ;
5. envoi du noyau, *initramfs* et fichier de configuration vers un nœud hôte ;
6. envoi d'un signal de démarrage.

La première étape du déploiement peut être accomplie une seule fois pour un groupe de nœuds en utilisant le plus petit dénominateur commun au niveau de l'architecture du processeur. Nous fournissons d'ailleurs un noyau compilé pour être utilisé en domaine U *Xen* sur une architecture *i686* ou plus récente. Un noyau pour *i686* est également fourni pour *QEMU* (cf. section 3.5.4).

Dans le cas où on préfèrerait configurer un noyau personnalisé, le temps requis pour réaliser cette étape est variable. Pour obtenir des performances vraisemblablement accrues sous *Xen*, on peut compiler un ensemble de plusieurs noyaux optimisés pour certaines architectures de processeurs. Au niveau des périphériques, il faudra configurer le noyau pour supporter ceux qui sont virtualisés par le VMM utilisé (sous *QEMU*, un ensemble minimal de périphériques décrit dans la page de manuel, sous *Xen*, on pourra considérer le support de tous les périphériques de *frontend*, bien qu'on dépende en fait des *backends* supportés sur le noyau en domaine 0 sur

le serveur *Napalm* utilisé). Nous fournissons des configurations de noyaux *Xen* et *QEMU* sur lesquelles on pourra se baser.

La création de la couche à superposer, pour le jeu standard de fichiers permettant de démarrer une distribution Gentoo clonée, prend moins d'une seconde. Le listage 4.1 montre une sortie console de l'exécution de la création d'une couche de différentiation :

Le temps requis pour générer cette couche supérieure de superposition est de l'ordre $O(n)$, pour n couches de différentiation de clone à générer (variant ensuite selon le nombre de fichiers à préparer et inclure dans cette couche — temps constant d'un clone à l'autre). Vu la performance observée, il ne nous semble pas nécessaire d'implanter de mécanismes supplémentaires pour accélérer la configuration d'un nœud clone.

Lorsqu'on compare ce que nous obtenons aux résultats présentés par MORIN (2006) pour la configuration de grappes, on constate un gain en rapidité d'exécution des scripts de mise en place des services. On attribue principalement ce gain à l'utilisation de *Python* au lieu de Bash. La plupart du code réside dans des modules qui sont compilés en *bytecode* lors de leur installation. Outre cette pré-optimisation, beaucoup de fonctionalités de *Python* sont programmées dans des modules C performants, notamment l'analyseur de syntaxe YAML. L'engin de gabarits *Cheetah* se base également sur un code C optimisé pour effectuer les remplacements de variables dans les fichiers gabarits.

Quant à la création manuelle d'un fichier de configuration décrivant la machine virtuelle qu'on veut démarrer, cela nécessite également un temps variable de l'ordre approximatif de quelques minutes, mais un jeu standard est généré automatiquement pour chacun des clones. Il est placé dans `/etc/napalm-clones` et peut être

Listage 4.1 Mesure de la performance de la création de la couche supérieure décrivant les particularités d'un système clone

```
thakur # /usr/bin/time -p clone -w clone01
*** Clone clone01:
Wiping ... done
Loading setup scripts ... DummySetupScript LoggerSetupScript NapalmSetupScript
--- DummySetupScript ---
DummySetupScript:execute
--- LoggerSetupScript ---
Clone will use the metalog logger.
--- NapalmSetupScript ---
Tip: You can start this clone in the stateful mode by using
     napalm [-u url_of_napalm_server] -c clone01-sample /etc/clones-napalm/↩
         stateful/clone01-sample
Tip: You can start this clone in the stateless mode by using
     napalm [-u url_of_napalm_server] -c clone01-sample /etc/clones-napalm/↩
         stateless/clone01-sample
Tip: You can start this clone in the stateless-qemu mode by using
     napalm [-u url_of_napalm_server] -c clone01-sample /etc/clones-napalm/↩
         stateless-qemu/clone01-sample
Tip: You can start this clone in the stateful-qemu mode by using
     napalm [-u url_of_napalm_server] -c clone01-sample /etc/clones-napalm/↩
         stateful-qemu/clone01-sample
--- end setup scripts ---
Merging trees ...
/var/state/systems/clone01/dummy
/var/state/systems/clone01/etc/runlevels/clone-default/register-clone
/var/state/systems/clone01/etc/runlevels/clone-default/metalog
/var/state/systems/clone01/etc/runlevels/default
/var/state/systems/clone01/etc/runlevels/boot
/var/state/systems/clone01/etc/modules.autoload.d/kernel-2.6
/var/state/systems/clone01/etc/init.d/register-clone
/var/state/systems/clone01/etc/conf.d/net
/var/state/systems/clone01/etc/conf.d/hostname
/var/state/systems/clone01/etc/hosts
/var/state/systems/clone01/etc/fstab
/var/state/systems/clone01/sbin/register-clone
/var/state/systems/clone01/sbin/dump-clone
/var/state/systems/clone01/sbin/unregister-clone
/var/state/systems/clone01/tmp/.wh.__dir_opaque
/var/state/systems/clone01/var/tmp/.wh.__dir_opaque
/var/state/systems/clone01/var/log/.wh.__dir_opaque
/var/state/systems/clone01/var/run/.wh.__dir_opaque
/var/state/systems/clone01/var/lib/postgresql/data/.wh.__dir_opaque
done
real 0.37
user 0.35
sys 0.03
```

utilisé directement, ou modifié relativement rapidement (très peu de lignes à modifier – le fichier XML au format *libvirt* généré fait moins de 20 lignes).

La découverte des machines sur un réseau local nécessite peu de temps. L'invocation de `napalm-browse` utilise les informations mises en cache par Avahi et effectue recherche active d'une durée (paramétrable en ligne de commande) de 5 secondes pour faire une requête sur le réseau et obtenir des réponses fraîches de la part des machines hôtes.

Le temps nécessaire à l'envoi du noyau et de la configuration vers une machine hôte dépend de plusieurs facteurs. La taille du noyau et le réseau sont les principaux. La taille de l'*initramfs* généré par *Genkernel* est de 3.6 Mo au format `cpio` (non compressé — lorsque l'*initramfs* est intégré au kernel, il est compressé avec `gzip`). Lorsque fusionné à un noyau contenant un minimum de caractéristiques (grossièrement) requises pour fonctionner sur les périphériques virtuels *Xen* et ayant un support *UnionFS*, nous obtenons un fichier d'environ 3 Mo, ce qui se transfère en près de 0.24 seconde sur un réseau *Fast Ethernet* 100 Mbit/s.

En ce qui a trait aux communications avec un serveur *Napalm*, l'analyse du transfert de paquets à l'aide de WireShark[3] montre les requêtes HTTP effectuées selon les opérations courantes. Le tableau 4.3 résume ce qui est observé.

Globalement, le processus de déploiement est très rapide, même sur du matériel d'ancienne génération, et tente de minimiser l'effort de l'administrateur : tous les fichiers et gabarits nécessaires pour démarrer un clone sont fournis. La génération des couches est rapide. Les transferts réseaux s'effectuent rapidement.

[3]WireShark, autrefois appelé Ethereal, est un observateur et analyseur de trafic réseau disponible à `http://wireshark.org`. Utilisé ici en version 0.99.4.

Tableau 4.3 Nombre de requêtes HTTP selon les opérations sous *Napalm*

	GET	PUT	DELETE
Création d'un domaine	0	1	0
Changer l'état d'un domaine	0	1	0
Supprimer un domaine	0	0	1
Obtenir status	1	0	0
Obtenir fichier	1	0	0
Envoyer fichier	0	1	0
Obtenir l'archive d'un domaine	1	0	0

4.6 Profilage du démarrage

Une fois les couches de superposition en place, les fichiers de configuration et la commande de démarrage envoyés à un hôte *Napalm*, le nœud est lancé : son noyau est chargé, qui passe la main à l'*initramfs*, qui superpose les systèmes de fichiers et qui, enfin, passe la main à `/sbin/init`, mimant ainsi en tous points un démarrage traditionnel de système *Linux*. Aucun service particulier ne doit être appelé pour configurer le nœud local.[4]

À l'aide de l'option *Show timing information on printks* (`CONFIG_PRINTK_TIME`) du noyau *Linux*, nous avons pu obtenir le temps nécessaire pour initialiser le noyau d'une machine virtuelle sous *Xen*. Nous avons ensuite mesuré le temps d'exécution de l'*initramfs* en lui faisant afficher le temps en sortie de son exécution, juste avant de faire pivoter la racine vers le système de fichiers final et de lancer `init`. Les résultats sont présentés au tableau 4.4. Il est à noter que le temps d'exécution de l'*initramfs* peut varier en fonction de facteurs extérieurs, soit le temps nécessaire à l'obtention d'une adresse IP par DHCP et le temps nécessaire au montage de deux

[4]Le seul ajout que nous faisons aux services de démarrage est le script `register-clone`, qui écrit essentiellement l'adresse IP et le nombre de processeurs dans 2 fichiers sur NFS — difficilement un facteur limitant les performances (cf. section 3.5.1.1).

Tableau 4.4 Profilage de la première partie du démarrage d'un clone sous *Xen*

Étape	Temps (s)
Initialisation du noyau	0.32
Exécution de l'*initramfs*	3.06
Total	3.38

systèmes de fichiers NFS.

Le profilage du démarrage des systèmes est fait à l'aide de l'outil *Bootchart* (BOOTCHART, 2007). En utilisant le support *BSD Process Accounting* du noyau, cet outil permet de visualiser une trace des processus démarrés lors de l'exécution des scripts des *runlevels*. Un processus *Bootchart* est démarré très tôt dans la séquence d'initialisation et surveille l'ensemble des processus démarrés, puis stocke à chaque 0.2 s différentes mesures décrivant leur exécution[5]. On traite ensuite ces données pour obtenir un rendu graphique du démarrage.

Les figures 4.9 et 4.10 montrent chacune une trace de profilage du démarrage post-*initramfs* de clones de `thakur` sous *Xen*. Dans le premier cas, on voit un démarrage de clone sans mémoire d'état, alors que dans le second, il s'agit d'un clone à mémoire d'état.

La durée du processus de démarrage n'est pas affectée de façon importante par le mode de fonctionnement (avec ou sans mémoire d'état) du nœud clone. La faible différence entre ces deux modes s'explique par un ratio lecture/écriture élevé. Le nœud sans mémoire d'état ayant une vitesse d'écriture significativement plus grande, il aurait démarré beaucoup plus rapidement que l'autre s'il y avait eu un rapport plus faible. La performance de démarrage est bornée par les lectures.

[5]Les processus dont l'exécution prend moins de 0.2 s peuvent donc être omis, expliquant les très légères différences entre les figures présentées.

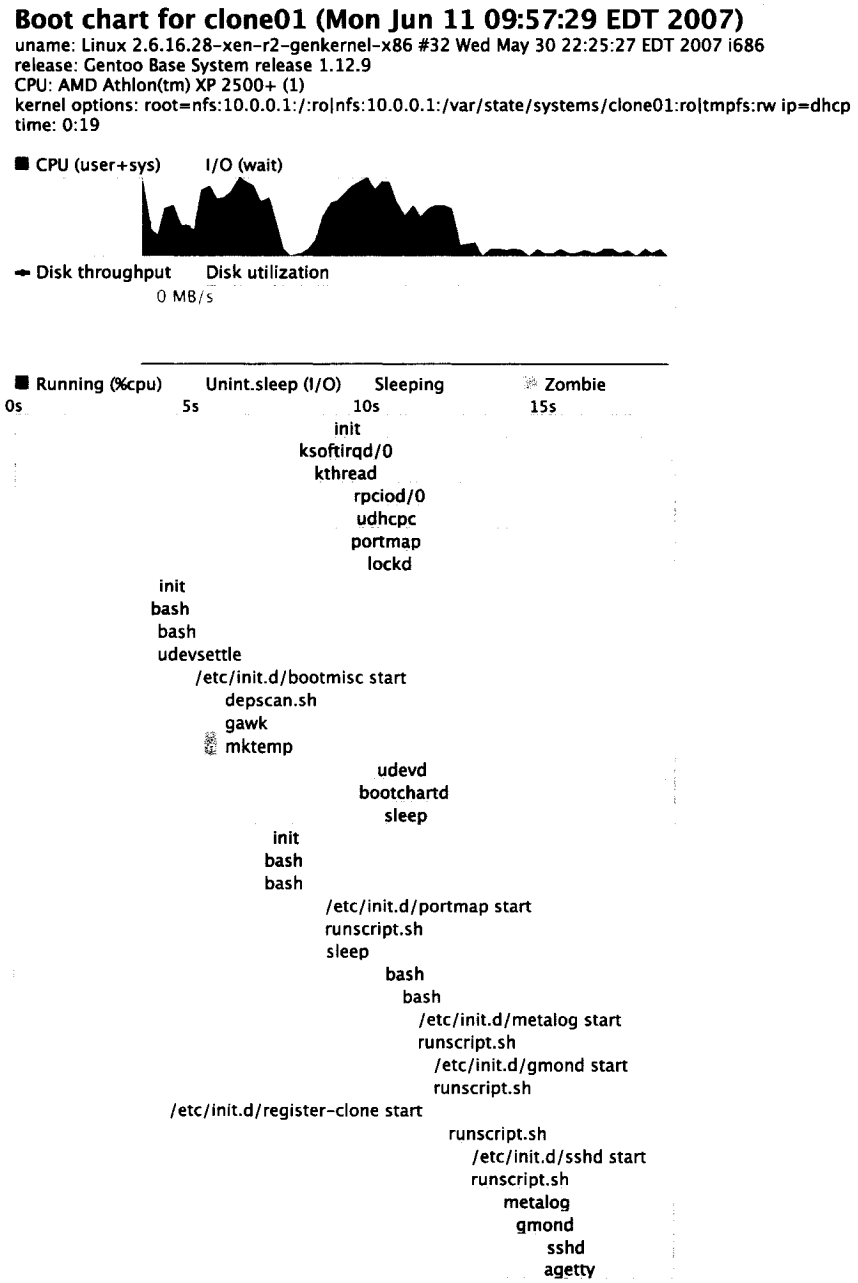

Figure 4.9 Profilage du démarrage d'un clone sans mémoire d'état

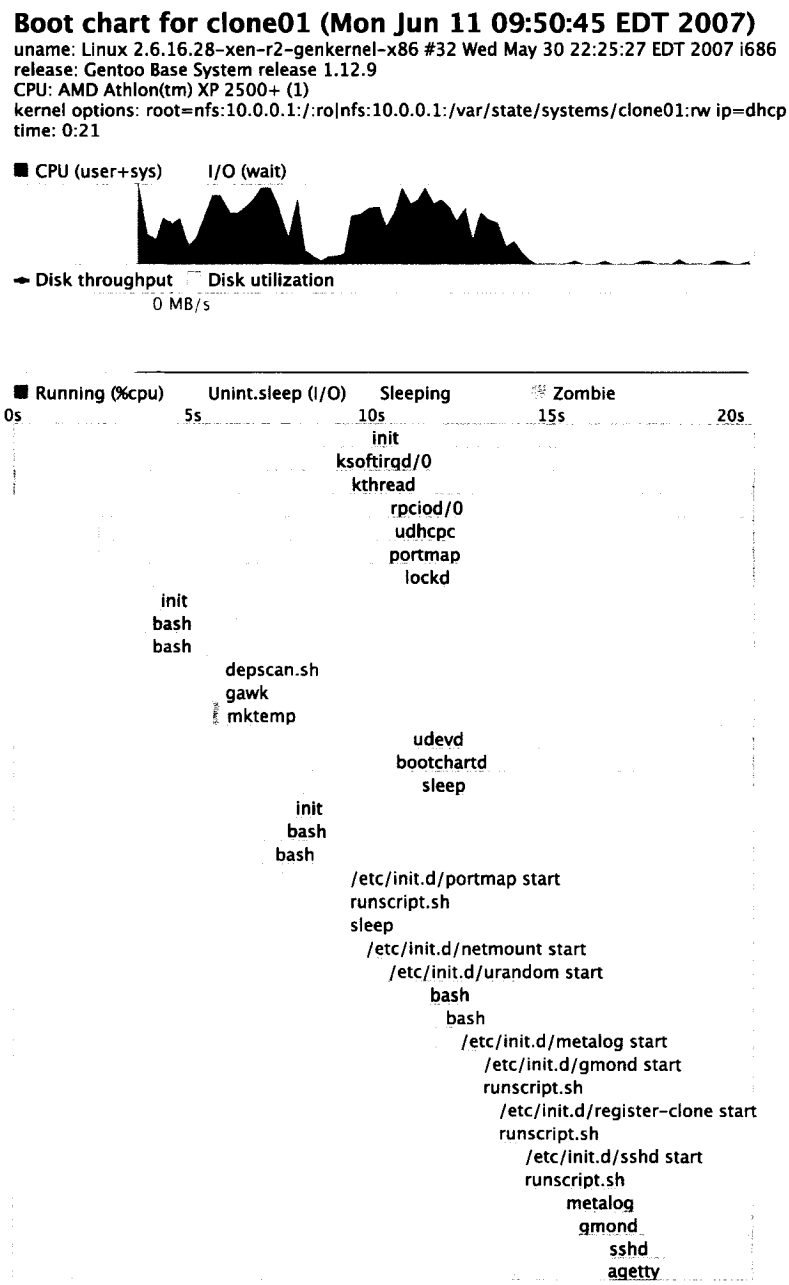

Figure 4.10 Profilage du démarrage d'un clone avec mémoire d'état

4.7 Sécurité

Une composante importante déterminant souvent le sort d'une technologie est la sécurité qu'elle assure aux données des usagers et aux infrastructures qui supportent le traitement informatique. Les logiciels développés dans le cadre de ce travail n'y font pas exception.

Nous faisons une analyse de la sécurité de l'utilisation de *Clone* et *Napalm*. Une étude du fonctionnement des logiciels et une analyse des paquets transmis entre les différents acteurs d'un système *Napalm* + *Clone* indique plusieurs facteurs à considérer pour rendre plus sécuritaire leur utilisation. Bien que nous n'ayons pas considéré la sécurité dans le développement de l'application pour se concentrer sur d'autres aspects, nous croyons qu'il est important de souligner les failles potentielles et, surtout, de proposer des correctifs pouvant empêcher les brèches.

4.7.1 Transferts de données sur HTTP

Le contrôle des machines virtuelles est effectué à l'aide d'une connexion HTTP vers un serveur *Napalm*. Dans l'implémentation actuelle, ce serveur répond diligemment à toute requête. Dans le « vrai monde », il s'agit là d'une importante faille de sécurité, puisque n'importe qui pourrait alors lancer des machines virtuelles sur un serveur *Napalm*.

Pour pallier à ce problème, il existe deux solutions classiques.

En premier lieu, utiliser SSL pour crypter les communications et authentifier les parties. Cela permet de vérifier l'identité à la fois de la machine effectuant la requête que du serveur *Napalm* lui-même.

En second lieu, on peut ajouter à cela une protection par mots de passe simplement en modifiant la configuration de *Apache* pour que ce dernier demande une paire *usager–mot de passe* à toute tentative de connexion vers `napalm.cgi`. Ce deuxième stratagème peut être couplé au premier. Nous croyons cependant que l'utilisation de comptes usagers peut être difficile à gérer à travers plusieurs domaines administratifs.

Une solution d'authentification globale — traversant les domaines administratifs — reste à être développée.

4.7.2 NFS

L'IO vers le système de fichiers des clones est supporté par le vénérable système de fichiers réseau NFS. NFS est le talon d'Achille de notre solution et doit être mis en place avec une extrême précaution, car il est difficile de sécuriser complètement ce sous-système.

L'utilisation de NFS présente un haut risque :

Accès clients Si un serveur NFS est configuré insouciamment, des clients malicieux peuvent monter des systèmes de fichiers distants. Dans notre cas, cela permet l'accès complet à la racine du système d'exploitation du système cloné, et l'accès en lecture-écriture sur le répertoire contenant les couches de différentiation des clones ;

Authentification usagers Il est possible pour un client malicieux de se faire passer pour un usager quelconque sur serveur et accéder à ses fichiers ;

Écoute clandestine Les paquets contenant les fichiers sont accessibles sur un réseau à accès public.

Bien que l'on puisse limiter l'accès par NFS à certaines adresses IP particulières, des méthodes d'attaque permettent de se faire passer pour quelqu'un d'autre et d'accéder à des répertoires NFS. L'utilisation d'un réseau physiquement sécurisé est l'option offrant le plus bas degré de risque.

Une piste de solution serait d'embarquer, dans l'*initramfs*, une implémentation de *Virtual Private Network* (VPN) et une clef cryptographique (on suppose que le transfert vers un serveur *Napalm* est aussi sécurisée, voir la section 4.7.1). Cela permettrait l'établissement d'un VPN *avant* le montage des systèmes de fichiers NFS. Il serait alors envisageable de monter ces derniers par-dessus un réseau VPN à communications cryptées, pour un probable surcoût en performance.

4.7.3 Authentification des serveurs *Napalm*

Rien n'empêche un attaquant de simuler un serveur *Napalm* pour hameçonner des usagers. Cela nécessite la publication par DNS-SD d'un service *Napalm*, et la mise en place de l'interface REST décrite à la section 3.3.1.

Pour contrer ce type d'attaque, on peut utiliser une extension DNS (WELLINGTON, 2000) imposant l'authentification des tiers parties qui veulent mettre à jour une zone DNS sur un serveur DNS contrôlant un sous-domaine. Un serveur *Napalm* authentifié pourrait alors mettre à jour la zone et insérer son champ SRV, alors que tout imposteur ne pourrait que publier par *multicast* ses champs SRV. Une modification des outils usagers permettrait de différencier les entrées DNS qui sont authentifiées des autres.

Toutefois, cette approche nécessite une implication plus importante des administrateurs d'un réseau local, qui devront maintenir une base de données de serveurs légitimes et leurs clefs d'authentification.

L'utilisation de la technologie SSL sur HTTP, avec laquelle on peut vérifier l'identité des serveurs est également une piste de solution.

4.7.4 Connexions des clones vers l'extérieur

Nous privilégions des connexions SSH (BARRETT et al., 2001) entre les différentes machines faisant parties d'un réseau de clones. Les autres solutions parfois utilisées pour les connexions distantes (`rsh`, `rlogin`, `telnet`) étant insécures puisque communiquant généralement les informations en plein texte sur le réseau.

Bien que SSH assure une bonne sécurité côté console distante et transfert de fichiers, différents services, comme par exemple *Ganglia*, doivent communiquer leurs données en plein texte à travers l'Internet sur des ports variés.

La prudence, encore une fois, est de mise : l'usager doit tenir compte des services qu'il exécute, des ports ouverts, des données transférées et des failles de sécurité connues sur les logiciels qu'il exécute.

Pour minimiser les risques, nous croyons qu'il serait, encore une fois, judicieux de lier les clones entre eux par des connexions en VPN. Cela permettrait l'échange de données sur des canaux sécurisés. Les services ne nécessitant que peu de performance réseau ne se trouveraient pas gênés par une telle approche.

4.7.5 Pollution inter-nœuds

Dans un contexte de SFI, puisque les systèmes partagent une racine, on doit s'assurer que le déploiement d'un nœud n'a pas d'impacts néfastes sur les fonctionalités des autres systèmes basés sur le même système de fichiers. Il faut donc minimiser la

« pollution » inter-nœuds : un nœud ne devrait pas pouvoir écrire dans un système de fichiers sur lequel un autre nœud tourne.

Il faut donc s'assurer qu'un nœud qui tourne

1. n'affecte nullement les *autres systèmes* démarrés en parallèle et partagent également le système de fichiers en racine (*pollution des paires*) ;
2. n'affecte nullement le fonctionnement du *serveur*, qui ne doit pas subir de modifications en termes de fonctionnalité (*pollution du père*)

Pour *Clone*, il est impossible de polluer les autres nœuds sauf en accédant directement au répertoire NFS contenant leur couche de différentiation. Des précautions au niveau de NFS (section 4.7.2) peuvent permettre de limiter cette possibilité.

On peut bien sûr polluer le nœud père par remplissage de son disque : plus on stocke de données sur le clone, plus on remplit le répertoire NFS exporté par le père. Une attention particulière doit être portée aux quotas d'utilisation des disques du serveur par les clones. Ce problème n'est cependant présent qu'en mode *avec mémoire d'état* puisqu'on a alors accès en lecture-écriture au répertoire. Dans le mode *sans mémoire d'état*, on peut assurer un accès en lecture seulement au système de fichiers du père.

4.7.6 Conclusion de l'étude de performance

Sur notre système de test, nous avons pu constater que les clones déployés sur des machines virtuelles *Xen* ont accès à la puissance nominale du réseau sous-jacent. La performance de calcul brute n'est pas affectée significativement par la virtualisation avec *Xen*. Ces résultats concordent avec ceux des recherches précédentes.

Cependant, avec *QEMU*, la perte de performance générale est très importante. Les

transferts réseaux sont également plus lents. Ce sont des résultats auxquels nous nous attendions après avoir fait la revue des solutions de virtualisation.

Les accès au système de fichiers, quant à eux, sont nettement moins performants qu'en accès direct sur un disque dur. Plusieurs facteurs contribuent à cela. En premier lieu, l'exportation du système de fichiers via NFS fait significativement décroître la performance vis-à-vis un système accédant directement à des disques locaux. En second lieu, le simple fait d'accéder à des données sur un système de fichiers *UnionFS* fait décroître les performances approximativement dans l'intervalle de 15% à 25%[6].

Notre étude de performance montre qu'il peut être avantageux d'utiliser le mode sans mémoire d'état pour les accès disques, avec les performances encourageantes présentées au tableau 4.1. L'utilisation du mode sans mémoire d'état n'est cependant pas la panacée, car il présente deux faiblesses, soit la consommation de la mémoire locale du nœud pour stocker les données et le manque de pérennité de l'état du nœud déployé. Certains scénarios en font cependant un mode à privilégier et la possibilité de déployer des clones avec ou sans mémoire d'état laisse la décision entre les mains de l'utilisateur final.

Globalement, les machines virtuelles sont moins performantes que leur équivalent natifs, avec les mêmes caractéristiques. Les techniques de paravirtualisation telles qu'implémentées dans *Xen* permettent cependant d'approcher des performances natives, alors que la l'émulation de système complets telle qu'implantée par *QEMU* s'en éloigne. Avec Xen, on retient que, pour des applications où les calculs sont dominants, il n'y a pratiquement pas de perte de performance. L'absence d'accès direct à certains mécanismes matériels d'optimisation des performances (DMA et

[6]Pour minimiser l'impact d'*UnionFS*, nous avons ajouté le support du montage des répertoires usagers (`/home`) sans *UnionFS*, directement par NFS.

autres systèmes d'*offloading* de charge vers le matériel) peut être une cause importante de la perte de performance sur le matériel moderne. Des recherches offrent cependant des méthodes permettant de pallier à ce problème et nous restons optimistes quant à l'amélioration des systèmes de virtualisation avec l'arrivée très récente de nouveaux joueurs (*KVM*, *lguest* (RUSSELL, 2007)) et de nouvelles extensions à l'architecture *x86*.

Le processus de déploiement et l'effort requis pour créer une grappe instantanée a également été mesuré. On montre qu'un effort minime est requis pour déployer des nœuds virtuels.

Au niveau de la sécurité, nous avons mis en lumière certaines failles importantes qui doivent être prises en compte lors de l'utilisation du système de déploiement de nœuds. Cette analyse amène des suggestions de correctifs à l'implémentation. Indirectement, elle remet aussi en cause certaines autres boîtes d'outils logicielles qui partagent des éléments avec la nôtre.

CONCLUSION

Ce travail a permis d'évaluer la viabilité d'une solution de grilles de calcul instantanées basées sur l'idée de transformation par différentiation d'un système d'exploitation installé sur un poste-référentiel. Nous avons réalisé l'intégration de plusieurs technologies et méthodes pour permettre la génération de grilles virtuelles articulées sur des nœuds SFI légers.

Nous avons montré que la génération de fichiers par gabarit, qui, dans une certaine mesure, fait le succès de plusieurs approches à l'écriture d'applications Web, se transpose très bien dans le cas de génération de fichiers de configuration. Nos résultats montrent qu'il est très rapide et flexible, en plus d'être intuitif, d'utiliser une solution basée sur des gabarits.

Nous avons créé plusieurs scripts et gabarits pour les cas d'utilisation standards, permettant de cloner un système d'exploitation.

Une réécriture du système *early-userspace* de *Genkernel* a amené de nombreuses améliorations en termes de structure et de flexibilité au processus de démarrage pré-`init` sous *Linux*, non seulement pour le contexte de notre projet, mais pour soutenir, nous l'espérons, de nombreux autres scénarios innovateurs.

Nous montrons que l'utilisation de *UnionFS* permet de créer des SFI flexibles et d'une performance acceptable pour de nombreux usages. Par la mémorisation d'état par couches superposées, nous amenons la possibilité d'obtenir des clones de systèmes où l'ensemble des fichiers est en lecture-écriture. *UnionFS* permet de répliquer un système entier tout en offrant l'illusion complète d'avoir une seconde machine indépendante.

Un système de déploiement non-intrusif et basé sur des protocoles standards comme HTTP et DNS-SD a été créé. Ce système permet, à un faible coût, de rendre les postes de travail peu sollicités utiles à des tâches distribuées.

La modularisation de l'ensemble logiciel développé assure une réutilisation simple des trois parties : les outils *early-userspace*, de déploiement (*Napalm*) et de génération de couches de différentiation (*Clone*) sont indépendants et peuvent être remplacés aisément ou réutilisés pour d'autres contextes, particulièrement les modifications à *Genkernel*. Ils sont tous distribués sous la licence GPL, version 2 (FSF, 1991).

Enfin, nous avons évalué la trousse d'outils développée en utilisation réelle et ses limites selon les axes de la performance et de la sécurité.

Sur la plate-forme *Xen*, nos résultats ont montré une performance en calcul brut très proche de ce qu'on peut atteindre sur un système natif. Les performances en transferts réseau sont également presqu'au par avec celles d'un système non-virtualisé. La piètre performance en accès au système de fichiers due à l'utilisation de NFS est cependant un facteur limitant important et réduit la performance générale des clones avec ou sans mémoire d'état. L'utilisation du système de superposition *UnionFS* a également un impact négatif sur la performance d'accès au système de fichiers.

En comparaison, une étude de l'utilisation de clones hébergés sur le logiciel *QEMU* nous a montré des performances générales moins intéressantes. D'abord pour les mêmes raisons que celles évoquées précédemment, mais également parce que la stratégie d'interprétation logicielle de *QEMU* est nettement moins performante que la méthode de paravirtualisation de *Xen* dans le cas général.

L'ensemble des outils développés permet de générer rapidement des grappes de cal-

culs dont la performance est intéressante pour certains domaines applicatifs. L'approche permet de déployer des nœuds légers : peu de données sont transférées pour les instancier et l'appareillage nécessaire sur les systèmes hôtes se limite à un VMM et un couple démon–interface Web. Elle s'inscrit dans une stratégie de récupération de ces fameux *cycles perdus*, mais en offrant une simplicité de déploiement et une grande flexibilité en s'appuyant sur la virtualisation.

Perspectives

Différentes améliorations à l'égard de notre projet sont à étudier. On en retient quatre principales, soit l'amélioration de la sécurité de l'implémentation, la gestion et le partage des ressources, la recherche d'optimisations possibles au système de fichiers réseau NFS utilisé et la migration transparente de machines virtuelles. La première a été traitée au chapitre 4 ; nous discutons de ces trois dernières dans cette section.

Gestion et partage de ressources

Il y a une quantité finie de ressources qui peuvent être libérées pour consommation par le public. Même si on tente d'en libérer plus par notre approche, cela ne suffira probablement jamais à la demande. Que faire alors pour diviser équitablement un nombre restreint de ressources entre différents consommateurs ?

Pour solutionner ce problème une méthode permettant de recenser ces ressources et de connaître en tout temps leur disponibilité s'impose. Ensuite, cela demande de développer des heuristiques de partage optimal de ressources, des systèmes de mise en queue intelligents et éventuellement des méthodes d'allocations par paiements,

s'appuyant sur l'authentification des usagers.

Une infime partie du problème est solutionnée en mettant en pratique la proposition à l'annexe V pour connaître quels types de machines virtuelles on peut lancer sur un serveur *Napalm* donné, mais il reste de cette problématique qui pourrait vraisemblablement alimenter plusieurs travaux de recherche.

Évaluation sur variantes et extensions pour NFS

Nos résultats ont montré sur notre système de test que la faiblesse fondamentale du déploiement de systèmes clones est au niveau de l'accès au système de fichiers. D'une part, *UnionFS* entraîne un surcoût non-négligeable, mais c'est surtout l'utilisation de NFS pour exporter les couches qui a un très grand impact.

Heureusement, une nouvelle génération du système de fichiers NFS, NFSv4, promet de nombreuses améliorations. Selon HARRINGTON et al. (2006), on peut s'attendre à des améliorations de performance notables et des caractéristiques nouvelles comme la réplication et la migration :

> *NFSv4 provides some support for filesystem replication and migration with a new* `fs_locations` *attribute that clients can use to find other servers exporting the same filesystem data.*
> *[...]*
> *We have patches implementing preliminary support for replication and migration; further work needs to be done to refine them and integrate them into mainline.*

De plus, NFSv4 a des performances beaucoup plus grandes que NFSv3 pour certaines opérations de lecture et écriture (notons l'écriture asynchrone, où on peut

s'attendre à un performance six fois plus grande). Dans les autres cas, on peut s'attendre à une performance au moins égale à celle de NFSv3.

Des mécanisme de *caching* sont également proposés pour NFSv4 :

> *Delegations are supported, and the client will take advantage of a file delegation where possible to perform opens and closes without contacting the server. Work is underway to provide even more aggressive caching on the client, if desired, using on-disk data caching.*

FS-Cache (HOWELLS, 2006) est également proposé comme couche permettant aux systèmes de fichiers réseau de demander un service de mise en cache au noyau sans devoir connaître l'implantation de ce mécanisme. FS-Cache se charge de gérer le *caching*, qui se fait soit sur un système de fichiers dédié (*CacheFS*) où dans un fichier *CacheFile* sur le disque du système. L'intérêt principal de cette solution est sa généricité : on cherche à développer un service que la plupart des systèmes de fichiers réseaux (on cite NFS, *Andrew File System*, *Common Internet Filesystem*) pourront utiliser. Ensuite, d'un point de vue génie logiciel, on maximise la réutilisation en centralisant la gestion du *caching* dans une interface générique utilisable par tous les systèmes de fichiers réseaux. Le développement de ce système amènerait des mécanismes de *caching* à NFSv3, améliorant par ricochet la performance de notre implémentation.

Migration transparente

Étant donné la variation possible des conditions d'exécution due à une utilisation parallèle plus ou moins importante du matériel soutenant un nœud virtuel, il peut être souhaitable de déplacer le travail d'un emplacement à un autre pour maximiser

les performances. Deux approches, sur deux niveaux différents, sont à étudier :

1. Déplacement dynamique des processus parmi une grappe de nœuds virtuels ;
2. Migration de machines virtuelles, à chaud ou non, d'une machine physique à une autre.

Le premier point pourrait être résolu en additionnant une couche SSI (cf. section 1.3.2), le deuxième en exploitant des caractéristiques présentes sur certaines implémentations permettant un déplacement dynamique de VMs. Tel que vu à la section 1.2.3.1, *Xen* permet de déplacer des machines virtuelles d'un ordinateur à un autre avec un minimum de temps d'arrêt. Il serait intéressant d'intégrer cette possibilité aux travaux présentés dans ce mémoire.

RÉFÉRENCES

ALMESBERGER, W. et LERMER, H. (1996). Using the initial RAM disk (initrd). Documentation du noyau Linux 2.6.16.28.

AMD (2006). *AMD I/O Virtualization Technology (IOMMU) Specification.* Advanced Micro Devices Inc. Document PID 34434.

ANDREWS, J. (2007). Interview : Avi Kivity. En ligne à : `http://kerneltrap.org/node/8088` [Consulté le 2007.04.23].

ANDREWS, M. ; JERPETJOEN, K. ; VERMEULEN, S., et NEYS, X. (2007). Diskless Nodes with Gentoo. En ligne à : `http://www.gentoo.org/doc/en/diskless-howto.xml` [Consulté le 2007.04.12].

AVAHI (2007). Avahi. En ligne à : `http://www.avahi.org/` [Consulté le 2007.03.02].

BAKER, S. (2007). A Brief History of The Utah Teapot. En ligne à : `http://www.sjbaker.org/teapot/` [Consulté le 2007.05.25].

BARAK, A. ; LA'ADAN, O., et SHILOH, A. (1999). Scalable Cluster Computing with MOSIX for LINUX. *Proceedings of the 5th Annual Linux Expo.* Raleigh, NC, USA.

BARHAM, P. ; DRAGOVIC, B. ; FRASER, K. ; HAND, S. ; HARRIS, T. ; HO, A. ; NEUGEBAUER, R. ; PRATT, I., et WARFIELD, A. (2003). Xen and the Art of Virtualization. *Symposium on Operating Systems Principles 2003.* Bolton Landing, NY, USA.

BARRETT, D. J. ; BYRNES, R. G., et SILVERMAN, R. E. (2001). *SSH : The Secure Shell.* O'Reilly and Associates, Sebastopol, CA, USA.

BECKER, D. J. ; STERLING, T. ; SAVARESE, D. ; DORBAND, J. E. ; RANAWAK, U. A., et PACKER, C. V. (1995). Beowulf : A parallel workstation for scientific computation. *Proceedings of the 24th International Conference on Parallel Processing*, pages I :11–14. Champaign-Urbana, IL, USA.

BELLARD, F. (2005). QEMU, a Fast and Portable Dynamic Translator. *Proceedings of the USENIX Annual Technical Conference 2005.* Anaheim, CA, USA.

BELLARD, F. (2007). QEMU Accelerator Technical Documentation. En ligne à : `http://fabrice.bellard.free.fr/qemu/kqemu-tech.html` [Consulté le 2007.05.28].

BEN-KIKI, O. ; EVANS, C., et INGERSON, B. (2005). YAML™ Specification Index. En ligne à : `http://yaml.org/spec` [Consulté le 2007.02.01].

BJERKE, H. K. F. (2005). HPC Virtualization with Xen on Itanium. Mémoire de maîtrise, Norwegian University of Science and Technology.

BOCHS (2007). bochs : The Open Source IA-32 Emulation Project (Home Page). En ligne à : `http://bochs.sourceforge.net/` [Consulté le 2007.02.01].

BOOTCHART (2007). Bootchart. En ligne à : `http://bootchart.org/` [Consulté le 2007.05.25].

BPROC (2006). BProc Manual. En ligne à : `http://bproc.sourceforge.net/bproc.html` [Consulté le 2006.12.04].

BURGER, T. W. (2006). *Intel Virtualization Technology Quick Reference Guide.* Intel Corporation.

CAMERON, D. ; ROSENBLATT, B., et RAYMOND, E. (1996). *Learning GNU Emacs.* O'Reilly and Associates, Sebastopol, CA, USA.

CHASE, J. ; IRWIN, D. ; GRIT, L. ; MOORE, J., et SPRENKLE, S. (2003). Dynamic Virtual Clusters in a Grid Site Manager. *Proceedings of the 2003 IEEE International Symposium on High Performance Distributed Computing.* Seattle, WA, USA.

CHEETAH (2007). Cheetah, The Python-Powered Template Engine. En ligne à : `http://www.cheetahtemplate.org/` [Consulté le 2007.04.02].

CHESHIRE, S. et KROCHMAL, M. (2006). Multicast DNS. En ligne à : `http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt` [Consulté le 2007.01.30].

CHESIRE, S. et KROCHMAL, M. (2006). DNS-Based Service Discovery. En ligne à : `http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt` [Consulté le 2007.01.30].

CLARK, B. ; DESHANE, T. ; DOW, E. ; EVANCHIK, S. ; FINLAYSON, M. ; HERNE, J., et MATTHEWS, J. N. (2004). Xen and the Art of Repeated Research. *Proceedings of the USENIX 2004 Annual Technical Conference, FREENIX Track*, pages 135–144. Boston, MA, USA.

CLARK, C. ; FRASER, K. ; HAND, S. ; GORM HANSEN, J. ; JUL, E. ; LIMPACH, C. ; PRATT, I., et WARFIELD, A. (2005). Live Migration of Virtual Machines. *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation.* Boston, MA, USA.

COLLINS-SUSSMAN, B. ; FITZPATRICK, B. W., et PILATO, C. M. (2007). *Version Control with Subversion.* O'Reilly and Associates, Sebastopol, CA, USA.

CONDOR (2007). Overview of the Condor High Throughput Computing System. En ligne à : `http://www.cs.wisc.edu/condor/overview` [Consulté le 2007.02.01].

CROFT, B. et GILMORE, J. (1985). RFC 951 - BOOTSTRAP PROTOCOL (BOOTP). En ligne à : `http://www.ietf.org/rfc/rfc951.txt` [Consulté le 2007.01.24].

CROSBY, S. et BROWN, D. (2007). The virtualization reality. *Queue*, 4(10).

DALHEIMER, M. K.; DAWSON, T.; KAUFMAN, L., et WALSH, M. (2003). *Running Linux*. O'Reilly and Associates, Sebastopol, CA, USA.

DAVOLI, R. (2005a). VDE : Virtual Distributed Ethernet. *First International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, 00 :213–220.

DAVOLI, R. (2005b). Virtual Square. *Proceedings of the First International Conference on Open Source Systems*. Genova, Italy.

DAVOLI, R. (2007). Virtual Machines. En ligne à : `http://virtualsquare.org/vm.html` [Consulté le 2007.02.01].

DEBIAN (2007). Basics of the Debian package management system. En ligne à : `http://www.debian.org/doc/FAQ/ch-pkg_basics` [Consulté le 2007.05.30].

DES LIGNERIS, B.; BARRETTE, M.; GIRALDEAU, F., et DAGENAIS, M. (2004). Thin-oscar : Diskless clustering for all. *Linux World Magazine*, 2(3) :38–44.

DES LIGNERIS, B.; SCOTT, S.; NAUGHTON, T., et GORSUCH, N. (2003). Open Source Cluster Application Resources (OSCAR) : design, implementation and interest for the [computer] scientific community. *Proceedings of the First OSCAR Symposium*. Sherbrooke, QC, Canada.

DHPC-COMPUTING-GROUP (2007). *MPIBench's User's Guide Version 1.1.* Distributed and High Performance Computing Group. En ligne à : `http://www.dhpc.adelaide.edu.au/projects/MPIBench/mpibench-usermanual-v1.1.pdf` [Consulté le 2007.05.10].

DOUGHERTY, D. et ROBBINS, A. (1997). *sed & awk.* O'Reilly and Associates, Sebastopol, CA, USA.

DROMS, R. (1993). Interoperation Between DHCP and BOOTP. En ligne à : `http://www.ietf.org/rfc/rfc1534.txt` [Consulté le 2007.01.24].

ETHERBOOT (2007). EtherBoot/GPXE Wiki. En ligne à : `http://www.etherboot.org/wiki/index.php` [Consulté le 2007.05.12].

FIELDING, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Thèse de doctorat, University of California, Irvine.

FIELDING, R. (2002). Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2.

FIELDS, S. (1993). Hunting for Wasted Computing Power. En ligne à : `http://www.cs.wisc.edu/condor/doc/WiscIdea.html` [Consulté le 2007.02.01].

FIGUEIREDO, R. J.; DINDA, P. A., et FORTES, J. A. B. (2003). A Case For Grid Computing On Virtual Machines. *Proceedings of the 23rd International Conference on Distributed Computing Systems.* Providence, RI, USA.

FINLAYSON, R. (1984). Bootstrap Loading using TFTP. En ligne à : `http://www.ietf.org/rfc/rfc906.txt` [Consulté le 2007.01.24].

FOLDINGATHOME (2000). Folding@home. En ligne à : `http://folding.stanford.edu/` [Consulté le 2007.05.18].

FOLDINGATHOMEPS3 (2007). Folding@Home on the PS3. En ligne à : `http://folding.stanford.edu/FAQ-PS3.html` [Consulté le 2007.05.18].

FOSTER, I. (2006). Globus Toolkit Version 4 : Software for Service-Oriented Systems. *Lecture Notes in Computer Science*, 3779 :2–13.

FSF (1991). GPL. En ligne à : `http://www.fsf.org/licensing/licenses/info/GPLv2.html`.

FSFBASH (2007). GNU BASH. En ligne à : `http://www.gnu.org/software/bash/bash.html` [Consulté le 2007.02.01].

FSFMAKE (2007). GNU Make. En ligne à : `http://www.gnu.org/software/make/` [Consulté le 2007.03.01].

GANGLIA (2007). Ganglia Monitoring System. En ligne à : `http://ganglia.sourceforge.net/` [Consulté le 2007.03.18].

GARFINKEL, S. et SPAFFORD, G. (1996). *Practical UNIX & Internet Security*. O'Reilly and Associates, Sebastopol, CA, USA.

GAVIN, P. ; TRYVGE KALLEBERG, K. ; FRYSINGER, M., et ROBBINS, D. (2007). Gentoo Developer Handbook. En ligne à : `http://www.gentoo.org/proj/en/devrel/handbook/handbook.xml` [Consulté le 2007.01.30].

GENTOOWIKI (2007). HOWTO : Xen and Gentoo. En ligne à : `http://gentoo-wiki.com/HOWTO_Xen_and_Gentoo` [Consulté le 2007.01.30].

GIMPS (1996). GIMPS. En ligne à : `http://www.mersenne.org/` [Consulté le 2007.05.18].

GIT (2007). Fast Version Control System. En ligne à : `http://git.or.cz/` [Consulté le 2007.05.29].

GLOBUSWORKSPACE (2007). Globus Workspace. En ligne à : `http://workspace.globus.org/` [Consulté le 2007.03.13].

GOLDBERG, R. P. (1972). *Architectural Principles for Virtual Computer Systems.* Thèse de doctorat, Harvard University.

GOLDSHMIDT, O. (2005). Virtualization – Advanced Operating Systems. IBM Corporation. En ligne à : `http://cs.haifa.ac.il/courses/AdvancedOS/` [Consulté le 2007.08.15].

GORM HANSEN, J. et JUL, E. (2005). Optimizing Grid Application Setup using Operating System Mobility. *Lecture Notes in Computer Science.*

GOSLING, James nad MCGILTON, H. (1996). The Java Language environment : a white paper. En ligne à : `http://java.sun.com/docs/white/langenv/`.

HARRINGTON, B. ; CHARBON, A. ; REIX, T. ; ROQUETA, V. ; FIELDS, J. B. ; MYKLEBUST, T. ; JAYARAMAN, S. ; NEEDLE, J., et MARSON, B. (2006). NFSv4 Test Project. *Proceedings of the Ottawa Linux Symposium 2006.*

HENNESSY, J. et PATTERSON, D. (1996). *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, USA, 2ième édition.

HOWELLS, D. (2006). FS-Cache : A Network Filesystem Caching Utility. *Proceedings of the Ottawa Linux Symposium 2006.*

HP-NETWORK-DIVISION (1995). Netperf : A Network Performance Benchmark. En ligne à : `http://www.netperf.org/netperf/training/Netperf.html` [Consulté le 2007.01.30].

INFINIBAND-TA (2007). InfiniBand. En ligne à : `http://www.infinibandta.org` [Consulté le 2007.05.30].

INTEL (1999). *Preboot Execution Environment (PXE) Specification, version 2.1.* Intel Corporation.

INTEL (2005). *Intel Virtualization Technology Specification for the IA-32 Intel Architecture.* Document C97063-002.

INTEL (2007). *Intel 64 and IA-32 Architectures Software Developer's Manual.* Volume 3A : System Programming Guide. En ligne à : `http://www.intel.com/products/processor/manuals/index.htm` [Consulté le 2007.08.15].

KERRHIGED (2007). Kerrhiged, Linux clusters made easy. En ligne à : `http://www.kerrighed.org/` [Consulté le 2007.05.13].

KNOPPER, K. (2000). Building a self-contained auto-configuring Linux system on an iso9660 filesystem. *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 373–376. Atlanta, GA, USA.

KOTSOVINOS, E. (2005). Global Public Computing. Technical Report 615, University of Cambridge.

KOTSOVINOS, E.; MORETON, T.; PRATT, I.; Russ, R.; FRASER, K.; HAND, S., et HARRIS, T. (2004). Global-scale service deployment in the XenoServer platform. *Proceedings of the First Workshop on Real, Large Distributed Systems 2004.* San Francisco, CA, USA.

KTRINTZ, S. et SUCU, S. (2006). Adaptive On-the-Fly Compression. *IEEE Transactions on Parallel and Distributed Systems.*

KUHLMANN, G. et MARES, M. (1997). Mounting the root filesystem via NFS (nfsroot). Documentation du noyau Linux 2.6.16.28.

LITZKOW, M. (1987). Remote Unix - Turning Idle Workstations into Cycle Servers. *Proceedings of Usenix Summer Conference 1987*, pages 381–384. Phoenix, AR, USA.

LIU, J. ; HUANG, W. ; ABALI, B., et PANDA, D. K. (2006). High Performance VMM-Bypass I/O in Virtual Machines. *Proceedings of the 2006 USENIX Annual Technical Conference*. Boston, MA, USA.

LOTTIAUX, R. ; GALLARD, P. ; VALLE, G. V., et MORIN, C. (2005). OpenMosix, OpenSSI and Kerrighed : A Comparative Study. *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid*. Cardiff, Wales, UK.

LWN (2002). Initramfs arrives. En ligne à : `http://lwn.net/Articles/14776/` [Consulté le 2007.04.13].

MASSIE, M. L. ; CHUN, B. N., et CULLER, D. E. (2004). The Ganglia Distributed Monitoring System : Design, Implementation, and Experience. *Parallel Computing*, 30.

MENON, A. ; COX, A. L., et ZWAENEPOEL, W. (2006). Optimizing Network Virtualization in Xen. *USENIX 2006 Annual Technical Conference Refereed Paper*. Boston, MA, USA.

MENON, A. ; SANTOS, J. R. ; TURNER, Y. ; JANAKIRAMAN, G. J., et ZWAENEPOEL, W. (2005). Diagnosing Performance Overheads in the Xen Virtual Machine Environment. *Proceedings of the First ACM/USENIX International Conference on Virtual Execution Environments*. Chicago, IL, USA.

MEWBURN, L. (2001). The Design and Implementation of the NetBSD rc.d system. *Proceedings of the FREENIX Track : 2001 USENIX Annual Technical Conference.* Boston, MA, USA.

MORIN, B. (2006). Parallélisme sur plateforme reconfigurable de calcul à mémoire répartie. Mémoire de maîtrise, École Polytechnique de Montréal.

MUEHLEN, M. ; NICKERSON, J., et SWENSON, K. D. (2005). Developing Web Services Choreography Standards – The Case of REST vs. SOAP. *Decision Support Systems*, 40(1) :9–29.

OETIKER, T. ; PARTL, H. ; HYNA, I., et SCHLEGL, E. (2006). *The Not So Short Introduction to LaTeX 2e.* En ligne à : `http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf` [Consulté le 2007.06.08].

OPENMOSIX (2007). openMosix, an Open Source Linux Cluster Project. En ligne à : `http://openmosix.sourceforge.net/` [Consulté le 2007.05.13].

OSCAR (2007). *Oscar 5.0 Users Manual.* En ligne à : `http://oscar.openclustergroup.org/public/docs/oscar5.0/OSCAR5.0_Users_Manual.pdf` [Consulté le 2007.05.20].

PAPADOPOULOS, P. M. ; KATZ, M. J., et BRUNO, G. (2003). NPACI Rocks : tools and techniques for easily deploying manageable Linux clusters. *Concurrency and computation.*

PENDRY, J.-S. et MCKUSICK, M. K. (1995). Union Mounts in 4.4BSD-Lite. *Proceedings of the 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems.* New Orleans, LA, USA.

PENGUINCOMPUTING (2007). Scyld. En ligne à : `http://www.penguincomputing.com/` [Consulté le 2007.02.02].

PLAN9 (1992). *Plan 9 from Bell Labs - Programmer's Manual (first edition).*

PLANETLAB (2007). PlanetLab : Home. En ligne à : `http://www.planet-lab.org/` [Consulté le 2007.02.01].

POPEK, G. J. et GOLDBERG, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17 :412–21.

PRATT, I. (2005). Xen Virtualization. Présentation BOF au LinuxWorld 2005, San Francisco, CA, USA.

PRATT, I. ; FRASER, K. ; HAND, S. ; LIMPACH, C. ; WARFIELD, A. ; MAGENHEIMER, D. ; NAKAJIMA, J., et MALLICK, A. (2005). Xen 3.0 and the Art of Virtualization. *Proceedings of the Ottawa Linux Symposium 2005.*

PRATT, I. ; MAGENHEIMER, D. ; BLANCHARD, H. ; XENIDIS, J. ; NAKAJIMA, J., et LIGUORI, A. (2006). The Ongoing Evolution of Xen. *Proceedings of the Ottawa Linux Symposium 2006.*

REDHAT (2007). rpm.org - The RPM Package Manager. En ligne à : `http://rpm.org/` [Consulté le 2007.05.01].

REYNOLDS, J. K. (1993). RFC 1497 - BOOTP Vendor Information Extensions. En ligne à : `http://www.ietf.org/rfc/rfc1497.txt` [Consulté le 2007.01.24].

ROBIN, J. S. et IRVINE, C. E. (2000). Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. *Proceedings of the 9th USENIX Security Symposium.* Denver, CO, USA.

ROSCOE, T. et HAND, S. (2003). Palimpsest : Soft-Capacity Storage for Planetary-Scale Services. *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX).* Lihue, HI, USA.

ROSENBLUM, M. (2004). The reincarnation of virtual machines. *Queue*, 4 :34–40.

RUSSELL, R. (2007). lguest (formerly lhype). En ligne à : `http://lguest.ozlabs.org/` [Consulté le 2007.05.25].

SCHNEIER, B. (1996). *Applied Cryptography.* John Wiley & Sons, New York, NY, USA.

SETIATHOME (1999). En ligne à : `http://setiathome.ssl.berkeley.edu/` [Consulté le 2007.05.18].

SILVER, L. M. (2001). What are clones ? *Nature.*

SINGH, A. (2007). An Introduction to Virtualization. En ligne à : `http://www.kernelthread.com/publications/virtualization/` [Consulté le 2007.05.2].

SMITH, C. (2006). Linux NFS-HOWTO. En ligne à : `http://nfs.sourceforge.net/nfs-howto/` [Consulté le 2007.05.10].

SMITH, J. et NAIR, R. (2005). The Architecture of Virtual Machines. *Computer*, 38(5).

SOLLINS, K. R. (1992). RFC 1350 - The TFTP Protocol (Revision 2). En ligne à : `http://www.ietf.org/rfc/rfc1350.txt` [Consulté le 2007.03.04].

SPENCE, D. et HARRIS, T. (2003). XenoSearch : Distributed Resource Discovery in the XenoServer Open Platform. *Proceedings of the 2003 IEEE International Symposium on High Performance Distributed Computing.* Seattle, WA, USA.

ST. LAURENT, A. M. (2004). *Understanding Open Source & Free Software Licensing.* O'Reilly and Associates, Sebastopol, CA, USA.

STONE, J. (2007). Tachyon Parallel / Multiprocessor Ray Tracing System. En ligne à : `http://jedi.ks.uiuc.edu/~johns/raytracer/` [Consulté le 2007.05.10].

SUCHOMSKI, M. (2002). *The OSDB Handbook.* En ligne à : `http://osdb.sourceforge.net/` [Consulté le 2007.08.14].

SUZAKY, K. ; YAGI, T. ; IIJIMA, K. ; KITAGAWA, K., et TASHIRO, S. (2006). HTTP-FUSE Xenoppix. *Proceedings of the Ottawa Linux Symposium 2006.*

TRIDGELL, A. (2007). Emulating Netbench. En ligne à : `http://samba.org/ftp/tridge/dbench/README` [Consulté le 2007.05.10].

UNIONFS (2007). A Stackable Unification File System. En ligne à : `http://www.filesystems.org/project-unionfs.html` [Consulté le 2007.05.08].

VEILLARD, D. et ZAK, K. (2007). The Virtualization API. En ligne à : `http://libvirt.org/` [Consulté le 2007.02.01].

VIRO, A. et ANVIN, H. P. (2002). initramfs buffer format. Documentation du noyau Linux 2.6.16.28.

WALKER, B. J. (2003). OpenSSI (Single System Image) Clusters for Linux. En ligne à : `http://openssi.org/ssi-intro.pdf` [Consulté le 2007.08.14].

WARFIELD, A. ; HAND, S. ; HARRIS, T., et PRATT, I. (2002). Isolation of Shared Network Resources in XenoServers. PlanetLab Design Note PDN–02–006.

WARFIELD, A. ; ROSS, R. ; FRASER, K. ; LIMPACH, C., et HAND, S. (2005). Parallax : Managing Storage for a Million Machines. *Proceedings of the 10th Workshop on Hot Topics in Operating Systems.* Santa Fe, NM, USA.

WELLINGTON, B. (2000). Secure Domain Name System (DNS) Dynamic Update. En ligne à : `http://www.ietf.org/rfc/rfc3007.txt` [Consulté le 2007.03.13].

WILLIAMS, T. et KELLEY, C. (2004). *GNUPLOT : An Interactive Plotting Program.* En ligne à : `http://www.gnuplot.info/docs/gnuplot.html` [Consulté le 2007.08.14].

WRIGHT, C. (2004). Virtually Linux, Virtualization Techniques in Linux. *Proceedings of the Ottawa Linux Symposium 2004.*

WRIGHT, C. P. ; DAVE, J. ; GUPTA, P. ; KRISHNAN, H. ; QUIGLEY, D. P. ; ZADOK, E., et NAYER ZUBAIR, M. (2006). Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage*, 2 :1–32.

WRIGHT, C. P. et ZADOK, E. (2004). Unionfs : Bringing File Systems Together. *Linux Journal.*

XENTEAM (2004). *Xen v2.0 Interface Manual.* The Xen Team. En ligne à : `http://xensource.com/files/xen_interface.pdf` [Consulté le 2007.02.17].

XENTEAM (2005). *Xen v3.0 Users' Manual.* The Xen Team. En ligne à : `http://www.cl.cam.ac.uk/research/srg/netos/xen/readmes/user/user.html` [Consulté le 2007.02.17].

XENWIKI (2007). XenSpecificGlibc. En ligne à : `http://wiki.xensource.com/xenwiki/XenSpecificGlibc` [Consulté le 2007.05.25].

YU, J. T. (2004). Performance Evaluation of Linux Bridge. *Proceedings of the Telecommunications System Management Conference 2004.* Monterey, CA, USA.

ZHANG, X.; KEAHEY, K.; FOSTER, I., et FREEMAN, T. (2005). Virtual Cluster Workspaces for Grid Applications. Technical report, Argonne National Laboratory and University of Chicago.

ANNEXE I

ANALYSE DE PERFORMANCE

I.1 Système de test `thakur`

Le sytème de test est constitué d'un serveur frontal, connecté sur le réseau institutionnel, et de quatre nœuds connectés entre eux et avec le frontal, sur un réseau en étoile *Fast Ethernet* 100 MBit/s.

Les sections suivantes décrivent le matériel contenu dans le système frontal et les quatre nœuds sur son réseau interne.

I.1.1 `thakur`

Le listage qui suit montre le contenu de `/proc/cpuinfo` sur `thakur`.

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 6
model           : 10
model name      : AMD Athlon(tm) XP 2500+
stepping        : 0
cpu MHz         : 1837.553
cache size      : 512 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
```

```
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov ↩
    pat pse36 mmx fxsr sse syscall mmxext 3dnowext 3dnow ts
bogomips        : 3679.32
```

La commande `lspci` retourne :

```
00:00.0 Host bridge: nVidia Corporation nForce2 AGP (different version?) (rev a2↩
    )
00:00.1 RAM memory: nVidia Corporation nForce2 Memory Controller 1 (rev a2)
00:00.2 RAM memory: nVidia Corporation nForce2 Memory Controller 4 (rev a2)
00:00.3 RAM memory: nVidia Corporation nForce2 Memory Controller 3 (rev a2)
00:00.4 RAM memory: nVidia Corporation nForce2 Memory Controller 2 (rev a2)
00:00.5 RAM memory: nVidia Corporation nForce2 Memory Controller 5 (rev a2)
00:01.0 ISA bridge: nVidia Corporation nForce2 ISA Bridge (rev a3)
00:01.1 SMBus: nVidia Corporation nForce2 SMBus (MCP) (rev a2)
00:02.0 USB Controller: nVidia Corporation nForce2 USB Controller (rev a3)
00:02.1 USB Controller: nVidia Corporation nForce2 USB Controller (rev a3)
00:02.2 USB Controller: nVidia Corporation nForce2 USB Controller (rev a3)
00:04.0 Ethernet controller: nVidia Corporation nForce2 Ethernet Controller (rev↩
    a1)
00:06.0 Multimedia audio controller: nVidia Corporation nForce2 AC97 Audio ↩
    Controler (MCP) (rev a1)
00:08.0 PCI bridge: nVidia Corporation nForce2 External PCI Bridge (rev a3)
00:09.0 IDE interface: nVidia Corporation nForce2 IDE (rev a2)
00:1e.0 PCI bridge: nVidia Corporation nForce2 AGP (rev a2)
01:09.0 Ethernet controller: Intel Corporation 82557/8/9 [Ethernet Pro 100] (rev↩
    0c)
02:00.0 VGA compatible controller: nVidia Corporation NV18 [GeForce4 MX - nForce↩
    GPU] (rev a3)
```

Le système est connecté au réseau interne à la grappe (celui utilisé pour les tests) par sa carte d'interface réseau *Intel Ethernet Pro 100*. Il est connecté à l'extérieur par sa carte *nVidia nForce2*. Le disque de `thakur` est un *QUANTUM FIREBALLP AS20.5*.

I.1.2 Nœuds de `thakur`

Le listage qui suit montre le contenu de `/proc/cpuinfo` sur chacun des nœuds.

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 6
model           : 10
model name      : AMD Athlon(tm) XP 2500+
stepping        : 0
cpu MHz         : 1830.023
cache size      : 512 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
flags           : fpu tsc msr pae mce cx8 apic mtrr mca cmov pat pse36 mmx fxsr ↩
    sse syscall mmxext 3dnowext 3dnow ts
bogomips        : 3661.19
```

L'exécution de `lspci` retourne :

```
00:00.0 Host bridge: nVidia Corporation nForce2 AGP (different version?) (rev a2↩
    )
00:00.1 RAM memory: nVidia Corporation nForce2 Memory Controller 1 (rev a2)
00:00.2 RAM memory: nVidia Corporation nForce2 Memory Controller 4 (rev a2)
00:00.3 RAM memory: nVidia Corporation nForce2 Memory Controller 3 (rev a2)
00:00.4 RAM memory: nVidia Corporation nForce2 Memory Controller 2 (rev a2)
00:00.5 RAM memory: nVidia Corporation nForce2 Memory Controller 5 (rev a2)
00:01.0 ISA bridge: nVidia Corporation nForce2 ISA Bridge (rev a3)
00:01.1 SMBus: nVidia Corporation nForce2 SMBus (MCP) (rev a2)
00:02.0 USB Controller: nVidia Corporation nForce2 USB Controller (rev a3)
00:02.1 USB Controller: nVidia Corporation nForce2 USB Controller (rev a3)
00:02.2 USB Controller: nVidia Corporation nForce2 USB Controller (rev a3)
00:04.0 Ethernet controller: nVidia Corporation nForce2 Ethernet Controller (rev↩
     a1)
00:06.0 Multimedia audio controller: nVidia Corporation nForce2 AC97 Audio ↩
    Controler (MCP) (rev a1)
```

```
00:08.0 PCI bridge: nVidia Corporation nForce2 External PCI Bridge (rev a3)
00:09.0 IDE interface: nVidia Corporation nForce2 IDE (rev a2)
00:1e.0 PCI bridge: nVidia Corporation nForce2 AGP (rev a2)
01:09.0 Ethernet controller: Intel Corporation 82557/8/9 [Ethernet Pro 100] (rev⇀
    0c)
02:00.0 VGA compatible controller: nVidia Corporation NV18 [GeForce4 MX - nForce⇀
    GPU] (rev a3)
```

Les nœuds sont connectés au réseau interne à la grappe par leur carte d'interface réseau *Intel Ethernet Pro 100*. Les nœuds ont tous des disques *QUANTUM FIREBALLP AS20.5*.

I.2 Résultats des bancs d'essais

Nous utilisons les abbréviations décrites à la section 4.2 pour les résultats présentés dans le reste cette annexe, soit

TH	`thakur`
N2	`node02`, domaine 0, 1 Go de mémoire
N4	`node04`, limité à 512 Mo de mémoire
clTH@N2	clone de `thakur`, sur `node02`, 512 Mo de mémoire

Les clones sont sous *Xen* sauf dans le cas d'indications contraires.

I.2.1 Performance générale, compilation du kernel

Nous mesurons le temps de compilation du kernel 2.6.16.36 standard, sans rustine et dans sa configuration par défaut.

Le script de test utilisé est présenté au listage I.1.

Listage I.1 Script de mesure du temps de compilation du kernel

```
#!/bin/bash

for i in `seq 1 6`; do
        cd linux-2.6.16.36
        make mrproper > /dev/null 2>&1
        make defconfig > /dev/null 2>&1
        f=`mktemp`
        /usr/bin/time -o ${f} -p make > /dev/null 2>&1
        cat ${f}
        rm ${f}
        cd -
done
```

Tableau I.1 Mesures du temps de compilation sur N4

Essai	real (s)	user (s)	sys (s)
1	290.08	270.98	18.27
2	291.44	270.90	18.35
3	289.96	270.99	18.35
4	289.63	271.42	17.93
5	289.72	270.97	18.39
Moyenne	290.17	271.05	18.26
Écart-type	0.73	0.21	0.19

Les résultats obtenus sont présentés aux tableaux I.1, I.2 et I.3.

Sous *QEMU*, nous obtenons les résultats présentés au tableau I.4.

I.2.2 Performance générale, *OSDB*

Nous avons mesuré la performance à l'aide du banc de test *OSDB* version 0.21, sur la base de donnée PostgreSQL version 8.0.8. Nous utilisons les mêmes paramètres que CLARK et al. (2004).

Le script de test utilisé est présenté au listage I.2.

Les résultats obtenus sont présentés aux tableaux I.5, et I.6.

Tableau I.2 Mesures du temps de compilation sur clTH@N2, sans mémoire d'état

Essai	**real (s)**	**user (s)**	**sys (s)**
1	403.88	253.48	39.5
2	403.99	254.29	38.48
3	403.80	254.08	39.44
4	404.05	251.99	40.74
5	403.86	254.39	38.14
Moyenne	403.92	253.65	39.26
Écart-type	0.1	0.99	1.02

Tableau I.3 Mesures du temps de compilation sur clTH@N2, avec mémoire d'état

Essai	**real (s)**	**user (s)**	**sys (s)**
1	442.43	250.17	40.77
2	443.54	249.97	41.98
3	442.21	249.77	42.61
4	441.97	249.01	43.79
5	441.79	249.07	42.12
Moyenne	442.39	249.6	42.25
Écart-type	0.69	0.53	1.09

Tableau I.4 Mesures du temps de compilation sur clTH@N2, sans mémoire d'état, sous *QEMU*

Essai	**real (s)**	**user (s)**	**sys (s)**
1	8324.98	7498.58	592.35
2	8318.42	7509.31	577.38
3	8314.16	7496.06	585.78
4	8325.41	7502.31	588.31
5	8298.67	7481.3	584.31
Moyenne	8316.33	7497.51	585.63
Écart-type	10.93	10.35	5.53

Listage I.2 Script de mesure de performance sous *OSDB*

```
#!/bin/bash

su postgres -c "osdb-pg --dbuser postgres --generate-files --size 40m --datadir ↩
    /tmp"

for i in 'seq 1 6'; do
        echo "------ "$i
        su postgres -c "osdb-pg --dbuser postgres --postgresql=no_hash_index --↩
            datadir /tmp"
done
```

Tableau I.5 Mesures de la performance de PostgreSQL lors de l'exécution du test *OSDB*, sur N4

Essai	**Mixed OLTP (tuples/s)**	**Mixed IR (tuples/s)**
1	1307.59	130.21
2	1323.68	127.10
3	1370.95	128.25
4	1319.24	129.67
5	1375.87	141.26
Moyenne	1339.47	131.30
Écart-type	31.59	5.70

Tableau I.6 Mesures de la performance de PostgreSQL lors de l'exécution du test *OSDB*, sur clTH@N2, avec mémoire d'état

Essai	**Mixed OLTP (tuples/s)**	**Mixed IR (tuples/s)**
1	203.25	104.90
2	211.32	120.90
3	172.43	116.82
4	161.07	107.82
5	167.53	118.16
Moyenne	183.12	113.72
Écart-type	22.61	6.95

Listage I.3 Script de mesure de performance disque avec *DBench*

```
#!/bin/bash

for i in 1 2 3 4 5 6 9 12 15 18 21 24; do
        echo BEGIN: `date`
        for j in `seq 1 10`; do
                dbench -c /usr/share/dbench/client_oplocks.txt $i | grep Throu
        done
        echo END: `date`
done
```

I.2.3 Performance disque, *DBench*

DBench permet de tester les performance en accès au système de fichiers. Nous utilisons *DBench* en version 2.0.

Nous utilisons le script simple présenté au listage I.3 pour générer les données.

Les résultats bruts obtenus sont présentés aux tableaux I.7, I.8, I.9 et I.10.

Tableau I.7: Mesures de la bande passante système de fichiers sur TH. Moyenne : 231.93 Mo/s, éart-type : 58.29 Mo/s

Nb procs *DBench*	**B. pass. (Mo/s)**	**Nb procs *DBench***	**B. pass. (Mo/s)**
1	323.83	9	187.64
1	331.99	9	203.07
1	322.71	9	220.04
1	326.49	9	216.17
1	98.06	9	203.78
1	325.14	9	186.99
1	327.69	9	206.87
1	316.74	9	221.16

Suite à la page suivante

Tableau I.7 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
1	325.16	9	179.71
1	328.16	9	200.13
2	201.75	12	260.76
2	328.37	12	293.00
2	120.63	12	226.26
2	178.77	12	201.40
2	147.55	12	220.73
2	126.39	12	267.98
2	173.07	12	205.99
2	327.73	12	249.98
2	166.83	12	224.40
2	137.40	12	229.26
3	161.44	15	277.70
3	320.78	15	205.29
3	147.60	15	224.81
3	326.10	15	208.13
3	153.11	15	213.65
3	330.70	15	246.06
3	170.10	15	244.46
3	325.37	15	237.47
3	147.25	15	208.54
3	328.05	15	202.68
4	176.14	18	258.93
4	151.62	18	275.15
4	289.27	18	247.30

Suite à la page suivante

Tableau I.7 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
4	183.94	18	231.84
4	323.96	18	269.43
4	323.47	18	263.06
4	195.28	18	274.00
4	325.46	18	230.57
4	223.94	18	224.50
4	173.56	18	250.89
5	216.36	21	276.97
5	199.34	21	230.30
5	177.05	21	212.41
5	163.93	21	286.44
5	145.73	21	212.89
5	157.27	21	202.48
5	318.12	21	290.75
5	159.61	21	271.74
5	167.74	21	280.65
5	186.00	21	248.41
6	230.60	24	268.00
6	176.57	24	189.68
6	205.30	24	242.94
6	177.01	24	271.77
6	212.93	24	208.74
6	280.18	24	244.86
6	187.35	24	289.60
6	184.15	24	258.81

Suite à la page suivante

Tableau I.7 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
6	170.22	24	268.53
6	180.67	24	268.39

Tableau I.8: Mesures de la bande passante système de fichiers sur N4 ayant monté un système de fichiers à partir de TH par NFS, en lecture-écriture. Moyenne : 10.94 Mo/s, écart-type : 0.32 Mo/s

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
1	11.52	9	11.07
1	11.22	9	10.91
1	11.26	9	10.98
1	11.28	9	11.20
1	11.25	9	10.91
1	11.44	9	11.07
1	12.24	9	11.13
1	11.17	9	11.16
1	11.51	9	11.24
1	11.69	9	11.19
2	10.60	12	11.14
2	10.65	12	11.04
2	10.53	12	10.89
2	10.66	12	11.22

Suite à la page suivante

Tableau I.8 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
2	11.13	12	11.10
2	10.46	12	11.07
2	10.45	12	11.17
2	10.97	12	11.26
2	10.84	12	10.89
2	10.84	12	11.24
3	11.06	15	10.96
3	11.17	15	11.04
3	11.18	15	10.86
3	10.53	15	11.14
3	11.07	15	11.11
3	11.33	15	10.94
3	10.83	15	10.85
3	10.84	15	10.97
3	11.10	15	11.07
3	11.04	15	10.99
4	11.31	18	10.79
4	10.62	18	10.58
4	10.69	18	11.09
4	10.99	18	10.76
4	11.08	18	10.68
4	11.11	18	10.69
4	10.90	18	10.72
4	10.81	18	10.63
4	10.55	18	10.97

Suite à la page suivante

Tableau I.8 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
4	10.96	18	10.76
5	10.86	21	10.71
5	10.67	21	10.82
5	10.90	21	10.71
5	10.63	21	10.54
5	11.04	21	10.68
5	10.83	21	10.46
5	11.15	21	10.57
5	11.16	21	10.57
5	10.60	21	10.63
5	11.07	21	10.59
6	11.39	24	10.47
6	11.29	24	10.54
6	11.20	24	10.46
6	11.37	24	10.43
6	11.47	24	10.34
6	11.28	24	10.50
6	11.15	24	10.33
6	11.20	24	10.53
6	11.07	24	10.40
6	11.30	24	10.67

Tableau I.9: Mesures de la bande passante système de fichiers sur clTH@N2, sans mémoire d'état. Moyenne : 360.59 Mo/s, écart-type : 4.63 Mo/s

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
1	345.27	9	359.89
1	349.76	9	360.97
1	378.71	9	357.55
1	353.67	9	359.04
1	375.18	9	361.72
1	346.95	9	361.08
1	377.89	9	361.95
1	347.09	9	356.79
1	369.67	9	361.23
1	352.76	9	361.33
2	363.41	12	357.77
2	375.15	12	361.85
2	363.44	12	360.09
2	363.85	12	357.95
2	359.85	12	361.68
2	364.76	12	360.83
2	363.09	12	358.35
2	361.70	12	360.77
2	365.82	12	357.91
2	361.08	12	359.66
3	365.22	15	356.26
3	358.20	15	358.91

Suite à la page suivante

Tableau I.9 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
3	364.70	15	358.77
3	355.13	15	358.97
3	365.92	15	360.18
3	357.11	15	360.57
3	362.51	15	360.48
3	356.77	15	358.79
3	366.44	15	358.49
3	363.31	15	359.16
4	360.01	18	358.75
4	362.13	18	358.98
4	360.40	18	359.98
4	359.23	18	359.67
4	361.32	18	359.85
4	362.69	18	359.12
4	358.98	18	358.54
4	365.68	18	356.95
4	359.57	18	361.03
4	360.99	18	359.93
5	364.13	21	359.84
5	362.58	21	359.95
5	358.94	21	357.66
5	364.57	21	357.83
5	361.83	21	360.49
5	359.51	21	359.51
5	363.30	21	360.87

Suite à la page suivante

Tableau I.9 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
5	361.44	21	356.32
5	358.03	21	360.72
5	363.90	21	360.46
6	360.15	24	358.31
6	361.34	24	359.25
6	366.42	24	360.73
6	359.72	24	360.06
6	363.75	24	359.77
6	359.98	24	360.32
6	361.51	24	360.68
6	361.07	24	359.56
6	357.96	24	359.95
6	361.09	24	360.02

Tableau I.10: Mesures de la bande passante système de fichiers sur clTH@N2, avec mémoire d'état. Moyenne : 9.00 Mo/s, écart-type : 0.29 Mo/s

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
1	8.77	9	8.76
1	8.94	9	8.88
1	8.97	9	9.16
1	8.72	9	9.01

Suite à la page suivante

Tableau I.10 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
1	9.05	9	9.14
1	8.96	9	9.23
1	9.33	9	9.28
1	8.99	9	9.39
1	8.16	9	9.03
1	8.70	9	9.00
2	8.79	12	8.97
2	8.73	12	8.95
2	8.27	12	8.89
2	9.17	12	9.10
2	8.46	12	9.03
2	9.03	12	8.93
2	8.69	12	8.99
2	8.64	12	8.99
2	8.60	12	9.21
2	8.46	12	8.83
3	9.52	15	9.09
3	9.08	15	8.75
3	9.06	15	9.06
3	9.13	15	8.86
3	9.24	15	8.89
3	9.14	15	8.80
3	9.42	15	9.03
3	8.98	15	8.89
3	9.02	15	8.98

Suite à la page suivante

Tableau I.10 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
3	9.20	15	8.96
4	9.54	18	8.86
4	9.16	18	8.98
4	9.22	18	8.90
4	8.97	18	9.14
4	9.39	18	8.89
4	9.49	18	8.84
4	9.38	18	8.79
4	9.54	18	8.83
4	9.23	18	8.95
4	9.31	18	8.90
5	9.31	21	8.92
5	9.45	21	8.78
5	9.25	21	8.77
5	9.33	21	8.69
5	9.32	21	8.77
5	9.41	21	8.75
5	9.27	21	8.76
5	9.32	21	9.01
5	9.60	21	8.74
5	9.54	21	8.84
6	8.75	24	8.51
6	9.08	24	8.50
6	9.35	24	8.57
6	9.06	24	8.66

Suite à la page suivante

Tableau I.10 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
6	9.40	24	8.73
6	8.92	24	8.74
6	9.33	24	8.65
6	9.33	24	8.53
6	9.35	24	8.77
6	9.29	24	8.64

Listage I.4 Script de mesure de performance avec *FourInARow*

```
#!/bin/bash
for i in makefiles/*; do
        echo "--- Using makefile $i"
        cp $i freebench-1.03/Makefile
        cd freebench-1.03
        make clean 1> /dev/null 2>&1
        make test 1> /dev/null 2>&1
        ./utils/runbenchmark ref fourinarow | grep seconds
        cd ..
done
```

Tableau I.11 Mesures de l'impact sur l'éxecution de *FourInARow* selon les options d'optimisation passées au compilateur GCC, sur N2

Paramètres d'optimisation	**Temps d'exécution (s)**
`-O0`	149.19
`-O1`	74.23
`-O2`	70.47
`-O3`	59.5
`-O0 -march=athlon-xp`	150.55
`-O1 -march=athlon-xp`	68.36
`-O2 -march=athlon-xp`	70.73
`-O3 -march=athlon-xp`	55.95

I.2.4 Performance en calcul, *FourInARow*

Nous utilisons le test *FourInARow* de la suite *FreeBench* en version 1.03.

Nous utilisons le script présenté au listage I.4 pour effectuer les tests. Il fait appel à plusieurs `Makefile` (FSFMAKE, 2007) contenant des options d'optimisation de compilation variées.

Nous avons d'abord comparé rapidement différentes options d'optimisation à la compilation et obtenu les résultats présentés au tableau I.11. Ce test a été exécuté sur N2, deux fois par chacune des optimisations. Nous avons conservé le deuxième résultat.

Tableau I.12 Mesures du temps d'exécution de *FourInARow* sur N4

Essai	**Temps d'exécution (s)**
1	55.14
2	55.16
3	55.15
4	55.13
5	55.16
Moyenne	55.15
Écart-type	0.01

Tableau I.13 Mesures du temps d'exécution de *FourInARow* sur clTH@N2, sans mémoire d'état

Essai	**Temps d'exécution (s)**
1	56.83
2	56.77
3	56.84
4	56.81
5	56.76
Moyenne	56.8
Écart-type	0.04

En utilisant seulement les optimisations de compilation ayant le mieux performé (`-O3 -march=athlon-xp`), nous obtenons les résultats donnés aux tableaux I.12, I.13 et I.14.

Tableau I.14 Mesures du temps d'exécution de *FourInARow* sur clTH@N2, avec mémoire d'état

Essai	**Temps d'exécution (s)**
1	56.61
2	56.73
3	56.67
4	56.71
5	56.74
Moyenne	56.69
Écart-type	0.05

I.2.5 Bande passante, *Netperf*

Pour évaluer la bande passante réseau TCP/IP, nous avons utilisé le script présenté au listage I.5, tiré du travail de MORIN (2006), avec quelques modifications mineures.

Listage I.5 Script d'évaluation de la performance TCP/IP avec *Netperf*

```
#!/bin/sh

if [ -z "${1}" ]; then
        echo "please specify other testing host as first parameter"
        exit 1
fi

host="${1}"

if [ -z "${2}" ]; then
    first=1
else
    first=${2}
fi

if [ -z "${3}" ]; then
    last=8388608
else
    last=${3}
fi

retval=255
while [ ${retval} -eq 255 ]; do
    ssh ${host} 'start-stop-daemon --start --background --exec `which netserver`
        ''
    retval=${?}
done

sleep 1

i=${first}

while [ ${i} -le ${last} ]; do
```

```
    if [ ${i} -eq ${first} ]; then
        netperf -l -$( expr ${i} \* 50) -H ${host} -t TCP_STREAM -- -m ${i} -s ↩
            32768 -S 32768
    else
        netperf -l -$( expr ${i} \* 50) -H ${host} -t TCP_STREAM -- -m ${i} -s ↩
            32768 -S 32768 | tail -n 1
    fi
    i=$(expr ${i} \* 2)
done

retval=255
while [ ${retval} -eq 255 ]; do
    ssh ${host} 'start-stop-daemon --stop --exec `which netserver`'
    retval=${?}
done
```

Le tableau I.15 montre la bande passante entre deux systèmes physiques, et entre un système physique et son clone sans mémoire d'état sous *Xen*.

Pour ce qui est de *QEMU*, le tableau I.16 présente la bande passante entre deux systèmes physiques, et entre un système physique et son clone sans mémoire d'état sous *QEMU*.

I.2.6 Application de rendu d'image

Nous avons obtenu les temps de traitement présentés au tableau I.17 lors de nos exécutions de *Tachyon* sur 4 nœuds virtuels ou 4 nœuds physiques.

I.3 Impact d'*UnionFS*

Nous avons mesuré la bande passante avec *DBench* de la même manière que précédemment (cf. annexe I.2.3) sur un système de fichiers en mémoire (`tmpfs`), et

Tableau I.15 Mesures de la bande passante (Mbit/s) entre TH et N2, puis entre TH et clTH@N2

Taille paquet (octets)	**TH vers N2**	**TH vers clTH@N2**
1	2.12	1.68
2	4.12	3.38
4	7.41	6.64
8	13.5	11.59
16	20.38	17.3
32	29.43	25.81
64	46.04	41.97
128	62.98	58.78
256	75.85	72.88
512	83.66	81.99
1024	85.58	86.74
2048	90.32	89.6
4096	92.44	92.22
8192	93.22	93.14
16384	93.68	93.59
32768	93.88	93.76
65536	93.98	93.96
131072	94.03	94.02
262144	94.05	88.97
524288	94.08	91.51
1048576	94.09	91.5
2097152	94.1	90.88
4194304	94.1	91.19
8388608	94.1	91.19

Tableau I.16 Mesures de la bande passante (Mbit/s) entre TH et N2, puis entre TH et clTH@N2 sous QEMU

Taille paquet (octets)	**TH vers N2**	**TH vers clTH@N2 *QEMU***
1	2.12	0.22
2	4.12	0.59
4	7.41	1.26
8	13.5	2.56
16	20.38	4.62
32	29.43	2.2
64	46.04	12.09
128	62.98	21.44
256	75.85	31.12
512	83.66	39.98
1024	85.58	27.79
2048	90.32	63.54
4096	92.44	50.44
8192	93.22	75.88
16384	93.68	80.98
32768	93.88	80.06
65536	93.98	82.85
131072	94.03	83.79
262144	94.05	84.18
524288	94.08	85.96
1048576	94.09	85.09
2097152	94.1	85.34
4194304	94.1	85.22
8388608	94.1	85.37

Tableau I.17 Mesures du temps de rendu d'un pot de thé par *Tachyon*

Essai	**Nœuds physiques (s)**	**Nœuds virtuels (s)**
1	18.05	18.34
2	18.2	19.08
3	19.18	19.02
4	18.24	19.09
5	18.18	18.87
Moyenne	18.37	18.88
Écart-type	0.46	0.31

sur un système de fichiers en mémoire inclu comme seul système de fichiers d'une superposition. Le tableau I.18 présente le premier cas. Le tableau I.19 présente le deuxième cas. Enfin, nous avons également mesuré la bande passante sur une union d'un système de fichiers `tmpfs` superposé sur six autres `tmpfs`. Le tableau I.20 présente les résultats.

Tableau I.18: Mesures de la bande passante système de fichiers sur `tmpfs` sur N4. Moyenne : 424.03 Mo/s, écart-type 1.11 Mo/s

Nb procs *DBench*	**B. pass. (Mo/s)**	**Nb procs *DBench***	**B. pass. (Mo/s)**
1	427.23	9	420.61
1	416.24	9	423.12
1	423.10	9	420.66
1	422.24	9	419.92
1	419.95	9	421.64
1	427.04	9	422.89
1	419.79	9	420.25
1	431.14	9	420.38
1	416.61	9	421.95
1	429.07	9	426.15
2	423.21	12	423.59
2	429.23	12	426.08
2	439.21	12	427.13
2	424.43	12	426.25
2	435.87	12	426.75
2	436.45	12	423.03
2	436.53	12	422.81

Suite à la page suivante

Tableau I.18 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
2	431.06	12	423.73
2	429.48	12	424.71
2	429.88	12	423.18
3	428.29	15	423.17
3	429.98	15	425.29
3	425.25	15	422.00
3	431.21	15	423.61
3	425.11	15	425.11
3	427.84	15	424.02
3	433.79	15	424.71
3	432.16	15	420.21
3	424.11	15	422.55
3	430.17	15	401.01
4	421.17	18	421.12
4	421.48	18	424.83
4	431.11	18	423.30
4	424.91	18	424.95
4	427.24	18	419.74
4	424.46	18	419.44
4	422.80	18	420.94
4	422.34	18	420.57
4	430.71	18	416.59
4	423.06	18	420.72
5	426.21	21	421.66
5	424.70	21	423.71

Suite à la page suivante

Tableau I.18 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
5	428.61	21	418.26
5	423.00	21	421.69
5	422.25	21	423.13
5	424.51	21	421.62
5	428.80	21	419.80
5	424.89	21	423.04
5	427.74	21	422.71
5	423.73	21	419.42
6	421.90	24	423.16
6	422.60	24	420.78
6	424.17	24	424.72
6	420.59	24	420.11
6	423.25	24	421.12
6	421.90	24	422.09
6	422.84	24	424.14
6	418.73	24	422.05
6	423.36	24	419.79
6	423.15	24	419.71

Tableau I.19: Mesures de la bande passante système de fichiers à une seule couche `tmpfs` dans un système de fichiers *UnionFS* sur N4. Moyenne : 332.77 Mo/s, écart-type : 1.24 Mo/s

Nb procs *DBench*	**B. pass. (Mo/s)**	**Nb procs *DBench***	**B. pass. (Mo/s)**
1	347.33 9		331.30
1	348.61 9		331.97
1	342.50 9		331.93
1	343.08 9		335.37
1	340.33 9		330.53
1	343.87 9		330.48
1	342.56 9		332.41
1	336.47 9		330.20
1	340.40 9		332.86
1	339.02 9		329.90
2	336.98 12		333.90
2	339.94 12		332.45
2	336.99 12		331.95
2	335.04 12		331.30
2	333.63 12		328.72
2	340.82 12		329.18
2	337.37 12		330.99
2	338.80 12		332.74
2	336.60 12		332.38
2	333.57 12		330.06
3	339.51 15		329.99

Suite à la page suivante

Tableau I.19 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
3	336.98 15		331.36
3	330.23 15		328.31
3	334.57 15		330.77
3	329.91 15		332.88
3	336.00 15		330.98
3	333.62 15		330.95
3	334.73 15		330.02
3	334.59 15		330.86
3	336.17 15		329.47
4	333.50 18		330.81
4	334.15 18		330.55
4	332.06 18		328.12
4	336.12 18		330.83
4	333.91 18		329.83
4	333.17 18		331.17
4	333.81 18		326.13
4	334.63 18		328.16
4	333.55 18		328.80
4	335.37 18		329.64
5	331.77 21		329.99
5	331.25 21		329.65
5	329.78 21		329.47
5	332.41 21		330.70
5	332.92 21		328.79
5	330.72 21		329.36

Suite à la page suivante

Tableau I.19 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
5	334.06 21	330.05	
5	334.88 21	328.98	
5	332.27 21	330.55	
5	330.38 21	328.35	
6	332.10 24	327.28	
6	332.19 24	327.71	
6	329.74 24	328.60	
6	331.47 24	327.12	
6	330.23 24	330.29	
6	330.98 24	328.68	
6	333.84 24	328.97	
6	331.76 24	328.48	
6	331.85 24	329.67	
6	333.33 24	331.17	

Tableau I.20: Mesures de la bande passante système de fichiers à sept couches (la plus en haut en lecture-écriture, les six autres en lecture seulement) `tmpfs` dans un système de fichiers *UnionFS* sur N4. Moyenne : 330.43 Mo/s, écart-type : 1.48 Mo/s

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
1	336.75 9	330.60	

Suite à la page suivante

Tableau I.20 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
1	337.53 9		328.82
1	344.28 9		328.38
1	346.43 9		329.16
1	343.57 9		328.34
1	336.77 9		327.95
1	342.87 9		329.97
1	339.07 9		328.39
1	338.54 9		328.39
1	333.47 9		329.96
2	338.32 12		327.91
2	334.36 12		328.94
2	338.29 12		329.16
2	339.37 12		325.02
2	333.91 12		327.20
2	341.52 12		330.25
2	337.41 12		331.09
2	332.92 12		327.33
2	341.98 12		328.47
2	343.34 12		326.00
3	335.15 15		326.49
3	323.94 15		325.53
3	337.68 15		327.06
3	334.35 15		325.35
3	330.38 15		325.88
3	338.27 15		327.53

Suite à la page suivante

Tableau I.20 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
3	331.52 15		324.36
3	330.23 15		325.70
3	330.62 15		325.96
3	337.49 15		325.70
4	335.18 18		327.41
4	332.74 18		328.31
4	326.84 18		326.51
4	332.37 18		326.28
4	332.38 18		323.80
4	329.34 18		324.72
4	333.38 18		324.90
4	326.64 18		324.56
4	331.30 18		323.81
4	331.68 18		325.80
5	328.08 21		323.38
5	335.60 21		323.57
5	331.86 21		323.91
5	330.92 21		325.50
5	331.09 21		327.36
5	333.45 21		323.43
5	330.31 21		324.93
5	332.13 21		326.05
5	332.35 21		325.71
5	329.79 21		328.56
6	329.84 24		325.72

Suite à la page suivante

Tableau I.20 – suite de la page précédente

Nb procs *DBench*	B. pass. (Mo/s)	Nb procs *DBench*	B. pass. (Mo/s)
6	333.68 24	328.36	
6	331.17 24	326.80	
6	331.38 24	325.05	
6	329.62 24	325.89	
6	329.15 24	324.07	
6	332.26 24	325.91	
6	329.17 24	327.58	
6	332.28 24	325.54	
6	330.12 24	327.02	

I.4 Profilage du démarrage

Le tableau I.21 présente les mesures de profilage de l'initialisation du noyau sous *Xen*, du lancement du système jusqu'au démarrage de l'*initramfs*. Puis, on mesure le temps passé à exécuter l'*initramfs*, jusqu'à ce que `init` soit appelé. Ces mesures sont effectuées sur un clone sans mémoire d'état, avec 512 Mo de mémoire. La moyenne de la différence entre le moment du début de l'exécution du noyau et le moment du début de l'exécution de l'*initramfs*, ainsi que la moyenne entre le début et la fin de l'exécution de l'*initramfs* sont retenues et présentées au chapitre 4.

Tableau I.21 Mesures du temps d'initialisation du noyau et du temps d'exécution de l'*initramfs*

Essai	**Début noyau (s)**	**Début initramfs (s)**	**Fin initramfs (s)**
1	695944.14	695944.47	695946.90
2	696172.95	696173.27	696176.74
3	696262.52	696262.84	696266.29
4	696399.84	696400.17	696403.66
5	696497.85	696498.18	696500.65

ANNEXE II

EXEMPLES DE SCRIPTS `FSLOADER`

Cette section contient deux scripts `fsloaders` permettant de monter la racine du système de fichiers final lors de l'exécution du processus `init` dans un *initramfs*.

Les scripts `fsloaders` suivent tous une interface standard, ils doivent au moins supporter les options de ligne de commande suivantes :

- `--help` donne un message d'aide à afficher à l'usager lors du démarrage, si demandé ;
- `--parameters` donne une liste concise des paramètres à afficher à l'usager lors du démarrage, si demandé ;
- `--do-mount liste_arguments` effectue le montage selon les paramètres envoyés. Ces paramètres correspondent aux options à placer après le nom du dispositif (dont le nom correspond au nom du script `fsloader`) sur la ligne de commande `root=` (cf section 2.2.2.1).

Les listages II.1 et II.2 présentent respectivement le script qui monte un système de fichiers NFS et un `fsloader` qui monte un système de fichiers à partir d'un (ou plusieurs) fichier d'image en boucle stocké sur un répertoire NFS.

Listage II.1 Exemple de script de montage supportant `root=nfs[...]` pour *Genkernel. Droit de reproduction partagé avec Tim Yamin et Eric Edgar, développeurs de Genkernel.*

```
#!/bin/sh
# [...]
# NFS root fs loader for genkernel
#
# ./nfs --do-mount mountpoint ip dir mount_opts
#

. /etc/initrd.defaults
. "${LIBGMI}/libgmi.sh"


parameters() {
        echo 'ip dir mount_opts'
}


help() {
cat << EOHELP
usage: nfs:<ip>:<dir>[:mount_opts]
example: 'root=nfs:192.168.0.1:/mnt/rootfs'
         'root=nfs:10.0.0.2:/mnt/system1:ro,nolock'
parameters are
<ip>            IP address of the NFS server
<dir>           directory to mount
[mount_opts]    parameters to pass at mounting of the filesystem
EOHELP
}


do_mount() {
        # unwrapping parameters for human beings
        local mountpoint="${1}"
        local server_ip="${2}"
        local dir="${3}"
        local mount_opts="${4}"
        local _mount_opts

        is_set ${mount_opts} && _mount_opts="-o ${mount_opts}"
```

```
39
40          dbg_msg "mount -t nfs ${_mount_opts} ${server_ip}:${dir} ${mountpoint}"
41          mount -t nfs ${_mount_opts} ${server_ip}:${dir} ${mountpoint}
42          assert "$?" "Could not mount NFS share '${server_ip}:${dir}'" || return ↩
                1
43  }
44
45
46  case "${1}" in
47          "--help")
48                  help
49                  ;;
50          "--parameters")
51                  parameters
52                  ;;
53          "--do-mount")
54                  shift
55                  do_mount $@
56                  exit $?
57                  ;;
58          *)
59                  exit 1
60                  ;;
61  esac
```

Listage II.2 Exemple de script de montage supportant `root=loop+nfs[...]` pour *Genkernel.* On remarque l'appel à `nfs` et `loop`. *Droit de reproduction partagé avec Tim Yamin et Eric Edgar, développeurs de Genkernel.*

```
#!/bin/sh
# [...]
#
# Loop file over NFS fs loader for genkernel
#
# ./loop+nfs --do-mount mountpoint ip path
#

. /etc/initrd.defaults
. "${LIBGMI}/libgmi.sh"


parameters() {
        echo 'ip path'
}


help() {
cat << EOHELP
usage: loop+nfs:<ip>:<path>
example: 'root=loop+nfs:10.0.0.1:/mnt/modules'
         'root=loop+nfs:10.0.0.9:/mnt/images/image.squashfs'
parameters are
<ip>            IP address of the NFS server
<path>          directory containing images OR a single image's full path
EOHELP
}


do_mount() {
        # unwrapping parameters for human beings
        local mountpoint="${1}"
        local server_ip="${2}"
        local path="${3}"
        local nfsmnt
        local umnt
        local lfile
        local retval
```

```
39
40          nfsmnt=$( mkmntpoint )
41
42          # try 1 : path is a dir, we mount it and loop mount every file in it
43          # in the $UNIONS directory, so they get unionized into the rootfs
44          # by init later on
45
46          ${FSLOADERS}/nfs --do-mount ${nfsmnt} ${server_ip} ${path} ro,nolock
47
48          if [ "$?" = "0" ]
49          then
50                  dbg_msg "Mounting loop files"
51                  cd ${nfsmnt}
52
53                  umnt=${mountpoint}
54                  got_good_root="no"
55                  for lfile in *
56                  do
57                          good_msg "\t\t${lfile}"
58                          ${FSLOADERS}/loop --do-mount ${umnt} ${lfile}
59
60                          if [ "$?" = "0" ]
61                          then
62                                  if [ "${USE_UNIONFS}" != "yes" ]
63                                  then
64                                          # get out as soon as we have one loop ↩
                                                mounted
65                                          return 0
66                                  fi
67
68                                  got_good_root="yes"
69
70                                  # room for another, please
71                                  umnt=$( mkumntpoint )
72                          fi
73                  done
74                  # remove the last unused union mountpoint
75                  rm -rf ${umnt}
76
77                  [ "${got_good_root}" = "no" ] && return 1
78          else
79                  # try 2 : path is a file, mount the dirname, then loop mount ↩
```

```
                    only this file
80
81              ${FSLOADERS}/nfs --do-mount ${nfsmnt} ${server_ip} $( dirname ${path} ) ro,nolock
82
83              if [ "${?}" = "0" ]
84              then
85                      cd ${nfsmnt}
86                      ${FSLOADERS}/loop --do-mount ${mountpoint} $( basename ${path} )
87                      return $?
88              else
89                      dbg_msg "Was neither a file or directory, bailing out"
90                      return 1
91              fi
92      fi
93
94      return 0
95  }
96
97
98  case "${1}" in
99          "--help")
100                 help
101                 ;;
102         "--parameters")
103                 parameters
104                 ;;
105         "--do-mount")
106                 shift
107                 do_mount $@
108                 exit $?
109                 ;;
110         *)
111                 exit 1
112                 ;;
113 esac
```

ANNEXE III

ACCÈS À LA DOCUMENTATION ET AU CODE

Le lecteur intéressé pourra accéder au code et à la documentation produits pour le projet à *Genkernel* l'aide de Subversion (COLLINS-SUSSMAN et al., 2007) à l'URL suivant :

`https://svn.gentooexperimental.org/genkernel`

Le code et la documentation de notre prototype expérimental *Napalm* sont disponibles à l'aide de l'outil *Git* (GIT, 2007) à l'URL suivant :

`git://repoz.org/napalm.git`

Pour *Clone* :

`git://repoz.org/clone.git`

Les extensions de *Clone* sont à :

`git://repoz.org/clone-extensions.git`

Enfin, le *Portage Overlay* utilisé est disponible à :

`git://repoz.org/gentoo-overlay-clone-napalm.git`

ANNEXE IV

FICHIERS DE CONFIGURATION

Cette annexe présente certains fichiers de configuration utilisés lors de notre travail.

IV.1 Configuration `clones.conf`

Nous avons utilisé le fichier de configuration des clones présenté au listage IV.1 lors de nos tests sur la grappe **thakur**.

Listage IV.1 Fichier de configuration des clones

```
#/etc/clones.conf

server: 10.0.0.1

os_root_device: 10.0.0.1:/
os_root_fs_type: nfs
os_root_fs_options: nfsvers=3

os_home_device: 10.0.0.1:/home
os_home_fs_type: nfs
os_home_fs_options: nfsvers=3

groups:
  # Start a group; all nodes in that group share settings
  attack of the clones:
    # Define hosts in this group, with their own private settings.
    hosts:
      clone01:
      clone02:
      clone03:
      clone04:
      clone05:
      clone06:
```

```
24        clone07:
25        clone08:
26        clone09:
27        clone10:
28        clone11:
29        clone12:
30        clone13:
31        clone14:
32        clone15:
33        clone16:
34        clone17:
35        clone18:
36        clone19:
37        clone20:
38        clone21:
39        clone22:
40        clone23:
41        clone24:
42
```

IV.2 Configuration NFS

Le fichier de configuration des exportations par le serveur NFS est présenté au listage IV.2.

Listage IV.2 Configuration du serveur NFS

```
1 # /etc/exports: NFS file systems being exported.  See exports(5).
2
3 /                    10.0.0.0/8(ro,async,no_root_squash)
4 /var/state/systems/  10.0.0.0/8(rw,async,no_root_squash)
5 /home/               10.0.0.0/8(rw,async)
```

ANNEXE V

PROPOSITION D'AMÉLIORATION : DÉTERMINATION DES TYPES DE MACHINES VIRTUELLES SUPPORTÉES SUR UN SERVEUR *NAPALM*

Pour permettre des déploiements plus efficaces, il serait nécessaire de connaître divers aspects des serveurs *Napalm* présents sur un réseau :

- les éléments (mémoire, bande passante, etc.) qu'ils sont prêts à fournir à des machines virtuelles (en considérant leurs ressources, l'utilisation actuelle de ces dernières, et d'éventuelles réservations) ;
- le type de machines virtuelles qu'ils supportent.

Alors que le premier élément est un sujet de travaux futurs, la deuxième partie du problème pourrait être solutionnée : il serait relativement simple d'exploiter une caractéristique ajoutée très récemment à la librairie *libvirt*.

Le listage suivant est tiré de la page Web du projet[1] :

```
Discovering virtualization capabilities (Added in 0.2.1)

As new virtualization engine support gets added to libvirt, and to
handle cases like QEmu supporting a variety of emulations, a query
interface has been added in 0.2.1 allowing to list the set of
supported virtualization capabilities on the host:

    char * virConnectGetCapabilities (virConnectPtr conn);

The value returned is an XML document listage the virtualization
capabilities of the host and virtualization engine to which @conn is
```

[1] `http://libvirt.org/format.html#Capa1`, page visionnée le 18 mai 2007.

connected. One can test it using virsh command line tool command 'capabilities', it dumps the XML associated to the current connection. For example in the case of a 64 bits machine with hardware virtualization capabilities enabled in the chip and BIOS you will see

```
<capabilities>
  <host>
    <cpu>
      <arch>x86_64</arch>
      <features>
        <vmx/>
      </features>
    </cpu>
  </host>

  <!-- xen-3.0-x86_64 -->
  <guest>
    <os_type>xen</os_type>
    <arch name="x86_64">
      <wordsize>64</wordsize>
      <domain type="xen"></domain>
      <emulator>/usr/lib64/xen/bin/qemu-dm</emulator>
    </arch>
    <features>
    </features>
  </guest>

  <!-- hvm-3.0-x86_32 -->
  <guest>
    <os_type>hvm</os_type>
    <arch name="i686">
      <wordsize>32</wordsize>
      <domain type="xen"></domain>
      <emulator>/usr/lib/xen/bin/qemu-dm</emulator>
      <machine>pc</machine>
      <machine>isapc</machine>
      <loader>/usr/lib/xen/boot/hvmloader</loader>
    </arch>
    <features>
    </features>
  </guest>
  ...
```

```
</capabilities>

The fist block indicates the host hardware capbilities, currently it
is limited to the CPU properties but other information may be
available, it shows the CPU architecture, and the features of the chip
(the feature block is similar to what you will find in a \textsl{Xen} fully
virtualized domain description).

The second block indicates the paravirtualization support of the \textsl{Xen}
support, you will see the os_type of xen to indicate a paravirtual
kernel, then architecture informations and potential features.

The third block gives similar informations but when running a 32 bit
OS fully virtualized with \textsl{Xen} using the hvm support.

This section is likely to be updated and augmented in the future, see
the discussion which led to the capabilities format in the
mailing-list archives.
```

Nous croyons que de lier ce détail à une interface publique via un chemin d'accès déterminé permettrait de résoudre cette partie du problème.

Par exemple, on verrait bien un chemin d'accès HTTP comme celui-ci :

`http://[...]/napalm.cgi/capabilities.xml`

Les outils d'auto-découverte de serveurs *Napalm* pourraient exploiter ces informations et filtrer les valeurs retournées en fonction des caractéristiques de machines virtuelles recherchées par un usager.