NFS-Ganesha évolutions d'un serveur NFS pour le HPC du Terascale à l'Exascale

Philippe Deniel

▶ **To cite this version:**

Philippe Deniel. NFS-Ganesha évolutions d'un serveur NFS pour le HPC du Terascale à l'Exascale. Système d'exploitation [cs.OS]. Université Paris-Saclay, 2023. Français. NNT : 2023UPASG056 . tel-04496216

HAL Id: tel-04496216

https://theses.hal.science/tel-04496216

Submitted on 8 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

universite
PARIS-SACLAY

NFS-Ganesha : évolutions d'un serveur NFS pour le HPC du Terascale à l'Exascale

NFS-Ganesha : evolutions of a NFS server for the HPC from Terascale to Exascale

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580 : Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique
Référent : Université de Versailles Saint-Quentin-en-Yvelines

Thèse préparée dans l'unité de recherche LI PARAD (Université Paris-Saclay, UVSQ), sous la direction de **Soraya ZERTAL**, Professeure

Thèse soutenue à Guyancourt, le 21 septembre 2023, par

Philippe DENIEL

Composition du jury

Membres du jury avec voix délibérative

William JALBY Professeur des universités, Université Paris-Saclay	Président
Emmanuel JEANNOT Professeur, HDR, Université de Bordeaux	Rapporteur et examinateur
Luiz Angelo STEFFENEL Professeur, HDR, Université de Reims Champagne-Ardenne	Rapporteur et examinateur
Joël FALCOU Maitre de Conférences, Université Paris-Saclay	Examinateur
Sophie ROBERT Maitre de Conférences, Université d'Orléans	Examinatrice

THESE DE DOCTORAT

NNT : 2023UPASG056

ÉCOLE DOCTORALE
Sciences et technologies
de l'information et de
la communication (STIC)

université
PARIS-SACLAY

Titre : NFS-Ganesha : évolutions d'un serveur NFS pour le HPC du Terascale à l'Exascale

Mots clés : NFS, HPC, Calcul Haute Performance, système de fichiers, Exascale, Stockage massif

Résumé : Cette thèse présente NFS-Ganesha, un serveur NFS en espace utilisateur pour le HPC, et ses évolutions depuis sa création à l'aube des années 2000 jusqu'à la période Exascale actuelle.

Créé à l'origine pour des besoins opérationnels liés à l'exploitation des grands systèmes de stockage, NFS-Ganesha a été pensé pour être générique et parallélisé. L'apparition conjointe des systèmes de fichiers parallèles, donnant naissance aux architectures "data-centriques" des centres de calcul, et celle du protocole NFSv4 vont faire évoluer NFS-Ganesha qui va devenir un serveur NFS générique capable de s'interfacer avec de nombreux *back-ends*. L'évolution de NFSv4, sous la forme de NFSv4.1 et du protocole pNFS, fera de NFS-Ganesha un standard adopté par une forte communauté open-source impliquant chercheurs et industriels.

NFS-Ganesha sera utilisé pour réaliser la fonctionnalité IO-Proxy, et la création de nouveaux protocoles parallèles afférents.

Impliqués dans des projets de R&D européens, NFS-Ganesha servira à implémenter la fonctionnalité de serveur éphémère afin de répondre aux exigences de l'Exascale.

Title : NFS-Ganesha : evolutions of a NFS server for the HPC from Terascale to Exascale

Keywords : NFS, HPC, High Performance Computing, file systems, Exascale, Mass storage

Abstract : This thesis exposes NFS-Ganesha, a NFS server running in userland, dedicated to HPC, as well as its evolutions from the Terascale period of the early 2000 to Today's Exascale periode.

Developped because of operational needs, with close relations to the exploitation of large HPC storage systems, NFS-Ganesha was designed to be generic and parallel. The apparition of large parallel files systems, leading to to « data centric » architectures, and the joint apparition of the NFSv4 protocol made NFS-Gaesha a generic NFS server capable of being interfaced to many different backends. The evolution of NFSv4, via NFSv4.1 and its new pNFS feature made NFS-Ganesha a standard adopted by a strong opensource community involving researchers and industrials.

NFS-Ganesha will be used to build the IO-Proxy feature, with the creation of new parallel protocols. Involved in different european R&D projects, NFS-Ganesha will be used to implement epehemeral services to fulfill the requirements of Exascale systems.

Glossaire

Access Control List Les ACLs, ou Access Control Lists, sont des enregistrements permettant de définir de manière fine les autorisations données à certains utilisateurs ou groupes d'utilisateur pour accéder à une ressource.. 104

Amazon S3 Amazon S3 (Amazon Simple Storage Service) est un site d'hébergement de fichiers proposé par Amazon Web Services. Amazon S3 propose des services de stockage à travers des services Web (REST, SOAP et BitTorrent)1. . 98, 176

automounter Un automounter est une fonctionnalité du système d'exploitation Linux qui permet de monter une ressource distante de manière automatique quand un utilisateur souhaite y accéder. Cette action est transparente pour l'utilisateur. 158

back-end Le terme back-end désigne une ressource exploité au final par une ou plusieurs couches logicielles au sein d'une architecture logicielle.. 67, 68

benchmarking En informatique, un benchmark est un banc d'essai permettant de mesurer les performances d'un système pour le comparer à d'autres. . 51

big endian De l'anglais big-endian (même sens), popularisé dans le domaine informatique par Dany Cohen, en référence aux Voyages de Gulliver, roman satirique de Jonathan Swift datant de 1721, où est décrit des habitants mangeant des œufs à la coque par le gros ou petit bout.. 126

buffer En informatique, *buffer* est le terme anglais équivalent à mémoire tampon, une zone de mémoire virtuelle ou de disque dur utilisée pour stocker temporairement des données, notamment entre deux processus ou deux pièces d'équipement ne fonctionnant pas à la même vitesse.. 40

cluster Ensemble cohérent de machines connectées en réseau mais fonctionnant ensemble pour rendre un servicedistribué. Aussi appelé "grappe de machines".. 55, 57

clusters Au sein d'un système informatique, un cluster de serveurs est un groupe de serveurs et d'autres ressources indépendantes fonctionnant comme un seul système. Les serveurs sont généralement situés à proximité les uns des autres, et sont interconnectés par un réseau dédié.. 29, 109

conteneur Un conteneur est une structure de données, une classe, ou un type de données abstrait, dont les instances représentent des collections d'autres objets. Autrement dit, les conteneurs sont utilisés pour stocker des objets sous une forme organisée qui suit des règles d'accès spécifiques. On peut implémenter un conteneur de différentes façons, qui conduisent à des complexités en temps et en espace différentes. On choisira donc l'implémentation selon les besoins. . 183

credentials Authentifiant de connexion à un espace à accès restreint (sur internet notamment). Association d'un identifiant avec au moins un facteur supplémentaire d'authentification, comme un mot de passe par exemple.. 166, 185

daemon Acronyme de Deferred Auxiliary Executive MONitor, il s'agit d'un processus système qui s'exécutent en arrière plan, et qui implémente généralement un service du système d'exploitation. 48, 133, 159

data node Machine d'une architecture dédiée à la gestion des données et généralement dédiées au fonctionnement d'un serveur de données.. 97

dataset Jeu de donées, ensemble de données qui ont une raison d'être utilisées ensemble.. 183

doxygen Doxygen est un générateur de documentation sous licence libre capable de produire une documentation logicielle à partir du code source d'un programme. Pour cela, il tient compte de la syntaxe du langage dans lequel est écrit le code source, ainsi que des commentaires s'ils sont écrits dans un format particulier avec l'utilisation de balises particulières.. 50

hard link Un *hard link* est un mécanisme par lequel une entrée d'un système de fichier peut exister sous plusieurs noms différents dans différents répertoires tout en continuant à adresser le même contenu.Classiquement opérer sur les fichiers, des hardlinks implicites existent pour les répertoires, c'est typiquement le cas des entrées '.' et '..'. Les hardlinks ne doivent pas être confondus avec les liens symboliques, ou *symlinks*, qui sont des entrées spécifiques et autonomes qui renvoient à un chemin complet ou relatif.. 102

hyperviseur Serveur disposant des ressources matérielles et logicielles lui permettant de faire fonctionner des machines virtuelles. 76

inode Un nœud d'index ou inode (contraction de l'anglais index et node) est une structure de données contenant des informations à propos d'un fichier ou répertoire stocké dans certains systèmes de fichiers (notamment de type Linux/Unix). À chaque fichier correspond un numéro d'inode (i-number) dans le système de fichiers dans lequel il réside, unique au périphérique sur lequel il est situé.. 57, 80, 163

keytab Un keytab est un fichier permettant, dans le fonctionnement de Kerberos, d'authentifier un utilisateur ou un service sans utiliser un mot de passe.. 183

KNC Acronyme des processeurs *Knight Corner* d'Intel, mettant en oeuvre des architectures de type ManyCore. 72, 88

KNL Acronyme des processeurs *Knight Landing* d'Intel, mettant en oeuvre des architectures de type ManyCore. 72, 88

Labeled NFS Labeled NFS est une extension des fonctionnalités du protocole NFS permettant à celui-ci de gérer des labels similaires à ceux exploités dans le fonctionnement de SELinux. Il fait désormais partie du protocole RPCSEC_GSSv3. 153

LibC GNU C Library (glibc) est la bibliothèque standard C écrite par Roland McGrath pour le projet GNU. Il s'agit d'un logiciel libre, distribué selon les termes de la Licence publique générale limitée GNU. Depuis 2001, son principal contributeur et mainteneur est Ulrich Drepper3. . 65, 72, 99, 100, 167, 175

little endian De l'anglais little-endian (même sens), popularisés dans le domaine informatique par Dany Cohen, en référence aux Voyages de Gulliver, roman satirique de Jonathan Swift datant de 1721, où est décrit des habitants mangeant des œufs à la coque par le gros ou petit bout.. 126, 165

machine virtuelle Une machine virtuelle ou VM est un environnement entièrement virtualisé qui fonctionne sur une machine physique, nommée hyperviseur. Elle exécute son propre système d'exploitation (OS) et bénéficie des mêmes équipement qu'une machine physique : CPU, mémoire RAM, disque dur et carte réseau.. 76

MIC Architecture ManyCore, architecture qui regroupe au sein du même processeur une grande quantité de coeurs de calcul. 72

middleware Un middleware (ou intergiciel) est un logiciel qui agit comme une passerelle entre les autres applications, outils et bases de données pour offrir aux utilisateurs des services unifiés.. 98

Modular Supercomputing Architecture L'architecture de centre de calcul MSA est portée par le centre de calcul allemand FZJ et s'articule autour de différents composants, ou modules, dôtés de ressources spéccifiques et identifiées. Les travaux sur une telle architecture viennent allouer des ressources dans les modules en fonctions de leurs spécificités (par exemple des GPUs ou des processeurs neuromorphiques).. 97

offset Un offset désigne une adresse de manière relative. C'est une valeur entière représentant le déplacement en mémoire nécessaire, par rapport à une adresse de référence, pour atteindre une autre adresse. Autrement dit, l'offset est la distance séparant deux emplacements mémoire. . 154

OOM killer Le mécanisme OOM killer (Out-Of-Memory Killer) est un mécanisme de la dernière chance qui est incorporé au noyau Linux en cas de dépassement de la capacité mémoire. Si le système n'a plus assez de mémoire à allouer aux processus et que le swap a été lui aussi entièrement rempli alors le noyau n'a pas d'autre choix que de faire appel à son tueur à gages préféré : OOM killer. . 71

Openstack Swift Le stockage objet d'OpenStack s'appelle Swift. C'est un système de stockage de données redondant et évolutif. Les fichiers sont écrits sur de multiples disques durs répartis sur plusieurs serveurs dans un Datacenter. Il s'assure de la réplication et de l'intégrité des données au sein du cluster.. 98

pool Informatique. Un pool désigne un ensemble de ressources réutilisables, géré de façon commune pour un ensemble d'usagers (processus informatique, utilisateurs, ordinateurs, etc.).. 154

processus Un processus (en anglais, process), est un programme en cours d'exécution par un ordinateur. De façon plus précise, il peut être défini comme un ensemble d'instructions à exécuter, pouvant être dans la mémoire morte, mais le plus souvent chargé depuis la mémoire de masse vers la mémoire vive ; un espace d'adressage en mémoire vive pour stocker la pile, les données de travail ; des ressources permettant des entrées-sorties de données, comme des ports réseau.. 104, 146

proxy Un *proxy* est un serveur de procuration, c'est clasiquement un élément logiciel qui se positionne entre un client et un serveur afin de se substituer au client face au serveur. Ainsi, des logiques de cache peuvent être mise en oeuvre ce qui conduit à des optimisations de la charge au niveau du serveur.. 66

SFTP SFTP signifie *Secure File Transfer Protocol* ou protocole de transfert de fichiers SSH. C'est un serveur faisant partie de SSH ou Secure Shell qui a pour rôle d'assurer un transfert de fichiers. Il s'agit d'un remplaçant efficace et sécurisé pour l'établissement d'une session de terminal sur des machines UNIX.. 43

Slurm SLURM (Simple Linux Utility for Resource Management) est une solution open source d'ordonnancement de tâches informatiques qui permet de créer des grappes de serveurs sous Linux ayant une tolérance aux pannes, type ip-failover, ferme de calcul, système d'ordonnancement des tâches. . 98

sparse file Un *sparse file*, ou "fichier creux" ou encore "fichier à trou" est un fichier dont le contenu n'est pas d'un seul tenant. Certaines plages ne contiennent aucun contenu au niveau du système de fichiers et constituent ainsi des "trous".. 154, 170

SSHFS SSHFS est un système de fichiers basé sur la bibliothèque FUSE et la fonctionnatlité SFTP du produit SSH. Il permet, via une connection SSH, d'exposer le système de fichiers d'une machine à une autre, en réseau.. 43

stateful Un protocole stateful est un concept de mise en réseau . Un serveur reçoit les requêtes des clients. La transaction a lieu soit dans une série d'échanges ou il s'agit d'une demande unique suivie d'une réponse . Un processus " stateful " , le serveur doit consigner les résultats des opérations afin qu'ils puissent être rappelés ou repris.. 161

stateless Un protocole sans état est un protocole de communication qui n'enregistre pas l'état d'une session de communication entre deux requêtes successives. La communication est formée de paires requête-réponse indépendantes et chaque paire requête-réponse est traitée comme une transaction indépendante, sans lien avec les requêtes précédentes ou suivantes. Autrement dit, un protocole sans état ne nécessite pas que le serveur conserve, au cours de la session de communication, l'état de chacun des partenaires. A contrario, un protocole qui impose la conservation des informations sur l'état interne du serveur est appelé protocole avec état (en anglais stateful protocol). . 166

striping Le striping est une méthode d'accès au matériel, généralement des disques, qui vient aggréger différentes ressources, regroupées sous la forme d'une ressource virtuelle.. 56

thread Un thread ou fil (d'exécution) ou tâche (terme et définition normalisés par ISO/CEI 2382-7:2000 est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions

semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. En revanche, tous les threads possèdent leur propre pile d'exécution. . 104, 146

Top500 Le TOP500 est un projet de classification, par ordre décroissant, des 500 superordinateurs les plus puissants au monde. . 54

transient Dans une communication client serveur une position *transient* est ue posture temporaire dans laquelle le client et le serveur échange leurs rôle: le serveur devient ainsi temporairement client d'un service éphémère mis à disposition par le client. Les mécanismes de ce type sont en particulier utile pour éviter les désagréments pouvant surgir quand des opérations potentiellement longues doivent être à gérer, ce qui peut induire une forte charge sur le serveur.. 147

user land Espace utilisateur, dans lequel résident la plupart des processus d'un système d'exploitation, par opposition à l'espace noyau.. 44, 60

user space Synonyme de user land. 58, 159

xid La xid est un champ d'une requête conforme aux protocoles ONCRPCv2 et RPCSEC_GSS. Produite par l'initiateur de la requête, elle identifie celle-ci de manière unique. 46

Acronymes

AAA Authentication Authorization Accounting. 157

AFS Andrew File System. 143

API Application Programming Interface. 31, 37, 77

BXI Bull eXascale Interconnect. 73, 75, 77

CRUD Create Read Update Delete. 107, 111, 176, 179

CSC Tieteen tietotekniikan keskus. 87

CXL Compute eXpress Link. 88

DAOS Distributed Asynchronous Object Storage. 87, 111

dd data dump. 82

DMA Direct Memory Access. 77

DoE Department of Energy. 29, 32, 73

DRC Duplicate Request Cache. 46, 160, 182

DS Data Server. 55, 78, 150

ENSTA Ecole Nationale Supérieure des Techniques Avancées. 26

FDDI Fiber Distributed Data Interface. 29

FIFO First In First Out. 49, 50

flop floating operations per second. 21, 53

FSAL File System Abstraction Layer. 39, 52, 63, 86, 98, 105, 166, 184

FZJ Forschungszentrum Jülich. 97

GPFS Global Parallel File System. 31, 55, 65

GSSAPI Global Security Service Application Programming Interface. 157, 158

HDD Hard Disk Drives. 87, 97, 154

HDF5 Hierarchical Data Format version 5. 98

HiPPI High Performance Parallel Interface. 34

HPC High Performance Computing, ou calcul hautes performances en français. Il s'agit de la branche de l'informatique qui s'intéresse aux supercalculateurs et à leur environnement. 21, 29, 51, 55, 86, 109

HPSS High Performance Storage System. 29, 34, 51, 81, 109

HSM Hierarchical Storage Manager. 22, 29, 32, 35, 51, 57, 97, 105

ICMP Internet Control Message Protocol. 126

IEEE Institute of Electrical and Electronics Engineers. 32

IETF Internet Engineering Task Force. 125, 158

IO-SEA I/O Software Environment Architecture. 105

ior Interlaced Or Random. 83

KDE K Desktop Environment. 71

knfs Kernel NFS client. 159

knfsd Kernel NFS server. 159

krb5 Kerberos v5. 157, 184, 185

KVS Key Value Stores. 96, 98, 99, 107

LIOP Lightweight IO Provider. 79, 166

LNet Lustre Network. 56

LRU Least Recently Used. 48

LUKS Linux Unified Key Setup. 183

Lustre Lustre est un système de fichiers distribué libre, généralement utilisé pour de très grandes grappes de serveurs. Le nom est la réunion de Linux et cluster. L'objectif du projet est de fournir un système de fichiers distribué capable de fonctionner sur plusieurs centaines de nœuds, avec une capacité d'un pétaoctet, sans altérer la vitesse ni la sécurité de l'ensemble. Lustre est distribué sous licence GPL.. 55

MDS Meta Data Server. 55, 78, 150

MDT Meta Data Target. 56

MGS Management Server. 55

MGT Management Target. 57

MIT Massachusetts Institute of Technology. 157

MPI Message Passing Interface. 31, 83

MPMD Multiple Programs Muliple Data. 75

MPP Massively Paralell Processor, désigne une architecture de super-calculateurs très présente dans les années 80 et qui regroupe au sein d'une même machine une grande quantité de processeurs connectés par un réseau selon une topologie présente,par exemple selon un tore en 3D dans le cas du Cray T3D. 21, 29, 30, 109

MSSRM Mass Storage System Reference Model. 32, 33

NFS Network File System. 29, 31, 166

NLM NFS Lock Management. 31, 133

NNSA National Nuclear Security Administration. 32

NSA National Security Agency. 155

NSM Network State Management. 133

NVMe Non Volatile Memory express. 87, 106

NVRAM Non Volatile Random Access Memory. 73, 87, 97

OID Object ID. 160

ONCRPCv2 Open Network Computing Remote Procedure Call version 2. 49, 166

OPA OmniPath. 73

ORNL Oak Ridge National Laboratory. 87

OSS Object Storage Server. 55

OST Object Storage Target. 56

p9p Paralell 9p. 79

Parallel Heterogeneous OBject Store PHOBOS est un logiciel destiné à gérer de grands volumes de données sur différents types de stockage, depus les bandes jusqu'au SSD. PHOBOS peut efficacement prendre en compte de grands datasets sur des supports bon marché sans sacrifier la scalabilité, les performances ou les exigences de tolérance aux pannes. 176

PFS Paralell Filesystem. 32

PoC Proof of Concept. 78

POSIX Portable Operating System Interface. 98, 100, 154, 167

PVM Parallel Virtual Machine. 31

RAID Redundant Array of Independant Disks. 66, 183

RDMA Remote Direct Memory Access. 73, 75, 77, 136, 161

RFC Request For Comments. 110, 125

RIKEN Rikagaku Kenkyusho. 87

RPCL Remote Procedure Call Language. 165

RPM Redhat Package Manager. 50

SAL State Abstraction Layer. 70

SELinux Security Enhanced Linux. 167

SENFS Security Enhanced NFS. 155

SHerPA Système Hiérarchique et PArtagé. 34, 109

SMP Symmetric Multi-Processor. 21, 29, 30, 51, 53, 55, 109

SNMP Simple Network Management Protocol. 44

SQL Structured Query Language. 44

SSD Solid State Device. 73, 87, 97, 154

SSH Secure SHell. 43

TGCC Très Grand Centre de calcul du CEA. 58

UBO Université de Bretagne Orientale. 26

UDP User Datagram Protocol. 46

URI Uniform Resource Identifier. 150

UVSQ Université Versailles Saint-Quentin. 26

VFS Virtual File System. 40, 44, 63, 78, 185

XFS eXtended File System. 63

ZFS Zettabytes File System. 66

Table des matières

Glossaire **3**

Table des figures **15**

Liste des tableaux **17**

Introduction **21**

1 La période Terascale et la naissance de NFS-Ganesha **29**
- 1.1 Contexte de la période Terascale 29
- 1.2 Les systèmes de stockage de données durant la période Terascale 30
  - 1.2.1 Les systèmes de fichiers parallèles 30
  - 1.2.2 Les systèmes de stockage de données hiérarchiques 32
  - 1.2.3 Les réseaux hautes performances pour le transfert des données 33
- 1.3 Interaction entre les systèmes de stockage 34
- 1.4 Limitations dans l'implémentation de NFS 36
- 1.5 Le serveur NFS-Ganesha 37
  - 1.5.1 La File System Abstraction Layer et la gestion des back-ends 39
  - 1.5.2 FSAL disponibles et produits dérivés 43
  - 1.5.3 les couches de cache 45
  - 1.5.4 La gestion du réseau en mode multi-thread 48
  - 1.5.5 Les protocoles d'un serveur NFS 50
  - 1.5.6 Méthodologie de développement 50
- 1.6 Bilan et perspectives pour la période Terascale 51

2 La période Petascale et les IO Proxies **53**
- 2.1 Contexte de la période Petascale 53
- 2.2 Des machines de calcul organisées en îlots 53
- 2.3 Des systèmes de fichiers parallèles et distribués au centre de l'architecture 55
  - 2.3.1 Le logiciel Lustre 55
  - 2.3.2 L'architecture de Lustre 55
  - 2.3.3 Connexion entre Lustre et les HSMs 57
- 2.4 Architecture *datacentric* des centres de calcul 58
- 2.5 Un effort de R&D accentué 59
- 2.6 NFSv4 : une nouvelle version du protocole NFS aux multiples facettes 59
- 2.7 Naissance d'une communauté open-source NFS-Ganesha 60
- 2.8 Évolution de NFS-Ganesha durant la période petascale 61
  - 2.8.1 Prise en compte de la sécurité 61
  - 2.8.2 Remplacement des arbres bicolores par des arbres AVL 61

2.8.3 Les nouvelles FSALs . . . 63
2.8.4 Refonte en profondeur des caches et de l'API FSAL . . . 67
2.8.5 Support des états de fichiers et apparition de la couche SAL . . . 70
2.8.6 Utilisation de nouveaux outils logiciels . . . 71
2.9 Les IO-Proxies . . . 72
2.9.1 Contexte et problématique . . . 72
2.9.2 Le concept d'IO Proxy . . . 73
2.10 Implémenter des IO Proxies en modifiant NFS-Ganesha . . . 76
2.10.1 Le protocole 9p . . . 76
2.10.2 Utilisation de RDMA . . . 76
2.10.3 Gestion du BXI avec Pasha . . . 77
2.11 Injecter du parallélisme dans les IO Proxies . . . 78
2.12 Mise en exploitation de NFS-Ganesha pendant la période petascale . . . 80
2.12.1 Exportation d'un système de fichiers Lustre sur architecture NEC . . . 81
2.12.2 Administration de l'espace de nommage HPSS au travers de NFS . . . 81
2.13 Expérimentation des IO-Proxies en vraie grandeur . . . 82
2.13.1 Benchmarks réalisés . . . 82
2.13.2 Résultats . . . 83
2.13.3 Résultats et discussions . . . 85
2.14 Bilans et Perspectives pourla période Petascale . . . 86

3 La période Exascale et l'émergence du stockage objet **87**
3.1 Contexte de la période exascale . . . 87
3.2 Les problématiques de l'exascale . . . 88
3.3 Le stockage objet : une nouvelle manière de stocker les données . . . 91
3.4 Nouveaux paradigmes pour les systèmes exascale . . . 92
3.4.1 Services éphémères et Data Nodes . . . 92
3.4.2 Vers un placement intelligent des données . . . 94
3.5 Résolution des problématiques via les nouvelles approches adoptées . . . 95
3.6 De nouvelles perspectives pour le stockage exascale . . . 95
3.7 Les outils exploités . . . 96
3.8 De nouvelles collaborations pour répondre aux enjeux de l'exascale . . . 96
3.9 Les IO Proxies deviennent des serveurs éphémères . . . 97
3.10 La bibliothèque KVSNS . . . 98
3.10.1 les couches de KVSNS . . . 98
3.10.2 Implémentation du namespace KVSNS . . . 100
3.10.3 KVSNS comme une FSAL de NFS-Ganesha . . . 105
3.11 Mise en œuvre de NFS-Ganesha dans le projet IO-SEA . . . 105
3.11.1 L'architecture logicielle IO-SEA . . . 106
3.11.2 Le serveur NFS-Ganesha dans le projet IO-SEA . . . 107
3.12 Bilan et perspectives pour la période Exascale . . . 107

Conclusion générale **109**

Bibliographie **113**

Annexes **119**

A Curriculum Vitae et liste de publications **121**

B Les protocoles NFSv2, NFSv3 et leurs protocoles auxiliaires **125**
B.1 RPC et XDR . . . 125
B.2 Le protocole NFS Version 2 . . . 126
B.2.1 Le Mount Protocol . . . 128
B.2.2 Limitations structurelles de NFSv2 . . . 129
B.2.3 Le protocole NFS Version 3 . . . 130
B.2.4 Le Mount Protocol v3 . . . 133
B.2.5 Autres protocoles auxiliaires . . . 133

C Les multiples RFC de NFSv4 et les versions mineures **135**

D Le protocole NFSv4 **139**
D.1 Les requêtes composées . . . 139
D.2 Les opérations de validation . . . 142
D.3 Les traversées de jonctions . . . 142
D.4 La gestion des exports et la disparition du Mount Protocol . . . 143
D.5 Prise en compte de RPCSEC_GSS dans NFSv4 . . . 144
D.6 La gestion des états . . . 144
D.6.1 Le protocole NFSv4.1 . . . 147
D.6.2 Les nouvelles opérations de NFSv4.1 . . . 148
D.6.3 La fonctionnalité pNFS du protocole NFSv4.1 . . . 150
D.6.4 Le protocole NFSv4.2 . . . 152

E Le protocole RPCSEC_GSS et la sécurité réseau **157**
E.1 kerberos 5 . . . 157
E.2 La GSSAPI . . . 158
E.3 RPCSEC_GSSv2 . . . 158
E.4 Prise en compte de RPCSEC_GSS dans NFSv3 . . . 158
E.5 Montages NFS kerbérisés . . . 159
E.6 Interaction avec le client knfs via RPCSEC_GSS . . . 159

F Le protocole 9p2000.L **161**

G Spécifications du protocole LIOP **165**
G.1 Notations utilisées pour décrire les appels du protocole . . . 165
G.2 Les spécifications du protocole LIOP . . . 166
G.2.1 Lustre_fid . . . 166
G.2.2 ucred . . . 166
G.2.3 version . . . 167

G.2.4 Codes d'erreurs 167
G.3 Les fonctions du protocole LIOP 167
G.3.1 ERROR 168
G.3.2 STATUS 169
G.3.3 READ 169
G.3.4 WRITE 170
G.3.5 RELEASE 171
G.3.6 GETFID 171

H Un exemple de fichier de configuration de KVSNS 173

I Les mécanismes du KVSNS pour gérer la sémantique CRUD 175
I.1 Accès séquentiel et sémantique CRUD 175
I.2 La machine à états du crud_cache 176
I.3 Implémentation de la machine à états 178
I.4 Déplacement des données : la couche OBJSTORE 179
I.5 Le GRH 180

J Gestion de la sécurité dans les IO Proxies 183
J.1 Les conteneurs chiffrés 183
J.2 9p/krb5 184
J.3 Gestion de la sécurité dans 9p 185

K Distributions logicielles 187

Table des figures

1	Frise chronologique des évolutions de NFS-Ganesha	23
2	Frise chronologique des réalisations principales de ma carrière professionnelle	25
1.1	Architecture du centre de calcul TERA durant la période Terascale	32
1.2	Schéma de principe d'un HSM	33
1.3	Schéma de principe du Cache SHerPA	35
1.4	Architecture du Cache SHerPA	35
1.5	Architecture de NFS-Ganesha, version Terascale et initale	38
1.6	Traitement des requêtes entre le dispatcher et les workers	49
2.1	Exemples de topologies réseaux	54
2.2	Architecture de Lustre (source : Lustre Wiki)	56
2.3	Mouvement des données Lustre/HSM	57
2.4	Architecture du centre de calcul TGCC durant la période pétascale	58
2.5	Positionnement de l'oncle dans le cas noir, ici à gauche	62
2.6	Schéma générique d'un serveur proxy / serveur entremetteur	66
2.7	Les IO-Proxies entre les calculateurs et les systèmes de stockage	74
2.8	Fonctionnement d'un IO-Proxy avec des LIOP auxiliaires	79
2.9	Résultats IOR sur Lustre, débits en lecture et écriture (MB/s) en fonction du nombre de tâches	84
2.10	Résultats IOR sur p9p/LIOP, débits en lecture et écriture (MB/s) en fonction du nombre de tâches	85
3.1	Empilement des ressources de stockage HPC	89
3.2	Architecture en couches de la bibliothèque KVSNS	99
3.3	L'architecture logicielle IO-SEA	106
B.1	Pile de protocoles NFS de la période Terascale	125
D.1	Schéma en tierce partie mis en jeu par NFSv4.1/pNFS	150
E.1	Négociation GSS via le rpc.gssd	160
I.1	Transitions d'états dans le fonctionnement du crud_cache	177
I.2	Fonctionnement des requêtes dans l'outil GRH	180

Liste des tableaux

1.1 Performances des machines du CEA durant la période Terascale 30
1.2 Principaux types génériques de la FSAL 39
1.3 Principaux appels génériques de la FSAL, avec adressage direct 42
1.4 Principaux appels génériques de la FSAL, avec adressage par parent et nom 43

2.1 Performances des machines du CEA durant la période Petascale 53

3.1 Performances des 3 premières machines du Top500 au début de la période pré-Exascale 87
3.2 Performances des machines du CEA durant la période pré-Exascale 88
3.3 Les fonctions de l'API extstore 100
3.4 Les fonctions de l'API KVSAL 101
3.5 Les structures de l'API KVSNS 102
3.6 Les structures de l'API KVSNS 108

B.1 Appels du protocole NFS Version 2 127
B.2 Appels du protocole Mount Protocol v1 129
B.3 Appels du protocole NFS Version 3 131
B.4 Appels du protocole Mount Protocol v3 133

D.1 Appels du protocole NFS Version 4 139
D.2 Opérations du protocole NFS Version 4.0 140
D.3 Opérations spécifiques du protocole NFS Version 4.1 148

F.1 Messages du protocole 9p2000.L 162

G.1 Codes d'erreur du protocole LIOP 168

I.1 Les Appels de l'API OBJSTORE 179

Remerciements

Cette thèse en VAE est une démarche toute spéciale pour moi, qui m'a permis de rassembler au même endroit les fruits d'un travail qui s'est étalé sur plus de vingt ans et plusieurs générations successives de centres de calcul.

Ce travail n'aurait pas pu être possible sans l'aide et le soutien bienveillants d'un grand nombre de personnes. S'il me semble difficile de toutes les mentionner (cela prendrait plusieurs pages !), je tiens à remercier certaines d'entre elles plus spécifiquement. Mes excuses à celles qui ne se retrouveront pas dans cette liste : soyez assurés que je ne vous oublie pas et que ma sympathie tout autant que ma gratitude vous sont acquises à jamais.

En premier lieu, je tiens à remercier ma directrice de thèse, le Professeur Soraya Zertal. J'avais eu l'occasion d'encadrer des thèses avec elle, me retrouver de "l'autre côté" et être à mon tour l'un de ses doctorants aura été une expérience unique autant qu'enrichissante. Je tiens en particulier à souligner son implication et son engagement pour gérer un étudiant un petit peu plus âgé que ses autres doctorants.

Un grand merci également à mes hiérarchiques successifs au CEA, au premier rang desquels Hervé Lozach et Jacques-Charles Lafoucrière, qui m'ont embauché au CEA en 1998 et ensuite encadré pendant de longues années, mais aussi Gilles Wiber qui m'a permis de réaliser cette thèse selon la procédure VAE sur une partie de mes heures de travail.

Ce travail n'aurait pas été possible sans les gens avec lesquels j'ai travaillé pendant toutes ses années, et dont j'ai parfois été le manager, en particulier Thomas Leibovici, Dominique Martinet, Henri Doreau, Sébastien Gougeaud, Patrice Lucas et Yoann Valéri. Merci également à Philippe Couvée de BULL S.A avec lequel j'ai souvent été amené à collaborer pour mon plus grand plaisir professionnel.

Enfin, un grand merci à tous les experts en informatique du département DSSI du CEA, qui ont su créer un environnement scientifique et technique dans lequel je me suis pleinement épanoui : Jean-Christophe Weill, Jean-Philippe Nominé et Guillaume Colin de Verdière.

Introduction et contexte

Introduction Générale

Le calcul haute performance, que nous abrégerons en «HPC» dans le reste du présent document, trouve ses racines dans la fin des années 80 et le tout début des années 90. Il a connu différentes évolutions majeures : si la puissance de calcul des premiers supercalculateurs était globalement comparable à ceux d'un *smartphone* actuel [1] avec quelques millions d'opérations flottantes à la seconde, ou floating operations per second (flop))s, les futures machines exaflopiques atteindront la puissance de 10^{18} flop, soit un facteur de multiplication de 10^{12}, c'est-à-dire mille milliards.

Ces évolutions comportent plusieurs grandes étapes que l'on peut résumer de la manière suivante :

- initialement, on a assisté à une très large domination de Cray sur le HPC avec ses machines MPP, durant la fin des années 1980 et le début des années 1990 ;
- la seconde partie des années 1990, et l'aube des années 2000 voient le début de l'ère terascale avec des clusters de Symmetric Multi-Processor (SMP)s. Ceux-ci regroupent autour d'un réseau rapide des machines aux architectures proches des stations de travail monoprocesseurs et monocore. En 2001, le CEA acquiert ainsi un calculateur du constructeur Dec-Alpha qui parvient à dépasser le téraflops (10^{12} flops) ;
- les machines SMP poursuivent leur croissance et comptent de plus en plus de nœuds, au point de regrouper plus de machines que les parcs de stations de travail bureautiques, permettant d'obtenir une puissance de calcul qui dépasse le petaflops, c'est l'ère petascale qui s'amorce (10^{15} flops) au tournant de 2005 ;
- les nœuds dans les machines SMPs de classe petaflopique se multiplient encore, ce qui va de pair avec l'utilisation de topologies réseau de plus en plus élaborées et complexes, en particulier avec des configurations sous forme « d'ilots » ; ces architectures atteignent les puissances de plusieurs dizaines de petaflops au tournant de 2010 et la centaine de petaflops aux alentours de l'année 2015 ;
- par la suite, les multiples évolutions des processeurs et des technologies de réseau rapide permettent d'augmenter les performances et d'atteindre l'exascale (10^{18} flop) ; Ces architectures sont actuellement au cœur des efforts de R&D des centres de recherche et des industriels.

1. Á tire indicatif, le machine Cray 2 déployait une puissance de calcul de 1,9 Gflops (soit 1.9 millions d'opérations flottantes par seconde), du même ordre de grandeur que les 1,6 Glops mesurables sur un iPhone4.

Les évolutions des moyens de calcul sont le fruit de différents facteurs :

- les innovations dans le domaine des processeurs dont la fréquence augmente et qui comptent de plus en plus de cœurs de calcul,
- les évolutions matérielles qui fournissent de meilleurs débits entre la mémoire et les processeurs,
- l'apparition de nouvelles technologies réseau haute performance.

Si l'on s'intéresse maintenant aux systèmes de stockage de données, dont la finalité est d'absorber, de rendre disponible et de conserver les résultats produits par les supercalculateurs, on peut distinguer plusieurs étapes également :

- **1990-2000** : durant cette période "pré-terascale", que l'on peut qualifier de *gigascale*, les machines Cray sont, d'une manière simpliste, des stations de travail massives et multi-utilisateurs, avec un stockage sur des systèmes de fichiers locaux, corrélés avec un gestionnaire de stockage hiérarchique ou Hierarchical Storage Manager (HSM) ;
- **2000-2010** : durant la période terascale, le stockage devient un système autonome, indépendant du calculateur. Il implémente un HSM qui centralise les données face à des systèmes de fichiers locaux utilisés comme des caches de données ;
- **2010-2018** : l'ère petascale voit l'émergence d'architectures datacentriques conçues autour de très grands systèmes de fichiers distribués (tels que Lustre ou GPFS), indépendants des supercalculateurs, couplés avec des HSM.
- **2018-** : le facteur d'échelle de l'exascale rend caduque les solutions précédentes. De nouveaux paradigmes doivent être mis en place pour répondre à ce nouveau besoin.

Dans ce contexte, des travaux, d'abord propres au développement logiciel, ont été réalisés. Ils se sont vite axés autour de la R&D avec la généralisation des approches dites *codesign* qui voient les centres de calcul conçus conjointement par les industriels et les centres de recherche qui les exploitent.

J'ai été recruté par le CEA en 1998, à l'heure où les centres de calcul se préparaient pour accueillir les nouvelles architectures terascale et abandonner les solutions monolithiques comme les calculateurs Cray. Le besoin de nouvelles solutions était fort et c'est dans ce cadre que j'ai initié le développement du logiciel NFS-Ganesha, un serveur NFS générique fonctionnant intégralement en espace utilisateur. Ce serveur avait été conçu pour être générique et facile à adapter à de nouvelles technologies, il a naturellement évolué au fil des années pour coller au mieux à l'évolution des besoins durant les périodes successives. Un résumé de ces évolutions et des différentes étapes qui leur correspondent est présenté dans La figure 1. Ces éléments seront par la suite présentés plus en détail dans les pages de ce manuscrit.

ce travail de recherche a été conduit dans le domaine du stockage. Il est

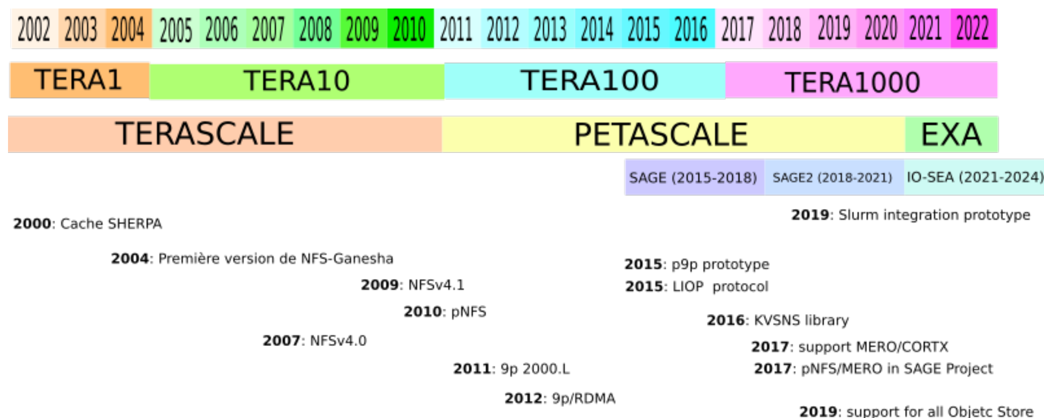


Figure 1 – Frise chronologique des évolutions de NFS-Ganesha

devenu indispensable au fur et à mesure des générations technologiques et face à la course à la puissance de calcul. Ce document présentera les différentes pistes que j'ai explorées, les travaux de recherche et les innovations que j'ai réalisés, souvent très en avance de phase sur les supercalculateurs.

Contexte de cette procédure de thèse en VAE

Cette petite section est assez inhabituelle dans un manuscrit de thèse, mais elle prend tout son sens dans le cadre d'une procédure de thèse par Validation des Acquis par l'Expérience (VAE). Par souci de simplicité, l'utilisation de la première personne du singulier (donc du "je") sera privilégiée.

Le stockage de données massif n'est pas la partie la plus visible du HPC. Depuis de nombreuses années, les communications scientifiques mettent bien plus souvent en avant les performances brutes en termes de puissance de calcul, mesurées grâce au célèbre benchmark LINPACK[35] qui fait littéralement figure de mètre-étalon dans ce domaine et qui donne lieu au Top500[93] qui définit à chaque édition de la conférence SuperComputing le classement des machines supposées les plus puissantes du monde.

Cependant, à quoi bon calculer des résultats très précis et coûteux autant en temps calcul qu'en temps humain s'il s'agit de les laisser en mémoire vive, les condamnant à disparaître quand la machine sera éteinte ? LINPACK n'est pas un véritable code de calcul, il se contente d'inverser une très grande matrice le plus vite possible, mais il ne conserve pas le résultat.

Les vrais codes de simulation numérique, ceux qui composent l'exploitation régulière d'un supercalculateur, tentent de reproduire des phénomènes physiques complexes et ainsi d'éviter de devoir produire des expériences réelles complexes et couteuses. La simulation des essais nucléaires conduite par le CEA/DAM est à ce titre un parfait exemple. Ces codes produisent de véritables

résultats qui constituent la « substantifique moelle » de la simulation et qu'il faut conserver, et qui pourra par la suite être exploitées par des physiciens spécialistes du domaine. Les simulations numériques font à ce titre figure d'expérimentations numériques dont la pertinence rivalise avec des expérimentations scientifiques réelles.

Pour héberger les données produites et les préserver afin de les exploiter, il faut construire de grandes installations dédiées, souvent désignées sous le nom de data centers, et des réseaux hautes performances pour les déplacer entre les différents "lieux" qui composent un centre de calcul. Cette activité suppose de développer des systèmes et des technologies dédiées, ce qui inclut de l'ingénierie de pointe, mais aussi une recherche en amont poussée.

J'ai longtemps exercé des fonctions d'ingénieur, et j'ai contribué, à mon échelle, à faire émerger certaines nouvelles idées. À cette occasion, j'ai été amené à travailler avec les scientifiques du domaine, et à bâtir des collaborations souvent étroites avec eux, autour de projets communs ou d'encadrements de doctorants et de stagiaires. Cette activité de recherche n'a cessé de s'intensifier, surtout à l'heure où l'avènement futur de "l'exascale" fait apparaître de nouveaux défis complexes.

Les années à venir, plus que toutes les autres périodes précédentes, vont être riches en innovations et vont probablement voir l'émergence de concepts et d'idées qui forgeront les futures technologies en exploitation. Pour suivre ce mouvement, j'ai choisi de m'impliquer davantage dans les activités de R&D, avec un focus sur les activités de recherche, mais afin de mener cette initiative, il me faut adopter les mêmes méthodes que les chercheurs avec lesquels j'ai déjà travaillé. Le chemin le plus sûr pour y arriver passe par la validation de mes travaux et de mon expérience dans le cadre d'une procédure dont ce manuscrit est la pierre angulaire.

Mon parcours professionnel

Avant de rentrer dans les aspects techniques avec les chapitres suivants, et dans le cadre de la procédure VAE, je vais vous présenter ma carrière et mon parcours professionnel depuis mon entrée au CEA-DAM suite à l'obtention de mon diplôme d'ingénieur de l'École Centrale paris en 1996, jusqu'à présent. Mon CV détaillé figure en annexe A.

Comme on peut le voir sur la frise chronologique de la figure 2, ma vie professionnelle s'est organisée selon les étapes suivantes :

- **1996-1997 :** Service National. J'effectue celui-ci en qualité de Scientifique du Contingent au CEA/DAM au sein de l'unité en charge des centres de calcul du CEA. Suite à ce "stage" d'une année, je suis engagé par mon unité d'accueil.
- **1997-2008 :** ingénieur système et de développeur système au CEA/DAM. On me confiera en particulier la mise en place des nouveaux systèmes

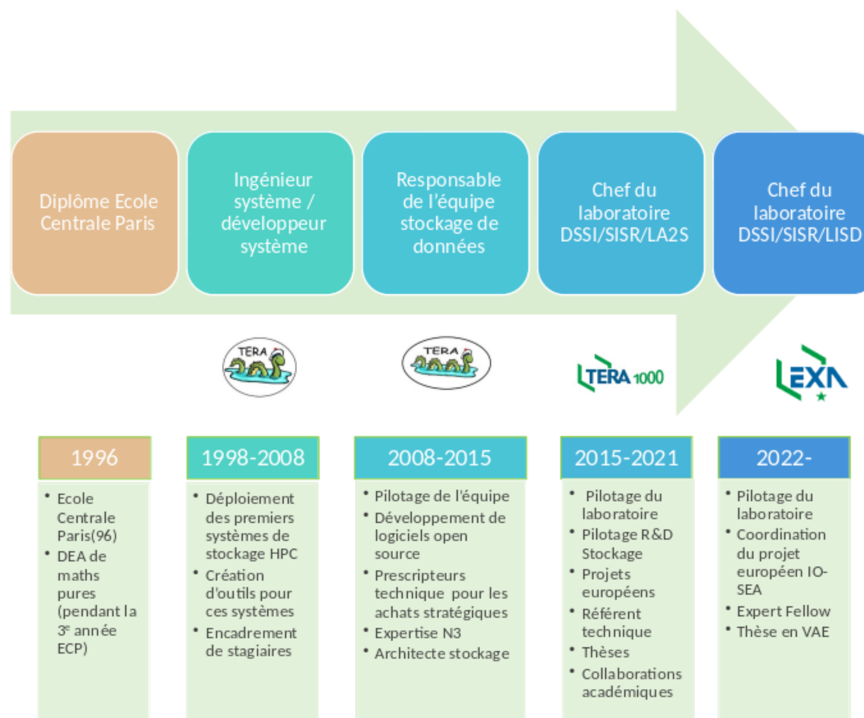


Figure 2 – Frise chronologique des réalisations principales de ma carrière professionnelle

de stockage. J'ai été amené à créer différents outils pour réaliser cette mission et je ai été amené à encadrer des stagiaires de fin de M2 et de cycle d'école d'ingénieurs.

- **2008-2015 :** Responsable des systèmes de stockage. L'équipe en charge des systèmes de stockage s'est agrandie autour de moi et je pilote cette petite structure pendant ces années. Je suis amené durant ces années à initier des développements opensource, dont Casimir, un analyseur de journaux systèmes, qui sera le premier logiciel publié en opensource par le CEA/DAM. Ces années voient aussi la naissance du produit NFS-Ganesha qui fait l'objet du présent document. Mes responsabilités opérationnelles me conduisent à réaliser des cahiers des charges pour piloter des acquisitions de grands systèmes de stockage pour les centres de calcul. En parallèle, je suis amené à réaliser différentes communications dans des conférences où je présente les activités du CEA/DAM autour des systèmes de stockage du CEA/DAM, et j'obtiens le grade d'Expert Senior (Expert N3 dans la nomenclature du CEA).
- **Octobre 2015-2020 :** Chef de laboratoire. Je prends en 2015 la tête du

laboratoire auquel j'appartiens et je deviens responsable du pilotage d'un groupe d'une quinzaine de personnes. Je conserve le pilotage des efforts de R&D conduits avec ATOS dans le cadre des accords de Recherche et Développements pour le HPC établis entre le CEA et ATOS. L'arrivée des projets financés par l'Europe dans le cadre de l'initiative H2020 [2] me conduit à travailler dans plusieurs projets financés dans ce cadre, dont SAGE et SAGE2. Mes activités me conduisent à encadrer des thèses et à établir des collaborations avec des universités en France, dont l'Université Versailles Saint-Quentin (UVSQ), devenue depuis l'Université Paris-Saclay, l'Université de Bretagne Orientale (UBO) et l'Ecole Nationale Supérieure des Techniques Avancées (ENSTA).
- **2020-2022 :** Ces dernières années me voient poursuivre le pilotage de mon laboratoire, mais voient aussi le lancement du projet européen IO-SEA financé par EuroHPC JU, dans la continuité de H2020, dont je suis l'initiateur et le coordinateur scientifique. Dans le même temps, j'accède au grade d'Expert Fellow (anciennement nommé Expert N4 dans le cadre de la nomenclature du CEA).

Afin de permettre le rapprochement avec les différentes générations de centres de calcul dont j'ai assisté au déploiement durant ces années, j'ai ajouté sur le schéma les logos qui leur ont été attribués. Dans le cadre de ce document, on retrouvera ces étapes : TERA1, puis TERA10, TERA100, TERA1000 et enfin EXA1 qui est actuellement en production.

Des détails sur mes publications, issues pour certaines de ces collaborations, peuvent être trouvés dans l'annexe A.

Structure du manuscrit

Le présent document reprend les étapes des évolutions du HPC présentées dans cette introduction. Il comprend donc trois chapitres, présentant les trois grandes périodes du HPC.

Le chapitre 1 est consacré à la période terascale et présentera le contexte dans lequel le logiciel NFS-Ganesha a été créé ainsi que les éléments d'architecture logicielle correspondants.

Le chapitre 2 s'intéressera à la période petascale durant laquelle NFS-Ganesha est devenu un produit opensource doté d'une communauté très active. Cette dernière permet au produit de voir une partie de son architecture modifiée. Par ailleurs, l'émergence de nouveaux protocoles comme NFSv4 puis NFSv4.1/pNFS, que j'ai rapidement implémenté dans NFS-Ganesha conduisent à d'autres évolutions de NFS-Ganesha qui le rendent encore plus générique et adaptable.

Le chapitre 3 s'intéresse à la période exascale, donc au passé récent et

2. Voir https ://www.horizon2020.gouv.fr/

au présent. Il décrit comment NFS-Ganesha est exploité pour fournir des solutions afin de gérer le stockage objet et l'implication du produit dans différents projets européens (dont le projet IO-SEA, dont je suis le coordinateur scientifique) et son rôle dans ces nouveaux écosystèmes logiciels.

Le manuscrit se terminera avec quelques réflexions et conclusions, en brossant un bilan du travail effectué et en énumérant les différentes perspectives.

Chapitre 1 : La période Terascale et la naissance de NFS-Ganesha

1.1 .Contexte de la période Terascale

Ce chapitre couvre la période entre la fin des années 90 et le début des années 2000. Le HPC a pendant des années subi l'hégémonie de Cray avec des machines à processeurs vectoriels, comme le Cray T90[24], et des machines à architecture MPP, notamment le Cray T3D et son successeur le Cray T3E[25]. Ce modèle de supercalculateur, qui regroupe au sein d'une seule machine tous les composants d'un centre de calcul actuel ou presque, commence à atteindre ses limites et les premières architectures à base de clusters de machines SMP font leur apparition.

Le stockage des données, auparavant réalisés sur des disques propres aux machines Cray, s'externalise et l'on commence à voir apparaître des systèmes qui sont explicitement des gestionnaires de stockage. Les systèmes de type HSM, qui fédèrent disques et bandes pour produire un stockage massif économiquement viable, font leur apparition, le prix de l'octet sur bandes étant très inférieur à celui de l'octet sur disques. Ces outils en sont à leurs débuts, et si des produits, comme Cray DMF [1] ou UniTREE, existent déjà, ils sont encore très rattachés à certaines architectures et peu génériques.

C'est l'apparition du logiciel High Performance Storage System (HPSS), développé par IBM Government Systems avec l'assistance du Department of Energy (DoE) Américain qui introduit un changement technique radical. Ce logiciel deviendra rapidement un standard de fait, en devançant ses concurrents. À ce jour, en 2023, HPSS occupe toujours une position de leader.

Les clusters SMP qui composent les toutes nouvelles générations de supercalculateurs, s'ils ont des performances bien meilleures que les monolithiques Cray, sont constitués d'éléments qui proviennent du monde des stations de travail : leurs « nœuds » [2] ressemblent à des stations de travail, et les réseaux qui les fédèrent se rattachent aux technologies de bureautique d'entreprise de haut de gamme, comme Fiber Distributed Data Interface (FDDI).

Au niveau des protocoles qui permettent de gérer les accès distants aux données, seul Network File System (NFS) répond à l'appel. Sa mouture la plus récente, c'est-à-dire NFSv3, n'est pas encore exploitée partout et c'est souvent NFSv2 qui lui est préféré, avec quantité de limitations structurelles qui s'avéreront bloquantes à court terme.

1. Cray DMF : Cray Data Migration Facility
2. les nœuds sont les différents éléments qui constituent des grappes ou *clusters* dans un calculateur SMP

Quelques chiffres de performances représentatifs de la période Terascale

Ce paragraphe présente quelques chiffres relatifs aux capacités et aux performances des machines de la période Terascale exploitées dans les centres de calcul du CEA. Ils permettront au lecteur de recontextualiser l'ensemble de ce chapitre.

Année	Machine	Performances	Stockage
années 90	Cray 7924 et Cray T3E	60 Gflops	80 To de bandes
2000	Tera 1	5 Tflops	50 To (disques), 1 Po (bandes)
2005	Tera 10	60 Tflops	2 Po (100 Go/s), 10 Po (bandes)

Table 1.1 – Performances des machines du CEA durant la période Terascale

1.2 . Les systèmes de stockage de données durant la période Terascale

La période Terascale voit l'apparition de plusieurs concepts qui seront intensivement exploités durant les générations postérieures de centres de calcul. Ici, ces technologies connaissent leurs premières moutures, les développeurs, les utilisateurs et les administrateurs de ces systèmes les découvrent et doivent se les approprier.

De nombreuses idées voient ainsi le jour, les chapitres suivants les exposeront et détailleront leurs évolutions et leurs conséquences technologiques.

Les technologies en question sont :

- les systèmes de fichiers parallèles ;
- les systèmes de stockage de données hiérarchiques ;
- les réseaux hautes performances pour le transfert des données.

Nous allons nous attarder sur chacune d'entre elles dans les sous-sections suivantes.

1.2.1 . Les systèmes de fichiers parallèles

Les nouveaux clusters de SMP, qui constituent la nouvelle génération de super-calculateurs et qui supplantent les machines MPP, ne sont fondamentalement que des stations de travail fédérées par un réseau dédié et optimisé, nommé *réseau d'interconnexion*. Dans ce nouveau schéma, les codes de calcul ne sont plus constitués de différents processus fonctionnant au sein

d'une même machine et échangeant des données via des segments de mémoire partagées, ce sont des processus répartis sur différentes machines et qui communiquent via des messages dédiés, réalisés sur ce réseau d'interconnexion. L'API et le standard Message Passing Interface (MPI) deviennent incontournables et remplacent l'ancien formalisme du produit Parallel Virtual Machine (PVM) qui manque de fonctionnalités telles que des communications collectives efficaces.

Si l'on considère ces machines du point de vue du stockage de données, le besoin de regrouper les données produites par différents processus éclatés dans toute la topologie de la machine se fait naturellement ressentir. Pour répondre à ce besoin, le concept de *système de fichiers parallèle* voit le jour.

Le principe est fondamentalement voisin de ce qu'offrent les systèmes de fichiers distribués comme NFS, mais la nature même des super-calculateurs rend la problématique beaucoup plus complexe :

- les exigences de performances, tant au niveau de l'accès aux données qu'à celui de la gestion des métadonnées, sont sans communes mesures avec ce que NFS peut produire ;
- la concurrence entre les accès doit être prise en compte : plusieurs processus de différentes machines doivent être capables d'accéder différentes portions d'un même fichier ouvert plusieurs fois, tout en conservant la cohérence et la pertinence des informations produites. C'est là une problématique bien plus complexe que ce que les protocoles de l'époque, tels que NFS Lock Management (NLM) utilisé par NFS, savent faire. Ces dernier se contentant de reproduire de manière distribuée le comportement de verrous de fichiers locaux tels que manipulés par les Application Programming Interface (API) en langage C *fcntl (FLCK, ...)*
- les volumes de données produits, et qui doivent donc être conservées, sont à l'échelle de ces puissantes machines dont la puissance, qui dépasse le teraflops, est considérable au regard des générations précédentes limitées à quelques centaines de gigaflops. Il s'agit de gérer des volumes qui atteignent des centaines de teraoctets, voire qui sont voisines du petaoctet. C'est là un défi de taille quand les volumes des disques se comptent en dizaines de Gigaoctets.

Pour répondre à la problématique de volume, on fera appel à des gestionnaires de stockage de données hiérarchiques qui fournissent une volumétrie hybride sur disques et sur bandes. Ce concept est décrit dans la section 1.2.2.

Des solutions dédiées sont produites pour répondre aux deux premiers points, à savoir les performances et la concurrence. C'est ainsi que le produit Global Parallel File System (GPFS) voit le jour dans les laboratoires d'IBM. Au niveau du centre de calcul du CEA, le super-calculateur est fourni par le constructeur Compaq qui vient juste d'acquérir la société DEC, qui porte la technologie processeur Dec Alpha. Leur système de fichiers parallèle, nommé

Paralell Filesystem (PFS), est une déclinaison du protocole NFS, modifiée pour se comporter de manière plus optimale dans l'environnement "cluster" du supercalculateur.

1.2.2 . Les systèmes de stockage de données hiérarchiques

Les nouvelles machines produisent des volumes de données importants qu'il faut conserver. Pour les moyens de l'époque, une production de un tera-octet en 24 heures représente un véritable défi. Le volume total du système, qui doit atteindre le Petaoctet ou la dizaine de petaoctets, est également un défi.

Les HSMs, permettent de répondre à la problématique de volume. Dotés d'une architecture en grappe, réunissant plusieurs serveurs de données, donc plusieurs interfaces réseaux, ils deviennent également capables de faire face à la problématique de débit.

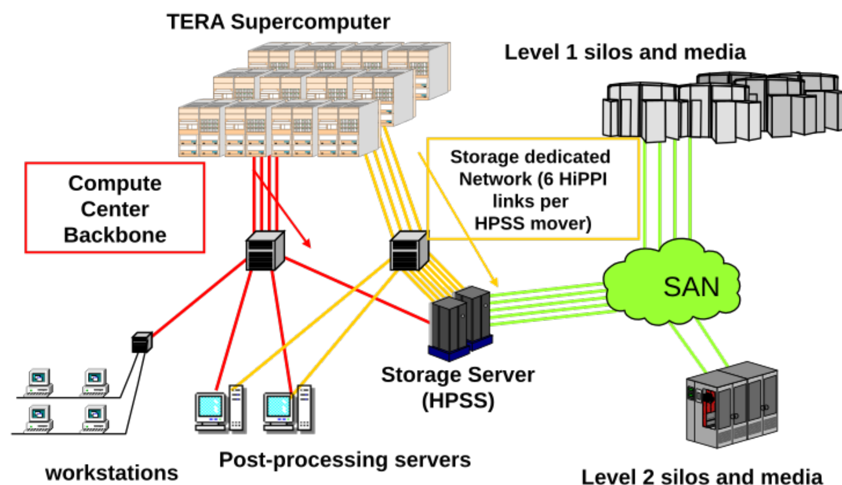


Figure 1.1 – Architecture du centre de calcul TERA durant la période Terascale

Il n'est pas possible de parler d'HSM sans parler du logiciel HPSS tant celui-ci est devenu rapidement un standard de fait, une position qu'il parvient toujours à tenir plus de 25 ans plus tard. HPSS répond à un besoin exprimé par les grands centres de calcul du DoE Américain, avec en première ligne les grands sites du National Nuclear Security Administration (NNSA) , équivalents du CEA/DAM outre Atlantique. Des besoins sont ainsi exprimés et un standard est produit par l'Institute of Electrical and Electronics Engineers (IEEE), le Mass Storage System Reference Model (MSSRM)[68]. Il est pressenti que la réalisation d'un tel logiciel va requérir des méthodes et des moyens propres à l'industrie. IBM Government System sera choisi pour piloter celle-ci et HPSS sera ainsi développé conjointement par le DoE et IBM, selon un schéma qui reste toutefois fondamentalement celui d'un produit commercial.

HPSS est capable de présenter un espace de nommage POSIX, avec toutes les fonctionnalités que cela suppose (répertoires, fichiers, droits d'accès, verrous, liens symboliques ou non. . .,) mais il est capable de conserver les fichiers sur différents types de médias, en particulier des disques et des bandes magnétiques.

Grâce à la capacité élevée des bandes et la rentabilité économique du prix par octet de ce support physique, des systèmes de stockage de plusieurs dizaines de Petaoctets sont désormais accessibles. HPSS gère ses ressources par un mécanisme de migration sophistiqué dont les grandes étapes sont définies dans le MSSRM.

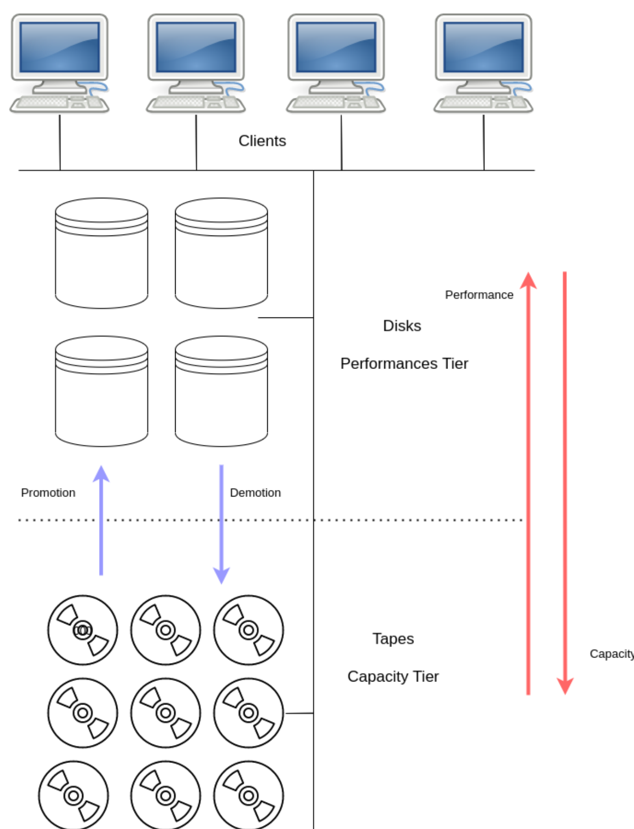


Figure 1.2 – Schéma de principe d'un HSM

1.2.3 . Les réseaux hautes performances pour le transfert des données

Les calculateurs ne sont pas capables de préserver sur leurs propres disques les données qu'ils produisent, malgré l'avènement des systèmes de fichiers parallèles. Comme nous l'avons vu, les architectures terascale introduisent des systèmes de stockage massifs et distribués pour assurer la conservation des données de simulation produites par ces grands systèmes. Déplacer des

données impose de disposer des canaux à très hauts débits convenablement dimensionnés pour réaliser cette action. La machine Tera1 du CEA/DAM s'appuiera sur la technologie High Performance Parallel Interface (HiPPI) pour fournir le débit et la latence nécessaire. La nature distribuée de l'ensemble des acteurs, qu'il s'agisse du stockage ou des calculateurs, rend possible l'utilisation de liens multiples en même temps, un parallélisme réseau qui permet d'atteindre ainsi les performances requises. Les acteurs logiciels qui exploitent ces réseaux se chargeront de fragmenter et de réassembler les informations transmises.

1.3 .Interaction entre les systèmes de stockage

Les sections précédentes ont dressé les grandes lignes des composants des architectures des centres de calcul de la période terascale. Un centre de calcul complet, incluant un calculateur, son système de stockage associé et le réseau dédié qui les relie, remplace donc les machines monolithiques de jadis. Il convient dans cette section de regarder de plus près les mécanismes mis en œuvre et les composants logiciels qu'il a fallu développer, notamment le cache SHerPA dont l'architecture est illustrée par la figure 1.3.

Dans le contexte des architectures de l'époque, ce genre d'outil est indispensable. Les données de référence sont conservées dans le logiciel HPSS, mais le calculateur est incapable de les accéder efficacement et avec des débits suffisants. Il devient impératif de recopier ces données sur un système de fichiers global au calculateur. Le cache SHerPA permet de voir celui-ci comme un cache des données de référence conservées dans HPSS.

Le fonctionnement du Système Hiérarchique et PArtagé (SHerPA) décrit dans cette section suit une stratégie simple : on considère que les systèmes de stockage des données, qui exécutent le logiciel HPSS, sont la référence des données préservées dans le centre de calcul. Les données manipulées par le supercalculateur sur ses systèmes de fichiers parallèles seront considérées comme des copies de travail, dupliquées dans un cache.

Ce cache n'est pas transparent, il est implémenté via une bibliothèque de fonctions qui est appelée explicitement par les codes de simulation qui tournent sur la machine Tera1 du CEA/DAM. Dans SHerPA, on considère deux espaces de nommages :

- l'espace de nommage des données stockées dans l'HSM, exposé à l'utilisateur via NFS, nommé **espace de référence** ;
- le système de fichiers parallèle du supercalculateur, nommé **espace cache**.

Les fichiers et les répertoires qui les contiennent sont toujours créés dans l'espace de référence, même quand il s'agit d'un nouveau fichier, donc vide. Le code demandera explicitement la mise en cache d'une sous-partie de l'arbo-

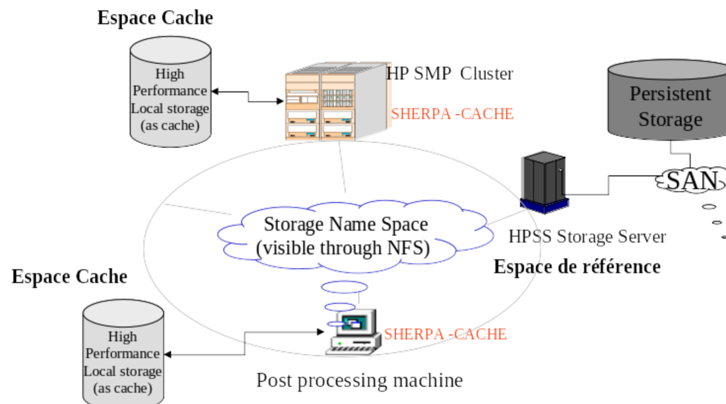


Figure 1.3 – Schéma de principe du Cache SHerPA

rescence. Celle-ci sera alors répliquée, produisant des chemins d'accès POSIX aux fichiers identiques au-delà du point de montage des espaces de référence et de cache.

Par exemple, un utilisateur créera un fichier vide */mnt_ref/dir1/dir2/file* et demandera sa mise en cache, ce qui produira automatiquement le chemin */mnt_cache/dir1/dir2/file*, incluant tous les répertoires intermédiaires avec les mêmes droits UNIX, dans l'espace de cache. L'utilisateur écrit toujours dans le cache (qui est un système de fichiers parallèle propre au calculateur et optimisé pour cela) jamais dans la référence. Il va donc produire des données dans */mnt_cache/dir1/dir2/file*. Une fois le travail terminé, il demandera la mise dans l'espace de référence, ce qui invoque un outil de mouvement de données optimisé, (ou *copy tool*) capable d'exploiter plusieurs liens réseau et le parallélisme de l'HSM et du système de fichiers parallèle.

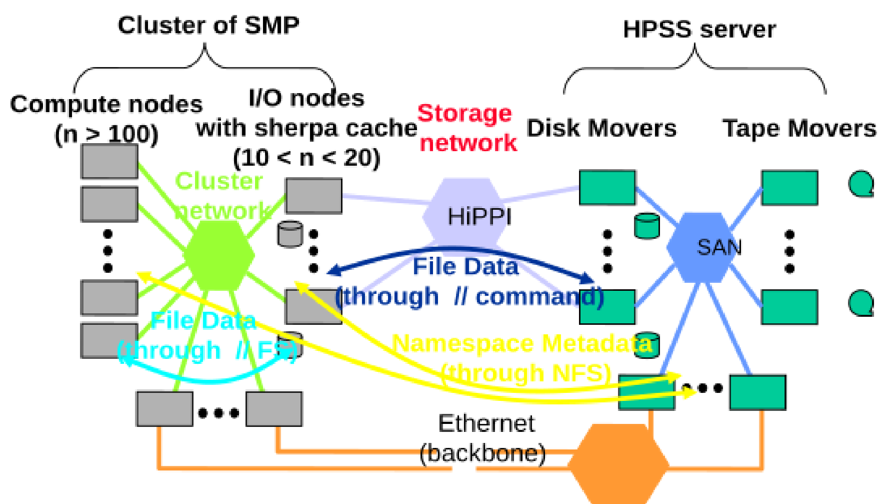


Figure 1.4 – Architecture du Cache SHerPA

L'utilisateur peut également demander des données déjà existantes, dans ce cas, lors de la création de l'arborescence cache, les données seront transférées par le *copy tool*. Les arborescences dans le cache ne sont pas nécessairement détruites en fin de calcul, elles sont préservées autant que possible pour limiter le nombre de fichiers à transférer. Un mécanisme de validation de cache est requis pour déterminer si la copie est à jour par rapport à la référence : on utilise alors une simple comparaison des derniers temps d'accès au fichier, obtenus par des appels à la fonction *stat()* de la libC sur chacune des entrées.

Enfin, il peut être fréquent que des arborescences inutilisées persistent dans le cache. Un mécanisme spécifique, lancé en tâche de fond et nommé *zapper* vient réaliser les opérations de ramasse-miettes. Là encore, c'est la comparaison entre les chemins et les métadonnées des fichiers et répertoires dans les espaces de référence et de cache qui est exploité : tout fichier ou répertoire du cache qui ne possède pas un équivalent dans la référence est supprimé. Tout fichier ou répertoire dont les temps d'accès sont antérieurs à ceux de son "jumeau" dans la référence est également effacé.

Si aucun orphelin n'est détecté, le zapper classe les sous-arborescences de la plus ancienne à la plus récente en se basant sur ses métadonnées POSIX et les efface à hauteur d'un seuil défini dans un fichier de configuration. L'occupation du cache oscille donc entre une marque haute (*High Water Mark*) et une marque basse (*Low Water Mark*).

1.4 .Limitations dans l'implémentation de NFS

Le fonctionnement du cache SHerPA requiert que soit possible des comparaisons des métadonnées POSIX des fichiers dont une partie est obtenue au travers de NFS. Dans ses premières versions, il s'appuie sur une implémentation de NFS fourni dans la suite logicielle HPSS. Cette dernière présente de nombreuses limitations intrinsèques qui posent de multiples soucis alors que de plus en plus d'utilisateurs utilisent les systèmes, produisant plus de données et des fichiers plus nombreux et plus volumineux.

En particulier, ce serveur NFS intégré dans la suite logicielle HPSS provient d'un portage d'un logiciel propriétaire de la société Sun Microsystems et ne supporte que NFSv2. L'auteur du présent mémoire a été en première ligne dans l'implémentation de NFSv3 dans ce produit en 1998. Les sources du produit sont fournies au titre du support du produit HPSS [3]. Cette nouvelle fonctionnalité sera intégrée par IBM dans les versions ultérieures de HPSS.

Si ce développement permet de s'affranchir des limitations protocolaires de NFSv2 (son absence de cache et son incapacité à gérer correctement des

3. la licence propriétaire autorise la mise à disposition des sources de l'ensemble des composants

fichiers de plus de 2 GB en particulier), il ne change rien aux autres limitations de cette implémentation de NFS, en particulier l'incapacité de ce logiciel à utiliser plus d'un processeur (à cette époque les processeurs ne disposent que d'un seul cœur) et l'absence de gestion des fonctionnalités de multi-threading. Il en résulte des performances très limitées qui rendent difficile l'exploitation du système.

De plus, ce service est instable, est facilement surchargé et tombe en panne de manière trop fréquente. Décision est donc prise de développer un nouveau serveur NFS optimisé pour bénéficier des fonctionnalités offertes par les threads multiples au sein du processus, une fonctionnalité encore assez jeune à ce moment-là. Un autre axe dans la conception devra permettre de disposer de caches mémoire importants afin d'accélérer les performances et la qualité du service final rendu à l'utilisateur. C'est dans ce but que j'ai proposé, conçu et développé le serveur NFS-Ganesha.

1.5 .Le serveur NFS-Ganesha

NFS-Ganesha est pensé pour être un serveur NFS doté des caractéristiques suivantes :

- la généricité des *back-ends* : le serveur doit être capable de montrer l'espace de nommage géré par HPSS, mais il doit être facilement possible de lui ajouter d'autres back-ends dans lesquels conserver les données ; NFS-Ganesha ne doit pas être dédié au seul espace de nommage HPSS ;
- la généricité protocolaire / généricité des *front-ends* : les versions 2 et 3 de NFS seront supportées, mais il doit être possible de facilement ajouter au produit le support d'autres protocoles de serveurs de fichiers ;
- l'utilisation intensive de cache : afin de limiter les accès à l'espace de nommage de HPSS, dont les accès sont relativement lents en comparaison d'un système de fichiers traditionnel, le produit gérera des caches de données et de métadonnées afin de limiter les requêtes dans HPSS ;
- l'utilisation du multi-threading : l'API des threads POSIX devient un standard et rejoint les appels généraux de la libC. Leur utilisation permet d'augmenter considérablement les performances des processeurs.

L'architecture du serveur NFS-Ganesha est décrite par la figure 1.5 :

Différentes couches apparaissent sur la figure 1.5

- la couche *Network* gère toutes les interactions avec le réseau ;
- la couche *Reqs Dispatch* distribue les messages lus sur le réseau et les répartit sur différents *workers threads* qui les interpréteront et effectueront les actions sur les couches inférieures ;
- la couche ONC/RPCv2 est une implémentation *thread safe* du protocole ONC/RPCv2 sur lequel s'appuie les différentes versions de NFS ;
- la couche suivante contient les implémentations des protocoles NFS

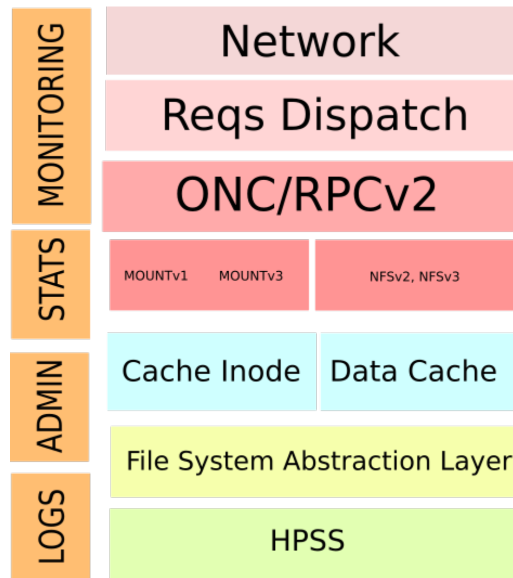


Figure 1.5 – Architecture de NFS-Ganesha, version Terascale et initale

dans leurs versions 2 et 3 et des protocoles auxiliaires *Mount Protocol* dans leurs versions 1 et 3 ;
- les deux modules qui viennent ensuite implémentent des caches utilisés par les couches supérieures, l'un pour les données, l'autre pour les métadonnées ;
- la couche *File System Abstraction Layer* (abrégée en FSAL) fournit une API générique appelée par les couches de cache. Cette API permet d'accéder au stockage sous-jacent.
- le niveau le plus bas est constitué par les librairies propriétaires fournissant l'API qui permet d'accéder à l'espace de nommage de HPSS, elles seront utilisées pour mettre en œuvre le module FSAL_HPSS qui permet de coder les différents appels de l'API FSAL afin qu'ils accèdent HPSS.

Les sections suivantes décriront plus en détail les différentes couches. On notera la présence de couches transverses qui ne seront pas détaillées, car ne présentant pas un intérêt scientifique et technique majeur :
- une couche pour effectuer la surveillance du bon fonctionnement du serveur ;
- une couche pour collecter des informations de statistiques quant à l'utilisation du serveur NFS-Ganesha, et la production de fichiers de rapports d'utilisation ;
- une couche destinée à faciliter l'administrabilité du serveur ;
- une couche très générique de journalisation, permettant de conserver des informations horodatées dans les fichiers de journaux (ou *logs*) du

serveur.

1.5.1 . La File System Abstraction Layer et la gestion des back-ends

La File System Abstraction Layer (FSAL) est un composant majeur de l'architecture de NFS-Ganesha. Le serveur NFS-Ganesha est générique et doit pouvoir interagir avec différents systèmes de fichiers même si la cible principale de cette version est l'exportation de l'espace de nommage du produit HPSS.

La FSAL est fondamentalement une API interne qui enveloppe tous les appels à une bibliothèque externe associée à la gestion de l'espace de nommage exporté par le serveur NFS-Ganesha. Elle propose des types et des structures de données génériques ainsi qu'une batterie d'appels génériques les utilisant et qui permet de réaliser l'ensemble des opérations nécessaires à l'implémentation du service NFS.

Les types utilisés dans l'API de la FSAL sont présentés dans le tableau 1.2. Les fonctions qui les exploitent seront décrites dans les tableaux 1.3 et 1.4.

Nom du type	Nature du type
fsal_size_t	Taille d'une entrée
fsal_off_t	Offset dans un fichier
fsal_uid_t	User Id d'un utilisateur
fsal_gid_t	Group Id d'un groupe d'utilisateur
fsal_attrib_list_t	Attributs d'une entrée
fsal_accessmode_t	Droits d'accès d'une entrée
fsal_acl_t	ACL associée à une entrée
fsal_openflags_t	Flags à l'ouverture d'un fichier
fsal_locktype_t	Type de verrou sur un fichier
fsal_status_t	Code retour d'une opération
fsal_buffdesc_t	Zone mémoire gérée par la FSAL
fsal_dir_t	Répertoire
fsal_dirent_t	Entrée dans un répertoire
fsal_file_t	Entrée de type fichier
fsal_handle_t	Handle d'une entrée
fsal_name_t	Nom d'une entrée
fsal_path_t	Chemin d'une entrée

Table 1.2 – Principaux types génériques de la FSAL

La structure FSAL Handle et la fonction lookup()

De tous les types de la FSAL, c'est le type *fsal_handle_t* qui est de loin le plus important, l'API de la FSAL est en effet construite autour de cette structure.

Un handle est une structure dont le contenu est opaque à l'API qui l'exploite, mais qui a une signification pour les couches situées au-dessous. De nombreuses APIs dans le noyau Linux, qui empilent les différentes couches d'abstraction, sont notamment construites de cette manière. Le FSAL Handle est vu comme un buffer mémoire défini simplement par une adresse mémoire et une taille (et qui sera géré par le type *fsal_buffdesc_t*). Le handle permet donc d'adresser de manière associative une entrée dans l'espace de nommage. Toute entrée est identifiée par un et un seul handle, en termes mathématiques, il existe une relation bi univoque entre l'ensemble des entrées et l'ensemble des handles.

Dans le cas du logiciel HPSS, c'est la structure du handle de l'API HPSS, non opaque pour l'API de HPSS, qui est recopiée dans le FSAL Handle.

Les appels de la FSAL reproduisent la sémantique des systèmes de fichiers, en particulier la couche Virtual File System (VFS) dans le noyau, mais ils reproduisent également fortement le comportement des appels des différentes versions du protocole NFS. On trouvera globalement deux familles d'appels qui se caractérisent par la manière dont une entrée est adressée :

- adressage direct : une entrée est adressée directement par son FSAL Handle.
- adressage indirect, par nom et parent : une entrée est adressée par le répertoire d'un répertoire qui la contient et le nom de celle-ci dans ce répertoire.

Il est évident que, contrairement à l'adressage direct, il n'y a pas unicité de l'adressage indirect : un fichier qui dispose de plusieurs liens physiques (*hardlinks*[4]) aura autant d'adresses de ce type que de liens. Ce type d'adressage est en particulier utilisé pour créer de nouvelles entrées (créer un fichier, un répertoire ou un lien symbolique par exemple), mais aussi pour les effacer ou les déplacer. En effet, ce type d'action impacte également le répertoire parent qui voit son propre contenu modifié (on ajoute ou l'on supprime des entrées dans le répertoire).

Les deux types d'adressage ne sont pas imperméables, bien au contraire, la fonction *lookup()* fera le lien entre les deux. Cette fonction est fondamentale dans la mise en place de la structure arborescente de l'espace de nommage.

La fonction lookup() prend en argument un couple d'adresses indirectes et produira en sortie le FSAL Handle qui donne un adressage direct vers l'entrée. Utilisé de manière itérative, s'il est associé avec la fonction *readdir()* qui permet d'obtenir la liste des noms des entrées d'un répertoire et la fonction *getroot()* qui retourne le FSAL Handle de la racine de l'espace de nommage.

On pourra parcourir le système de fichier avec un algorithme aussi simple

---

4. Dans la suite du document, le terme «hardlink», en un seul mot, sera utilisé pour désigner les liens physiques dans les systèmes de fichiers. Le but est ici de bien distinguer les liens physiques des liens symboliques

que celui-ci :

1. On commence avec la racine de l'espace de nommage (dont le schéma est «/»), qui devient le répertoire courant, connu par son adressage direct par FSAL Handle ;

2. (a) on réalise un readdir() pour connaître le contenu du répertoire courant, on obtient la liste des noms du répertoire ;
   (b) on construit pour chaque nom un adressage indirect pour chaque entrée de ce répertoire ;
   (c) on traduit ce couple (parent, nom) en le Handle unique de cette entrée grâce à la fonction lookup() ;
   (d) on utilise ce FSAL Handle pour connaître les attributs de l'entrée en question, si cette entrée est un répertoire, on appelle récursivement cette fonction, l'entrée en question devenant le répertoire courant dans cet appel.

On note que le processus itératif de lookup permet de convertir un chemin, par exemple «/mnt/directory/file», en un FSAL Handle :

- le premier «/» désigne la racine
- les chaînes entre deux «/» ou entre un «/» et la fin de la chaîne sont des noms d'entrées (dans l'exemple il s'agira de «mnt», «directory» et «file»)
- partant de la racine qui me donne le premier répertoire courant, on parcourt la chaîne en résolvant les couples (parent, nom), si l'entrée n'est pas un répertoire ou n'existe pas, on retourne une erreur et sinon on poursuit jusqu'à atteindre la fin de la chaîne et obtenir le FSAL handle final désiré.

Optimisation du processus de readdir() et lookup() : la fonction readdirplus()

Il existe une optimisation classique à la logique de l'utilisation couplée de lookup() et readdir(), il s'agit de l'introduction de l'appel readdirplus(). Celle-ci ne va pas simplement retourner la liste des noms contenus dans le répertoire, mais aussi le FSAL Handle qui correspond à chaque entrée. C'est le comportement par défaut de l'appel readdir() qui est implémenté dans l'API de la FSAL. Celui est même étendu et l'appel retournera aussi les attributs de chaque entrée. En effet, un cache de métadonnées est connecté à la FSAL et celui-ci va cacher chaque entrée avec ses attributs. On économise ainsi une boucle qui répéterait des appels à *getattr()* pour obtenir l'ensemble des attributs. Ce cache est décrit à la section 1.5.3.

Les fonctions de la FSAL

Les principales fonctions de la FSAL sont décrites dans les tableau 1.3 et 1.4, organisées selon le type d'adressage utilisé dans cet appel.

Fonction	Nature de la fonction
FSAL_lookup()	Retourne le FSAL Handle à partir d'un couple (Handle Parent, nom)
FSAL_lookupPath()	Retourne le FSAL Handle à partir d'un chemin depuis la racine
FSAL_access()	Indique si une action est autorisée, pour un utilisateur, sur une entrée
FSAL_truncate()	Effectue une troncature sur un fichier
FSAL_getattrs()	Retourne les attributs d'une entrée
FSAL_setattrs()	Modifie les attributs d'une entrée
FSAL_readdir()	Lit le contenu d'un répertoire (avec optimisation readdirplus)
FSAL_open()	Ouvre un fichier, on obtient en sortie un descripteur de fichier ouvert
FSAL_read()	Effectue une opération de lecture dans un fichier ouvert
FSAL_write()	Effectue une opération d'écriture dans un fichier ouvert
FSAL_close()	Ferme un fichier précédemment ouvert
FSAL_readlink()	Suit le contenu d'un lien symbolique
FSAL_lock()	Pose un verrou sur un fichier.
FSAL_changelock()	Change le type d'un verrou posé sur un fichier
FSAL_unlock()	Relâche un verrou sur un fichier

Table 1.3 – Principaux appels génériques de la FSAL, avec adressage direct

Cas particulier de la gestion des verrous

Les appels associés à la gestion des verrous sont directement réalisés sur l'espace de nommage, et il n'existe pas de méthode de cache dans cette version de NFS-Ganesha (les versions ultérieures apporteront de nouvelles fonctionnalités dans ce domaine). La principale FSAL utilisée dans cette version de NFS-Ganesha, qui s'appuie sur le produit HPSS, ne dispose pas de verrous, car HPSS ne dispose pas lui-même de cette fonctionnalité. Par conséquent, ces appels seront des «coquilles vides» dans cette version. Du code y sera ajouté dans les versions ultérieures, quelques années plus tard.

Les différentes FSAL disponibles durant la période terascale

NFS-Ganesha est pensé pour être un produit open-source et générique. Ce n'est pas qu'un simple outil en production, mais aussi un objet qui permet de participer à la formation de futurs ingénieurs via de nombreux sujets de stages. En particulier, le développement de nouvelles bibliothèques FSALs est

Fonction	Nature de la fonction
FSAL_create()	Création d'un nouveau fichier
FSAL_mkdir()	Création d'un nouveau répertoire
FSAL_link()	Création d'un lien physique,
FSAL_open_by_name()	Identique à l'appel FSAL_open(), un adressage par couple
FSAL_symlink()	Création d'un nouveau lien symbolique
FSAL_rename()	Change le nom d'une entrée
FSAL_unlink()	Détruit une entrée dans l'espace de nommage

Table 1.4 – Principaux appels génériques de la FSAL, avec adressage par parent et nom

un sujet plusieurs fois mis en avant.

1.5.2 . FSAL disponibles et produits dérivés

La FSAL "Fuse Like"

FUSE [33] est un produit bien connu de la communauté Linux. Cet acronyme signifie **F**ilesystem in **U**ser**S**pac**E**, en français : « système de fichiers en espace utilisateur ». Diffusé en logiciel libre, FUSE permet à un utilisateur sans privilège particulier d'accéder à un système de fichiers sans qu'il soit nécessaire de modifier les sources du noyau.

FUSE fournit un module noyau spécifique pour s'interfacer avec du code résidant en espace utilisateur, lequel agit donc comme une passerelle entre l'espace utilisateur et l'espace noyau. Le code résidant dans l'espace utilisateur est implémenté sous forme de *call-backs* qui permettent de rapidement mettre en place une logique de système de fichiers.

Une grande quantité de petits systèmes de fichiers virtuels sont ainsi disponibles, l'un des plus connus est SSHFS qui permet de visualiser les données d'un site distant via Secure SHell (SSH) et SFTP comme s'il s'agit d'un simple point de montage. On peut également citer le module NTFS-3G qui rend possible, depuis Linux, l'accès à un disque formaté selon le système de fichiers NTFS du monde Windows.

FUSE est également très populaire parmi les communautés qui développent de nouveaux systèmes de fichiers [49], en particulier pour le prototypage. L'API de FUSE s'appuie sur des *handles* comme l'API FSAL de NFS-Ganesha. Interfacer les deux se fait assez naturellement

La FSAL_FUSELIKE fait le lien entre l'API FUSE et l'API FSAL : elle encode chacun des appels de l'API FUSE en utilisant les appels de l'API FSAL. Les différents sous-modules de FUSE sont ainsi utilisables directement dans NFS-Ganesha.

Le serveur NFS qui en résulte expose les fonctionnalités du système de fichiers virtuels encodés dans la bibliothèque dédiée à FUSE, et peut être monté comme un système de fichiers. Le résultat réside à 100% dans l'espace utilisa-

teur, ce qui présente de nombreux intérêts pour le déverminage

Cette FSAL contribuera beaucoup à l'essor du produit NFS-Ganesha au sein des communautés open-source après avoir été présentée en 2008 au Linux Symposium [31]. Elle sera en particulier citée dans différents documents de référence en ligne, dont celui de Ben Martin [63].

La FSAL POSIX

Si l'API de la VFS est basée sur la gestion des handles, elle ne disposait pas encore des appels nécessaires à manipuler ces structures depuis le user land [5]. Exporter via NFS-Ganesha un système de fichiers POSIX, fait alors figure de quadrature du cercle. L'étude relative à cette FSAL sera effectuée dans le cadre d'un stage que j'ai encadré.

Cette FSAL consiste à mettre en place un mécanisme qui permet un adressage direct basé sur un handle à partir de l'adressage basé sur des chemins, via une base de données Structured Query Language (SQL) [6] qui assure l'association entre ce handle et le chemin complet, permettant de convertir l'un en l'autre. Par la suite, il était simple d'utiliser les routines présentes dans la libC pour agir sur les systèmes de fichiers.

Cette implémentation souffre cependant d'un défaut de taille : si l'espace de nommage est modifié en dehors du périmètre du serveur NFS, les informations contenues dans la base de données deviennent incohérentes et inutilisables. Des mécanismes de récupération sont mis en place, mais ils ne donnent pas de bons résultats, et on n'utilisera cette FSAL que pour exposer des systèmes de fichiers en lecture seule.

Cette FSAL sera rendue caduque quelques années plus tard, quand les appels *name_to_handle_at()* et *open_by_handle_at()* seront ajoutés à la VFS et à la libC. Cette nouvelle FSAL sera détaillée dans le chapitre suivant.

La FSAL SNMP

Le protocole Simple Network Management Protocol (SNMP)[41] est à cette époque le protocole standard qui permet à des services, des machines ou des équipements connectés en réseau de faire remonter des informations de statut, des erreurs ou des statistiques à une grappe de serveurs de surveillance centraux. L'étude relative à cette FSAL sera faite par le biais d'un stage de fin d'études d'école d'ingénieurs, axé sur le protocole SNMP que j'ai encadré.

Dans le cadre de cette étude conduite avec le stagiaire, une FSAL en mode «lecture seule», qui n'implémente que les appels permettant de parcourir l'arbre SNMP et de lire les valeurs terminales des branches, est réalisée. Elle

5. les appels en question ne feront leur apparition que quelques années plus tard

6. A cette époque, il n'existait pas dans le paysage informatique d'outil plus simple à mettre en place. De nos jours, un tel mécanisme serait probablement architecturé autour d'un Key-Value Store comme REDIS

permettra le montage via NFS en *read-only* de cette arborescence qui peut dès lors être parcourue et manipulée depuis un shell avec des commandes telles que *cd*, *ls* ou *cat*.

Cette fonctionnalité n'a jamais été mise en production, mais a fait l'objet de différentes communications, dont celle produite lors du Linux Symposium de 2007 [31] pour démontrer le caractère générique de NFS-Ganesha et du formalisme de la FSAL.

1.5.3 . les couches de cache

Les couches de cache sont intrinsèquement liées au fonctionnement du serveur NFS-Ganesha, lui permettant d'opérer différentes optimisations en termes de performances. Elles ont un fort impact dans le cadre de cette première version afin d'accélérer l'accès à l'espace de nommage de HPSS qui est particulièrement lent.

On trouvera trois types de cache dans NFS-Ganesha :

- le *Duplicate Request Cache* ou DRC qui vient optimiser le fonctionnement du protocole RPC
- le cache de métadonnées qui accélère le rendu des requêtes utilisateurs en évitant de systématiquement solliciter la couche FSAL
- le cache de données qui permet de réaliser des lectures et des écritures sur un système de fichiers rapide, local au serveur, ce qui décharge la FSAL.

Les arbres bicolores

La gestion d'un cache est très centrée sur la manière dont les différents enregistrements sont adressés. Dans le cas de cette version terascale, on utilise un adressage associatif basé sur les arbres bicolores (ou *red-black trees*). Ces arbres sont l'une des pierres angulaires sur lesquelles s'appuient les différents caches de NFS-Ganesha

Les arbres bicolores forment un type particulier dans la famille des arbres binaires de recherche équilibrés. Cet algorithme revient à Rudolf Bayer qui, en 1972, nomme ces arbres *symmetric binary B-trees* [13]. Chaque nœud de l'arbre possède un attribut binaire, qui peut être vu comme sa "couleur" (rouge ou noire). Cet attribut binaire permet de garantir l'équilibre de l'arbre lors de l'insertion ou la suppression d'éléments en conservant certaines propriétés sur les relations entre les nœuds

Il a été démontré [13] [85] que malgré les possibles actions de ré-arrangements et de re-coloriages durant les suppressions et les insertions, les temps de réponse évoluent d'une manière logarithmique avec le nombre total de nœuds contenus dans un tel arbre.

Architecture générique des caches

Les caches dans NFS-Ganesha sont des tableaux statiques de références vers des arbres bicolores. La taille de ce tableau est un nombre premier P. Chaque cache est associé à deux fonctions de hachage :

- la première produira un résultat entre 0 et $P-1$, elle permet de choisir l'arbre bicolore dans lequel insérer la valeur,
- la seconde fonction calcule une valeur positive sur 64 bits, qui accélère le placement dans les RBT : c'est celle-ci qui définit la "valeur" du nœud RBT qui correspond à l'entrée, ce qui déclenchera de potentielles opérations de réorganisation de l'arbre bicolore. Cette fonction de hachage permet d'identifier la zone de l'arbre où insérer le nouveau nœud portant la nouvelle entrée.

L'utilisation de plusieurs arbres est motivé par les exigences de performances : les fonctions de hachage sont des opérations en $O(1)$ tandis que les opérations sur un arbre bicolore sont en $O(log(N))$ où N est le nombre total d'entrées. P arbres bicolores peuplés de N/P éléments occuperont la même place qu'un seul arbre contenant P éléments, mais promettent des performances en $O(log(N/P))$, il en résulte des accroissements de performances surtout si P est grand,

Le Duplicate Request Cache (DRC)

Certaines des opérations du protocole NFS sont idempotentes : elles peuvent être effectuées plusieurs fois à la suite, le résultat sera celui obtenu dès la première action (par exemple changer le propriétaire d'un fichier avec un appel comme *chmod()*). D'autres, en revanche, ne sont pas idempotentes, si elles sont exécutées plusieurs fois, il en résultera une erreur, par exemple, effacer un fichier une seconde fois alors qu'il a déjà été détruit par le premier appel.

Les clients NFS de cette période peuvent utiliser différentes couches de transport, et il est très fréquent de voir des montages NFSv2 s'appuyer sur le protocole User Datagram Protocol (UDP), avec lequel il est fréquent de perdre des messages quand la bande passante réseau est trop sollicitée. Le client, s'il n'obtient pas de réponse à un message, l'émet à nouveau après un délai configurable. Le message porte les mêmes informations d'enveloppe, en particulier, il a la même *xid* que l'original. Il suffit dès lors de conserver dans un cache les réponses précédemment envoyées pour le émettre à nouveau, en particulier pour les actions non idempotentes.

Ce cache est pensé pour être assez petit, au plus une à deux centaines d'entrées, ce qui représente quelques dizaines de secondes de fonctionnement. Les entrées sont indexées par la xid du message pour leurs insertions dans les arbres bicolores. On utilisera donc la configuration de cache suivante :

- Le nombre d'arbres bicolores exploités est petit, mais premier, une va-

leur de 11 ou 13 s'avère suffisante à l'usage ;

- la xid est une valeur sur 32 bits, il n'est pas utile de la hacher, on l'utilise directement pour positionner l'enregistrement dans l'arbre bicolore correspondant ;
- la fonction de hachage permettant de choisir l'arbre bicolore est la xid *modulo* P, le nombre (premier) d'arbres utilisés.

L'emploi d'une mécanique aussi lourde que des arbres bicolores n'est pas indispensable pour un cache de si petite taille, une liste chainée aurait pu suffire avec des performances proches. Toutefois, la structure de cache étant existante dans le code, le choix a été fait d'appliquer les mêmes mécanismes pour tous les caches.

Le cache de métadonnées

Le cache de métadonnées est le cache le plus important dans l'architecture de NFS-Ganesha. Sa taille justifie donc l'utilisation d'une structure aussi complexe que les arbres bicolores pour sa gestion.

L'espace de nommage exposé par cette version de NFS-Ganesha, à savoir celui du produit HPSS, est très lent, surtout au regard des opérations en mémoire. On choisit alors de conserver autant d'informations que possible en mémoire afin de répondre à toutes les requêtes qui ne modifient pas l'état de HPSS, et de l'espace de nommage exposé au sens large. On retrouve ici une dichotomie que pour les opérations idempotentes et non idempotentes du DRC mais qui n'est pas identique : en effet, l'opération SETATTR, qui modifient les attributs POSIX d'une entrée dans l'espace de nommage, est belle et bien idempotente bien qu'elle modifie l'état de l'entrée concernée.

Pour implémenter ce cache de métadonnées, on utilisera le mécanisme de cache basé sur les arbres bicolores décrit plus haut avec la configuration suivante :

- Le nombre d'arbres bicolores exploités est grand, car l'on s'apprête à cacher des dizaines de milliers d'entrées, on utilisera de l'ordre de 10^2 à 10^3 arbres bicolores. La configuration choisie utilisera généralement les valeurs 241, 919 et 1531.
- les fonctions de hachage sont dépendantes de la FSAL (voir paragraphe 1.5.1), elles opèrent sur le *FSAL Handle*. Ces premières fonctions de hachage s'appuient sur des "ou exclusif" (XOR) et des décalages de bits.

L'API du cache de métadonnée est similaire à celle de la FSAL, et pour chaque appel de cette API, on va trouver un appel correspondant dans l'API du cache de métadonnées. Cette API sera détaillée dans les chapitres suivants qui montreront notamment ses évolutions avec les différentes versions de NFS-Ganesha.

Le cache de données

Le cache de données est construit comme un corollaire du cache de métadonnées. Chaque entrée de type "fichier" dispose d'un pointeur vers un fichier dans le répertoire utilisé comme cache local. La logique de fonctionnement est la suivante :

- le fichier existe dans le cache de métadonnées décrit au paragraphe 1.5.3;
- lors du premier accès en lecture ou en écriture, la totalité du fichier est recopiée depuis la FSAL vers le cache disque local au serveur NFS;
- les accès suivants auront lieu dans le fichier présent dans le cache local
- un fichier qui a été créé dans l'espace de nommage, et qui est donc vide, est associé avec une entrée dans le cache local dès la première écriture dans ce fichier, l'opération écrit alors dans le cache local.

Ce cache est destiné à gérer les très petits fichiers présents dans HPSS, laissant les gros fichiers sous la gouvernance du cache SHerPA décrit à la section 1.3. Les petits fichiers représentent beaucoup moins de 0.5% des fichiers. Les requêtes de lecture dans ces fichiers peuvent être nombreuses et coûteuses en termes de ressources, en particulier, elles vont saturer rapidement le logiciel HPSS, très inadapté à ce type de profil. Ce cache de données vise à réduire autant que faire se peut ces actions de lecture et d'écriture.

Le système de fichiers exploité pour héberger les données de ce cache local est dimensionné pour être assez grand pour héberger la totalité de ces petits fichiers. Ce cache, contrairement aux DRC et au cache de métadonnées qui exploitent la RAM, est implémenté via une ressource de stockage sur disques, et il survit au redémarrage du daemon et au redémarrage électrique du serveur.

Ramassage de miettes

Le ramasse-miettes utilise la stratégie basée sur une liste doublement chaînée. Quand le cache atteint une "marge haute" (ou *High Water Mark*) un thread dédié commence à retirer les entrées les plus anciennes du cache, forçant l'écriture de l'entrée correspondante dans la FSAL. Il agira ainsi jusqu'à ce que le cache passe en dessous d'une marque basse (ou *Low Water Mark*).

L'implémentation de ce ramasse-miette passe par la mise en œuvre de l'algorithmique Least Recently Used (LRU) agissant sur une liste doublement chainée pour faciliter les insertions et les suppressions.

1.5.4 . La gestion du réseau en mode multi-thread

Les threads sont un formalisme qui devient très populaire durant cette période terascale. Les processeurs sont de plus en plus efficaces pour gérer différentes activités simultanément, et l'API des threads POSIX permet de regrouper efficacement ces activités.

Ce formalisme remplace les anciens modes de programmation ou différents processus Unix, un pour chaque activité, fonctionnent avec des APIs dédiées (System V en particulier). Du côté des codes de calcul, c'est le formalisme OpenMP qui fait une entrée en force, mais du côté de la programmation système, c'est l'API POSIX, avec les *pthread* qui est massivement utilisée dans NFS-Ganesha.

La prise en compte des threads est faite dès les couches réseaux du serveur afin d'opérer le parallélisme, via les pthreads, au niveau de la gestion des messages.

Dans cette première version de NFS-Ganesha, un unique thread *dispatcher* vient lire tous les messages incidents. Il sait identifier le début et la fin du message grâce aux informations d'enveloppe du protocole Open Network Computing Remote Procedure Call version 2 (ONCRPCv2) sans en lire le contenu ou *payload*. Il doit alors choisir un worker auquel confier le traitement du message en privilégiant celui dont la file d'attente est la moins remplie.

L'architecture du serveur NFS-Ganesha est décrite par la figure 1.6.

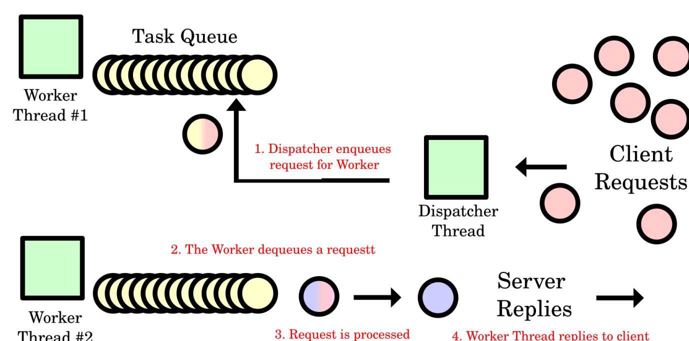


Figure 1.6 – Traitement des requêtes entre le dispatcher et les workers

Chaque worker vient traiter les requêtes présentes dans sa propre file en mode First In First Out (FIFO). Il vient interpréter la payload conformément à ONCRPCv2, interprète le message selon le protocole adapté (NFS ou Mount Protocol) et applique les actions qui correspondent sur le backend, au travers des couches de cache et de la FSAL.

Le status retourné par les actions via les caches et la FSAL sont encodés conformément à NFS, Mount Protocol et ONCRPCv2,et le message est retourné au client à travers les couches réseau. Le worker est alors libre de traiter une nouvelle requête stockée dans sa file d'attente, le cas échéant.

1.5.5 . Les protocoles d'un serveur NFS

Cette version de NFS-Ganesha implémente les protocoles NFSv2 et NFSv3 et les protocoles auxiliaires Mountv1 et Mountv3. Ces derniers sont décrits en annexe dans la partie B.

NFSv2 est présent pour des raisons de compatibilité avec les standards de l'époque, mais il présente des limitations structurelles majeures qui le rendent inutilisable dans un environnement HPC qui utilise de très gros fichiers. En particulier, NFSv2 gérant la taille des fichiers sur seulement 32 bits, il est impropre à gérer des fichiers de plus de 4 Gigaoctets. L'exploitation en production de NFS-Ganesha se fera via NFSv3 qui s'appuie sur des entiers 64 bits.

Implémentation des protocoles NFS et Mount dans NFS-Ganesha

Dans la forme, NFS-Ganesha implémente, sous la forme de deux bibliothèques de fonctions à usage interne, toutes les fonctions nécessaires pour implémenter NFSv2 et ses routines XDR afférentes. Ces routines sont amenées à travailler sur les buffers stockés dans les files d'attente des différents workers par le thread dispatcher. Les workers viennent les dépiler en FIFO avant de lire les informations d'enveloppe (grâce à XDR) pour décoder le message et réaliser les actions nécessaires au travers des couches de cache de données et de métadonnées qui viennent ensuite faire des appels explicites à la FSAL

1.5.6 . Méthodologie de développement

NFS-Ganesha est conçu comme un serveur générique et doit pouvoir être compilé et déployé sur autant d'architecture que possible.

Son code est intégralement écrit en C, il embarque des balises spécifiques dans les commentaires permettant la compilation de la documentation à partir des fichiers sources grâce au produit *doxygen*

La gestion de la compilation est assurée par la suite *Autotools*, véritablement incontournable à cette époque et seule solution véritablement envisageable [7]. Il en résulte, une séquence très classique dans les logiciels open-source de l'époque afin de construire et d'installer le binaire NFS-Ganesha, via la séquence de commande suivante :

```
# autoreconf -fi && ./configure && make && make install
```

Afin de faciliter l'intégration de NFS-Ganesha dans un environnement Linux standard basé sur une distribution Red Hat ou Fedora, un fichier *specfile* listant les recettes et les directives pour produire un fichier Redhat Package Manager (RPM). Le pendant pour les distributions Debian et Ubuntu est également réalisé, permettant de produire le fichier paquetage idoine.

7. Le produit CMake commence à percer durant la même période, mais il est encore très marginal

Durant cette période, il y a peu de FSALs disponibles. Le choix de la FSAL qui sera utilisée se fera au moment de la configuration de la construction du binaire, par une option au script *./configure*. Le binaire qui sera alors produit sera dédié à cette FSAL, il faudra donc compiler potentiellement autant de binaire que de FSALs différentes.

Tests de non-régression

La communauté qui développe des clients et des serveurs NFS est très active, et elle a très vite adopté les notions d'interopérabilité. Des solutions clients et serveurs provenant de sources différentes doivent interagir conformément aux règles décrites par la norme POSIX. Cette communauté se réunit régulièrement lors d'évènements nommés "Connectathon".

La norme POSIX est complexe, aussi les acteurs des Connectathons ont-ils rédigés des suites de tests qui permettent de vérifier si un système de fichiers, ou dans le cas présent le montage via NFS d'un système de fichiers, respecte parfaitement la norme POSIX, jusque dans ces cas d'usage les plus complexes.

NFS-Ganesha sera validé par la suite de tests du Connectathon depuis ses premières versions jusqu'aux plus récentes. Durant la période Terascale, c'est l'un des seuls outils disponibles. Les générations suivantes verront l'apparition d'outils complémentaires.

Un test commercial existe : le test SpecSFS. Vendu par l'organisme Spec, c'est avant tout un outil dédié au benchmarking mais il est connu pour soumettre le serveur qu'il teste à un stress intense et sait produire des motifs d'accès très inhabituels, voire marginals, mais parfaitement conformes. C'est notamment le cas lorsqu'il simule le comportement d'un moteur de base de données. Cet aspect en fait un outil couramment utilisé pour valider la tenue en charge d'un serveur NFS et il sera mis en œuvre dans ce cadre de non-régression.

1.6 .Bilan et perspectives pour la période Terascale

Dans ce chapitre a été présenté le contexte de la période Terascale et comment la modification des solutions de stockage des nouvelles architectures de centre de stockage HPC, architecturées autour de clusters de SMPs, imposait des refontes importantes des solutions logicielles pour accéder aux données.

Dans ce cadre, le CEA a mis en œuvre un mécanisme de cache élaboré nommé SHerPA qui a imposé le développement d'une nouvelle implémentation du protocole NFS pour accéder efficacement à HPSS, l'HSM qui contient l'ensemble des données des utilisateurs.

En effet, les solutions proposées dans la suite logicielle adossée à HPSS ne permettaient pas une mise en production optimale, il a donc été décidé d'implémenter une nouvelle solution pour les exports NFS.

Dans ce contexte, j'ai proposé, conçu et développé NFS-Ganesha, un serveur NFS générique qui fonctionne intégralement en espace utilisateur. L'architecture du logiciel et ses rouages internes, dont plusieurs caches, ont été détaillés. Les limitations des solutions existantes ont été identifiées et discutées, et il a été montré comment l'architecture de NFS-Ganesha que j'ai mise en place permet de s'en affranchir.

NFS-Ganesha dispose d'une API génétique, la FSAL, basée sur des `handles`, lui permettant de s'interfacer avec HPSS mais aussi avec d'autres espaces de nommage. Différentes sortes de FSALs sont implémentées dans le cadre de stages et d'études en amont, montrant la généricité et la viabilité de l'API.

Une première version de NFS-Ganesha voit le jour durant cette période Terascale, elle est mise en production sur le centre de calcul du CEA au tournant de l'année 2005.

Chapitre 2 : La période Petascale et les IO Proxies

Ce chapitre couvre la période Petascale, durant laquelle les supercalculateurs deviennent capables de performances de l'ordre du petaflops, soit 10^{15} flops, voire de quelques dizaines ou centaines de petaflops. Des performances proches du dixième d'exaflops seront atteintes quelques années plus tard, à la fin de cette période, soit 10^{17} flops).

Ce chapitre va présenter les innovations technologiques propres à cette période et leurs impacts sur les architectures des centres de calcul. Ces évolutions imposent des changements dans NFS-Ganesha dont les nouvelles versions voient le support de nouveaux protocoles et qui le mettent au centre d'une nouvelle fonctionnalité de centre de calcul, les IO-Proxies.

2.1 .Contexte de la période Petascale

Du point de vue du stockage de données, on est amené à gérer des volumes de plusieurs centaines de petaoctets, voire de plusieurs exaoctets (10^{18} octets). Au fil des années, les débits agglomérés incidents sur les systèmes de stockage atteignent des débits de plusieurs Teraoctets par seconde.

Le tableau 2.1 présente les performances des calculateurs SMPs déployés au CEA durant la période Petascale. Il est donné à titre de référence de comparaison.

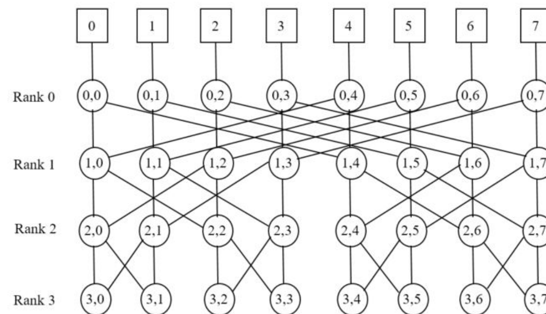
Année	Machine	Performances	Stockage
2005	Tera 10	60 Tflops	2 Po (100 Go/s), 10 Po (bandes)
2010	Tera 100	1.05 Pflops	20 Po (500 Go/s), 30 Po (bandes)
2015	Tera 1000	30 Pflops	40 Po (767 Go/s), 2 Po (SSD, 1 To/s), 100 Po (bandes)

Table 2.1 – Performances des machines du CEA durant la période Petascale

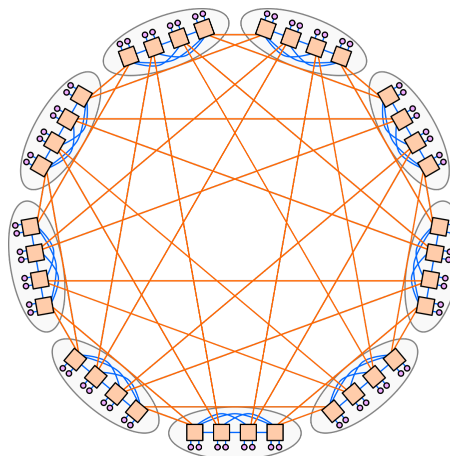
2.2 .Des machines de calcul organisées en îlots

Les architectures des supercalculateurs enrôlent de plus en plus de nœuds de calcul. Pour organiser ceux-ci, les réseaux d'interconnexion deviennent de

plus en plus complexes, on va ainsi voir apparaître parmi les machines du Top500[93] telles que *Fat Tree*[98], *HyperX*[4], *Dragonfly*[53], *Butterfly* ou *Flattened Butterfly*[52].



(a) Topologie Butterly



(b) Topologie Dragonfly

Figure 2.1 – Exemples de topologies réseaux

On ne rentrera pas dans les détails des différentes topologies ni sur leurs avantages et inconvénients. On notera toutefois qu'elles possèdent une caractéristique commune : les nœuds dans les topologies ne sont pas à égale distance les uns des autres, on peut leur appliquer des qualificatifs tels que "proches" ou "éloignés". Il est ainsi légitime de parler de voisinage d'un nœud ou d'un ensemble de nœuds. La distance entre deux nœuds pourra est ici définie par le plus petit nombre de composants réseaux à traverser pour aller de l'un à l'autre.

Dans le cadre de ces nouvelles architectures des réseaux d'interconnexion, on va voir émerger, des *îlots*, c'est à dire des ensembles de machines très proches et qui peuvent donc fonctionner ensemble de manière très efficace. Ces îlots sont assez proches, en termes de puissance de calcul, des super-

calculateurs de la génération précédente. Les machines tendent ainsi à évoluer de "clusters de SMPs" à "clusters de clusters de SMPs".

L'allocation des travaux de calcul est favorisée sur des machines qui appartiennent à un même îlot, pour maximiser les performances en réduisant les coûts des échanges réseaux. Les ressources de stockage et leur utilisation doit prendre en compte cet aspect et fournir des stratégies adaptées. Le modèle des IO-Proxies, décrits dans la section 2.9.2, est un exemple d'une telle tentative.

2.3 . Des systèmes de fichiers parallèles et distribués au centre de l'architecture

Les systèmes de fichiers parallèles et distribués sont l'innovation la plus importante à apparaître dans cette période Petascale, avec en particulier les logiciels GPFS et Lustre. Le CEA fera le choix technique de s'appuyer sur Lustre, qui est une solution opensource, contrairement à GPFS, produit commercial vendu par IBM.

2.3.1 . Le logiciel Lustre

Lustre est un système de fichiers distribué libre, très utilisé dans le cadre du HPC. Le nom "Lustre" est un "mot valise" qui est la réunion du mot "Linux" et du mot "cluster". Le projet vise à fournir un système de fichiers distribué capable de fonctionner sur plusieurs centaines de nœuds, avec une capacité de l'ordre du pétaoctet, sans altérer la vitesse ni la sécurité de l'ensemble. Lustre est distribué sous licence GPL. Il est initialement développé par la société CFS (Cluster File Systems), en 2007, la société Sun Microsystems distribue et maintient le produit.

Depuis le rachat de Sun par Oracle (2009), Lustre a été maintenu par Oracle pour les machines utilisant exclusivement son matériel, puis par sa propre communauté Open Source ainsi que certaines entreprises spécialisées.

2.3.2 . L'architecture de Lustre

L'architecture fondamentalement distribuée de Lustre est décrite par la figure 2.2. Elle regroupe trois types de serveurs : des Meta Data Server (MDS), dédiés à la gestion des métadonnées, et des Data Server (DS), nommés aussi Object Storage Server (OSS), dédiés à la gestion des données et des Management Server (MGS) dédié à l'administration et à la gestion du système lui-même.

Cette séparation explicite du traitement des flux de données et de métadonnées est l'un des éléments clefs qui permettent à Lustre de passer à l'échelle et d'assurer des performances élevées sur de très grands volumes de données et de très larges infrastructures.

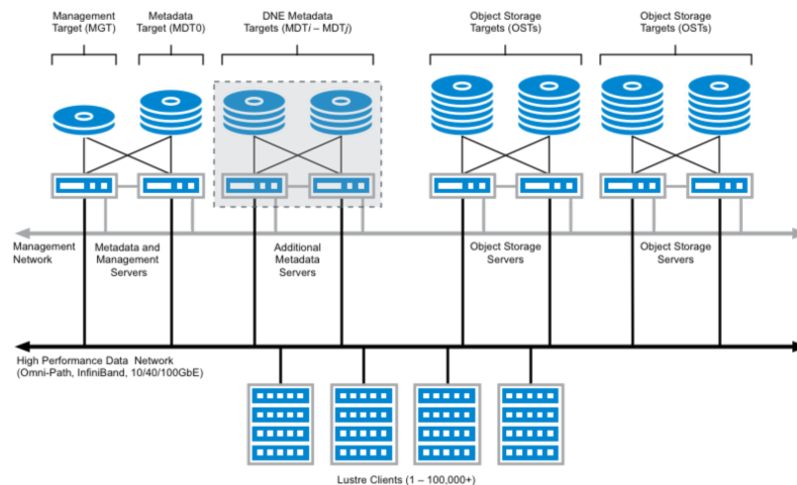


Figure 2.2 – Architecture de Lustre (source : Lustre Wiki)

Lustre distingue différents types d'objets. Les "objets" [1] sont des structures similaires à des tableaux et qui conservent les données brutes. Ils sont associés aux contenus des fichiers tandis que les index sont des associations clef-valeur qui permettent un adressage associatif de structures plus complexes comme les métadonnées POSIX. Lustre exploite ses propres protocoles réseaux dédiés au transport des données via la couche Lustre Network (LNet).

Les Meta Data Servers

Les MDS gèrent toutes les opérations sur l'espace de nommage exporté par Lustre. Ils stockent leurs informations sur des disques désignés sous le nom de Meta Data Target (MDT). Un système de fichiers Lustre possède au moins une instance de MDS et au moins un MDT, mais d'autres ressources de ces types peuvent être ajoutés durant la vie du système de fichiers, pour étendre ses performances et sa capacité. C'est le MDS qui pilote l'allocation des objets dans lesquels sont conservées les données.

Les Object Storage Servers

Les OSS fournissent une capacité de stockage brute en gérant de multiples disques, désignés sous le nom d'Object Storage Target (OST). Les OSTs contiennent des objets de stockage qui vont héberger les contenus des fichiers. De manière usuelle, un OSS va gérer plusieurs OSTs, et il est très fréquent de voir plusieurs OSS dans une configuration de Lustre. Cette multiplicité offre un striping potentiellement grand qui permet de répartir les gros fichiers sur des objets et des serveurs multiples, ce qui garantit des accès

1. Les objets dont il est ici question ne sont pas ceux dont il sera question dans le chapitre suivant, dédié à la période exascale

performants. Il est par ailleurs simple d'étendre la volumétrie de Lustre en lui ajoutant de nouveaux OSTs gérés par de potentiellement nouveaux OSS (en général de l'ordre de la dizaine, voire de la centaine).

Les Management Servers

Les MGS conservent l'ensemble des données de configuration d'un système de fichiers Lustre. Dans le cadre d'une architecture fortement distribuée comme celle de Lustre, déployée sur une grappe de machines ou cluster, les instances de MGS permettent de centraliser la gestion et la configuration de Lustre. Le MGS gère en particulier les états des différents MDS et OSS et les supervise. Les informations persistantes des MGS sont conservées sur des disques désignés sous le terme de Management Target (MGT).

2.3.3 . Connexion entre Lustre et les HSMs

Lustre dispose à cette époque d'une nouvelle fonctionnalité permettant de l'interfacer avec les HSMs[97].

Un HSM est conçu pour gérer une hiérarchie de stockage avec différentes ressources de stockage (disques et bandes). La nouvelle fonctionnalité de Lustre permet de rajouter le système de fichiers Lustre comme le niveau le plus élevé d'une hiérarchie de stockage.

Pour réaliser cette fonctionnalité, Lustre a rendu plus élaborée la façon dont les inodes sont gérées, permettant en particulier de gérer des "inodes creuses" dont le contenu n'est plus sur les OSTs du système de fichiers Lustre mais dans les ressources HSM. Ces inodes creuses peuvent voir leur contenu restauré de manière transparente pour l'utilisateur, à la première lecture dans les fichiers correspondants. Ces actions sont conduites par le coordinateur, un nouveau composant ajouté au fonctionnement du MDS.

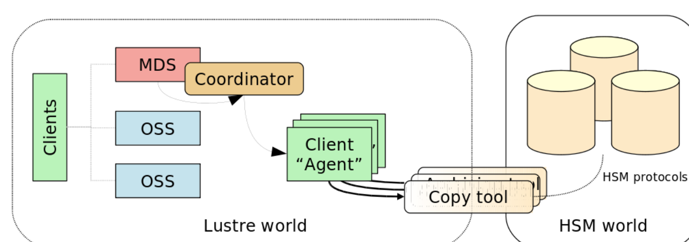


Figure 2.3 – Mouvement des données Lustre/HSM

Cette fonctionnalité implique la gestion des mouvements de données, lesquels sont pilotés par le logiciel *RobinHood Policy Engine*[57][26][56] développé par les équipes du CEA/DAM.

Les mouvements de données sont réalisés, comme indiqué par la figure 2.3 par un outil nommé *copy tool*, analogue dans son principe sur celui mis en œuvre dans le cadre du cache SHerPa. Ce dernier est un simple programme

résident en user space, capable de réaliser des IO à la fois dans Lustre et dans l'HSM. Il est déclenché par le coordinateur à chaque fois qu'une copie des données, dans un sens ou dans l'autre, est nécessaire.

2.4 .Architecture *datacentric* des centres de calcul

Dans le cas des centres de calcul du CEA/DAM, une architecture centrée sur les données, dite *data-centrique*, est mise en œuvre aux alentours de 2005. Elle est bâtie autour du logiciel opensource Lustre[16]. Le schéma 2.4 indique sa mise en œuvre dans le cadre du Très Grand Centre de calcul du CEA (TGCC) [2] au CEA.

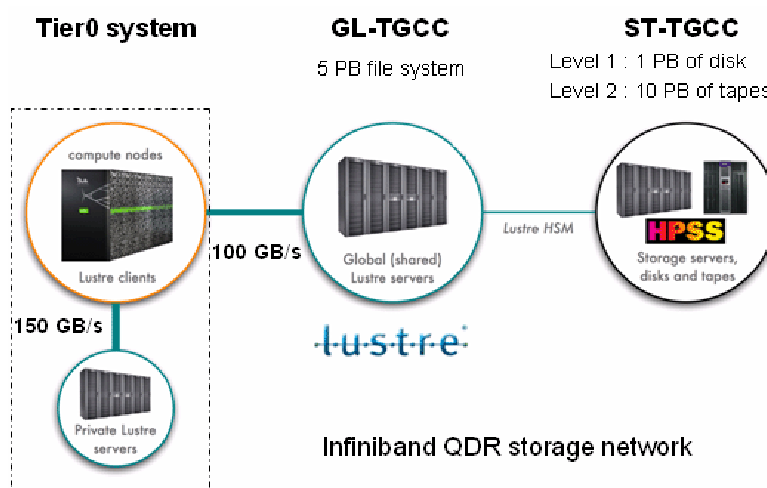


Figure 2.4 – Architecture du centre de calcul TGCC durant la période pétascale

Ce nouveau paradigme dans l'architecture des centres de calcul va faire disparaître les caches non transparents comme le cache SHerPA mis en œuvre durant la période terascale (voir le paragraphe 1.3). Lustre les rend en effet totalement inutiles : l'ensemble de machines de calcul vont directement monter le système de fichiers Lustre central.

Il n'est ainsi plus nécessaire de faire une séparation explicite entre des systèmes de fichiers globaux et locaux, des produits comme Lustre sont conçus pour être capables de gérer le très grand nombre de clients à accéder à cette ressource.

Grâce à la fonctionnalité d'interconnexion entre Lustre et un HSM, il sera possible de conserver l'ensemble des données dans le gestionnaire de stockage hiérarchique mais de référencer tous les fichiers dans Lustre, certains sous forme d'inodes creuses (voir section 2.3.3).

2. https ://www-hpc.cea.fr/fr/TGCC.html

Cette nouvelle approche est totalement transparente : les utilisateurs parcourent l'ensemble des données disponibles au travers du point de montage du système de données central. Si le fichier correspondant est stocké uniquement sur l'HSM, sur des bandes magnétiques, une opération de "démigration" dans l'HSM suivie d'une recopie des données dans Lustre, via le *copy tool* sur les OSTs gérés par Lustre. L'utilisateur n'aura pas d'action spécifique à mener, tout juste constatera-t-il un temps d'attente lors du mouvement de données qui verra celles-ci remonter depuis les bandes.

2.5 .Un effort de R&D accentué

Une convention commune de Recherches et Développement est signée en 2002 entre le CEA et l'entreprise ATOS/Bull. Cette dernière s'articule autour de différentes thématiques. le stockage de données massif est identifié comme la deuxième thématique de la convention.

J'ai été désigné comme pilote, pour la partie CEA, de cette thématique "stockage HPC". Cette nouvelle position m'a permis d'établir des collaborations étroites avec ATOS et de partager mes idées avec les équipes de ce constructeur. Cet effort a rendu possibles des évolutions impliquant NFS-Ganesha, comme les IO-Proxies que nous verrons dans la section 2.9.2, certains travaux seront menés conjointement avec ATOS, qui vient avec sa force de développement logiciel.

2.6 .NFSv4 : une nouvelle version du protocole NFS aux multiples facettes

Le protocole NFS version 4 est une vraie révolution dans le domaine des systèmes de fichiers distribués.

S'il a en commun certains points avec son prédécesseur[86], il change radicalement la logique de fonctionnement du protocole NFS en introduisant des requêtes composites qui offrent une grande souplesse. Ce protocole devient un outil pour s'attaquer aux nouveaux défis de cette période, dont ceux propres au HPC.

Le lecteur souhaitant avoir plus de détail sur NFSv4 et ses différentes versions mineures peut consulter les annexes suivantes :

- l'Annexe C qui recense les différents documents qui définissent ce protocole,
- l'Annexe D qui décrit en détail le protocole NFSv4, en particulier,
  - l'Annexe D.6.1 qui décrit le protocole NFSv4.1, qui étend NFSv4.0 ;
  - l'Annexe D.6.3 qui explicite le sous-protocole pNFS ;
  - l'Annexe D.6.4 qui donne des détails sur le futur protocole NFSv4.2 et ses nouvelles fonctionnalités.

2.7 .Naissance d’une communauté open-source NFS-Ganesha

NFS-Ganesha devient un logiciel open-source [3] en 2004. Dans les années qui suivent, je réalise plusieurs communications sur le sujet dans différentes conférences internationales, dont une présentation avec proceeding au Linux Symposium de 2007 [31] et une intervention dans les sessions *Work In Progress* de la conférence FAST’07[30].

Cette mise en lumière du logiciel, associée à l’avènement du protocole NFSv4, porte l’attention des industriels sur lui. NFS-Ganesha est alors le seul serveur NFS open-source en userland qui supporte NFSv4. C’est précisément son positionnement en dehors du noyau [4] qui va intéresser les unités de R&D de IBM puis de Panasas.

L’arrivée de NFSv4.1 et de sa fonctionnalité pNFS, laquelle s’appuie souvent sur des mécanismes et des services qui résident en user land (par exemple certains composants GPFS et les serveurs OSD2 de Panasas), font de NFS-Ganesha une solution viable pour injecter rapidement le support de pNFS dans leur propre produit.

Une communauté open-source va ainsi se former. Elle se réunit deux fois l’an lors des "Connectathons" et les Bake-A-Thon [5]. Cette communauté va ajouter des ressources de développement importantes au produit NFS-Ganesha, en particulier les contraintes des industriels à produire un logiciel facilement commercialisable [6]. En particulier, le serveur va subir une large série de tests correspondants à des cas d’usage, parfois en dehors du HPC tels que l’hébergement d’une base de données, qui vont permettre de diagnostiquer et de corriger les défauts. Certaines nouvelles fonctionnalités seront également le fruit de cette collaboration.

La communauté NFS-Ganesha qui a émergé alors est toujours très active de nos jours, elle est maintenue par Red Hat qui en a fait un élément majeur de ses solutions gravitant dans l’écosystème de Ceph et RADOS. On le trouve disponible dans les dépôts dérivant de Red-Hat Enterprise Linux [7] tels que Fedora ou SuSE. Il est également fourni dans les dépôts des distributions Linux Ubuntu et Debian. La communauté est structurée autour de son dépôt GitHub [8], qui héberge également son wiki.

---

3. à cette époque, le site qui sert de référence est sourceforge.net, github.com n’existe pas encore
4. Le serveur knfsd qui figure dans le noyau supporte lui aussi NFSv4
5. Les Bake-A-Thon sont pour l’essentiel des "connectathons additionnels", leurs dates et emplacements sont moins figés que ceux des connectathons
6. L’utilisation commerciale de NFS-Ganesha est en effet possible puisqu’il est diffusé sous les termes de la LGPLv3 ou "GPL amoindrie" [42]
7. NFS-Ganesha y apparaît dans le dépôt EPEL
8. https ://github.com/nfs-ganesha/nfs-ganesha

2.8 .Évolution de NFS-Ganesha durant la période petascale

Les évolutions décrites ci-après proviennent pour la plupart des interactions avec la communauté open-source NFS-Ganesha. J'ai réalisé celles-ci avec l'aide des contributeurs de cette communauté. Lorsque la contribution est due à un développeur de cette communauté, cela sera explicitement mentionné.

2.8.1 . Prise en compte de la sécurité

La question de la sécurité revêt un intérêt double pour NFS-Ganesha : la communauté opensource qui s'est construite autour du logiciel comprend différents industriels (dont IBM et Panasas) qui vont intégrer le logiciel dans leur solution commerciale [9], d'autre part la sécurité est un point clef pour le CEA.

J'ai réalisé la première implémentation du protocole sécurisé RPCSEC_GSS au détour des années 2003 et 2004. Ce travail est décrit plus en détail dans l'annexe E. . Par la suite, cette fonctionnalité sera maintenue et étendue par les industriels, en particulier CohortFS qui mettra en place la nouvelle bibliothèque libtirpc.

2.8.2 . Remplacement des arbres bicolores par des arbres AVL

Les arbres AVL, comme les arbres bicolores, sont des arbres binaires balancés. Ils ont été décrits initialement par Adelson, Velsky et Landis en 1962 [3]. Ils sont également décrits par Donald Knuth sous le nom de *Balanced Trees* [54]. Ils sont probablement l'une des premières implémentations des arbres de recherche binaire et partage certains points communs avec les arbres bicolores décrits à la section 1.5.3.

On peut démontrer que pour un même nombre d'entrées, les arbres AVL tendent à être moins hauts que les arbres bicolores [88]. Ainsi, pour N éléments, les arbres AVL auront une hauteur de l'ordre de $1.44log_2(N+2)-0.328$ tandis que cette hauteur sera de l'ordre de $2log_2(N)$ pour les arbres bicolores.

Les arbres bicolores sont très efficaces pour les opérations d'insertion et de suppression, mais l'algorithme a tendance à produire des arbres très profonds. Les arbres AVL sont plus simples et l'algorithme produit des arbres moins profonds. Par conséquent, ces arbres sont plus rapides pour les opérations de consultation, car il est nécessaire de traverser moins de nœuds. Cela conduit au constat suivant : les arbres bicolores sont plus efficaces quand les opérations d'insertion et de suppression dominent, les arbres AVL sont plus efficaces quand les opérations de consultation dominent. La charge sur un système de fichiers se range clairement dans la seconde catégorie. Les arbres

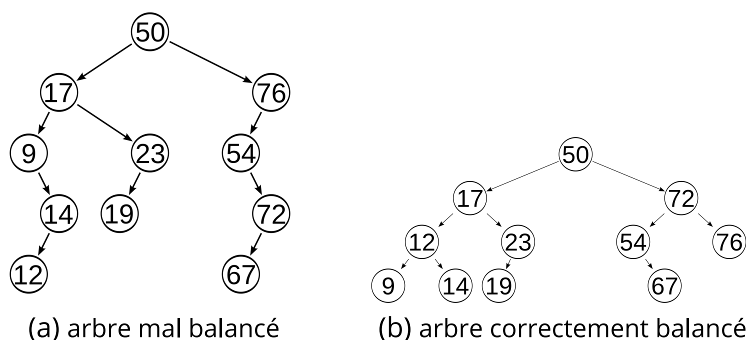
9. NFS-Ganesha est diffusé sous les termes de la licence LGPLv3 qui autorise ce type d'utilisation

bicolores de NFS-Ganesha sont donc remplacés par des arbres AVL, afin d'en optimiser les performances.

L'algorithmique des arbres AVL La notion de *hauteur* est là encore centrale. Un arbre AVL respecte les conditions suivantes :

1. La hauteur H(t) est 0 si l'arbre est vide ;
2. La hauteur H(t) d'un arbre t non vide est 1+max(H(t.l), H(t.r)) où t.l et t.r désignent les sous-arbres gauche et droit ;
3. Pour chaque nœud de l'arbre, la différence entre la hauteur du sous-arbre droit et du sous-arbre gauche H(t.r)-H(t.l) prend les valeurs 0, 1 ou -1.

La figure 2.5 montre dans sa partie gauche un arbre déséquilibré, donc non-AVL. En effet, si l'on s'attarde sur les nœuds qui portent la valeur 9, on voit que son arbre droit possède une hauteur de 2 quand la gauche a une hauteur de 0, soit une différence de 2 non conforme. Un déséquilibre similaire apparaît pour les nœuds qui portent les valeurs 54 et 76. La partie droite de cette même figure 2.5 montre le même arbre ré-équilibré qui respecte cette fois-ci les contraintes, il s'agit bien d'un arbre AVL.



(a) arbre mal balancé (b) arbre correctement balancé

Figure 2.5 – Positionnement de l'oncle dans le cas noir, ici à gauche

Un arbre AVL est rebalancé par des actions de rotations à droite ou à gauche comme décrit dans la section 1.5.3. Les arbres AVL suivent les mêmes règles de parcours infixes que les arbres bicolores.

Le remplacement des arbres bicolores par des arbres AVL est le fait de la société CohortFS [10], membre de la communauté NFS-Ganesha. Convaincu du bénéfice de cette optimisation, je suis intervenu pour les assister dans cette intégration.

10. Cette société sera rachetée et absorbée par red Hat

Utilisation de l'algorithme de hachage Murmur3

Une autre des innovations notables proposées par les développeurs de chez CohortFS est l'utilisation de la fonction de hachage non cryptographique `Murmur3`[7] dans le fonctionnement des arbres.

`Murmur3` permet de hacher un buffer selon des résultats sur 32, 64 ou 128 bits, c'est une évolution de la fonction de hachage `Lookup3`, afin de la rendre plus rapide. Comme indiqué, il ne s'agit pas d'une fonction de hachage cryptographique et elle est sujette aux collisions, mais cette situation est déjà gérée par les mécanismes internes des caches de NFS-Ganesha. L'usage d'une vraie fonction cryptographique, comme MD5 ou SHA256 ne se justifie pas en termes de coûts en ressources et de performances.

`Murmur3` va remplacer toutes les fonctions de hachage non triviales utilisées dans la version terascale de NFS-Ganesha, lui apportant un gain de performances notable.

2.8.3 . Les nouvelles FSALs

Différentes nouvelles FSAL apparaissent pendant cette période. Les premières, associées à Lustre et la VFS du noyau Linux, dérivent de la FSAL_XFS, les autres permettent d'accéder des `tanks` ZFS, le système de fichiers GPFS d'IBM. La dernière rend possible l'utilisation de NFS-Ganesha comme un proxy NFS.

Les FSALs basées sur XFS, LustreAPI et VFS

La version Terascale de NFS-Ganesha n'était pas capable de gérer efficacement les systèmes de fichiers gérés dans le noyau Linux par la couche VFS. En effet, il n'existait pas d'API basée sur des handles qui permettait de coder facilement un FSAL, la seule solution qui existait alors est la FSAL_POSIX avec ses limitations inhérentes.

Les évolutions technologiques dans le domaine des systèmes de fichiers vont changer cette situation et elles seront exploitées par NFS-Ganesha.

La FSAL_XFS

Le premier changement vient de eXtended File System (XFS) [94] un système de fichiers local développé par la société SGI. Parmi ses multiples fonctionnalités innovantes, on trouve la *libhandle*. Cette bibliothèque de fonctions expose en user land les fonctions de manipulations normalement réservées au kernel space, mais limitées à l'emploi sur un système de fichiers formaté en XFS. La libhandle expose seulement deux appels :

1. **name_to_handle_at()** qui permet, pour peu que l'on dispose d'un descripteur de fichier ouvert sur un répertoire [11], de traduite un couple (des-

[11] Obtenu avec les appels *open()* ou *openat()*, ces appels peuvent agir sur un répertoire s'ils sont utilisés avec le drapeau O_DIRECTORY

cripteur de répertoire, nom d'une entrée dans le répertoire) en handle XFS de l'entrée en question ;

2. **open_by_handle_at()** qui permet d'ouvrir une entrée de l'espace de nommage, telle qu'un fichier ou un répertoire, pour peu que l'on connaisse son handle.

Ces deux appels fournissent tout ce qui est nécessaire pour construire une logique d'adressage direct par handle et d'adressage indirect par "parent et nom", et donc d'implémenter une fonction qui implémente l'action de *lookup* telle que cela est décrit dans la section 1.5.1.

Les appels de la libC permettent d'implémenter les autres fonctions de la FSAL, via deux familles de fonctions standards :

- les fonctions préfixées par un "f", notées **fonctions f***, comme fchown(), fchmod() : elles permettent d'opérer sur les métadonnées d'une entrée pour peu que l'on dispose d'un descripteur de fichier ouvert sur cette entrée [12] ;
- les fonctions suffixées par "at", notées **fonctions *at** comme readlinkat() ou fstatat() : elles permettent d'agir sur une entrée dont on connait le nom et le file descriptor du répertoire parent.

J'ai réalisé cette FSAL_XFS lors du premier semestre 2010, convaincu de l'intérêt de cet API handle. Cette FSAL sera la "mère" de bien d'autres FSALs codées plus tard et construites sur un schéma similaire, en particulier FSAL_VFS, FSAL_GPFS et FSAL_LUSTRE.

La FSAL_LUSTRE

Le système de fichiers parallèle et distribué Lustre, présenté dans la section 2.4 sera le second à bénéficier d'un adressage par handle. Elle dérive fortement de la FSAL_XFS et sera codée par mes soins durant le second semestre de l'année 2010.

Lustre identifie ses entrées de façon unique par un identifiant nommé *fid*. Ces fids jouent le rôle des handles dans le formalisme de la FSAL, surtout si l'on s'appuie sur le pseudo répertoire `.lustre/fid` qui permet de faire une correspondance rapide entre fids et entrées dans le système de fichiers.

A partir de là, la mécanique établie pour FSAL_XFS peut être transposée.

La FSAL_VFS

Cette FSAL a été développée durant le premier semestre 2011 et est l'aboutissement de la démarche lancée par FSAL_XFS. Elle est également le fruit de la collaboration très active au sein de la communauté opensource de NFS-Ganesha, en particulier la synergie entre l'équipe de développement originelle

12. Il est à nouveau rappelé que POSIX autorise d'avoir des "file descriptors" sur autre chose que de simples fichiers

et les équipes de R&D d'IBM Research. A ce moment-là, IBM est déjà très engagé dans NFS-Ganesha qu'il utilise comme passerelle NFS vers son produit GPFS comme décrit plus haut dans la section 2.8.3.

IBM voulait à l'époque proposer d'implémenter les appels `name_to_handle_at()` et `open_by_handle_at()`, qui existent dans XFS, dans tous les systèmes de fichiers supportés par Linux, via la couche VFS, et d'inclure ces appels dans la LibC.

Pour justifier une telle action, il est nécessaire de disposer d'un cas d'usage auquel adosser ce développement. La possibilité de porter le code de FSAL_XFS pour en faire une FSAL générique capable d'exporter via NFS-Ganesha, sera ce cas d'usage. Le fait que NFS-Ganesha supporte le protocole NFSv4.0 et bientôt NFSv4.1[13], donc pNFS, sera l'élément qui incitera IBM à investir dans ce développement. Les deux appels seront rajoutés dans l'ensemble des couches du noyau Linux concerné, incluant la VFS mais aussi chacun des systèmes de fichiers gérés par la VFS, et ils seront ajoutés à la LibC.

Une fois ce développement réalisé, il suffit de changer moins d'une centaine de lignes de code C pour transformer FSAL_XFS en FSAL_VFS.

L'existence de cette FSAL, et la possibilité pour NFS-Ganesha de gérer depuis le *user land* l'ensemble des fichiers du monde Linux, va avoir un effet majeur sur la popularité du produit qui va alors connaître un réel essor dans le monde opensource. Il a aussi comme conséquence directe de rendre caduque la FSAL_POSIX développée durant la période terascale.

Les autres FSALs

En marge de la FSAL_XFS et de sa "fille", la FSAL_VFS, d'autres FSALs sont développées pour supporter des systèmes de fichiers ou des espaces de nommage non gérés par la VFS.

La FSAL_GPFS

Cette FSAL est le fait de IBM. Elle permet de monter via NFS les données contenues dans un système de fichiers GPFS.

La FSAL_GPFS dérive, elle aussi, de la FSAL_XFS mais son code diverge rapidement pour coller au plus près des fonctionnalités spécifiques de GPFS. Le mariage en NFS-Ganesha et GPFS est particulièrement fructueux, car de nombreux composants de GPFS résident également en user land. LA FSAL_GPFS sera l'une des premières à implémenter le parallélisme dans NFS-Ganesha via NFSv4.1/pNFS.

Le développement de FSAL_GPFS a été réalisée par IBM, avec mon support ponctuel pour certains points spécifiques du formalisme de la FSAL.

---

13. A cette époque, la RFC5661, qui définit NFSv4.1 a moins d'un an, son support dans NFS-Ganesha est en cours de réalisation

La FSAL_ZFS

Zettabytes File System (ZFS) est un système de fichier novateur apparu à l'aube des années 2000 [83]. Bien qu'il s'agisse d'un système de fichiers local, il affiche d'emblée sa volonté de gérer de très grands volumes de données et un très grand nombre d'entrées. ZFS est disponible en opensource, mais sous les termes d'une licence inhabituelle, la licence CDDL [14].

ZFS est architecturé autour des *B-trees* [12], des arbres binaires équilibrés introduits par Rudolf Bayer en 1971 (il introduira les arbres bicolores [13] l'année suivante). Ce système de fichiers regroupe ses supports physiques dans des structures nommées *tanks* dotés de fonctionnalités de Redundant Array of Independant Disks (RAID), et de miroirs.

ZFS dispose d'une bibliothèque de fonctions totalement user land qui permet d'exposer un "tank ZFS", tant au niveau des données que des métadonnées.

Dans le cadre d'un stage de fin d'école d'ingénieur, que j'ai encadré à l'été 2011, les appels de cette bibliothèques de fonctions sont utilisés pour construire une nouvelle FSAL. Pendant longtemps, cette FSAL sera celle qui présentera les meilleures performances (débits et opérations de métadonnées).

La FSAL_PROXY

Cette FSAL représente une nouvelle fonctionnalité de NFS-Ganesha, elle a été initiée sous la forme d'un stage de fin d'école d'ingénieur en 2006, sous mon encadrement, et offre la possibilité d'exploiter NFS-Ganesha comme un serveur NFS proxy À ce jour, NFS-Ganesha est le seul serveur NFS à avoir ce type de fonctionnalité, et elle sera l'un des fers de lance qui rendra NFS-Ganesha populaire dans la communauté NFS. Elle sera développée durant le second semestre 2009.

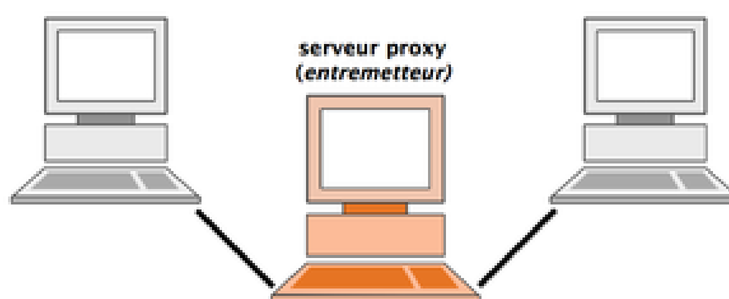


Figure 2.6 – Schéma générique d'un serveur proxy / serveur entremetteur

Le formalisme de l'API de la FSAL est très proche de celui des protocoles NFS, quelle que soit la version considérée. On transformera NFS-Ganesha en

14. Voir https ://opensource.org/licenses/cddl1.php

serveur NFS proxy de la manière suivante : la FSAL se positionne comme un client NFSv4 vers le serveur dont NFS-Ganesha doit être le proxy. Les différents appels de l'API FSAL sont encodés comme des requêtes composées, sous formes de tableaux d'opérations NFSv4. La souplesse extrême du protocole NFSv4 montre ici toute son efficacité. Par exemple, la fonction *FSAL_create* utilisera l'enchaînement d'opérations suivantes :

```
PUTFH(fh_parent)        -- positionner le CurrentFH
                             sur le parent
OPEN_CREATE(newname)    -- creer "newname" et
                             l'ouvrir
GETFH                   -- obtenir le fhandle de
                             newname dans CurrentFH
GETATTR                 -- obtenir les attributs
```

Cette implémentation comprend bien sûr des fonctions de conversion qui permettent de gérer les structures de la FSAL et leurs équivalents dans le protocole NFSv4 (par exemple les attributs des entrées du système de fichiers).

La gestion du FSAL Handle est ici simplifiée : il s'agira ici ni plus ni moins que du handle de l'entrée correspondante, mais fourni par le serveur NFSv4 dont NFS-Ganesha est le proxy. La conversion d'un FSAL Handle vers le fhandle NFSv4 fourni cette fois-ci par NFS-Ganesha est ici triviale.

2.8.4 . Refonte en profondeur des caches et de l'API FSAL

La version terascale de NFS-Ganesha disposait d'un faible nombre de FSAL différentes. On avait alors fait le choix technique de compiler la FSAL avec le binaire du daemon, ce qui spécialisait fortement celui-ci, le rendant incapable de gérer une autre FSAL, donc un autre back-end que celui qui aura été décidé lors de la compilation du daemon.

La période petascale voit le nombre de FSAL disponible exploser, sans compter les implémentations propriétaires qui ne sont pas visibles de la communauté opensource, et le besoin de pouvoir gérer plusieurs FSAL en même temps se fait sentir. La société Panasas propose alors à la communauté de développeurs NFS-Ganesha une solution très efficace, basée sur la fonctionnalité offerte par la libC de charger des librairies lors de l'exécution d'un programme [15].

Cette refonte sera faite à l'initiative de la société Panasas, et elle provoque un réel engouement dans la communauté NFS-Ganesha. L'ensemble des développeurs de NFS-Ganesha, moi-inclus, s'engagera donc dans ca grand chantier qui touche toutes les FSALs mais aussi les structures de cache.

Les FSALs sont des bibliothèques de fonctions dynamiques autonomes [16]

15. Voir en particulier les *manpages* de la fonction *dlopen()*

16. Sur des systèmes Linux, ces fichiers ont traditionnellement les extensions *.so ou *.so.*

qui sont chargées au démarrage du daemon. Lors de son initialisation, celui-ci vient lire son fichier de configuration qui contient la liste des systèmes de fichiers à exporter et la FSAL à utiliser pour accéder au back-end correspondant. Il réalise ensuite l'initialisation des ressources nécessaires au fonctionnement des FSALs après avoir chargé "à la volée" la bibliothèque de fonctions dynamique qui correspond. Pour faciliter le traitement et bien isoler les différents appels des différentes FSALs présentes au sein d'une même instance du daemon NFS-Ganesha, un formalisme orienté objet est utilisé.

Une API plus orientée objet

La nouvelle API FSAL est architecturée autour de trois concepts, similaires à des classes dans la programmation orientée objet. NFS-Ganesha et ses FSALs sont écrits en C [82], la stratégie d'implémentation ne passera pas par C++ mais sera assez voisine de celle utilisée dans certaines couches du noyau Linux ou dans l'implémentation classique de ONC/RPC.

Les concepts de cette API sont regroupés autour de trois "classes" :

1. une première classe qui représente la FSAL concernée ;
2. une deuxième classe qui représente un export réalisé par le serveur en exploitant cette FSAL ;
3. une troisième classe qui représente un FSAL handle, représentant une entrée dans la FSAL et le back-end associé à celle-ci.

Les méthodes attachées à une des trois classes sont des fonctions dont le premier argument est un pointeur vers une instance de cette classe. Les fonctions concernées seront regroupées dans un tableau rattaché à la classe "bibliothèque FSAL" et dont chaque entrée est un pointeur vers la fonction concernée. Chaque objet est une structure en C dont l'un des champs sera un pointeur vers le tableau contenant les méthodes opérant sur lui.

L'exemple ci-après, basé sur la fonction `fsal_lookup()` illustre le nouveau mode de fonctionnement. La structure FSAL handle est la suivante :

```
struct fsal_obj_handle {
  (...)
  struct fsal_filesystem *fs;   /*< Owning filesystem */
  struct fsal_module *fsal;     /*< Link back to
                                    fsal module */
  struct fsal_obj_ops *obj_ops; /*< Operations vector */
  object_file_type_t type;      /*< Object file type */
  fsal_fsid_t fsid;             /*< Filesystem on which
                                    the object is stored */
  uint64_t fileid;              /*< Unique identifier
                                   (e.g. inode number) */
  struct state_hdl *state_hdl;  /*< State of the handle */
  (...)
```

```
};
```

La fonction `lookup()` sera déclarée comme suit

```
fsal_status_t (*lookup)(struct fsal_obj_handle *dir_hdl,
                        const char *path,
                        struct fsal_obj_handle **handle,
                        struct fsal_attrlist *attrs_out);
```

La fonction fsal_lookup() est ensuite codée de cette manière [17]

```
fsal_status_t fsal_lookup(struct fsal_obj_handle *parent,
                          const char *name,
                          struct fsal_obj_handle **obj,
                          struct fsal_attrlist *attrs_out)
{
    (...)
    return parent->obj_ops->lookup(parent,
                                   name,
                                   obj,
                                   attrs_out);
}
```

Cette manière de structurer le code permet d'utiliser l'appel *fsal_lookup()* tout en dissimulant aux autres couches la multiplicité des FSALs. Ces couches peuvent manipuler des FSAL handles qui appartiennent à différentes FSALs sans difficultés.

Ce formalisme sera d'abord mis en œuvre sur FSAL_VFS, identifiée comme la FSAL la plus généraliste. La communauté NFS-Ganesha dans son ensemble joindra alors ses forces pour porter toutes les FSALs existantes pour les rendre conformes à cette nouvelle API. Cet effort prendra plusieurs mois entre les années 2012 et 2013.

Les FSALs empilables

Dans certaines situations de production, il peut être nécessaire de brider certains appels de la FSALs et de faire échouer des appels tout à fait licites. Par exemple, on peut interdire les opérations de lecture et d'écriture, mais autoriser tout le reste.

Afin de ne pas avoir à recopier le code d'une FSAL pour en faire une "FSAL modifiée" conforme au comportement souhaité, j'ai introduit dans NFS-Ganesha le concept de "FSAL empilable" (*stackable FSAL*).

Une FSAL empilable construit ses appels en utilisant les appels d'une autre FSAL. Dans la logique de chargement dynamique des FSALs au démarrage

17. des lignes sont supprimées, elles réalisent essentiellement des mesures de "programmation défensive" en s'assurant que les arguments ont les bons types, et contiennent des valeurs cohérentes

du service NFS-Ganesha, cette FSAL viendra charger une autre FSAL dont elle exploitera les services. Si l'on reprend l'exemple du début du paragraphe, une FSAL *"no read and write"* ne codera que les appels de lecture et d'écriture (pour les limiter) et passera la main à la FSAL sous-jacente dans tous les autres cas.

Il faut noter que ce formalisme n'interdit pas à une FSAL empilable de s'associer à une autre FSAL empilable.

Le concept de FSAL empilable sera exploitée par la suite pour construire la FSAL_MDCACHE.

La FSAL_MDCACHE et l'évolution des couches de cache

La FSAL_MDCACHE est développée par Red Hat dans le premier semestre de l'année 2016. Les versions précédentes de NFS-Ganesha utilisent une couche de cache de métadonnées (voir la section 1.5.3) qui est très explicite avec des appels qui sont bien identifiés et qui utilisent les structures et les types de la FSAL.

Cette nouvelle FSAL est une FSAL empilable, conforme au formalisme que j'ai introduit, et qui reprend les mécanismes de la couche de cache de métadonnées mais lui donne explicitement une API de FSAL. Il s'agit d'une FSAL d'un genre particulier, car elle est implicitement rajoutée au-dessus de chaque FSAL utilisée par NFS-Ganesha, sans déclaration explicite dans le fichier de configuration.

La couche de cache de données devient caduque : elle exploite un cache persistent sur disque local dont les performances sont généralement du même ordre de grandeur que les espaces de nommage exposés par les FSALs et l'utilisation centralisée de systèmes de fichiers Lustre (voir section 2.3.1). Le cache de données disparaît donc purement et simplement de la version petascale de NFS-Ganesha.

2.8.5 . Support des états de fichiers et apparition de la couche SAL

La naissance d'une communauté opensource autour de NFS-Ganesha et l'activité de celle-ci conduisent à une explosion du nombre de FSALs disponibles comme décrit dans les sections 2.8.3 et 2.8.3.

Plusieurs back-ends, en particulier les systèmes de fichiers gérés dans le noyau par la VFS, supportent les verrous sur les fichiers. La popularité de FSAL_VFS et les facilités offertes par NFSv4 pour gérer ces verrous au niveau du protocole amènent à considérer la gestion des états correspondants aux verrous, et par conséquent les états qui correspondent à des fichiers ouverts, dans le serveur NFS-Ganesha.

Une nouvelle couche logicielle dans le produit, baptisé *State Abstraction Layer* et désignée par l'acronyme State Abstraction Layer (SAL) dans le reste du document. L'architecture de ce nouveau composant et son interaction avec les couches logicielles déjà existantes sont discutés au sein de la commu-

nauté, notamment pendant des réunions présentielles durant les Connectathon et Bake-A-Thon, puis IBM se chargera de l'implémenter durant le premier semestre de l'année 2012.

Support du protocole auxiliaire NLM de NFSv3

La section B.2.5 a décrit le protocole auxiliaire de NFSv3 destiné à la gestion des verrous. La version terascale de NFS-Ganesha ne supportait pas ce protocole. L'apparition de la SAL fournit tous les outils nécessaires à sa mise en œuvre. IBM fera le nécessaire pour intégrer le support de NLMv4 dans le code, tout en garantissant que les verrous posés au travers des protocoles NFSv4 et NLMv4 interagissent correctement ensemble.

2.8.6 . Utilisation de nouveaux outils logiciels

Deux outils font leur entrée dans NFS-Ganesha durant cette période. On notera en particulier les apports liés au logiciel CMake et l'exploitation de nouveaux gestionnaires de mémoire.

Remplacement de la suite autotools par CMake Durant cette période, on observe un intérêt de l'industrie très fort pour NFS-Ganesha. L'utilisation qui est alors faite de la suite autotools pour gérer la compilation du produit apparaît alors comme un frein. Le projet gagne beaucoup en taille et la gestion des fichiers qui pilotent cette gestion de la compilation [18] devient peu à peu un obstacle, voire un point potentiellement bloquant à brève échéance.

La société Panasas propose d'utiliser le produit CMake [64] pour remplacer la suite autotools, une solution qui a prouvé son efficacité pour gérer les compilations de projets de grandes tailles, en particulier K Desktop Environment (KDE), un environnement graphique bien connu des habitués des distributions Linux.

Panasas réalisera le support de CMake en 2012. CMake tiendra vite ses promesses et il sera rapidement adopté par toute la communauté. Le support d'autotools sera supprimé du projet en décembre 2012.

Gestion de la mémoire

NFS-Ganesha est un logiciel qui fait la part belle aux caches, et il applique une forte pression sur la mémoire disponible sur la machine hôte. Des surconsommations de mémoire apparaissent lors de l'utilisation du serveur NFS dans des environnements de production et il n'est pas inhabituel de voir celui-ci être arrêté par l'*OOM killer*.

Les développeurs de la communauté NFS-Ganesha proposent alors d'optimiser la gestion de la mémoire avec d'autres allocateurs de mémoire que

18. En particulier les fichiers *configure.ac* et les multiples déclinaisons de *Makefile.am*

celui fourni par la LibC Des tests seront menés avec les produits TC-Malloc [44], porté par Google, et Jemalloc [34] porté par FreeBSD, Mozilla puis Facebook.

Les résultats montreront que Jemalloc produit les résultats les plus efficaces sans impact notable sur les performances. En particulier, Jemalloc va permettre de diviser la consommation mémoire par un facteur vingt. Jemalloc deviendra l'allocateur de mémoire dynamique de NFS-Ganesha en 2013.

2.9 .Les IO-Proxies

Les IO-Proxies, ou "délégateurs d'entrées-sorties" sont des outils développés dans le cadre de la R&D conduite avec l'entreprise ATOS, dans le cadre de la convention décrite à la section 2.5.

Dans le cadre de cette collaboration, j'ai défini les concepts, l'architecture du produit, fait le choix des protocoles de bases puis défini les protocoles dérivés de la solution initiale. J'ai implémenté et conçu la totalité de la partie serveur, la partie client pour les nouveaux protocoles. Le code correspondant dans le noyau Linux a été réalisé par ATOS dans le cadre de la convention de R&D conjointe avec le CEA.

Cette section présente les travaux menés autour de cette thématique, lesquels exploitent les fonctionnalités de NFS-Ganesha, lequel se voit étendu avec le support de nouveaux protocoles et qui est dès lors décliné dans un contexte un peu différent de l'exploitation pure de NFS.

2.9.1 . Contexte et problématique

Les IO-Proxies s'inscrivent dans le contexte défini par les architectures des supercalculateurs exploités dans les centres de calcul. Différentes problématiques apparaissent, liées aux nouvelles architectures.

Les architectures « manycores » et les processeurs MIC

On notera en particulier l'émergence des processeurs manycores (ou processeurs MIC) , tels que les architectures KNC ou KNL (Knights Landing. Ces processeurs disposent d'un très grand nombre de cœurs de calcul, mais de performances individuellement modestes. Si la puissance en flops explose avec ce type d'approche, de nombreuses problématiques apparaissent. Les cœurs de calcul sont en effet dédiés aux applications, le système d'exploitation doit fonctionner sur des processeurs peu rapides, en faible nombre et avec une empreinte mémoire réduite à la portion congrue car il faut réserver autant de mémoire que possible aux applications de simulation.

Vis-à-vis des IO, ce constat soulève différents problèmes. Les outils et protocoles s'appuient sur des mécanismes qui supposent la conservation d'informations d'état sur les clients et sur les serveurs afin de garantir une parfaite

cohérence des données et de bonnes performances. Ainsi, le protocole NFS fait un usage intensif de caches sur les clients, et un système de fichiers parallèle et hautes performances comme Lustre doit son efficacité aux algorithmes à états, comme le Distributed Lock Manager, par exemple, qu'il utilise.

Sur une plateforme aussi restreinte, en termes de ressources disponibles pour le fonctionnement du système d'exploitation, que ce que laissent présager les processeurs MICs, de tels produits n'ont plus leur place.

Explosion du nombre de clients

Même si elle s'appuie sur des processeurs MIC, donnant aux futurs nœuds de calcul exascale de faux airs de machines MPP, une puissance de classe exaflopique ne sera pas atteinte sans un très grand nombre de nœuds, avec des IO cohérentes. Dans le contexte de machines, où l'empreinte système est réduite, il sera compliqué de conserver des caches et des états.

Nouvelles couches réseaux

Les évolutions des technologies réseaux, que ce soit l'Infiniband, le Bull eXascale Interconnect (BXI) de Bull ou le futur OmniPath (OPA) d'Intel, les interconnexions rapides entre machines, se basent sur des paradigmes de type Remote Direct Memory Access (RDMA) dans lesquels la carte réseau accède directement à la mémoire de son hôte. Les protocoles réseaux sont simples et très performants, avec un faible nombre de couches logicielles. Les systèmes de stockage doivent s'affranchir de TCP/IP, jugé trop couteux à l'époque.

2.9.2 . Le concept d'IO Proxy

L'option retenue par le DoE (Department of Energy) repose sur l'utilisation de *Burst Buffers* : une couche intermédiaire constituée de disques Solid State Device (SSD) ou de Non Volatile Random Access Memory (NVRAM), peu capacitifs (au regard des volumes manipulés) mais très rapides, est interposée entre les nœuds de calcul et le stockage. Cela nécessite l'utilisation d'une couche logicielle dédiée (telle que la couche DAOS), avec un impact très intrusif au niveau du code de calcul. Cette démarche a principalement vocation à gérer les fichiers produits par le code au fur et à mesure de son avancement, notamment lorsque des fichiers de `checkpoint/restart` sont produits.

L'approche conduite dans le cadre de la R&D avec ATOS, ainsi que les études menées en vue de concevoir une architecture exascale est différente et bien moins intrusive. Elle repose sur la délégation d'IO au travers d'agents dédiés à cet usage et nommés IO Proxies. J'ai dirigé cette étude dans le cadre de cette convention de R&D, dans le cadre de mes activités de pilotage de la thématique "stockage de données HPC" de cette convention. J'ai opéré la plupart des choix technologiques déterminants, mais j'ai disposé du support de l'entreprise ATOS pour réaliser le code et l'intégration dans la pile logicielle,

en particulier au niveau de la programmation dans le noyau.

IO Proxy : un niveau d'inférence supplémentaire
Le principe de l'IO Proxy repose sur une idée simple qui relève de différents constats :

1. un code de calcul n'a pas besoin, pour fonctionner, de voir toutes les données stockées sur l'architecture de stockage, ni même d'avoir accès à toutes les données que peut voir l'utilisateur qui lance ce code de calcul ;
2. il est généralement rarissime qu'un code de calcul occupe l'ensemble du supercalculateur, une machine exaflopique tournera pendant plus de 99% de sa vie une majorité de jobs qui nécessitent moins de 10% de sa puissance maximale ;
3. plusieurs exécutions de codes de simulation numérique sont corrélées par le biais des données qu'elles utilisent ou des résultats qu'elles créent. Par exemple, on observe des études paramétriques qui utilisent des jeux de données en entrée quasiment identiques à un paramètre près dont on étudie l'impact de la variation. Par ailleurs, il est courant de voir des « chaînes de calcul » au cours desquelles plusieurs codes de simulation s'exécutent les uns derrière les autres, le suivant utilisant dans ses entrées les sorties du programme précédent.

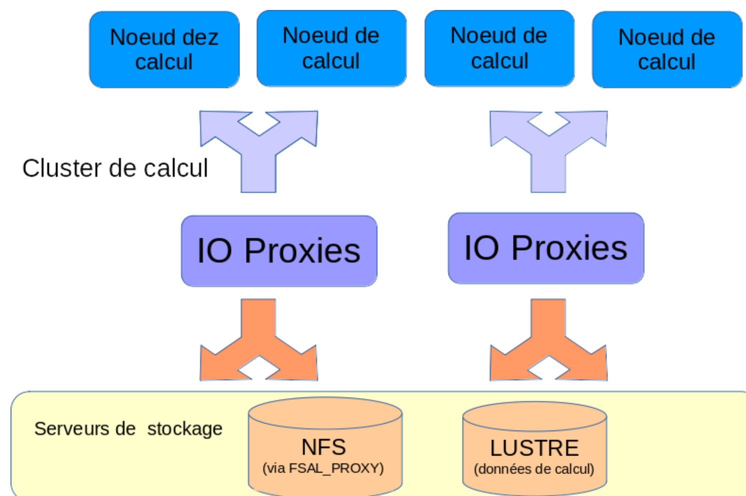


Figure 2.7 – Les IO-Proxies entre les calculateurs et les systèmes de stockage

Au sein du centre de calcul, les IO-Proxies viennent réaliser un niveau d'inférence supplémentaire, entre les nœuds de calcul et les serveurs de stockage (voir figure 2.7).

Comme tout proxy, ou serveur mandataire, les IO-Proxies sont à la fois des clients et des serveurs.

Un IO Proxy se base sur différentes technologies, toutes choisies en fonction des contraintes décrites ci-dessus.

Choix du protocole RDMA

La couche de transport choisie pour les IO-Proxies est RDMA, elle s'utilise en lieu et place de TCP/IP. Elle repose sur les couches *ibverbs* et *rdmacm* pour InfiniBand et sur l'API native de BXI. .

Choix du protocole 9p

Un protocole de système de fichiers distant doit être choisi. Ce choix se porte sur le protocole 9p. Ce dernier dispose de différents avantages. Pour commencer, il fournit la totalité des appels indispensables pour exposer une interface totalement POSIX à l'utilisateur (dont les verrous de fichiers et les attributs étendus) tout en ayant une très faible empreinte mémoire et CPU.

Le protocole est fortement *stateful* : le serveur conserve trace des états par "activité", en déchargeant le client, au contraire de NFS. Le protocole 9p est simple, il utilise des structures élémentaires (tableaux d'octets, nombres sur 16 bits, sur 32 bits, sur 64 bits), ce qui permet de gérer rapidement l'encodage et le décodage des messages en mémoire.

Utilisé en association avec un protocole comme RDMA, il devient très simple de mettre en œuvre des mécanismes de *zero copy*, avec de nombreuses optimisations afférentes. Le protocole 9p est décrit en détail dans l'annexe F.

Couplage de l'IO Proxy avec le job sur le calculateur

Un IO Proxy vient donner à un utilisateur la vision des données dont il a besoin pour effectuer une simulation numérique. Par opposition à l'approche américaine des **Burst Buffers** qui reposent sur une bibliothèque de fonctions et une API spécifique, l'IO Proxy expose les informations au travers d'un point de montage et une arborescence POSIX, très classique, qui suppose peu de changements dans les codes de simulation.

Un IO Proxy est associé à un job lancé en machine, ou un ensemble de jobs lancés en machine et partageant les mêmes données. Par conséquent, l'IO Proxy est lancé par le gestionnaire de jobs, ce qui fait des jobs de simulations numériques des codes fondamentalement Multiple Programs Muliple Data (MPMD). Cette allocation fait des IO des ressources à allouer de la même manière que les ressources mémoires ou CPU.

Les études pour dimensionner la bande passante via le nombre de threads attribués à un IO Proxy ont été réalisées par Bull/Atos et le CEA dans le cadre de la R&D. Il apparaît que le facteur à prendre en compte est le nombre de threads processeurs accordés à un IO Proxy. Ces derniers tournant sur des

machines « passerelles » dédiées, capables de voir à la fois le réseau haute performance dédié aux nœuds de calcul et celui dédié au stockage, l'allocation des ressources CPU de ces équipements devient représentatif de la bande passante désirée.

2.10 .Implémenter des IO Proxies en modifiant NFS-Ganesha

Dans cette section, on décrira les opérations à mener dans le logiciel NFS-Ganesha pour en faire un IO-Proxy. J'ai réalisé la majorité des intégrations dans le code de celui-ci.

On décrira le protocole 9p puis son intégration avec RDMA et l'utilisation conjointe de ces deux protocoles dans NFS-Ganesha pour en faire un IO-Proxy.

2.10.1 . Le protocole 9p

Le protocole 9p (parfois appelé *Styx* ou *Plan 9 Filesystem Protocol* est un système de fichiers distribués développé dans le cadre du projet Plan 9.

9p et le système d'exploitation Plan 9

Plan 9 est un système d'exploitation développé par Bell Labs [51] [45][45][76][75] comme une alternative aux systèmes d'exploitation Unix, à la fin des années 80 et le début des années 90.

Si Plan 9 n'a jamais été réellement exploité en production et a fini par disparaître, le protocole 9p lui a survécu. Il est en particulier utilisé pour exporter, dans une machine virtuelle, des systèmes de fichiers de l'hyperviseur, via la fonctionnalité `virtio`, implémentée grâce à 9p [19].

9p a été revue avec la quatrième édition de Plan 9, sous le nom 9p2000, lui donnant des fonctionnalités additionnelles à même d'implémenter toutes les fonctionnalités attendues dans un système de fichiers distribué. Le protocole 9p2000.L est décrit dans l'Annexe F.

2.10.2 . Utilisation de RDMA

La mise en œuvre de RDMA passe par la bibliothèque Mooshika qui vient faciliter l'emploi des bibliothèques systèmes complexes qui adressent ces couches réseau (*libibverbs* et *librdmacm*).

Cette bibliothèque est développée dans le cadre d'un stage de fin d'études d'école d'ingénieurs que j'ai encadré en 2009.

19. Voir la page https ://www.linux-kvm.org/page/9p_virtio

La bibliothèque de fonctions Mooshika

L'utilisation du protocole RDMA est incontournable dans l'optique du HPC. En effet, les réseaux haute performance employés s'appuient sur cette technologie, de plus elle offre la possibilité d'accéder directement aux ressources matérielles de la carte réseau et donc d'éviter de traverser une multitude de couches logicielles coûteuses en termes de mémoire et de CPU, donc impactantes sur les performances (cf paragraphes précédents).

En quelques mots, RDMA s'appuie sur des ressources réseaux capables de faire du Direct Memory Access (DMA), un paradigme dans lequel une carte (branchée dans un slot PCI) peut accéder directement à la mémoire de l'hôte. Si cette carte est une carte réseau, c'est le protocole réseau qui offre la visibilité d'une partie de la mémoire des machines.

On a ainsi un nouveau modèle de communication dans lequel des machines rendent visible une fenêtre mémoire dédiée à la communication avec les autres machines. Cette approche est particulièrement efficace, car la zone mémoire concernée peut être celle dans laquelle les données ont été produites ou celle depuis laquelle les données seront lues, il n'est donc pas nécessaire de les déplacer, le protocole RDMA les apporte tout de suite in situ.

Le modèle RDMA a été utilisé dans les réseaux Infiniband, même si d'autres technologies l'utilisent (comme BXI, RoCE, iWarp), et son API dérive des bibliothèques `ibverbs` et `rdmacm`, très ancrées dans le monde IB. Ces dernières sont devenues des standards de-facto, implémentées sur tous les hardwares s'appuyant sur RDMA, mais elles sont très complexes à mettre en œuvre.

La bibliothèque de fonctions Mooshika offre une abstraction de plus haut niveau, permettant l'usage de ces bibliothèques de façon plus simple et plus intuitive pour le développeur, tout en restant compatible avec toute implémentation de RDMA (en particulier celle qui figure dans le kernel). Libmooshika a été utilisée pour implémenter le support de la couche de transport RDMA pour le protocole 9p dans Ganesha, rendant ainsi possible l'implémentation d'un délégateur d'IO 9p/RDMA.

2.10.3 . Gestion du BXI avec Pasha

La technologie BXI[9][32] est portée par ATOS afin de permettre de développer un réseau d'interconnexion de supercalculateur. BXI s'appuie sur la technologie *Portals 4* [11] souvent désigné sous l'acronyme `ptl4`.

Portals 4 est une API qui permet d'implémenter du *Message Passing* ou échange de messages de très bas niveau. En résumant les choses à l'extrême, on peut voir BXI comme une implémentation matérielle de Portals 4.

La technologie BXI est en particulier mise en service dans les calculateurs Tera100 et Tera1000 du CEA/DAM.

Dans le cadre de la R&D conduite conjointement entre le CEA et ATOS, une expérience est conduite autour de BXI : il s'agit de montrer qu'il est possible

de faire fonctionner les IO-Proxies en exploitant la technologie BXI en lieu et place de RDMA. L'intégration envisagée est assez simple : on utilisera l'API BXI, qui est une variation de l'API Portals 4, en lieu et place de l'API Mooshika décrite dans la section 2.10.2.

Du côté du serveur, j'ai réalisé l'implémentation en introduisant de nouveaux threads destinés à écouter les messages véhiculés sur les interfaces BXI. La partie cliente est prise en charge par ATOS qui va développer un petit projet nommé *Pasha*. Celui-ci vient modifier le client 9p/RDMA pour remplacer les appels à la couche RDMA par des appels à la couche Portals 4, déjà ajoutée par ATOS dans le cadre de son implémentation physique de BXI.

Le projet sera un succès et il fera la démonstration de la faisabilité de l'implémentation de 9p, donc des IO-Proxies, sur BXI. Les protocoles de tests comprennent les étapes suivantes :

- exportation d'un système de fichiers supporté par VFS via un IO-Proxy qui utilise la FSAL_VFS, et qui est capable d'écouter sur une interface BXI, et capable de comprendre nativement le protocole Portals 4 ;
- montage de cette exportation sur une machine client, via une interface BXI ;
- test du montage ainsi réalisé avec les tests de non-régression utilisés dans le développement de NFS-Ganesha, en particulier les tests du Connectathon.

Pasha est resté au rang d'expérimentation et de Proof of Concept (PoC), une démonstration de la généricité du concept d'IO-Proxy et de sa faculté à s'adapter à différents paradigmes réseaux puisqu'il est fonctionnel sur TCP/IP, RDMA et BXI.

2.11 .Injecter du parallélisme dans les IO Proxies

Le modèle des IO-Proxies est initialement pensé sans parallélisme : un unique serveur discute avec plusieurs clients, mais Le besoin de performances conduit naturellement à envisager d'utiliser plusieurs interfaces réseaux à la fois. Mécaniquement, on va exploiter plusieurs machines serveurs et donc induire du parallélisme.

Pour implémenter ce parallélisme, on adopte une approche très voisine de celle qui a été conduite dans la fonctionnalité pNFS du protocole NFSv4.1 décrite dans l'annexe D.6.3 du présent document. On aura donc un premier IO-Proxy qui endossera le rôle de MDS tandis que d'autres serveurs, qui ne sont pas nécessairement du même type, endosseront quant à eux le rôle de DS. Le MDS sera monté au travers du protocole 9p2000.L, les DS seront accédés au travers de requêtes émises par le client sur la base d'informations retournées par le MDS.

Cette fonctionnalité induit un schéma en tierce-partie, qui fait agir 3 types

d'acteurs (MDS, DS et clients). Or, ces acteurs communiquent entre eux, par conséquent pour injecter du parallélisme, on devra disposer des outils suivants et réaliser les adaptations suivantes :

- modifications du protocole 9p, en lui ajoutant de nouvelles opérations, afin d'indiquer aux clients à quels DS ils doivent parler, cela suppose de modifier le noyau ;
- implémenter des DSs, pour ce faire, on va créer d'un nouveau type de serveur, le Lightweight IO Provider (LIOP) ;
- les clients ne communiquent pas seulement avec le MDS, ils parlent avec les DS, il faut donc définir un protocole de communication pour agir avec les serveurs LIOP. Le client est dans le noyau, ce nouveau protocole sera ajouté dans le noyau Linux,

Le plus souvent, le duo p9p/LIOP sera utilisé pour exporter un système de fichiers parallèle sur des clients qui ne peuvent pas supporter ce type de systèmes de fichiers. Dans le cas du CEA, il s'agira de Lustre. La figure 2.8 décrit le schéma qui en découle.

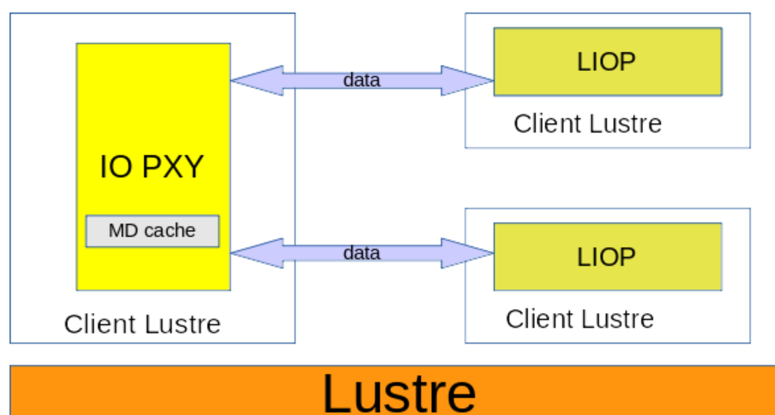


Figure 2.8 – Fonctionnement d'un IO-Proxy avec des LIOP auxiliaires

Le protocole 9p est ainsi modifié pour prendre en compte les LIOPs, on nomme ce nouveau protocole *Paralell 9p (p9p)*. Il sera par la suite désigné sous l'acronyme *10p*. D'une manière synthétique, on pourrait résumer la genèse de ce protocole avec "l'équation" suivante : `9p + pNFS = 10p`.

Cette mise en place du parallélisme aboutit donc à la création de deux protocoles, p9p, très proche de 9p2000.L et LIOP qui est créé. J'ai rédigé et défini ces deux nouveaux protocoles. Les protocoles p9p et LIOP ont besoin de couches de transport. Ils s'appuieront sur les protocoles TCP/IP et RDMA. Au niveau de l'implémentation, il faut modifier l'implémentation 9p pour en faire une implémentation de p9p, puis créer le serveur LIOP en implémentant le serveur et le client afférent.

J'ai réalisé les codes des parties serveurs. Dans le cadre de la convention

de R&D avec ATOS, c'est ATOS qui réalisera l'implémentation du code client dans le noyau.

Le protocole LIOP est décrit en détail dans l'annexe G

Les modifications faites dans 9p par p9p

Le protocole p9p n'ajoute qu'une seule fonction au protocole 9p2000.L, **HWALK**, implémentée par la paire de messages **THWALK**/**RHWALK**. Certaines structures du protocole sont également amendées.

Il s'agit ici de faire le lien entre la logique à états de 9p et le fonctionnement totalement sans état de LIOP. Dans le protocole 9p2000.L, la structure *qid* permet de représenter les entrées d'un système de fichiers, cette structure joue dans le protocole un rôle analogue à celui d'une inode dans les systèmes de fichiers traditionnels. Dans le cadre de p9p, on va ajouter à la structure *qid* un buffer qui identifie totalement le fichier par adressage direct. Ce buffer n'est fondamentalement rien d'autre qu'un fhandle NFS.

Dans 9p2000.L, le parcours dans l'arborescence d'un système de fichier exporté est réalisé par l'opération WALK, chaque étape est représentée par un fid [20]. Un fid au sens de 9p est donc un curseur placé sur le système de fichiers, il est supprimé par l'opération CLUNK. Quand une nouvelle entrée est créée, qu'il s'agisse d'un fichier (créé par LCREATE), d'un répertoire (créé par MKDIR) ou d'un lien symbolique (créé par SYMLINK), une nouvelle fid, pointant sur la nouvelle entrée, est retournée, mais l'opération retourne également une *qid* qui représente celle-ci.

Dans le cadre de p9p, on ajoutera simplement à la qid un buffer qui contiendra un fhandle. L'opération HWALK permettra simplement d'obtenir le fhandle correspondant à un fichier si cette valeur n'est pas connue du client, par exemple s'il l'a obtenu lors d'un précédent appel (et conservé dans un cache), ou s'il a lui-même créé l'entrée et obtenu la qid qui correspond.

Très concrètement, le fhandle qui est retourné par le serveur est un handle NFSv3. Le service IO-Proxy est implémenté par NFS-Ganesha qui implémente déjà NFSv3, par ailleurs la structure de fhandle est facilement construite à partir des handles de la FSAL. Dans le cas précis de Lustre, le fhandle n'est autre que la *Lustre fid* qui est recopiée dans le buffer représenté par le fhandle. Ce dernier sera récupéré par le client p9p qui s'en servira pour effectuer des opérations de lecture et d'écriture au travers du protocole LIOP.

2.12 .Mise en exploitation de NFS-Ganesha pendant la période petascale

Le serveur NFS-Ganesha a été mis en production et a été utilisé sur différentes plateformes de tests et de validation sur la période petascale au sein

20. à ne pas confondre avec les fids de Lustre,

des centres de calcul administrés par le CEA. Ces mises en œuvre diffèrent de celles qui ont été à l'origine du produit et qui ont été décrites dans le chapitre précédent. On retiendra en particulier trois utilisations particulièrement saillantes du produit.

2.12.1 . Exportation d'un système de fichiers Lustre sur architecture NEC

Au début des années 2000, les centres de calcul du CEA ont adopté une architecture centrée autour des données, dans laquelle des supercalculateurs, fonctionnant sous Linux, accèdent des systèmes de fichiers massifs gérés par le logiciel Lustre. NEC ne supporte pas Lustre et les calculateurs SX8 et SX9 ne seront jamais capables d'accéder directement ce système de fichiers. Une solution alternative sera exploitée via la FSAL_LUSTRE de NFS-Ganesha qui permet d'exporter Lustre. Les machines ne disposant pas d'implémentation de NFSv4, les données stockées dans Lustre seront rendues visibles par des montages NFSv3.

Cette exploitation s'avère difficile et complexe, à cause de la relative vétusté logicielle du système d'exploitation de NEC. Il faudra ainsi ajouter des fonctionnalités spécifiques On notera en particulier l'implémentation d'un cache négatif pour la gestion du contenu des répertoires dans la couche de cache des métadonnées de NFS-Ganesha.

2.12.2 . Administration de l'espace de nommage HPSS au travers de NFS

Dans la période Petascale, les architectures des centres de calcul adoptent une configuration dite *data centrique* autour de systèmes de fichiers Lustre, ce qui rend caduque les caches comme SHerPA (voir section 1.3).

Si les montages NFS ne sont plus exposés aux utilisateurs finaux, ils sont encore exploités par les administrateurs systèmes. En effet, si les mécanismes de *binding Lustre/hsm* (cf 2.3.3 et 2.3) dissimulent le gestionnaire de stockage hiérarchique HPSS, l'espace de nommage de celui-ci ne disparaît pas pour autant, il est toujours exploité par le *copytool*,qui vient opèrer les mouvements de données entre Lustre et HPSS.

La visibilité de l'espace de nommage de HPSS reste indispensable pour les administrateurs, via La FSAL "historique" de NFS-Ganesha, la FSAL_HPSS dont l'usage, via NFS-Ganesha et ses caches, reste plus efficace que les outils d'administration fournis avec HPSS.

Ce point de montage, basé sur NFS-Ganesha, permet d'utiliser des outils élaborés sur cet espace de nommage, en particulier l'utilisation de l'outil Robinhood[57] pour auditer cet espace de nommage.

2.13 .Expérimentation des IO-Proxies en vraie grandeur

Cette section décrit les tests conduits sur une configuration mettant en jeu des IO-Proxies et les performances mesurées. Ils mettent en œuvre NFS-Ganesha, utilisé comme IO-Proxy, la bibliothèque de communication Mooshika, basée sur le protocole RDMA et décrite dans la section 2.10.2, ainsi que le protocole p9p et son protocole auxiliaire LIOP, tout deux décrits dans la section 2.11 est issu des efforts de R&D communs menés par le CEA et ATOS. Ces tests ont été conduits par les équipes du CEA, sur du matériel dédié situé dans une plateforme de tests dédiée à l'exécution de benchmarks et à la validation de nouvelles technologies.

Configuration des machines impliquées dans le test Les tests vont mettre en œuvre différentes configurations de machines, résumées dans le tableau suivant :

Type	Nb	CPU	RAM	Rôle Lustre	Rôle Ganesha
MDS p9p	1	2 Xeon E5	128 GB	Client	serveur
DS LIOP	4	2 Xeon E5	128 GB	Client	serveur
Client p9p/ LIOP	4	2 Xeon E5	128 GB	néant	client

Toutes les machines utilisent un réseau IB et la distribution CentOS 7.4

2.13.1 . Benchmarks réalisés

Lors de ces tests, on va comparer les performances natives du système de fichier Lustre et les performances observées au travers du système de fichiers mettant en jeu p9p et LIOP. Dans la mesure où se dernier va ré-exporter les données contenues dans Lustre, en tant qu'IO-Proxy, on s'attend à mesurer des performances inférieures puisqu'une couche logicielle supplémentaire est exploitée. Le but de cette batterie de tests est de mesurer cet écart.

Les tests comprendront les actions suivantes :

Lecture/écriture mono-nœud avec dd

Ce test utilise l'outil système data dump (dd) qui est fourni en standard dans toutes les distributions Linux [21]. Ce programme lit depuis une source et écrit dans une destination un nombre de blocs de taille fixe. Si le source est le pseudo-fichier `/dev/zero`, on va mesurer les performances de la destination. Dans ce test, on écrira des blocs de 1024 kilo-octets, au nombre de 10000, soit un total de 10 gigaoctets, une taille standard pour les entrées-sorties de cette époque. On exécutera donc la commande suivante pour le test en écriture :

21. Voir la man page https ://man7.org/linux/man-pages/man1/dd.1.html

```
dd if=/dev/zero of=/root_lustre_path/subpath/dd.data \
        bs=1024K count=10000 conv=fdatasync
```

Pour le test en lecture, on prendra soin de vider les caches systèmes auparavant :

```
echo 3 > /proc/sys/vm/drop_caches
dd if=/root_lustre_path/subpath/dd.data of=/dev/null bs=1024K
```

Lecture et écriture parallèle avec ior

Dans ce texte, on utilie l'outil Interlaced Or Random (ior), un benchmark parallèle basé sur MPI [22]. L'outil ior permet de produire des accès parallèles sur un tel système de fichiers. Dans la mesure où `ior` est un programme MPI, il doit être lancé dans un environnement associé à cette bibliothèque de fonctions, ici la commande `mpirun`.

Les tests sont lancés par les commandes suivantes, sachant que $n représente le nombre de tâches MPI et que le test effectue à la fois des lectures et des écritures :

```
mpirun -N node[2115-2119,2124-2125]" -n $n -N 4 -c 1 -x \
            -m cyclic  ./ior/src/ior -w -k -v -F -t 4m  \
            -b $((128*4/$n))g                           \
            -o /root_lustre_path/subpath/data
```

2.13.2 . Résultats

On observe les résultats suivants avec les benchmarks précedemment décrits :

Accès via dd

Type I/O	Lustre	p9p/LIOP
écriture	1.5GB/s	720 MB/s
lecture	921 MB/s	958 MB/s

Lecture et écriture parallèle avec ior

Accès dans Lustre :

Nombre de Tâches	Écriture	lecture
1	1436.22 MB/s	568.43 MB/s
2	2971.33 MB/s	1065.30 MB/s
4	5930.28 MB/s	2166.25 MB/s
8	6331.16 MB/s	3704.75 MB/s
16	6336.18 MB/s	4711.55 MB/s
32	6379.35 MB/s	6107.20 MB/s

22. https://github.com/hpc/ior

Accès dans p9p/LIOP :

Nombres de Tâches	Écriture	lecture
1	709.22 MB/s	847.99 MB/s
2	1169.56 MB/s	838.71 MB/s
4	2197.16 MB/s	1732.04 MB/s
8	3939.18 MB/s	3256.78 MB/s
16	6113.47 MB/s	4006.24 MB/s
32	6105.81 MB/s	5157.38 MB/s

D'où l'on peut déduire les rendements suivants :

Nombres de Tâches	Écriture	lecture
1	49.4%	149.2%
2	39.4%	78.7%
4	37.0%	80.0%
8	62.2%	87.9%
16	96.5%	85%
32	95.7%	84.4%

Représentation graphique des résultats

Cette section synthétise les résultats obtenus sous forme de graphes.

La figure 2.9 les performances dans Lustre en lecture et en écriture, en fonction du nombre de tâches. La courbe bleue indique les performances en écriture, la courbe orange les accès en lecture.

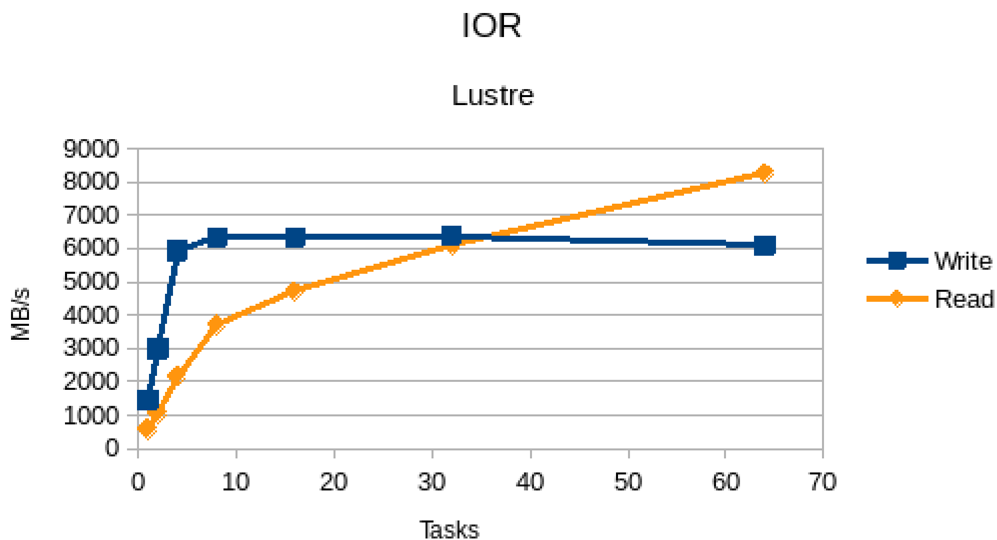


Figure 2.9 – Résultats IOR sur Lustre, débits en lecture et écriture (MB/s) en fonction du nombre de tâches

Les performances (en MB/s) dans le montage p9p/LIOP en lecture et en écriture varient ainsi, en fonction du nombre de tâches sont montrées par la

figure 2.10. La courbe bleue indique les performances en écriture, la courbe orange les accès en lecture.

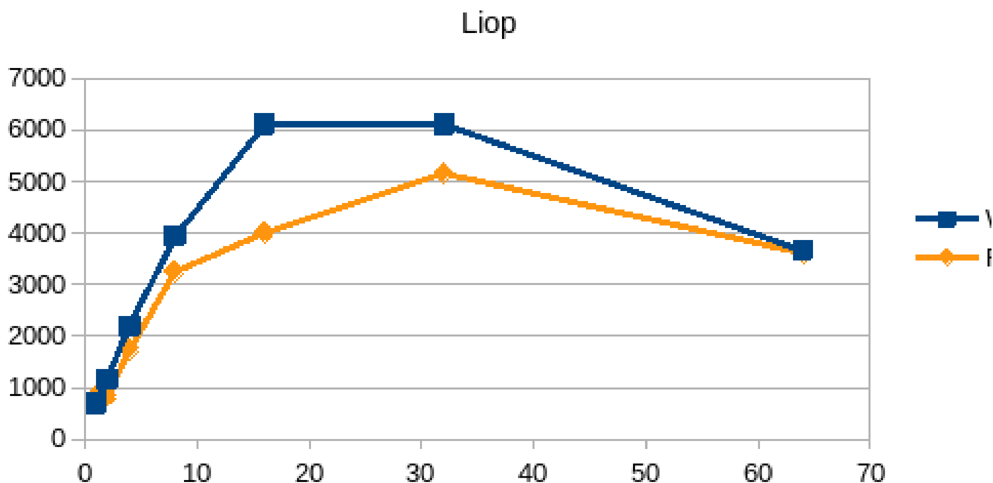


Figure 2.10 – Résultats IOR sur p9p/LIOP, débits en lecture et écriture (MB/s) en fonction du nombre de tâches

2.13.3 . Résultats et discussions

Ces résultats mettent en évidence plusieurs effets. En premier lieu, on peut voir que les IO-Proxies ont clairement un comportement de cache. les performances élevées en lecture mono-nœud, mesurées à la fois via `dd` et `ior` montrent une écriture dans les caches mémoires. En parallélisant les lectures via `ior`, on observe des performances inférieures dès lors que le nombre de tâches augmente. C'est particulièrement visible sur la seconde courbe, les performances augmentent jusqu'à 40 tâches, puis se stabilisent et commencent à baisser. Le même genre de phénomène a lieu dans les accès à Lustre mais il se produit plus rapidement, la stabilisation intervient dès 10 tâches.

On peut expliquer ce phénomène par différents constats. Pour commencer, les volumes lus et écrits augmentent avec le nombre de tâches, puisque chaque tâche écrit ou lit un volume fixe. Cet effet induit une pression plus forte sur les ressources qui perdent les bénéfices des effets de cache. D'autre part, on utilise deux processeurs quadricœurs, soit 8 cœurs par machine, donc au plus 32 cœurs pour les clients et les serveurs DS. Au-delà de 32, on aura des processeurs plus sollicités, qui doivent tourner plus d'une tâche. Cela explique partiellement le "décrochement" que l'on peut voir sur la seconde courbe, au-delà de 30 tâches.

En écriture, les rendements sont médiocres en mono-nœud, de l'ordre de 50% des performances natives de Lustre, mais ils deviennent bien meilleurs alors que le nombre de nœuds clients, et donc la parallélisation, augmentent.

Les IO-Proxies mettent en œuvre des couches supplémentaires par-dessus Lustre, avec un coût qui apparaît ici. Cependant, ce coût s'estompe avec la parallélisation, l'efficacité en écriture passe de 49,4% sur une tâche à 95.7% sur 32 tâches. Un phénomène similaire est visible en lecture (on atteint 84.4% sur 32 tâches). On rejettera l'efficacité en lecture de 149.2% en lecture sur une tâche, potentiellement due à une erreur de mesure ou une contention dans Lustre ayant conduit celui-ci à ne pas être efficace dans ce cas précis (très probablement un effet de *single shared object* qui a conduit à une contre-performance du système de fichiers parallèle).

Quand la parallélisation est forte, l'écart de rendements entre IO-Proxies et Lustre devient inférieur à 5%. Si des optimisations restent nécessaires, l'IO-Proxy demeure utilisable en environnement très parallélisé tout en apportant la couche d'inférence supplémentaire promise par cette configuration, celle-ci permettant de réduire significativement la charge sur le serveur de données.

2.14 .Bilans et Perspectives pourla période Petascale

Ce chapitre a replacé le contexte de la période Petascale. Les problématiques propres à cette époque ont été décrite, en particulier l'émergence des architectures *data-centriques* dans les centres de calcul HPC.

Le chapitre présente les évolutions du protocole NFS, via NFSv4 et NFSv4.1. Il a décrit également comment NFS-Ganesha évolue pour les supporter.

De nombreuses nouvelles FSALs voient le jour durant cette période, un fait corrélé avec l'émergence d'une communauté open-source autour de NFS-Ganesha, laquelle est toujours très active de nos jours.

Devenu le cœur d'un travail collaboratif, NFS-Ganesha connaît plusieurs refontes qui en font un produit plus industriel.

Enfin, le chapitre montre comment NFS-Ganesha est utilisé pour construire la fonctionnalité "IO-Proxy", un nouveau paradigme introduit pour répondre aux exigences de cette période. Le support du protocole 9p2000.L est introduit dans NFS-Ganesha, puis cette fonctionnalité est étendue via la création de p9p/LIOP qui lui permet de devenir parallèle, en définissant LIOP, un nouveau protocole ancillaire. Des tests de cette nouvelle fonctionnalité parallèle sont enfin présentés.

Chapitre 3 : La période Exascale et l'émergence du stockage objet

3.1 .Contexte de la période exascale

On ne peut attester que le HPC soit déjà rentré dans l'ère de l'exascale ou non, même si Les machines qui peuvent produire plus d'un exaflop, donc plus de 10^{18} flops sont techniquement envisageables. En effet, le podium actuel du Top500, avec les machines de Oak Ridge National Laboratory (ORNL), Rikagaku Kenkyusho (RIKEN) et Tieteen tietotekniikan keskus (CSC), décrit par le tableau 3.1 ne montre qu'une seule machine dont la puissance dépasse l'exaflop et elle est deux fois plus puissante que la seconde et deux fois plus que la troisième.

L'exascale est sur le point de devenir une réalité, mais ne constitue pas encore la norme, on est donc actuellement dans une période charnière que l'on pourrait désigner comme une période "pré-exascale".

Rang	Nom	Site	Performances
1	Frontier	ONRL	1.102 Eflops
2	Fugaku	RIKEN	537 Pflops
3	LUMI	CSC	214 Pflops

Table 3.1 – Performances des 3 premières machines du Top500 au début de la période pré-Exascale

Les technologies de stockage regroupent différents types de support. Les bandes sont toujours exploitées, une bande de type LTO-9 conserve 18 To sans compression et dispose d'une capacité estimée de 45 To en utilisant la compression matérielle des dérouleurs qui viennent les écrire et les lire. Le stockage sur disques fait la part belle à l'hybridation de support qui mêle des disques rotatifs, ou Hard Disk Drives (HDD), et des disques SSD basés sur les mémoires flash. L'utilisation conjointe des deux technologies permet d'avoir le meilleur des deux mondes, les HDD offrant une large capacité tandis que les SSD garantissent un débit élevé. Il n'est pas rare de voir des systèmes dont le débit est de l'ordre de grandeur de plusieurs To/s.

La technologie NVRAM apparaît, offrant des débits très élevés, de l'ordre de grandeur des bus mémoires. Le logiciel open-source Distributed Asynchronous Object Storage (DAOS)[62], porté par Intel, reste un cas d'usage très efficace des NVRAM qu'il sait exploiter pour assurer des performances très élevées, en particulier en nombre d'opérations pouvant être concurremment réalisées. NVRAM a cependant donné naissance au protocole matériel Non

Volatile Memory express (NVMe), en passe d'être supplanté par le protocole Compute eXpress Link (CXL).

Les chiffres clefs du centre de calcul du CEA

Les supercalculateurs du CEA pendant la période pré-exascale sont décrits dans le tableau 3.2. La puissance de calcul est assurée en multipliant le nombre de cœurs de calcul par processeurs. Les architectures Intel Xeon Phi, avec les processeurs KNC et KNL seront exploitées sur Tera 1000 avant d'être supplantées par les GPU portés par des constructeurs comme AMD ou NVidia.

Année	Machine	Performances	Stockage
2015	Tera 1000	30 Pflops	40 Po (767 Go/s), 2 Po (SSD, 1 To/s), 100 Po (bandes)
2022	Exa 1	100 Pflops	100 Po (HDD+SSD, 2.5 To/s), 300 Po (bandes)

Table 3.2 – Performances des machines du CEA durant la période pré-Exascale

3.2 .Les problématiques de l'exascale

À l'heure de l'exascale, de nouvelles problématiques se révèlent au niveau des systèmes de stockage de données massifs, en charge de conserver les données produites par les supercalculateurs.

Les solutions actuellement utilisées reposent sur des systèmes de fichiers distribués et parallèles. Cette approche risque de rencontrer une série de problèmes structurels et d'atteindre ses limites, la rendant de facto caduque. En particulier, différents algorithmes utilisés reposent sur des machines à états complexes dont la mise en œuvre deviendra impossible au-delà d'un volume important d'enregistrements à gérer.

Afin de construire des systèmes de stockage à même de répondre aux exigences futures, il devient indispensable de mettre en place de nouveaux paradigmes capables de s'adapter au contexte de l'exascale, tout en limitant l'impact sur l'utilisateur final. Cela passe notamment par l'utilisation d'outils basés sur l'exploitation intensive du stockage objet, l'adjonction de considérations liées à l'allocation explicite de ressources propres au stockage dans l'ordonnancement des travaux du supercalculateur (par exemple la bande passante) et l'implémentation de services éphémères, spécifiques à l'exécution de gros travaux sur la machine de calcul, en lieu et place de services globaux qui ne passent plus à l'échelle. Enfin, les évolutions des technologies de stockage de données, en particulier avec l'exploitation des mémoires non volatiles (ou

NVRAM) et l'explosion des performances des disques Flash complexifient le placement des données. La figure 3.1 illustre cette situaton.

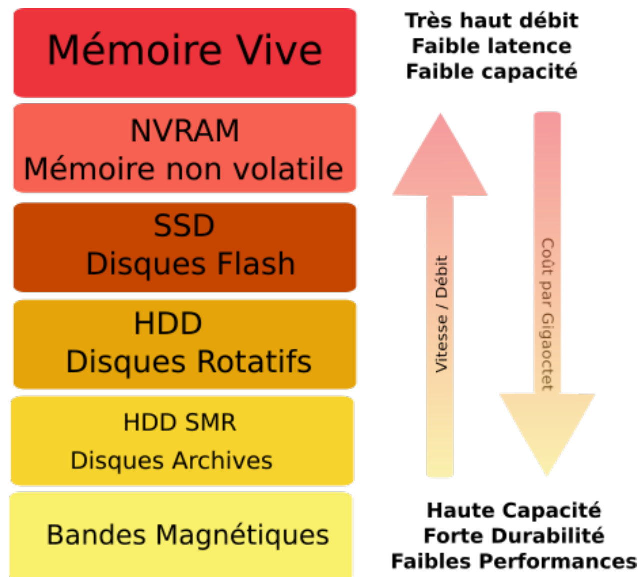


Figure 3.1 – Empilement des ressources de stockage HPC

Il est important de noter que les travaux présentés ici sont à caractère prospectif, ils couvrent en particulier des considérations liées aux architectures qui succéderont à Tera 1000 et Exa 1 (les actuels supercalculateurs en service au CEA–DAM). Cependant, ces évolutions majeures imposent de revoir en profondeur les principes sur lesquels s'appuie le stockage de données. Elles doivent donc être prises en compte très en amont, une démarche qui est restituée ici.

Les supercalculateurs de classe exaflopique tirent leurs accroissements des performances de différents facteurs :

- l'évolution des processeurs (qu'il s'agisse de CPU ou de GPU) qui fonctionnent à des fréquences plus hautes ;
- l'évolution des technologies de réseau haute performance ;
- l'introduction massive du parallélisme ;
  - en multipliant le nombre de machines au sein d'une ferme de calcul (ou cluster) ;
  - en multipliant le nombre de processeurs fonctionnant de manière coopérative dans chaque machine impliquée dans le cluster ;
  - en multipliant le nombre de cœurs de calcul par processeur ;

Il découle de ces évolutions différentes problématiques pour les systèmes de stockage massif, déclinées dans les paragraphes suivants.

Problématique 1 : l'inflation du volume de données et de métadonnées

La première problématique est naturellement l'augmentation du volume stocké. Il faudra gérer non seulement les nouvelles données produites, mais aussi les données anciennes qui auront été migrées. Cette inflation concerne les données, mais aussi les métadonnées, appelées, elles aussi, à augmenter jusqu'à atteindre des valeurs très grandes au regard des standards actuels.

Problématique 2 : l'inflation du nombre de systèmes clients du stockage

Il est ici fait écho au développement rapide du parallélisme : les calculateurs compteront un nombre de machines bien plus grand, toutes étant clientes des ressources de stockage. Quand ce chiffre commencera à se mesurer en dizaines, voire en centaines de milliers, l'infrastructure qui devra répondre au déluge de requêtes émanant de si nombreux clients devra être robuste, répliquée et fortement distribuée. Or, les systèmes de fichiers parallèles, pour lesquels la cohérence des données est critique, tendent au contraire à centraliser les informations.

Par ailleurs, ils mettent en œuvre des algorithmes dont la complexité explose à cette échelle, comme les gestionnaires de verrous distribués (appelés Distributed Lock Managers ou DLM) qui assurent la cohérence des données accédées.

Problématique 3 : l'inversion du ratio mémoire vive/cœur de calcul

Les innovations technologiques produisent des processeurs comprenant toujours plus de cœurs de calculs. C'est particulièrement vrai pour ceux exploités dans les supercalculateurs.

Chaque génération voit un accroissement de la mémoire vive et du nombre de cœurs de calcul, mais un nouveau phénomène se produit : le nombre de cœurs de calcul s'accroît désormais plus rapidement que la taille de la mémoire vive. En conséquence, le ratio mémoire vive par cœur décroît.

Cet effet n'est pas anodin, car les codes de simulation numérique exigent une taille mémoire par cœur incompressible, de l'ordre de la dizaine de gigaoctets, pour fonctionner. Le système d'exploitation doit donc réduire sa propre empreinte mémoire et se contenter de la portion congrue. Cette restriction de ressources rend très compliquée, voire impossible, la mise en œuvre des algorithmes exploités par les outils de stockage actuels (notamment les Distributed Lock Managers cités plus haut).

Problématique 4 : la non-localité des données

Les supercalculateurs sont des "clusters" de machines fédérées par un réseau d'interconnexion dont la complexité est bien plus grande que celle

du réseau qui connecte le calculateur au système de stockage. Lorsqu'une simulation démarre, les programmes qui composent la simulation vont être placés sur des machines proches, afin d'optimiser les communications, donc les performances.

Cette proximité n'est pas reproductible sur le réseau de stockage, plus simple, et certains composants vont donc se retrouver loin des serveurs qui hébergent les données qu'ils exploitent. Un surcoût en termes de latence en résulte dans certains cas.

Problématique 5 : l'hétérogénéité des ressources et des données, le placement de la donnée.

L'hétérogénéité est le dernier obstacle, mais pas le moindre. Sa première facette est l'hétérogénéité des supports. L'évolution des architectures de stockage durant ces dernières années a vu le développement massif des types SSD, mais aussi l'éclosion des mémoires non volatiles (NVRAM) qui permettent d'implémenter du stockage doté d'un débit rapide et d'une latence d'accès faible. Le coût financier élevé de ces supports ne rend pas caduque l'utilisation des disques rotatifs (HDD) et des bandes magnétiques.

Au contraire, on voit apparaître des architectures de stockage hiérarchisées qui empilent des supports peu coûteux et capacitifs, mais peu performants (bandes magnétiques, HDD) en dessous d'autres plus chers et beaucoup plus performants (SSD et NVRAM). Cette solution multi-niveaux permet des optimisations technologiques et financières, mais il en résulte de fortes disparités entre les supports de stockage des données.

La seconde facette recoupe l'hétérogénéité des données : celles-ci ont naturellement des profils très variés et des criticités diverses, ainsi certains fichiers vont nécessiter des volumes ou des performances importants (fichiers vidéo, bases de données, résultats de codes de calcul) tandis que d'autres se contenteront de moyens plus modestes (fichiers « checkpoints » des codes de simulations, souvent jamais relus).

Ces deux faces de l'hétérogénéité (des supports et des données) rentrent mutuellement en résonance : chaque type de support est plus adapté à certains types d'utilisation des données. La manière dont les données sont placées sur ces médias, et la manière dont elles sont déplacées entre médias différents deviennent des facteurs d'optimisation critiques.

3.3 . Le stockage objet : une nouvelle manière de stocker les données

Avant de poursuivre, il est indispensable de parler plus en détail du stockage objet. Ce concept rend possibles des actions devenues techniquement inenvisageables sur les systèmes de fichiers parallèles à l'échelle de l'exascale.

L'idée qui sous-tend le stockage objet remonte aux premières heures de l'informatique : il s'agit du mode d'adressage associatif.

De nombreuses différences en découlent :

- Là où les fichiers résidant dans un système de fichiers ont de nombreux voisins et sont enfouis dans un empilement complexe de répertoires, un objet n'a pas de voisin ni de répertoire parent qui le contient.
- Là où un fichier est adressé par un chemin parfois compliqué et susceptible de changer, un objet conservera la même clef, de manière immuable.
- Là où l'implémentation d'un système de fichiers complet va nécessiter la mise en œuvre d'environ une vingtaine de mécanismes différents, la gestion d'un objet n'en requiert que quatre. Ces quelques points font ressortir une facette très importante du stockage objet : sa sémantique d'accès est extrêmement simple.

Les technologies de stockage objet ne sont pas nouvelles. Les services de diffusion de vidéo en ligne les utilisent déjà massivement. De même, les fournisseurs de stockage "dans le cloud" s'appuient massivement sur elles pour gérer les immenses volumes de données associés à ces solutions.

Cette simplicité allège énormément la gestion des objets, puisque chacun d'entre eux est indépendant. Le relâchement des contraintes qui en résulte est conséquent et il devient plus facile de passer à de grandes échelles.

Par ailleurs, cette simplicité de la sémantique rend inutile la mise en œuvre de mécanismes complexes pour garantir les relations entre objets, puisque celles-ci n'existent pas au niveau du système. Cette autre levée des contraintes permet de facilement distribuer sur de nombreux serveurs les informations qui permettent d'adresser les objets, de les répliquer le cas échéant, et ainsi de mettre en place une architecture capable de faire face aux futurs supercalculateurs exascale.

Enfin, une sémantique simple est une brique de base sur laquelle implémenter des sémantiques plus complexes. Il est par exemple parfaitement possible de construire des systèmes de fichiers conformes à la norme POSIX (qui est très complexe) par-dessus des sémantiques propres au stockage objet.

3.4 .Nouveaux paradigmes pour les systèmes exascale

Des solutions existent pour répondre à ces nouvelles problématiques issues de l'exascale.

3.4.1 . Services éphémères et Data Nodes

La cohérence dans les systèmes de fichiers parallèles est coûteuse, surtout quand le nombre de systèmes clients augmente et que s'impose l'exi-

gence de garder une empreinte de mémoire faible.

Considérons l'utilisation des systèmes de fichiers par les utilisateurs finaux : ils exploitent des jeux de données propres à leur domaine de recherche (biologie, mécanique des fluides, thermodynamique,...) plus ou moins larges et spécifiques. Un code de calcul n'a donc pas le besoin d'accéder à l'ensemble des données, seulement à celles qui lui sont indispensables pour effectuer ses calculs. De plus, cet accès n'est utile que sur les nœuds de calcul qui sont impliqués dans les travaux en machine afférents.

La situation actuelle, qui permet un accès cohérent à l'ensemble des données à tout instant, relève par conséquent de la sur-qualité. On reformulera le besoin de l'utilisateur en rappelant qu'un travail de simulation connaît par avance les jeux de données dont il a besoin pour fonctionner et dans quel(s) jeu(x) de données il va produire.

On peut dès lors introduire plusieurs nouveaux paradigmes : les data nodes et les services éphémères. La description d'un travail de simulation sur le supercalculateur contient la liste exhaustive des jeux de données qui seront utilisés. On est donc en mesure de démarrer un service spécifique qui fournira au code de simulation ces données, et seulement celles-ci. Du point de vue de l'implémentation, cela revient à adosser au travail initial [1], demandé explicitement par l'utilisateur, un second travail implicite qui contiendra un serveur disposant de l'interface requise pour interagir avec le code et lui fournir l'accès à ses jeux de données. Il s'agira souvent d'une interface de type "système de fichiers", même si d'autres interfaces sont possibles, par exemple l'interface S3 soutenue par Amazon.

Le service n'a de sens que tant qu'il est adossé à un travail de simulation : il naît avec lui et disparaît avec lui, il n'est pas permanent comme les systèmes de fichiers des architectures pré-exascale. À cause de leur nature transitoire par construction, ces services alloués à la volée seront nommés "services éphémères".

Des machines sont nécessaires pour faire tourner ces serveurs. Elles seront choisies de manière à être proches des nœuds impliqués dans le travail de simulation qui déclenche le lancement du service éphémère. Dans la suite du document, on nommera *data nodes* les machines, proches des nœuds de calcul, sur lesquelles tournent les services éphémères.

L'introduction des services éphémères répond aux problématiques liées au changement d'échelle qui se manifestent dans les architectures exascale, à savoir l'inflation du nombre de clients et l'inversion du ratio "mémoire vive par cœur de calcul". En effet, les services éphémères ont à subir la pression d'un nombre plus faible de systèmes clients, la qualité de service requise correspond à l'échelle d'un travail de simulation, pas à celle de la machine tout entière. En dehors de rares occasions (tels que des passages de "grands dé-

1. le code de simulation soumis en machine

fis" qui requièrent la mise en place d'outils spécifiques), un très grand nombre de travaux fonctionnent en parallèle sur un calculateur, la charge sur les serveurs éphémères diminue donc d'un facteur équivalent. Par ailleurs, la proximité entre les nœuds de calcul et les data nodes permet la mise en place de mécanismes plus simples qui consomment peu de mémoire.

3.4.2 . Vers un placement intelligent des données

La gestion de l'hétérogénéité des données recoupe l'hétérogénéité des ressources utilisées pour les stocker. Les architectures de stockage pour l'exascale s'appuieront en effet sur différents types de supports organisés en piles :

- des supports très coûteux (de type NVRAM ou NVMe), avec de très hautes performances, en particulier des latences extrêmement faibles, mais peu capacitifs ;
- des disques SSD, dotés de bonnes performances et un peu plus capacitifs ;
- des disques rotatifs HDD, très capacitifs, mais peu performants ;
- des disques rotatifs HDD SMR, encore plus capacitifs, mais encore moins performants ;
- des bandes magnétiques, représentant plus de 80% des ressources de stockage, mais aux performances très réduites.

Les données hétérogènes correspondent à des usages hétérogènes, il s'agit d'optimiser les performances offertes aux utilisateurs en positionnant au mieux les données : on mettra une donnée "dormante" sur le niveau le plus bas composé de bandes magnétiques, mais à l'opposé une série de fichiers constituant une base de données très utilisée sera maintenue autant que possible dans les étages les plus hauts (NVRAM ou SSD).

Pour identifier ces comportements, on utilise des outils dits "d'apprentissage supervisé". Dès lors que l'on dispose d'éléments et de données décrivant les comportements des utilisateurs vis-à-vis des données, ces algorithmes permettent de regrouper ceux-ci en différentes populations.

Dès lors, on dispose d'un outil qui permet l'implémentation d'un gestionnaire de politique de placement. Ces résultats peuvent être utilisés en amont ; on observe en effet que la prédiction de comportement effectuée alors même que la donnée est en cours de création et d'accroissement est fiable[92], ce qui permet de la positionner très vite et d'éviter ensuite des déplacements coûteux en temps et en ressources.

3.5 . Résolution des problématiques via les nouvelles approches adoptées

L'utilisation de stockages objet à partir desquels on construira des services éphémères permet donc de répondre à la plupart des cinq problématiques déjà abordées :

1. par construction, les serveurs éphémères répondent à la problématique d'inflation du nombre de clients ;
2. l'utilisation des gestionnaires de stockage objet, qui passent beaucoup mieux à l'échelle et répartissent les données facilement entre différents serveurs, permet de répondre à la problématique liée à l'inflation du volume des données et des métadonnées ;
3. les serveurs éphémères servent moins de clients, car ils fonctionnent à l'échelle d'un travail de simulation et non du supercalculateur entier, ce qui permet d'utiliser des protocoles plus simples qui consomment moins de ressources sur les machines clientes (voire la section sur la "Problématique 3") ;
4. l'association entre services éphémères et travaux de simulation, par le biais du gestionnaire de ressources du supercalculateur, ainsi que la prise en compte des aspects liés au stockage de données par celui-ci, permettent de démarrer les services éphémères sur des data nodes proches des nœuds de calcul concernés. Ainsi, les données sont rendues aussi proches que possible des machines qui les produisent et qui les exploitent. Cela résout la problématique de non-localité ;
5. l'utilisation d'algorithmes de placement des données, exploitant des méthodes propres à l'apprentissage supervisé, permet de construire une solution à la problématique d'hétérogénéité en adaptant le comportant du gestionnaire de stockage HPC aux caractéristiques propres de chaque donnée.

3.6 . De nouvelles perspectives pour le stockage exascale

L'émergence de l'exascale fait apparaître différentes problématiques propres aux systèmes de stockage massif pour le HPC. Ces considérations peuvent rapidement devenir stratégique si l'on considère que le coût des systèmes de stockage représente une part notable du coût total d'un centre de calcul.

À l'heure où les anciens modèles utilisés pour exploiter les données atteignent leurs limites, il devient indispensable de construire de nouvelles solutions basées sur de nouveaux paradigmes, comme le stockage objet. D'autres solutions naissent au sein de technologies déjà bien ancrées dans le HPC, mais voient leur périmètre s'élargir. Ainsi, les technologies d'ordonnancement

et de placement des travaux de simulation en machine doivent prendre en compte les ressources de stockage et la bande passante des I/O.

De par mon expérience professionnelle, les supercalculateurs petaflopiques et leurs successeurs, tels que ceux actuellement en exploitation à la DAM et au TGCC, ont vu l'avènement des architectures dites "data-centriques" qui viennent placer les serveurs de stockage de données au centre des centres de calcul. Avec l'exascale, l'accent est encore mis sur les données, mais avec un focus sur l'utilisateur. Les données ne sont plus banalisées, elles subissent un traitement différent selon leur nature, que celle-ci soit exposée explicitement par leurs propriétés ou que le système dispose d'outils pour la détecter automatiquement. Les interfaces dont l'utilisateur a besoin pour exploiter ses données feront l'objet de services éphémères qui leur seront dédiés et qui optimiseront le service rendu et l'utilisation des ressources disponibles. C'est donc bien un focus "user-centrique" qui s'opère dès lors.

Même si elle débute à peine, cette approche est riche de promesses et permettra de faire face aux défis auxquels seront confrontées les architectures de stockages du futur.

3.7 .Les outils exploités

Les nouvelles fonctionnalités construites dans le contexte exascale s'appuient sur les technologies de stockage objet, puisque celles-ci lèvent une quantité de contraintes bloquantes, grâce à l'emploi de l'adressage de type associatif.

Les `object stores` seront utilisés conjointement avec les Key Value Stores (KVS) qui associent une valeur de type chaine de caractères ou buffer à une clef. En apparence, les deux notions peuvent sembler assez analogues dans leurs principes de base. Ce n'est pas un hasard si certains produits implémentent les deux fonctionnalités. Dans les paragraphes qui suivent, les différents produits disponibles seront parcourus et brièvement analysés.

3.8 .De nouvelles collaborations pour répondre aux enjeux de l'exascale

Le CEA fait sur cette période le choix stratégique d'élargir ses collaborations de R&D, en s'engageant dans différents projets européens. Ceux-ci s'inscrivent dans le mouvement de grands "méta projets" comme Horizon 2020 [2].

Dans ce cadre, j'ai été amené à être un partenaire très actif dans différents projets. Trois d'entre eux retiendront impliquent de près le stockage massif pour le HPC et le serveur NFS-Ganesha. Ces efforts constituent la suite des

2. https ://www.horizon2020.gouv.fr/

travaux fait durant la période petascale, lesquels s'étendent sur la période exascale.

Le projet SAGE (de 2015 à 2018) avait l'ambition de financer le développement du logiciel MERO tout en lui construisant un écosystème logiciel complet. Dans le contexte de SAGE, j'ai réalisé les couches logicielles nécessaires pour rendre NFS-Ganesha capable d'exposer le contenu d'objets au travers d'une interface mettant en action les différentes versions du protocole NFS et en particulier la fonctionnalité pNFS du protocole NFSv4.1 (c.f. D.6.3). Ce projet a ainsi donné naissance à la bibliothèque KVSNS (voir 3.10) et à la FSAL_KVSFS (voir 3.10.3).

Le projet Sage2 (de 2018 à 2021), concrétise les ambitions SAGE en étendant l'écosystème logiciel de MERO et son intégration dans des applications scientifiques.

Au cours de Sage2, MERO deviendra un produit open-source et changera de nom pour devenir CORTX-MOTR.

Le projet IO-SEA (de 2021 à 2024) sera détaillé dans la section 3.11. il a pour ambition de construire un environnement logiciel complet dédié au stockage de données massif et capable de répondre aux exigences de l'exascale. IO-SEA s'articule avec le projet RED-SEA, porté par ATOS et qui s'intéresse aux réseaux hautes performances, et avec le projet DEEP-SEA, porté par Forschungszentrum Jülich (FZJ), dont le but est la définition d'un environnement logiciel pour l'exascale s'appuyant sur les concepts développés autour de l'architecture Modular Supercomputing Architecture et des machines de calcul DEEP de FZJ. NFS-Ganesha est impliqué dans IO-SEA sous a forme d'un "service éphémère" permettant d'exposer, à la demande, des données à des codes de simulation.

J'ai assuré la supervision active de la rédaction de la proposition du projet IO-SEA. Je suis actuellement en charge de la coordination scientifique de ce projet.

3.9 .Les IO Proxies deviennent des serveurs éphémères

La période exascale introduit de nouvelles problématiques qui doivent être explicitement gérées (voir section 3.2). L'approche retenue repose sur les composants et les choix techniques suivants :

- réaliser le stockage des données au travers de stockage objet ;
- exploiter différentes technologies matérielles de stockage (mémoire non-volatile NVRAM, disques HDD et SSD, bandes magnétiques) gérées par des mécanismes de type HSM.
- adosser à l'exécution d'un travail de calcul sur un supercalculateur un serveur de stockage dédié, aussi appelé **serveur éphémère**, fonctionnant sur des **nœuds de gestion de données** (ou *data nodes*) ;
- la gestion des ressources associées au serveur éphémères au travers

d'un gestionnaire de ressources tel que Slurm [10], avec une allocation similaire à celle qui s'applique aux ressources de types CPU/GPU ou Mémoire Vive des nœuds de calcul;
- la versatilité des interfaces proposées aux codes de calcul. Outre les interfaces de type Portable Operating System Interface (POSIX), des interfaces de type Amazon S3 ou Openstack Swift sont envisagées, ainsi que le support d'agents permettant d'optimiser le fonctionnement de différentes couches de middleware, tel que Hierarchical Data Format version 5 (HDF5) [77].

Si l'on s'attarde sur l'implémentation d'un serveur éphémère fournissant une interface POSIX, de type "système de fichiers", il est intéressant de noter que cette fonctionnalité présente de nombreuses similarités avec la fonctionnalité IO-Proxy de la période petascale, décrite dans la section 2.9.2.

En effet, dans un cas comme dans l'autre, il s'agit d'exposer sous la forme d'un point de montage un sous-ensemble de données à un sous-ensemble bien identifié d'utilisateurs. Le travail effectué durant la période exascale sera donc réutilisé, et on retrouve ici le serveur NFS-Ganesha utilisé comme un serveur NFS éphémère.

La problématique de gestion du stockage objet apporte un nouveau défi puisqu'il s'agit de construire une structure arborescente de fichiers et de répertoires en s'appuyant sur des outils, les KVS et les *object stores*, qui adressent leur contenu de manière "plate" sans aucune organisation en arbre a priori.

Pour répondre à ce nouveau besoin, j'ai développé la bibliothèque de fonctions KVSNS qui fournit une série de fonctions IO similaires aux appels de la libC grâce aux services d'un KVS. Un object store est par ailleurs utilisé par KVSNS pour gérer le contenu des fichiers

Par la suite, l'interface nécessaire pour exploiter cette bibliothèque de fonctions dans NFS-Ganesha, donc sous la forme d'un module FSAL, la FSAL_KVSFS.

3.10 .La bibliothèque KVSNS

Cette section décrit la bibliothèque open-source de fonctions KVSNS.

3.10.1 . les couches de KVSNS

Bien qu'elle soit née dans l'environnement du produit Mero/MOTR, la bibliothèque KVSNS est pensée pour être totalement indépendante et autonome de ce produit. La même approche que celle suivie dans le cadre de la conception de NFS-Ganesha a été reprise ici : l'utilisation de couches d'abstraction. La figure 3.2 montre l'architecture en couches de la bibliothèque KVSNS.

Les rôles des couches sont les suivantes, en partant de la couche la plus haute :

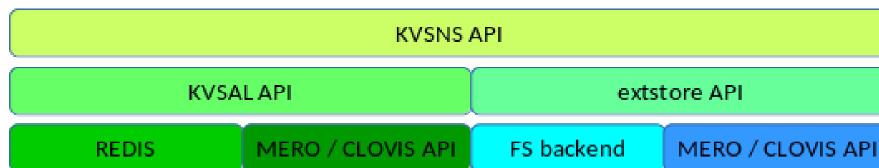


Figure 3.2 – Architecture en couches de la bibliothèque KVSNS

- la couche **KVSNS** fournit l'interface de plus haut niveau, elle implémente des fonctions de base qui reproduisent les appels POSIX associés aux systèmes de fichiers dans la LibC ;
- la couche **KVSAL**, acronyme de **K**ey-**V**alue **S**tore **A**bstraction **L**ayer, fournit une API générique qui permet d'exploiter les services d'un KVS ;
- la couche **extstore** fournit, quant à elle, une abstraction permettant d'utiliser via une API générique les services d'un Object Store.

Les couches **KVSAL** et **extstore** sont elles-mêmes implémentés par-dessus des produits qui fournissent les serices requis. Ces couches d'abstractions sont, dans leur logique, très proches des FSALs de NFS-Ganesha décrites à la section 1.5.1. On dispose donc de différentes bibliothèques différentes qui affichent la même interface mais qui s'appuient sur des services différents. Ces modules sont totalement indépendants et interchangeables. Seule la bibliothèque KVSNS est unique.

Le fonctionnement de la couche extstore

La couche extstore propose une abstraction de l'accès à un gestionnaire de stockage objet, ou *object store*. L'API est très simple et ses appels sont présentés dans la table 3.3. Les objets dans cette API sont identifiés par un type générique `extstore_id_t`. Il s'agit d'un simple buffer opaque destiné à être transmis à l'*object store* sous-jacent sur la base duquel on vient implémenter cette API.

Le fonctionnement de la couche KVSAL

Les appels de la couche d'abstraction KVSAL sont décrit dans le tableau 3.4. Cette couche permet d'interfacer de manière très générique les services d'un KVS.

Cette API dispose d'une logique simple et d'opérations élémentaires , puisqu'il s'agit de gérer l'association entre une clef et une valeur.

La clef est toujours une chaine de caractères. La valeur qui y est associée généralement une chaîne de caractères, mais il peut s'agir d'un buffer. Cette distinction opère en particulier entre les appels `kvsal_set()` / `kvsal_get()` et les appels `kvsal_set_binary()` / `kvsal_get_binary()`. Pour le cas très spécifique de l'association entre une clef et des métadonnées POSIX, manipulées

Nom de la fonction	Action réalisée
extstore_init()	Initialise la bibliothèque de fonctions
extstore_create()	Crée un nouvel objet, l'object id ayant été créée
extstore_new_objectid()	Crée une nouvelle object id sans créer d'objet
extstore_read()	Lecture séquentielle dans un objet
extstore_write()	Ecriture séquentielle dans un object
extstore_del()	Détruit un objet
extstore_truncate()	Tronque un objet
extstore_attach()	Réalise une opération d'attachement d'un objet
extstore_getattr()	Récupère les métadonnées "propres au stockage" sur un objet
extstore_archive()	Réalise une action "archive" (CRUD_CACHE)
extstore_restore()	Réalise une action "restore" (CRUD_CACHE)
extstore_release()	Réalise une action "release" (CRUD_CACHE)
extstore_state()	Obtient l'état d'un objet (CRUD_CACHE)
extstore_cp_to()	Copie le contenu d'un objet dans un file descriptor externe (bulk cp)
extstore_cp_from()	Recopie le contenu d'un file descriptor externe dans un objet (bulk cp)

Table 3.3 – Les fonctions de l'API extstore

via la structure `struct stat`, les fonctions spécifiques `kvsal_set_stat()` / `kvsal_get_stat()`.

L'API KVSAL permet de gérer des listes de clefs discriminées par des "motifs". Ceux-ci sont de simples "globs" tels que ceux qui sont classiquement manipulés dans les interpréteurs de Shell Script [101].

3.10.2 . Implémentation du namespace KVSNS

La couche KVSNS fournit tous les appels indispensables pour implémenter un comportement conforme à la norme POSIX. Dans le cas de NFS-Ganesha, ceux-ci sont utilisés pour construire le module FSAL_KVSFS qui permet de bâtir une interface de types "système de fichiers" en s'appuyant sur des Object Stores et des KVS. Les appels de l'API KVSNS sont décrits dans la table 3.5. Ceux-ci utilisent les structures décrites dans le tableau 3.6

Les types décrits dans la table 3.6 sont les types propres à la couche logicielle KVSNS. Les types standards de la LibC sont également utilisés.

La couche KVSNS va gérer plusieurs familles de clefs, toutes gérées par la couche KVSAL, pour mettre en place un espace de nommage POSIX. Ces familles sont présentées ci-après. Les noms des clefs sont variables, elles dé-

Nom de la fonction	Action réalisée
kvsal_init()	Initialise la bibliothèque de fonctions
kvsal_fini()	Stoppe la bibliothèque de fonctions
kvsal_begin_transaction()	Place le début d'une transaction
kvsal_end_transaction()	Place la fin d'une transaction
kvsal_discard_transaction()	Annule une transaction en cours
kvsal_exists()	Vérifie si une clef existe
kvsal_set_char()	Associe une clef avec une valeur chaîne de caractères
kvsal_get_char()	Récupère une chaine de caractères connaissant la clef
kvsal_set_binary()	Associe une clef avec un buffer
kvsal_get_binary()	Récupère un buffer connaissant la clef
kvsal_set_stat()	Associe une clef avec un buffer de type "struct stat"
kvsal_get_stat()	Récupère un buffer de type "struct stat" connaissant la clef
kvsal_get_list_size()	Récupère la taille d'une liste
kvsal_del()	Détruit une association clef-valeur
kvsal_incr_counter()	incrémente atomiquement un compteur associé à une clef
kvsal_get_list_pattern()	Récupère une liste au travers d'un "motif de clef"
kvsal_get_list()	Récupère la liste des clefs qui correspondent à un "motif de clef"
kvsal_fetch_list()	Récupère l'ensemble des clefs + valeurs qui correspondent à un "motif de clef"
kvsal_dispose_list()	Libère les ressources associées à une liste
kvsal_init_list()	Initialise une liste

Table 3.4 – Les fonctions de l'API KVSAL

pendent de certains paramètres tels que le nom d'une entrée ou son inode.

Gestion des métadonnées des entrées

Chaque entrée, qu'il s'agisse d'un fichier, d'un répertoire ou d'un lien symbolique, possède des attributs, ou métadonnées. C'est la structure `struct stat` de la libC qui sera utilisée. La section 3.10.1 a montré comment l'API KVSAL dispose de fonctions spécifiques pour gérer celle-ci.

Les attributs des entrées seront conservés dans des clefs dont le format est le suivant

```
‘‘%llu.stat’’(KVSNS_ino_t inode)
```

Nom du type	Nature
kvsns_ino_t	représente une inode dans l'espace de nommage
kvsns_cred_t	représente l'identité d'un utilisateur
kvsns_fsstat_t	représente les informations statiques de l'espace de nommage
kvsns_dentry_t	Représente une entrée à l'intérieur d'un répertoire
kvsns_open_owner_t	Représente l'acteur qui a ouvert un fichier
kvsns_file_open_t	Représente un fichier ouvert
kvsns_type_t	Type d'une entrée dans l'espace de nommage
kvsns_dir_t	Entrée de type répertoire
kvsns_xattr_t	Contenu d'un attribut étendu

Table 3.5 – Les structures de l'API KVSNS

Gestion des structures de répertoires et de liens avec des clefs

On doit ici gérer l'association qui existe entre une entrée, quels que soient son type, et le répertoire qui la contient. On retrouve ici les mêmes problématiques et les mêmes formalismes que nous avons vus dans les sections précédentes, relatives aux manières d'adresser les entrées dans un système de fichiers. Dans le cas de KVSNS, l'adressage direct d'une entrée sera réalisé en connaissant son numéro d'inode qui l'identifie de manière unique, l'adressage relatif sera réalisé en connaissant le couple formé par l'inode du répertoire parent et le nom de l'entrée dans le répertoire en question.

La gestion mise en place doit permettre de gérer les hard links. Comme nous l'avons vu, ceux-ci ne sont pas des entrées spécifiques comme les liens symboliques. Fondamentalement, cela revient à rendre possible, pour une entrée de type fichier, d'exister à plusieurs endroits dans l'espace de nommage de KVSNS, ce qui revient à dire qu'un fichier donné pourra avoir plusieurs répertoires parents différents.

La gestion des répertoires parents, pour un inode connue, sera obtenu avec une clef de la forme suivante :

```
‘‘%llu.parentdir’’(KVSNS_ino_t)
```

La valeur qui est retournée est une chaine de caractères qui représente une liste de répertoires parents, identifiés par leur numéro d'inodes. Il y aura autant d'entrée dans cette liste que l'entrée a de parent(s), et l'on fait le choix, si un fichier possède plusieurs hard links dans le même répertoire, et y existe donc sous différents noms, de référencer le répertoire parent un nombre de fois équivalent dans la liste. La longueur de cette liste, pour une inode donnée, est donc égale à la valeur du champ `st_nlink` de la structure `struct stat`.

Dans l'autre sens, il faut faire une correspondance entre un adressage direct et un adressage relatif, en d'autres termes établir une famille de clef qui permettent de connaître le numéro d'inode d'une entrée dont on connaît l'inode du répertoire parent et le nom dans le répertoire en question. Cette association est réalisée par la clef suivante :

```
‘‘%llu.dentries.%s’’(KVSNS_ino_t inode, char * name)
    → list by ‘‘%llu.dentries.*’’(KVSNS_ino_t inode) pattern
```

La valeur obtenue sera la valeur de l'inode de l'entrée concernée.

Par ailleurs, les liens symboliques seront gérés par les clefs suivantes :

```
‘‘%llu.link’’(KVSNS_ino_t)
```

Si l'inode en question est bien un lien symbolique alors cette clef existe et son contenu est une chaîne de caractères qui représente le contenu du lien. Implémenter la fonctions `kvsns_readlink()` revient donc à construire et à déréférencer la clef correspondante.

Gestion des fichiers ouverts et effacés

L'une des questions les plus complexes dans une implémentation de la sémantique POSIX est celle des fichiers ouverts. Là où des systèmes d'exploitation comme Windows refusent que soient effacés des fichiers qui sont ouverts, en retournant un message erreur explicite, POSIX va autoriser cet effacement. On procédera donc à l'effacement en deux temps : quand on efface un fichier, les métadonnées qui lui correspondent sont supprimées de l'espace de nommage, mais pas les données. Celles-ci restent accessibles par les acteurs qui ont déjà des références sur ces données [3] jusqu'à ce que ceux-ci relâchent la ressource [4]. Les données ne seront effectivement détruites que lorsque la dernière référence aura été rendue. Les fichiers possèdent donc un état intermédiaire *"opened and deleted"* quand ils sont effacés et toujours ouverts. Les transitions entre les différents états qui correspondent sont décrits dans le tableau suivant :

	Not Open	Open	Open Deleted	Deleted
Not Open		open	INTERDIT	delete
Open	close		delete	INTERDIT
Open Deleted	INTERDIT	INTERDIT		close
Deleted	INTERDIT	INTERDIT	INTERDIT	

La bibliothèque KVSNS implémente des clefs pour conserver les états qui permettent de gérer cet état intermédiaire. On rappelle que POSIX efface les

3. typiquement des processus qui disposent de `file descriptors` sur ces fichiers

4. Typiquement avec un appel à la fonction `close()`

fichiers selon deux méthodes : avec un effacement explicite réalisé par la commande shell `rm` ou la fonction `unlink()` de la libC, mais aussi en renommant un fichier avec un nom d'une entrée qui existe déjà, l'ancien possesseur du nom est ainsi effacé.

Il convient de référencer les acteurs qui ont ouvert les fichiers, que nous nommerons *open owners*. Dans KVSNS, un *open owner* sera un thread dans un processus POSIX, identifiés respectivement dans la LibC par les types `pthread_t` et `pid_t`. Un *open owner* sera identifié par une valeur répondant au format `‘‘%llu.%llu’’(pid_t pid, pthread_t thrid)’’`

A chaque fois qu'un thread ouvre un fichier, un nouvel *open owner* est créé, le fichier en question en conservera la trace via la clef suivante :

```
‘‘%llu.openowner’’(KVSNS_ino_t)
```

Un fichier peut être ouvert plusieurs fois par différents acteurs [5], la valeur associée à la clef sera ainsi une liste séparée par des "symboles pipes". A chaque fois que l'on fermera un fichier, le open owner qui correspond sera retiré de la liste et la valeur de la clef sera mise à jour.

Gestions des attributs étendus

La norme POSIX a apporté de manière récente la possibilité d'attacher des métadonnées additionnelles aux entrées d'un système de fichiers. Ces attributs étendus, ou *extended attributes* seront désignés par l'acronyme `xattr`. Les attributs étendus permettent de rajouter des métadonnées qui ont du sens du point de vue des applications et de certaines applications systèmes. Par exemple, les mécanismes propres à SELinux[99] sont implémentés par le biais des attributs étendus nommés *security.selinux*. De la même manière, le support des Access Control List, qui permet un contrôle fin [6] des contrôles d'accès sur les entrées d'un espace de nommage POSIX, est implémenté par les attributs étendus `system.posix_acl_access` et `system.posix_acl_default`.

Dans KVSNS, un attribut étendu dont on connaît le nom, et qui est associé à un fichier dont on connaît le numéro d'inode sera géré par une clef dont le format sera le suivant :

```
‘‘%llu.xattr.%s’’(KVSNS_ino_t inode, char *xattr_name)
    → list by ‘‘%llu.xattr.*’’(KVSNS_ino_t inode)
```

Configuration de KVSNS

Les sources de KVSNS et de ses couches de services KVSAL et EXTSTORE, qui mettent en œuvre des couches d'abstractions, sont organisées de la même

5. uniquement en *read-only*
6. surtout au regard de ce que les "modes" de POSIX permettent

manière que NFS-Ganesha. On s'appuie ainsi sur une gestion de la compilation via CMake[64] qui permet de bien isoler les différentes dépendances logicielles.

L'utilisation de KVSNS passe par un fichier de configuration. Les couches KVSAL et EXTSTORE sont gérés comme des bibliothèques dynamiques, chargées à la volée lors de l'initialisation de KVSNS, via la fonction *dlopen()*. Le fichier de configuration indique les chemins complets des bibliothèques en question, ainsi que des configurations propres aux outils sur lesquels s'appuient les couches en question. Comme dans NFS-Ganesha, cette gestion des dépendances permet de simplifier énormément la gestion des dépendances logicielles et la distribution logicielle du produit.

3.10.3 . KVSNS comme une FSAL de NFS-Ganesha

KVSNS est conçu pour s'interfacer avec NFS-Ganesha par le biais d'un module FSAL comme cela est décrit dans la section 1.5.1. Dans son formalisme, KVSNS est conçu pour de manière aussi simple et rapide que possible avec NFS-Ganesha. On va ainsi produire un module FSAL_KVSFS qui va simplement effectue la "couche de colle" (glue layer) entre FSAL et KVSNS.

LA FSAL_KVSFS a été écrite dans le cadre du projet SAGE afin de produire une interface NFS et pNFS pour le gestionnaire de stockage objet MERO, lequel deviendra le produit open-source MOTR quelques années plus tard. Le produit a été maintenu par la suite pour fournir d'autres interfaces sur des gestionnaires de stockage objet, dont le produit Phobos[18] dans le projet I/O Software Environment Architecture (IO-SEA). Un exemple est donné en annexe (voir annexe H)

3.11 .Mise en œuvre de NFS-Ganesha dans le projet IO-SEA

Les problématiques qui sont à l'origine du projet IO-SEA sont associées à l'Exascale. Les modèles actuellement utilisés durant l'ère petascale pourraient en effet ne plus passer à l'échelle dans le cadre de l'Exascale comme cela est décrit dans la section 3.2. IO-SEA a l'ambition de proposer une nouvelle architecture logicielle ou *Software Exascale Architecture*, afin de répondre aux exigences futures.

IO-SEA s'appuie sur une approche de *co-design* impliquant des applications de simulation appartenant à différents domaines de recherche, depuis l'astrophysique à la physique des particules. Il tente de répondre aux problématiques décrites dans la section 3.2 et il implémente les solutions définies dans cette même section et il les met concrètement en œuvre via des collaborations avec des scientifiques et des industriels. (cf 3.8).

On trouvera donc dans IO-SEA des tâches et des *Work Packages* implémentant des fonctionnalités d'HSM, des services éphémères, et des interfaces spé-

cifiques pour faciliter l'intégration de cette nouvelle couche logicielle dans les applications.

3.11.1 . L'architecture logicielle IO-SEA

L'architecture logicielle du projet IO-SEA peut être résumé par la figure 3.3.

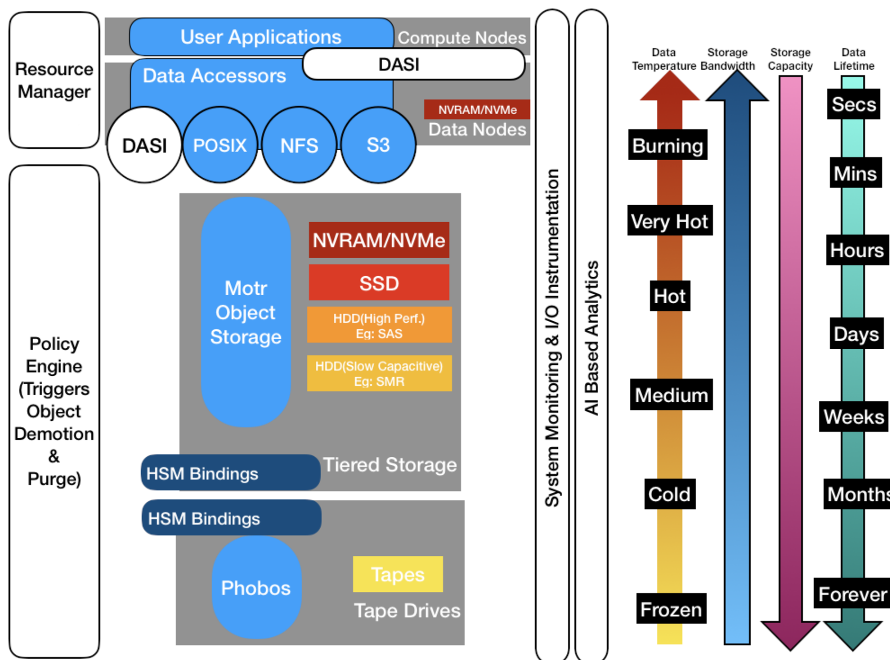


Figure 3.3 – L'architecture logicielle IO-SEA

Le projet IO-SEA va mettre en place différents éléments logiciels exposés dans la partie gauche de la figure 3.3, à savoir :

- la mise en place de mécanisme HSM au sein des gestionnaires de stockage objet CORTX/MOTR et Phobos, mais aussi entre les deux logiciels qui permettent ensemble de construire des *storage tiers* permettant de gérer tout le cycle de vie des données, depuis les supports rapides capables de gérer le protocole NVMe, jusqu'aux bandes magnétiques;
- l'utilisation du logiciel Robinhood Policy Engine modifié pour le rendre à même de gérer le stockage objet en plus des systèmes de fichiers pour lesquels il a été initialement créé;
- l'utilisation transverse intensive de la surveillance et de la télémétrie logicielle de manière à collecter des traces à tous les niveaux de l'architecture; ces données vont nourrir des algorithmes d'apprentissage automatique afin d'identifier des motifs et des comportements permettant des optimisations des différents acteurs;
- le développement d'interface utilisateurs de différents types permet-

tant d'exposer de manière simple les informations conservées dans les objets.

3.11.2 . Le serveur NFS-Ganesha dans le projet IO-SEA

IO-SEA reprend les travaux menés précédemment et les poursuit. En particulier, il met en jeu NFS-Ganesha comme un service éphémère. Les travaux précédents ont rendu possible cette intégration de la manière suivante :

- le modèle des IO-Proxies, décrit dans la section 2.9.2 permet de facilement utiliser NFS-Ganesha comme un service éphémère dans la sémantique de IO-SEA ;
- SAGE avait permis le développement de la couche KVSNS. Cette couche fournit une base solide au projet IO-SEA, et elle s'est vue dotée de nouvelles fonctionnalités telles que la gestion des object stores conforme à Create Read Update Delete (CRUD).
- les couches afférentes à KVSNS et NFS-Ganesha, comme FSAL_KVSFS qui seront portées dans ce nouvel environnement,
- des couches et des outils génériques, comme GRH (voir section I.5), et le CRUD_CACHE (voir section I.1), viennent compléter cet écosystème logiciel basé sur NFS-Ganesha.

3.12 .Bilan et perspectives pour la période Exascale

Ce chapitre a présenté brièvement la période Exascale, période dans laquelle nous nous trouvons actuellement. Cette période va voir arriver les premières machines dont les performances dépasseront 1 exaflop, ce qui soulève de nouvelles problématiques, en particulier pour les systèmes de stockage massif de données. Ces nouvelles problématiques sont abordées dans une section dédiée, ainsi que des propositions de pistes pour y répondre efficacement.

Parmi les solutions proposées figurent les *services éphémères*, qui peuvent être implémentés à partir de composants de base créés lors des périodes Terascale et Petascale. Le serveur NFS-Ganesha et les IO-Proxies en font partie.

La période Exascale offre de nouveaux outils, comme les KVS et les gestionnaires de stockage objet. La bibliothèque KVSNS est ensuite présentée. Elle permet de construire, sur les bases de ces nouveaux éléments, un système de nommage totalement conforme à POSIX et de l'exposer aux utilisateurs au travers de NFS-Ganesha.

Le projet IO-SEA est ensuite exposé. Il vise à construire un écosystème logiciel européen pour le stockage, et être à même de répondre aux nouvelles contraintes de cette période. IO-SEA exploite NFS-Ganesha sous forme de service éphémère, une fonctionnalité qui s'inscrit dans la continuité de la fonctionnalité IO-Proxy présenté dans la section précédente.

Nom de la fonction	Action réalisée
kvsns_start()	Initialise la bibliothèque de fonctions
kvsns_stop()	Ferme la bibliothèque de fonctions
kvsns_init_root()	Initialise la racine
kvsns_access()	Retourne les types d'accès possibles
kvsns_creat()	Crée une entrée "fichie"r
kvsns_mkdir()	Crée une entrée "répertoire"
kvsns_symlink()	Crée une entrée "lien symbolique"
kvsns_readlink()	Lit le contenu d'un lien symbolique
kvsns_rmdir()	Efface une entrée de type répertoire
kvsns_unlink()	Efface une entrée
kvsns_link()	Crée un lien hard
kvsns_rename()	Renomme une entrée
kvsns_lookup()	Effectue une opération de lookup
kvsns_lookupp()	Identifie le parent d'une entrée
kvsns_get_root()	Retourne la racine
kvsns_getattr()	Retourne les attributs POSIX
kvsns_setattr()	Positionne les attributs POSIX
kvsns_fsstat()	Retourne les informations statiques de l'espace de nommage
kvsns_opendir()	Ouvre un répertoire
kvsns_readdir()	Lit les dentries d'un répertoire ouvert
kvsns_closedir()	Ferme un répertoire
kvsns_open()	Ouvre un fichier dont on connaît l'inode
kvsns_openat()	Ouvre un fichier dont on connaît le nom et l'inode du parent
kvsns_close()	Ferme un fichier ouvert
kvsns_write()	Écrit dans un fichier ouvert
kvsns_read()	Lit à l'intérieur d'un fichier ouvert
kvsns_setxattr()	Positionne un attribut étendu sur une entrée
kvsns_getxattr()	Lit le contenu d'un attribut étendu
kvsns_listxattr()	Liste les attributs étendus d'une entrée
kvsns_removexattr()	Supprime un attribut étendu
kvsns_remove_all_xattr()	Supprime tous les attributs étendus
kvsns_cp_from()	Copie un fichier vers un *file descriptor*
kvsns_cp_to()	Copie un *file descriptor* externe dans un fichier
kvsns_attach()	Attache une entrée dans le namespace
kvsns_lookup_path()	Convertit un chemin complet en inode
kvsns_archive()	Archive une entrée (CRUD_CACHE)
kvsns_restore()	Restaure une entrée (CRUD_CACHE)
kvsns_release()	Détruit la copie de travail (CRUD_CACHE)
kvsns_state()	Donne le statut d'archivage (CRUD_CACHE)

Table 3.6 – Les structures de l'API KVSNS

Conclusion générale

NFS-Ganesha a prouvé être un logiciel très facile à adapter et il a su apporter des solutions à différentes contraintes apparues dans les centres de calcul du CEA depuis la période Terascale jusqu'à la période Exascale. Pour chacune de ses périodes, on a identifié les contributions que j'ai apportées via NFS-Ganesha, en précisant les problématiques qu'elles résolvent.

Dans la partie suivante, j'ai décrit les travaux toujours en cours, les évolutions attendues des paradigmes de stockage à l'heure de l'Exascale ainsi que le rôle que NFS-Ganesha pourrait être appelé à y jouer.

La dernière partie brosse un ultime bilan synthétique et posera des pistes d'évolutions futures.

Contributions apportées

Cette partie rappelle mes contributions lors des différentes périodes mentionnées dans ce manuscrit.

Période Terascale

La période Terascale voit le visage du HPC évoluer en profondeur quand les anciennes machines de type MPP, sont remplacées par des clusters de machines SMP. Les machines Cray disparaissent peu à peu des centres de calcul, mais de nouveaux systèmes avec de nouvelles fonctionnalités font leur apparition. En particulier, les systèmes de stockage massif, ancêtre de nos *datacenters* viennent centraliser la gestion et la conservation de l'ensemble des données produites par les supercalculateurs. Le besoin de performances conduit à la construction de mécanismes de cache, comme SHerPA. Le fonctionnement de ces caches suppose de manipuler efficacement les métadonnées des fichiers et des répertoires, mais les logiciels existant sur le marché ne sont alors pas assez performants. Pour pallier ce manque, j'ai architecturé, construit et déployé le serveur NFS-Ganesha.

NFS-Ganesha est un serveur NFS qui fonctionne dans l'espace utilisateur. Sa première mouture implémente les protocoles NFSv2 et NFSv3, et il est pensé pour traiter les requêtes parallèles et être capable de gérer différents *back-ends* de données. NFS-Ganesha exportera dans un premier temps l'espace de nommage géré par le logiciel HPSS dans le cadre de son exploitation sur les centres de calcul du CEA.

Période Petascale

En 2004, le logiciel NFS-Ganesha devient un logiciel open-source disponible sur la plateforme SourceForge. Grâce à différentes interventions dans des conférences externes [31] et à des publications [28][29], je parviens à attirer l'attention des industriels sur NFS-Ganesha qui devient le cœur d'une communauté open-source impliquant des centres de recherche et des entreprises qui intègrent des logiciels libres dans leurs solutions.

Un événement sera à ce titre particulièrement déterminant : l'arrivée du protocole NFSv4[14][15]. Je ferai l'implémentation du support de NFSv4 dans NFS-Ganesha dès la publication des Request For Comments (RFC) correspondantes, faisant de NFS-Ganesha l'un des premiers serveurs NFS supportant NFSv4, mais également le seul serveur en espace utilisateur à disposer de cette fonctionnalité.

La licence LGPLv3[42] choisie pour NFS-Ganesha facilitera l'intégration dans des produits commerciaux tout en facilitant la réversion des modifications sous forme de logiciel libre. Une collaboration s'est ainsi construite en impliquant différents acteurs industriels, dont IBM et RedHat, ce qui permet de faire une profonde refonte du produit, un effort conduit par tous les acteurs de cette nouvelle communauté. NFS-Ganesha en bénéficiera grandement, gagnant en stabilité et en performances. Cela finira par positionner NFS-Ganesha comme un standard *de-facto* pour les serveurs NFS en espace utilisateur.

Durant la même période, les études de recherches et développements du CEA soulignent l'importance de revoir les méthodes selon lesquelles les IO sont réalisées durant les simulations sur les supercalculateurs. Je deviens le pilote de la thématique dédiée au stockage de données massif dans cette nouvelle convention de R&D commune établie entre le CEA et ATOS. Dans ce contexte, je construis et mets en place la fonctionnalité IO-Proxy. Celle-ci me fera mettre en œuvre dans NFS-Ganesha d'abord le protocole 9p2000.L[51], issu du système d'exploitation Plan 9, puis les protocoles p9p et LIOP, qui injectent du parallélisme dans 9p2000.L et que je vais créer à cette occasion. J'assurerai l'implémentation de la partie serveur (dans NFS-Ganesha) de ces nouveaux protocoles.

Période Exascale

Cette période voit arriver des machines toujours plus puissantes, dont les performances flirtent avec l'Exascale. Les ressources de stockage, jadis souvent surdimensionnées, subissent des contraintes fortes et inédites. Dans ce contexte, le stockage objet, initialement associé aux technologies *Cloud*, apparaît comme une solution de stockage adaptée à cette situation.

Le CEA est désormais partie prenante de plusieurs projets financés par l'Europe qui travaille à définir des solutions innovantes. Dans cette optique, je

suis le responsable scientifique du projet IO-SEA dont l'ambition est la création d'une pile logicielle dédiée au stockage de données pour l'Exascale. Je mets NFS-Ganesha en œuvre, dans ce projet, en reprenant certaines idées des IO-Proxies et en les adaptant à ce nouvel environnement. NFS-Ganesha devient alors un *service éphémère*. Dans ces différents projets, je développe la bibliothèque KVSNS , qui permet d'interfacer NFS-Ganesha avec du stockage objet. Cette bibliothèque sera modifiée à plusieurs reprises, notamment pour prendre en compte les gestionnaires de stockage objet qui ne proposent que la seule sémantique CRUD, souvent au travers de l'API S3.

Travaux en cours et travaux futurs

Différents travaux sont toujours en cours à l'heure où ce manuscrit est rédigé.

La communauté NFS-Ganesha est toujours très active, plusieurs solutions commerciales, mais aussi opensource sont basées dessus [7]. Il est plus que jamais le standard pour les serveurs NFS en espace utilisateur. Il a ainsi débordé de son écosystème d'origine, à savoir le HPC, vers celui des services de bureautique, en particulier les serveurs de fichiers basés sur Linux.

Le paradigme du stockage objet, initialement né pour le stockage dans le *cloud*, est désormais une solution incontournable pour répondre aux contraintes de l'Exascale, et j'ai développé la bibliothèque KVSNS pour permettre d'exploiter ce paradigme via NFS-Ganesha. Cette bibliothèque est appelée à poursuivre ses évolutions pour supporter davantage de gestionnaires de stockage objet. Des travaux sont en particulier en cours pour l'interfacer avec des nouveaux *back-ends* tels que le logiciel DAOS d'Intel.

KVSNS pourrait devenir une souche logicielle autonome à moyen ou long terme, comme ce fut le cas pur NFS-Ganesha qui dispose maintenant de sa propre communauté de développeurs. Des efforts sont à poursuivre dans ce sens.

Les actions entamées dans le cadre de IO-SEA sont toujours en cours, le projet devant livrer ses conclusions en avril 2024. L'intégration de NFS-Ganesha dans la pile logicielle correspondante va se poursuivre.

Par ailleurs, le concept de *services éphémères*, porté par le projet IO-SEA, et implémenté via NFS-Ganesha, vise à permettre aux IO de franchir le cap de l'exascale sans devenir un goulot d'étranglement, grâce à un niveau d'inférence additionnel apporté par cette solution. Des évolutions de NFS-Ganesha sont en cours de réalisation pour finaliser cette intégration logicielle.

---

7. en particulier autour des logiciel CEPH, et GlusterFS dans des solutions portées par IBM/RedHat

Bibliographie

[1] C. Adams. The Simple Public-Key GSS-API Mechanism (SPKM). RFC 2025, October 1996.

[2] A. Adamson and N. Williams. Remote Procedure Call (RPC) Security Version 3. RFC 7861, November 2016.

[3] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146 :263—-266, 1962.

[4] J.H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. Hyperx : topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[5] Ch. Anderson, J. Lehnardt, and N. Slater. *CouchDB - The Definitive Guide : Time to Relax.* O'Reilly, 2010.

[6] D. Anton and E. Simion. Linux unified key setup (luks) - the good, the bad, the ugly. In *2018 10th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–6, 2018.

[7] A. Appleby. Murmur3 hash, 2008.

[8] L. Astrand and T. Yu. Deprecate DES, RC4-HMAC-EXP, and Other Weak Cryptographic Algorithms in Kerberos. RFC 6649, July 2012.

[9] ATOS. Bull exascale interconnect v2. urlhttps ://atos.net/fr/solutions/calcul-haute-performance-hpc/bxi-bull-exascale-interconnect.

[10] N. Azginoglu, M. U. Atasever, Z. Aydin, M. Celik, and H. Erbay. Open source slurm computer cluster system design and a sample application. In *2017 International Conference on Computer Science and Engineering (UBMK)*, pages 403–406, 2017.

[11] B. Barrett, R. Brightwell, K. Underwood, and S. Hemmert. Portals 4 network programming interface. In *2012 SC Companion : High Performance Computing, Networking, Storage and Analysis (SCC).*, pages 1467–1467, 11 2012.

[12] R. Bayer. Binary b-trees for virtual memory. In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and ControlNovember 1971*, pages 219–235, 1971.

[13] R. Bayer. Symmetric binary b-trees : Data structure and maintenance algorithms. *Acta Informatica*, 1(4) :290—-306, 1972.

[14] C. Beame, D. Robinson, B. Callaghan, D. Noveck, R. Thurlow, M. Eisler, and Shepler. NFS version 4 Protocol. RFC 3010, December 2000.

[15] C. Beame, R. Thurlow, B. Callaghan, D. Robinson, D. Noveck, M. Eisler, and S. Shepler. Network File System (NFS) version 4 Protocol. RFC 3530, April 2003.

[16] P. Braam. The lustre storage architecture. https://arxiv.org/abs/1903.01955, 2019.

[17] Brent Callaghan and Satinder Singh. The autofs automounter. In *USENIX Summer*, 1993.

[18] CEA. Phobos storage on github.com, 2019. https://github.com/cea-hpc/phobos.

[19] M. Chadalapaka, J. Satran, K. Meth, and D. Black. Internet Small Computer System Interface (iSCSI) Protocol (Consolidated). RFC 7143, April 2014.

[20] K. Chodorow and M. Dirolf. *MongoDB - The Definitive Guide : Powerful and Scalable Data Storage.* O'Reilly, 2010.

[21] Oracle Company. Rpc language specification.

[22] OSD Consortium. The osdfs filesystem. http://osdfs.sourceforge.net/.

[23] Panasas Corp. Panasas panfs parallel file system. https://www.panasas.com/products/panfs/.

[24] Cray Corporation. Cray history : the cray t90range. https://cray-history.net/cray-history-front/fom-home/cray-t90/.

[25] Cray Corporation. Inside the cray t3e/900 sn6702. https://cray-history.net/2021/07/16/2152/.

[26] T. Declerck. Using robinhood to purge data from lustre file systems. *CUG, Cray User Group*, 2014.

[27] S. K. Deepak. Rest based api. *International Journal of Trend in Scientific Research and Development*, 1(4) :571–575, June 2017.

[28] Ph. Deniel. Introduction à la gssapi. *Linux Magazine*, 1(52), jul 2003.

[29] Ph. Deniel. Authentification dans les onc/rpc. *M.I.S.C*, 1(17), jan 2005.

[30] Ph. Deniel, Th. Leibovici, and J-Ch. Lafoucrière. Ganesha, a multi-usage with large cache nfsv4 serve, wip session at fast'97. https://www.usenix.org/legacy/event/fast07/wips/slides/deniel.pdf, 2007.

[31] Ph. Deniel, Th. Leibovici, and J-Ch. Lafoucrière. Ganesha, a multi-usage with large cache nfsv4 server. In *Proceedings of the Linux Symposium 2007*, volume 1, pages 113–124, 2007.

[32] S. Derradji, Th. Palfer-Sollier, J-P. Panziera, A. Poudes, and F. Wellenreiter. The bxi interconnect architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 18–25, 2015.

[33] FUSE developers team. Fuse github repository. https://github.com/libfuse/libfuse.

[34] Jemalloc develpement team. Jemalloc. https://jemalloc.net/.

[35] J. Dongarra. Linpack. https://netlib.org/linpack/.

[36] M. Eisler. NFS Version 2 and Version 3 Security Issues and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5. RFC 2623, June 1999.

[37] M. Eisler. LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM. RFC 2847, June 2000.

[38] M. Eisler. RPCSEC_GSS Version 2. RFC 5403, February 2009.

[39] M. Eisler. IANA Considerations for Remote Procedure Call (RPC) Network Identifiers and Universal Address Formats. RFC 5665, January 2010.

[40] M. Eisler, L. Ling, and A. Chiu. RPCSEC_GSS Protocol Specification. RFC 2203, September 1997.

[41] M. Fedor, M. Lee S., J. Davin, and J. Case. Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[42] Free Software Foundation. Gnulesser general public license (lgplv3)version 3, jun 2007.

[43] S. Fridella, D. Black, and J. Glasgow. Parallel NFS (pNFS) Block/Volume Layout. RFC 5663, January 2010.

[44] S. Ghemawat and P. Menage. Tcmalloc : Thread-caching malloc.

[45] A. Goos. Is Plan 9 sci-fi or UNIX for the future ? *UNIX/world*, 7(10), October 1990.

[46] Th. Haynes. Network File System (NFS) Version 4 Minor Version 2 External Data Representation Standard (XDR) Description. RFC 7863, November 2016.

[47] Th. Haynes and D. Noveck. Network File System (NFS) Version 4 Protocol. RFC 7530, March 2015.

[48] S. Hernández, J. Fabra, P. Álvarez, and J. Ezpeleta. A reliable and scalable service bus based on amazon sqs. In Kung-Kiu Lau, Winfried Lamersdorf, and Ernesto Pimentel, editors, *ESOCC*, volume 8135 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 2013.

[49] S. Ishiguro, J. Murakami, Y. Oyama, and O. Tatebe. Optimizing local file accesses for fuse-based distributed storage. In *2012 SC Companion : High Performance Computing, Networking Storage and Analysis*, pages 760–765, 2012.

[50] T. Kacperski. Highly available nfs based kerberos kdc aka. ganesha + glusterfs + haproxy, 3 2018. https://www.loadbalancer.org/blog/highly-available-shared-nfs-server/.

[51] B. Kernighan. Beyond Unix : Plan 9. *Information Week*, 293, October 1990.

[52] J. Kim, W. Dally, and D. Abts. Flattened butterfly : a cost-efficient topology for high-radix networks, 6 2007.

[53] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88, 2008.

[54] D. Knuth. *The Art of Computer Programming, volume 3 : « Sorting and Searching », 3e édition*. Addison-Wesley, 1997. Voir en particulier les pages 458 à 475 de la section 6.2.3 : « Balanced Trees », expression qui désigne les arbres AVL chez Knuth.

[55] Th. Leeper. *aws.s3 : AWS S3 Client Package*, 2020. R package version 0.3.21.

[56] Th. Leibovici. Taking back control of HPC file systems with robinhood policy engine. *CoRR*, abs/1505.01448, 2015. http://arxiv.org/abs/1505.01448.

[57] Th. Leibovici, A. Cedeyn, G. Delbary, S. Gougeaud, and A. Degrement. Robinhood policy engine. https://github.com/cea-hpc/robinhood/wiki.

[58] J. Linn. Generic Security Service Application Program Interface. RFC 1508, September 1993.

[59] J. Linn. The Kerberos Version 5 GSS-API Mechanism. RFC 1964, June 1996.

[60] J. Linn. Generic Security Service Application Program Interface, Version 2. RFC 2078, January 1997.

[61] J. Linn. Generic Security Service Application Program Interface Version 2, Update 1. RFC 2743, January 2000.

[62] J. Lombardi. Essa 2022 invited speaker daos : Nextgen storage stack for hpc and ai. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1101–1101, 2022.

[63] B. Martin. Run your nfs server in the user address space with nfs-ganesha, 2008. https://www.linux.com/news/run-your-nfs-server-user-address-space-nfs-ganesha/.

[64] K Martin and B Hoffman. Mastering cmake : A cross-platform build system, 2010. ISBN 978-1930934207.

[65] Sun MIcrosystems. XDR : External Data Representation standard. RFC 1014, June 1987.

[66] Sun Microsystems. RPC : Remote Procedure Call Protocol specification : Version 2. RFC 1057, June 1988.

[67] Sun Microsystems. NFS : Network File System Protocol specification. RFC 1094, March 1989.

[68] S.W. Miller. Mass storage reference model special topics. In *Digest of Papers Ninth IEEE Symposium on Mass Storage Systems, 1988. 'Storage Systems : Perspectives'*, pages 3–7, 1988.

[69] J. Morris. Security enhanced nfs (senfs) requirements - draft 06, June 2007. http://namei.org/lnfs/senfs-requirements-draft-06.txt.

[70] C. Neuman, S. Hartman, K. Raeburn, and T. Yu. The Kerberos Network Authentication Service (V5). RFC 4120, July 2005.

[71] C. Neuman and Th. Ts'o. The Kerberos Network Authentication Service (V5). RFC 1510, September 1993.

[72] D. Noveck, M. Eisler, and S. Shepler. Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description. RFC 5662, January 2010.

[73] D. Noveck, M. Eisler, and S. Shepler. Network File System (NFS) Version 4 Minor Version 1 Protocol. RFC 5661, January 2010.

[74] D. Noveck and Ch. Lever. Network File System (NFS) Version 4 Minor Version 1 Protocol. RFC 8881, August 2020.

[75] R. Pike, D. Presotto, . Thompson, and H. Trickey. Plan 9 from Bell Labs. In *UKUUG. UNIX - The Legend Evolves. Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9 (of xi + 260), Buntingford, Herts, UK, 1990. UK Unix Users Group.

[76] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *EUUG Newsletter*, 10(3) :2–11, 1990.

[77] M. Poinot. Five good reasons to use the hierarchical data format. *Computing in Science & Engineering*, 12(5) :84–90, 2010.

[78] Ceph Project. Ceph main page. https://ceph.io/en/.

[79] Openstack Project. Openstack main page. https://www.openstack.org/.

[80] Openstack Project. Openstack/swift : github page. https://github.com/openstack/swift.

[81] SELinux Project. Labeled nfs. https://selinuxproject.org/page/Labeled_NFS.

[82] D.M. Ritchie and B.W. Kernighan. *The C programming language*. Bell Laboratories, 1988.

[83] O. Rodeh and A. Teperman. zfs - a scalable distributed file system using object disks. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings.*, pages 207–218, 2003.

[84] Seagate. Cortx/motr main page. https://github.com/Seagate/cortx-motr.

[85] R. Sedgewick and Leonidas J. Guiba. A dichromatic framework for balanced trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[86] P. Staubach, B. Pawlowski, and Callaghan B. NFS Version 3 Protocol Specification. RFC 1813, June 1995.

[87] DDN Storage. Hierarchical storage management (hsm) (lustre 2.5). https://github.com/DDNStorage/lustre_manual_markdown/blob/master/03.15-Hierarchical%20Storage%20Management%20(HSM).md.

[88] S. Strbac-Savic and M. Tomasevic. Comparative performance evaluation of the avl and red-black trees, 09 2012.

[89] Th. Talpey and B. Callaghan. Network File System (NFS) Direct Data Placement. RFC 5667, January 2010.

[90] Th. Talpey and B. Callaghan. Remote Direct Memory Access Transport for Remote Procedure Call. RFC 5666, January 2010.

[91] Haynes Th. Network File System (NFS) Version 4 Minor Version 2 Protocol. RFC 7862, November 2016.

[92] L. Thomas, S. Gougeaud, S. Rubini, Ph. Deniel, and J. Boukhobza. Predicting file lifetimes for data placement in multi-tiered storage systems for hpc. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '21, New York, NY, USA, 2021. ACM.

[93] Top500. Top 500 - homepage. https://www.top500.org.

[94] R.Y. Wang and T.E. Anderson. xfs : a wide area mass storage file system. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*, pages 71–78, 1993.

[95] R. Weber. Scsi object-based storage device commands -2 (osd-2). T10-1355D, January 2009. http://www.t10.org/cgi-bin/ac.pl?t=f&f=osd2r05a.pdf.

[96] B. Welch, J. Zelenka, and B. Halevy. Object-Based Parallel NFS (pNFS) Operations. RFC 5664, January 2010.

[97] Whamcloud. Configuring and using hierarchical storage management. https://whamcloud.github.io/Online-Help/docs/Config_and_using_HSM_6_0.html.

[98] Wikipedia. Fat tree. https://en.wikipedia.org/wiki/Fat_tree.

[99] Wikipedia. Selinux. https://fr.wikipedia.org/wiki/SELinux.

[100] E.H. Williams, N.G. Sullivan, J.G. Rusnak, J.E. Menges, D.M. Ogle, R.A. Floyd, and W.W. Chung. The andrew file system on os/2 and sna. In *Proceedings of TRICOMM '91 : IEEE Conference on Communications Software : Communications for Distributed Applications and Systems*, pages 181–191, 1991.

[101] F. Yesmin. Bash globbing tutorial, 2018. https://linuxhint.com/bash_globbing_tutorial/.

[102] E. Zeidner, C. Sapuntzakis, M. Chadalapaka, J. Satran, and K. Meth. Internet Small Computer Systems Interface (iSCSI). RFC 3720, April 2004.

Annexe A : Curriculum Vitae et liste de publications

Formation

- **Juin 1990 :** Baccalauréat section C
- **Septembre 1990-Juin 1993 :** Classes Préparatoires (Maths Sup et Maths Spé M') au Lycée Condorcet à Paris
- **Juin 1996 :** Diplôme d'ingénieur de l'École Centrale Paris (Option Informatique Temps Réel)
- **1994-1995 :** maitrise et un DEA de mathématiques pures (en parallèle avec mes études à l'École Centrale Paris).

Postes occupés

- **Du 01/09/2021-Présent** : Expert Fellow, CEA-DAM
- **2002-Présent** : Responsable R&D Stockage HPC, CEA-DAM
- **Du 01/11/2015 au 31/12/2023** : Chef du laboratoire « Stockage HPC et stations scientifiques », Architecture des systèmes de stockage des centres de calcul, CEA-DAM
- **Du 01/01/2008 au 01/11/2015** : Responsable de l'équipe Stockage HPC, Architecture des systèmes de stockage des centres de calcul, CEA-DAM
- **Du 01/06/1998 au 01/01/2008** : Administrateur/Développeur Système pour le Stockage HPC, CEA-DAM

Activités scientifiques et techniques

- **2004 :** Conception et développement du logiciel NFS-GANESHA (serveur NFS en user-space)
- **2009 :**Fondateur de la communauté open-source correspondante (lien GitHub : http ://nfs-ganesha.github.com/)
- **2005-2009 :** Encadrement d'une vingtaine de stagiaires de niveau M2 et fin d'études d'écoles d'ingénieur, suivi d'un post-doctorant sur la période 2005-2020

Participation a des projets européens et français

- **2021-2024 :** Coordinateur scientifique du projet IO-SEA (financé sur appel à projet EuroHPC)
- **2017-2021 :** Responsable du WP3 du projet SAGE2
- **2014-2017 :** Task Leader de WP3.2 dans SAGE
- **2017 :** : Task Leader de SP4.2 dans le projet IDIOM4 (Systèmes fortement couplés)

- **2002-Présent :** Membre du comité de lecture et du board des workshops Per3S (France) et WOPSSS (Europe)
- **2002-Présent :** Responsable de la Thématique 2 (Stockage de données) dans le cadre des conventions R&D entre le CEA et Atos/BULL

Coencadrement de thèses et participation à des jurys de thèse Dans le cadre de collaboration avec des universitaires, j'ai été co-encadrant de plusieurs thèses. À ce titre, j'ai fait partie des jurys de ces thèses.

- **2020** : Thèse d'Éloïse Billa (UVSQ, Laboratoire LiPaRaD) : Développement d'une méthode de distribution de métadonnée équilibrée pour un flux de requêtes exascale, directrice de thèse : Soraya Zertal (UPSaclay)
- **2019** : Thèse de Fotis Nikolaïidis (UVSQ, Laboratoire PriSM) : Tromos : a software development kit for virtual storage systems, directrice de thèse : Soraya Zertal (UVSQ)
- **2015** : Thèse de Jordan De La Houssaye (Evry I, labo IBISC) : Modèle de stockages distribués appliqué aux caches hiérarchiques , directeur de thèse Frank Pommereau (Evry I, IBISC)
- **2013** : Thèse de Joseph Emeras (INRIA Grenoble, équipe MESCAL) : Workload Traces Analysis and Replay in Large Scale Distributed Platforms , directeurs de thèse Yves Denneulin et Olivier Richard (CNRS/INRIA)

Présentations avec publication en ligne

- **11/2015** : Future of IO : a long and winding road (2015, présenté à l'OpenSource Summit, Paris, 2015, publication en ligne)
- **2014 et 2015** : Animation de BOF session dans les « Connectathons Events» qui regroupent les acteurs de la communauté NFS.
- **12/2014** : Working with FOSS Comunities at CEA (2014, présenté à l'OpenSourceSubmit, Paris, 2014, publication en ligne)

Publications

- **08/2020** : Workload Evaluation Tool for Metadata Distribution Method, EAI SIMULTool 2020, Éloïse Billa, Philippe Deniel et Soraya Zertal
- **2018** : Designing a parallel OGSSim through library specificities. SpringSim (TMS) 2018 : 10 :1-10 :12, Sebastien Gougeaud, Soraya Zertal, Jacques-Charles Lafoucriere, Philippe Deniel
- **12/2018** : Garantir la meilleure protection, Clef #67 (article de vulgarisation dans la revue interne du CEA), Philippe Deniel
- **06/2017** : Préserver l'intégrité des données, Clef #64 (article de vulgarisation dans la revue interne du CEA), Philippe Deniel
- **2017** : Optimizing Data Robustness in Large-Scale Storage Systems. HPCS 2017 : 236-243, Sébastien Gougeaud, Soraya Zertal, Jacques-Charles Lafoucriere, Philippe Deniel

- **2017** : Using ZeroMQ as communication/synchronization mechanisms for IO requests simulation. SPECTS 2017 : 1-8, Sebastien Gougeaud, Soraya Zertal, Jacques-Charles Lafoucriere, Philippe Deniel
- **2017** : Formal Modelling and Analysis of Distributed Storage Systems, Jordan de la Houssaye, Franck Pommereau, Philippe Deniel : Trans. Petri Nets Other Model. Concurr. 12 : 70-90 (2017)
- **2016** : Block shifting layout for efficient and robust large declustered storage systems. HPCS2016 : 342-349, Sébastien Gougeaud, Soraya Zertal, Jacques-Charles Lafoucriere, Philippe Deniel
- **2016** : A generic and open simulation tool for large multi-tiered hierarchical storage systems. SPECTS 2016 1-8, Sébastien Gougeaud, Soraya Zertal, Jacques-Charles Lafoucriere, Philippe Deniel
- **2016** : Formal Modelling and Analysis of Distributed Storage Systems.(Jordan de la Houssaye, Franck Pommereau, Philippe Deniel) : PNSE @ Petri Nets 2016 : 56-75
- **2015** : OGSSim : Open Generic data Storage systems Simulation tool", S. Gougeaud, S. Zertal, J-C. Lafoucrière and Ph. Deniel. EAI Endorsed Transactions on Scalable Information Systems, Vol 16, Num 9, ACM, 2015
- **2015** : OGSSim : open generic data storage systems simulation tool. SimuTools 2015 : 238-247, Sébastien Gougeaud, Soraya Zertal, Jacques-Charles Lafoucriere, Philippe Deniel
- **2013** : Contemporary High Performance Computing – From Petascale Towards Exascale / CRC Press (écriture de l'article sur le stockage de TERA100), Philippe Deniel
- **06/2007** : GANESHA, a NFSv4 Server running in user space with large cache, Proceedings du Linux Symposium 2007, juin 2007, Philippe Deniel
- **09/2006** : IPv6, une brève introduction, revue M.I.S.C n°27, septembre 2006, Philippe Deniel
- **01/2005** : Authentification dans les ONC/RPC, revue M.I.S.C n°17, janvier 2005, Philippe Deniel
- **07/2003** : Introduction à la GSSAPI, revue Linux Magazine n°52, juillet-août 2003 , Philippe Deniel
- **02/2007** : GANESHA, work in Progress, FAST07 (USENIX conference on File And Storage Technologies), février 2007, San José, CA, USA, Philippe Deniel
- **06/2007** : GANESHA, a NFSv4 Server running in user space with large cache, Linux Symposium 2007, juin 2007 Ottawa, Canada, Philippe Deniel et Thomas Leibovici

Annexe B : Les protocoles NFSv2, NFSv3 et leurs protocoles auxiliaires

Les spécifications des différentes versions du protocole NFS utilisées durant la période Terascale, et leurs protocoles auxiliaires, sont totalement décrites dans des documents nommés RFC et diffusés par l'Internet Engineering Task Force (IETF) , l'organisme en charge de standardiser les protocoles qui régissent les interactions entre ordinateurs, en particulier sur Internet. Cette version de NFS-Ganesha va implémenter les protocoles définis par les RFC suivantes :

- RFC1057 : *RPC : Remote Procedure Call Protocol Specification Version 2* [66]
- RFC1014 : *XDR : External Data Representation Standard* [65]
- RFC1094 : *NFS : Network File System Protocol Specification* [67]
- RFC1813 : *NFS Version 3 Protocol Specification* [86]

Les protocoles décrits dans ces documents interagissent avec d'autres protocoles des couches inférieures selon le schéma décrit à la figure B.1

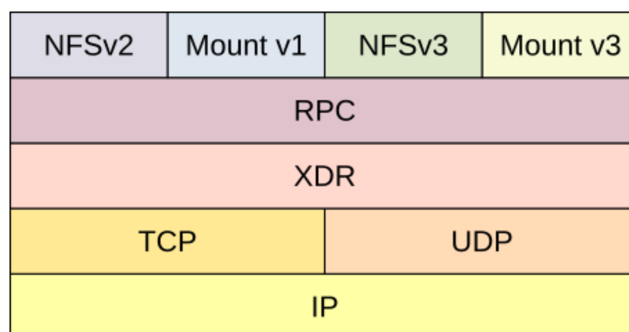


Figure B.1 – Pile de protocoles NFS de la période Terascale

B.1 .RPC et XDR

Le principe de fonctionnement de RPC est conçu pour faciliter l'intégration des appels clients-serveurs au sein d'une application : les interactions entre les acteurs connectés en réseau sont opérés au travers de fonctions très similaires aux fonctions qui constituent l'application. En particulier, chaque fonction a des arguments en entrée, potentiellement des arguments en sortie et un statut. Ces arguments sont souvent organisés selon des structures complexes, mais toujours analogues à la syntaxe du langage C [1].

1. la syntaxe reproduit notamment le comportement des mots clefs *struct* et *union* en langage C

Le protocole RPC s'appuie sur XDR qui est tout autant avant tout une façon de formater les données. Durant la période terascale, il est en effet très classique de trouver des machines dont la représentation interne et externe différent : si certains processeurs opèrent déjà en 64 bits, il est encore courant de trouver des machines qui fonctionnent sur 32 bits. Par ailleurs, certains constructeurs organisent les bits selon le format *little endian* quand d'autres favorisent le format *big endian*. XDR vient encoder les informations sur l'émetteur de telle sorte qu'elles seront décodées avec exactement la même signification sur le récepteur.

Ces informations encodées sont ensuite véhiculées sur le réseau via UDP et TCP. RPC fournit une totale abstraction de ces protocoles qui sont totalement transparents aux développeurs qui mettent en place une application en invoquant des fonctions basées sur ce protocole. Il faut noter que UDP est un protocole qui autorise la perte de paquets en cas de congestion, contrairement à TCP, mais il est toutefois encore très utilisé à cette époque. Les "bonnes pratiques" recommandent de réaliser les montages NFSv2 sur UDP, et ainsi de disposer d'une logique parfaitement sans état, comme nous le verrons dans le paragraphe suivant. L'avènement de NFSv3, puis de NFSv4 qui apportent des logiques à états, mettront fin à cet usage de UDP auquel on préfèrera toujours TCP.

B.2 .Le protocole NFS Version 2

NFSv2 est la première version publiée du protocole NFS [2].

Le protocole est structuré autour de 18 fonctions qui définissent les actions réalisables sur un système de fichiers. Ces fonctions sont listées dans le tableau B.1.

Le protocole respecte quelques règles dans son formalisme, que l'on retrouvera dans tous les protocoles opérant avec NFS, y compris dans les périodes postérieures :

- La première fonction du protocole, dont le numéro est O, est une action qui ne fait rien, sans argument ni code retour. Cette opération "PROC_NULL" permet au client de vérifier que le serveur est présent et qu'il est prêt à accepter des requêtes. Cette action, analogue à la commande shell *ping* qui envoie une trame Internet Control Message Protocol (ICMP) à une adresse IP pour tester sa présence, est utilisé en particulier par la commande shell *rpcinfo* ;
- les arguments en entrées sont réunis au sein d'une seule et même structure souvent dédiée à l'appel concerné. Par exemple, la fonction NFSPROC_READ prend un argument de type *readargs* ;

2. La version 1 ne franchira jamais les portes des laboratoires de Sun Microsystems, qui porte ce protocole

	Nom de la fonction	Action réalisée
0	NFSPROC_NULL	Action nulle
1	NFSPROC_GETATTR	Obtient les attributs
2	NFSPROC_SETATTR	Positionne les attributs
3	NFSPROC_ROOT	**(dépréciée)**
4	NFSPROC_LOOKUP	Descend dans une arborescence
5	NFSPROC_READLINK	Lit la valeur d'un lien symbolique
6	NFSPROC_READ	Lecture d'un fichier
7	NFSPROC_WRITECACHE	**(dépréciée)**
8	NFSPROC_WRITE	Écriture d'un fichier
9	NFSPROC_CREATE	Création d'un fichier
10	NFSPROC_REMOVE	Effacement d'une entrée non-répertoire
11	NFSPROC_RENAME	Renommage d'une entrée
12	NFSPROC_LINK	Création d'un lien
13	NFSPROC_SYMLINK	Création d'un lien symbolique
14	NFSPROC_MKDIR	Création d'un répertoire
15	NFSPROC_RMDIR	Effacement d'un répertoire
16	NFSPROC_READDIR	Lecture du contenu d'un répertoire
17	NFSPROC_STATFS	Récupération des statistiques

Table B.1 – Appels du protocole NFS Version 2

- en règle général, la même règle s'applique pour les valeurs en sortie de la fonction, sauf si les valeurs de sortie sont simples (un simple statut). Ainsi, la fonction NFSPROC_READ retourne une structure de type *readres* ;

Une entrée dans le système de fichiers est identifiée par un file handle [3], totalement opaque du point de vue du client.

Le fhandle NFS est totalement analogue dans son fonctionnement au FSAL Handle décrit au paragraphe 1.5.1, c'est d'ailleurs précisément le FSAL Handle qui y sera recopié, associé à quelques autres informations, pour le composer. De même que dans la FSAL, on va trouver un parcours d'arborescence qui respecte la même logique que la FSAL, avec usage itératif de LOOKUP et de READDIR. Il faut noter que NFSv2 n'implémente que les *readdir* simples, l'appel *readdirplus* n'y existe pas. On va retrouver dans NFSv2 la même logique de double type d'adressage indirect (par "parent et nom") et direct ("par handle") que dans le FSAL.

Les appels du protocole peuvent se regrouper ainsi (en dehors de PROC_NULL et des fonctions dépréciées) :

1. les fonctions READ et WRITE qui lisent et écrivent dans les fichiers et

3. noté fhandle dans le reste de la section

opèrent sur les contenus des fichiers ;

2. les appels qui changent l'état du système de fichiers
   (a) les appels qui créent de nouvelles entrées : CREATE, SYMLINK, MKDIR
   (b) les appels qui détruisent des entrées : REMOVE, RMDIR
   (c) la fonction RENAME qui renomme (et potentiellement déplace) une entrée
   (d) la fnncion LINK qui crée un nouveau link [4]
   (e) la fonction SETATTR qui modifie les attributs d'un fichier ;
3. les appels qui ne changent pas l'état du système de fichiers
   (a) la fonction READDIR qui liste les noms des entrées d'un répertoire ;
   (b) la fonction LOOKUP qui convertit une adressage parent et nom en fhandle
   (c) la fonction READLINK qui lit le contenu d'un lien symbolique
   (d) la fonction GETATTR qui lit les attributs d'une entrée
   (e) la fonction STATFS qui retourne des informations quant à l'utilisation du système de fichiers (nombres et proportions d'inodes utilisées, volume occupé)

B.2.1 . Le Mount Protocol

La logique basée sur des fhandles du protocole NFSv2 suppose qu'on dispose d'un fhandle qui identifie la racine de l'espace de nommage exporté. Le lecteur notera dans le tableau B.1 la présence de la fonction NFSPROC_ROOT() qui semble être un parfait candidat pour cela mais qui est dépréciée. En effet, la logique de la toute première version de NFS et qui n'est jamais sortie des laboratoires de Sun Microsystems, obtenait le fhandle de la racine grâce à cet appel. Il souffre cependant d'une forte limitation : on ne peut obtenir ainsi qu'un seul handle de racine, donc ne parcourir qu'un seul espace de nommage. Pour apporter une solution, le *Mount Protocol* a été introduit.

Il s'agit d'un protocole auxiliaire, indépendant de NFS, mais généralement implémenté au sein du même serveur qui expose donc les deux services. Les fonctions du Mount Protocol V1 (qui est adossé à NFSv2) sont décrites dans le tableau B.2. Les fonctions principales sont MOUNTPROC_MNT() et MOUNTPROC_UMNT(), la première permettant à la fois de traduire un chemin absolu en fhandle à partir duquel l'espace de nommage exporté peut être parcouru et qui réserve également des ressources pour assurer cette gestion (ce que l'on appelle une *export entry*) ; la seconde est utilisée par le client pour indiquer au serveur qu'il cesse de s'intéresser à cette arborescence et qu'il peut

4. Il s'agit d'un lien hard donc qui ne crée pas d'inode, contraitement au lien symbloque créé par SYMLINK

	Nom de la fonction	Action réalisée
0	MOUNTPROC_NULL	Action nulle
1	MOUNTPROC_MNT	Traduit un chemin en fhandle
2	MOUNTPROC_DUMP	Retourne la liste des montages
3	MOUNTPROC_UMNT	Démonte une arborescence
4	MOUNTPROC_UMNTALL	Démonte tous les montages gérés
5	MOUNTPROC_EXPORT	Retourne les chemins montables

Table B.2 – Appels du protocole Mount Protocol v1

libérer les ressources associées dans l'exportlist. Ce formalisme, que l'on retrouve dans les commandes d'administration Unix *mount* et *unmount* est très flexible, car il autorise le montage d'une sous-arborescence, avec des droits d'accès différents (généralement plus restrictifs) en même temps que l'arborescence principale. Les deux points de montage, qu'ils soient gérés par le même client ou non, manipuleront les mêmes fhandles ce qui garantira la cohérence des actions menées.

On remarque que le protocole Mountv1 ajoute des fonctions qui sont spécifiquement conçus pour aider à l'administration des points de montage, en plus de MOUNTPROC_MNT() et MOUNTPROC_UMNT() :

- MOUNTPROC_DUMP() permet de connaître toutes les montages actuellement gérés par un serveur, ce qui correspond à toutes les fois où MOUNTPROC_MNT() a été invoquée avec succès pour résoudre un chemin et le traduire en fhandle, créant ainsi une export entry
- MOUNTPROC_EXPORT() retourne toutes les entrées décrites dans le fichier de configuration du serveur et qui sont donc des arguments licites pour MOUNTPROC_MNT()
- MOUNTPROC_UMNTALL() démonte toutes les export entries connues du serveur, cet appel n'est généralement utilisé par aucun client.

B.2.2 . Limitations structurelles de NFSv2

Le protocole NFSv2 souffre de multiples limitations qui rendent difficile à maintenir en exploitation. La plus forte d'entre elle est sa syntaxe lest intriquée avec des structures 32 bits. Via NFSv2, il est impossible de gérer les fichiers dont la taille dépasse 2^{31} octets, soit 2 giga-octets : en effet, la structure Numéro qui décrit un offset dans un fichier est un entier signé sur 32 bits, il est impossible d'effectuer des lectures et des écritures de fichiers au-delà de cette barrière. La taille des fichiers est quant à elle gérée sur 32 bits non signés, cela conduit à des erreurs. La gestion de cette taille (par exemple celle affichée par la commande *ls*) dès lors qu'un fichier plus volumineux que 4 giga-octets. La taille du fhandle subit, elle aussi, une restriction du même type, cette structure ne doit pas dépasser la taille de 32 octets, ce qui peut être rapidement limita-

tif pour mettre en œuvre des back-ends sophistiqués dont les FSAL Handles seront volumineux.

Une autre limitation importante est présente et moins évidente à déceler : la nature profondément "sans état" ou *stateless* du protocole. Sur le principe, cette propriété du protocole, qui est clairement un choix de ses concepteurs, peut présenter des avantages. Notamment, elle favorise l'usage de UDP/IP comme couche de transport de RPC, un protocole qui est plus simple que TCP/IP et qui sait à l'époque gérer plus efficacement les épisodes de congestion réseau. Cette logique "sans état" permet de ne pas conserver d'état persistent dans le serveur lui-même, les arguments des fonctions et les valeurs en sortie des fonctions du protocole suffisent à eux-seuls pour gérer les états nécessaires aux modifications opérées sur l'espace de nommage exporté.

La prise en compte d'un arrêt brutal du serveur, qu'il ait des causes matérielles (comme un reboot de la machine) ou logicielles (crash du service, suivi d'un redémarrage, est simplifié : la nouvelle instance se comportera précisément comme son prédécesseur. ce mode de fonctionnement est efficace quand il y a un faible volume de requêtes et peu de clients, et si les clients s'intéressent chacun à des portions différentes de l'espace de nommage. Le cas d'usage principal de NFSv2 est celui d'un réseau qui héberge des postes de bureautique dont les *home directories*, les répertoires qui contiennent les données des utilisateurs, sont contenus dans un serveur central qui utilise NFSv2.

Dans cette situation, on aura un utilisateur par poste de travail, chacun étant un client du serveur NFSv2, et chacun accueille un utilisateur dédié. Cet utilisateur va s'intéresser à ses propres données, généralement pas à celle de ses collègues et les requêtes dans les zones de l'espace de nommage seront bien isolés entre les clients. Cette situation, qui force les clients à conserver trace des états puisque le serveur ne les gère pas, force ceux-ci à cacher les informations sans donner les outils pour les maintenir cohérents dans le cas de fichiers manipulés par plusieurs clients. Cette situation, qui prend tout son sens dans un contexte de bureautique, atteint ses limites dans le cas d'une utilisation HPC où de très nombreux clients viennent manipuler les mêmes entrées concurremment, ce qui produit de nombreux problèmes techniques liés à des incohérences qui apparaissent dans les caches.

Pour ces raisons, NFSv2 sera peu utilisé dans le centre de calcul et on basculera vite sur le protocole NFSv3 qui apporte des réponses propres à ce domaine.

B.2.3 . Le protocole NFS Version 3

Le protocole NFSv3 apporte quelques fonctionnalités supplémentaires au protocole NFSv2 et corrige certains défauts bien connus de NFSv2 et décrits au paragraphe précédent. Les fonctions du protocole sont décrites dans le

tableau B.3. Les nouvelles fonctions ajoutées au protocole y sont indiquées en **gras**.

	Nom de la fonction	Action réalisée
0	NFSPROC3_NULL	Action nulle
1	NFSPROC3_GETATTR	Obtient les attributs
2	NFSPROC3_SETATTR	Positionne les attributs
3	NFSPROC3_LOOKUP	Descend dans une arborescence
4	NFSPROC3_ACCESS	Vérification des droits d'accès
5	NFSPROC3_READLINK	Positionne les attributs
6	NFSPROC3_READ	Lecture d'un fichier
7	NFSPROC3_WRITE	Écriture d'un fichier
8	NFSPROC3_CREATE	Création d'un fichier
9	NFSPROC3_MKDIR	Création d'un répertoire
10	NFSPROC3_SYMLINK	Création d'un lien symbolique
11	NFSPROC3_MKNOD	Création d'un fichier spécial
12	NFSPROC3_REMOVE	Effacement d'une entrée non-répertoire
13	NFSPROC3_RMDIR	Effacement d'un répertoire
14	NFSPROC3_RENAME	Renommage d'une entrée
15	NFSPROC3_LINK	Création d'un lien
16	NFSPROC3_READDIR	Lecture d'un répertoire
17	NFSPROC3_READDIRPLUS	Lecture d'un répertoire
18	NFSPROC3_FSSTAT	Récupération des informations dynamiques
19	NFSPROC3_FSINFO	Récupération des informations statiques
20	NFSPROC3_PATHCONF	Implémente *pathconf()*
21	NFSPROC3_COMMIT	Acquitte une action asynchrone

Table B.3 – Appels du protocole NFS Version 3

Au chapitre des nouveautés, NFSv3 apportent bien des fonctionnalités qui faisaient défaut à NFSv2 et qui trouvent tout leur intérêt dans un contexte HPC.

Le support total de l'adressage du contenu des fichiers en 64 bits est la première d'entre elles. Il devient dès lors possible de gérer des fichiers dont la taille peut atteindre 2^{63} octets soit 9223 peta-octets [5] ou 9 exa-octets [6], des valeurs très au delà des exigences d'exploitation.

La taille du fhandle passe de 32 octets à 64 octets, ce qui facilite grandement l'encodage de ceux-ci.

5. 1 peta-octets = 10^{15} octets
6. 1 exa-octets = 10^{18}. octets

Le protocole NFSv3 prend en compte une notion d'états dans le serveur, mais sans toutefois abandonner totalement la logique sans état de NFSv2. Les appels à la fonction NFSPROC3_WRITE() peuvent à présent être réalisé dans ce que les spécifications du protocole[86] désignent sous le terme de *unstable storage*. Les buffers ainsi écrits ne sont déplacés en *persistent storage* qu'une fois réalisée une action de *commit* via la fonction NFSPROC3_COMMIT(). Cette nouvelle fonctionnalité permet la mise en œuvre de cache de données qui exploitent la RAM, vue comme du *unstable storage* ce qui permet un accroissement des performances notables. Comme indiqué, l'abandon de la logique sans état n'est pas total, car c'est au client de gérer le déplacement des données vers le *persistent storage*, la gestion de l'état est assurée par le client une nouvelle fois, ce qui facilite la gestion de la reprise sur panne.

On note enfin l'arrivée plus que bienvenue de la fonction READDIRPLUS() qui sera très rapidement préférée à NFSPROC3_READDIR() qui reprend le fonctionnement de son homonyme dans le protocole NFSv2. Dans NFS-Ganesha, l'implémentation de cette fonction tire grand bénéfice du comportement de la fonction FSAL_readdir() qui retourne les attributs et les handles des entrées contenus dans un répertoire en même temps que leurs noms.

La fonction NFSPROC3_ACCESS() est une nouveauté dont NFS-Ganesha tire grandement partie : en effet, cette fonction permet de vérifier si un utilisateur a le droit d'effectuer une action sur le système de fichiers avant de réellement la faire. NFS-Ganesha dispose d'un cache de méta-données qui contient toutes les informations nécessaires pour réaliser cette action. Cette fonction ne sollicitera donc pas les appels de la FSAL, apportant une optimisation des plus efficaces.

La nouvelle fonction NFSPROC3_PATHCONF() permet d'obtenir des informations relatives au comportement du système de fichiers vis-à-vis des fichiers, en retournant des informations telles que le nombre maximum de hardlinks autorisé, la taille maximum d'un nom de fichier ou d'un chemin de fichier, le caractère restreint de l'appel *chown()* qui permet de changer le propriétaire d'un fichier ou le fait de pouvoir ou non tronquer un fichier. Ces informations sont traditionnellement retournées par la fonction *pathconf()* de la libC dont cette fonction du protocole NFSv3 reprend la logique.

Enfin, on note l'apparition de la fonction NFSPROC3_MKNOD() qui permet la gestion au travers de NFS des fichiers spéciaux, tels que les *block files* ou les *character files* que l'on trouve classiquement dans le répertoire */dev* d'une machine sous Unix. Les FSALs de NFS-Ganesha alors implémentées ne disposent pas de tels fichiers, l'implémentation de cette fonction sera donc minimaliste, retournant simplement une erreur qui indique qu'elle n'est pas supportés

B.2.4 . Le Mount Protocol v3

Le Mount Protocol évolue lui aussi pour s'adapter à NFSv3. Ses appels sont référencés dans le tableau B.4

	Nom de la fonction	Action réalisée
0	MOUNTPROC3_NULL	Action nulle
1	MOUNTPROC3_MNT	Traduit un chemin en fhandle
2	MOUNTPRO3_DUMP	Retourne la liste des montages
3	MOUNTPROC_UMNT3	Effectue une opération de démontage
4	MOUNTPROC_UMNTALL3	Démonte tous les montages gérés
5	MOUNTPROC_EXPORT3	Retourne la liste chemins montables

Table B.4 – Appels du protocole Mount Protocol v3

Comme pourra le constater le lecteur, les appels sont très similaires à ceux du protocole Mountv1 adossé à NFSv2. L'ajout majeur de cette version 3 est la prise en compte de la nouvelle taille des fhanles dans NFSv3, qui passe à 64 bits.

B.2.5 . Autres protocoles auxiliaires

D'autres protocoles sont adossés à NFSv3, mais ils ne sont pas disponibles dans la version de NFS-Ganesha développée durant la période terascale. Ces protocoles sont :

- **NLM** : *NFS Lock manager protocol*, décrit en appendice de RFC1813 ;
- **STATUS** Ce protocole est associé au service *rpc.statd*, il permet de connaître les états des autres services du système. C'est indispensable pour gérer correctement les situations de reprise sur panne.

Le protocole NLM permet de gérer des verrous posés sur les fichiers. Ce protocole suit une logique *stateful*, à états, contrairement à NFSv3 qu'il vient compléter. La gestion propre d'un tel service suppose de gérer les situations où le serveur est amené à redémarrer, afin d'assurer une reprise correcte des états sur pannes. Sous Linux, c'est le daemon *rpc.statd* qui se charge de savoir quand les autres services ont démarré et quand ils se sont arrêtés brutalement. Le *rpc.statd* met en œuvre le protocole Network State Management (NSM) qui permet à un service de s'enregistrer à son démarrage afin de publier sur l'ensemble du système sa date et son heure de naissance. Une implémentation correcte d'un serveur supportant NLM ne peut se faire sans que le serveur en question soit client NSM.

Les FSALs supportées durant la période terascale ne supporte pas les verrous (HPSS en particulier) les appels de la FSAL qui correspondent resteront donc des coquilles vides et ni NLM ni la partie client de NSM ne seront dé-

ployés. Les chapitres suivants, couvrant les périodes postérieures, montreront comment cette situation sera appelée à évoluer.

Annexe C : Les multiples RFC de NFSv4 et les versions mineures

Du temps de NFSv2 et NFSv3, les choses étaient relativement simples sur le plan bibliographique : chaque protocole était décrit dans un seul et unique document bien identifié qui se suffisait à lui seul. Ce document restait d'une taille relativement modeste, par exemple RFC1813 ne compte qu'une grosse centaine de pages.

Les choses deviennent plus complexes avec NFSv4. En premier lieu, les documents sont considérablement plus volumineux : il faudra quelque 200 à 300 pages pour définir NFSv4 et plus de 600 pour NFSv4.1, ces documents sont souvent complétés par d'autres qui apportent des précisions sur des protocoles auxiliaires. Bien plus qu'une simple évolution de NFSv3, NFSv4 est un protocole totalement nouveau qui s'inscrit dans l'esprit de NFS. C'est également une logique assez complexe à mettre en œuvre, ce qui conduira à la découverte de petites incohérences internes qui rendent impossible le développement dans des produits de certaines fonctionnalités. Les RFCs de NFSv4 seront ainsi amendées.

D'autre part, l'ajout tardif de nouvelles fonctionnalités majeures dans NFSv4, telle que l'introduction du parallélisme avec pNFS, conduira à l'émergence de versions mineures du protocole. C'est ainsi que RFC5661[73] verra le jour. Plus mûr, car bénéficiant des retours sur l'implémentation de NFSv4, maintenant souvent désigné comme NFSv4.0, cette nouvelle mouture remporte rapidement l'adhésion des communautés de développeurs systèmes qui trouve ici un cadre plus simple et plus efficace.

Ultérieurement, l'ajout de nouvelles fonctionnalités importantes (copie en tierce partie, support des attributs étendus) donnera naissance à la sous version NFSv4.2.

Afin de se retrouver dans ces nombreux documents, il convient de faire le point et de décrire la variété de la bibliographie disponible et de préciser le contenu de chacun d'entre eux.

Les documents qui définissent NFSv4 (dans la version mineure initiale) sont :

- RFC3010 : *NFS version 4 Protocol* [14]
- RFC3530 : *Network File System (NFS) version 4 Protocol* [15]
- RFC7530 : *Network File System (NFS) version 4 Protocol* [47]

La première version a donc été amendée avant que celle-ci soit à son tour amendée pour aboutir à la version définitive. Ce processus est un travail de longue haleine, quinze ans séparent RFC3010 (publiée en 2000) de RFC7530 (publiée en 2015).

Dans la suite document, NFSv4.0 désignera le protocole NFSv4 initial, ini-

tié par RFC3010 et entériné par RFC7530, afin d'éviter les confusions avec les versions mineures. On réservera l'acronyme NFSv4 au protocole NFS version 4 en général, il englobe toutes les sous versions du protocole.

La première version mineure de NFSv4, sobrement nommée NFSv4.1, voit le jour en 2010, dix ans après NFSv4.0. Outre une simplification majeure du fonctionnement de NFSv4, grâce à une gestion plus simple des séquences d'opérations, cette version mineure apporte la fonctionnalité pNFS. Elle sera détaillée dans les paragraphes D.6.1 et D.6.3.

NFSv4.1 est défini par un premier document très volumineux (RFC5661 compte 617 pages), complété par différents documents qui viennent préciser certains points particuliers, souvent associés à différentes déclinaisons de pNFS. Le protocole subira le même type de corrections et d'amendements que NFSv4.0, on trouvera donc plusieurs RFCs, chacune rendant obsolète la version d'avant. noté Les documents qui définissent NFSv4.1, incluant sa fonctionnalité pNFS, sont :

- RFC5661 : *Network File System (NFS) Version 4 Minor Version 1 Protocol* [73]
- RFC5662 : *Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description* [72]
- RFC5663 : *Parallel NFS (pNFS) Block/Volume Layout* [43]
- RFC5664 : *Object-Based Parallel NFS (pNFS) Operations* [96]
- RFC5665 : *IANA Considerations for Remote Procedure Call (RPC) Network Identifiers and Universal Address Formats* [39]
- RFC8881 : *Network File System (NFS) Version 4 Minor Version 1 Protocol* [74]

Le protocole NFSv4.2, que nous évoquera brièvement au paragraphe D.6.4 et qui n'a que peu pris en compte par NFS-Ganesha est défini par les documents suivants :

- RFC7862 : *Network File System (NFS) Version 4 Minor Version 2 Protocol* [91]
- RFC7863 : *Network File System (NFS) Version 4 Minor Version 2 External Data Representation Standard (XDR) Description* [46]

Cette longue liste, qui fait un peu figure de "liste à la Prévert" ne serait toutefois pas complète sans mentionner d'autres documents qui tous vont avoir des conséquences sur les versions successives de NFS-Ganesha durant la période petascale. Ils couvent en particulier des évolutions liées au protocole RPC, qui devient RPCSEC_GSS et sera capable de gérer la sécurité fournie par le service Kerberos v5 et le formalisme de la GSSAPI, ainsi que la prise en compte du nouveau paradigme réseau que constitue RDMA :

- RFC2203 : *RPCSEC_GSS Protocol Specification* [40]
- RFC5403 : *RPCSEC_GSS Version 2* [38]
- RFC7861 : *Remote Procedure Call (RPC) Security Version 3* [2]
- RFC1510 : *The Kerberos Network Authentication Service (V5)* [71]
- RFC4120 : *The Kerberos Network Authentication Service (V5)* [70]
- RFC6649 : *Deprecate DES, RC4-HMAC-EXP, and Other Weak Cryptographic*

Algorithms in Kerberos [8]

- RFC1964 : *The Kerberos Version 5 GSS-API Mechanism* [59]
- RFC4121 : *he Kerberos Version 5 Generic Security Service Application Program Interface* [70]
- RFC5666 : *Remote Direct Memory Access Transport for Remote Procedure Call* [90]
- RFC5667 : *Network File System (NFS) Direct Data Placement* [89]

Annexe D : Le protocole NFSv4

Cette section décrit le protocole NFSv4 dans ses grandes lignes. Il sera très rapidement implémenté au sein de NFS-Ganesha qui deviendra le premier serveur NFS open-source à le mettre en œuvre en dehors de l'implémentation réalisée dans le noyau Linux.

D'une manière très simpliste, on peut regrouper les différences entre NFSv4.0 et NFSv3 sur quatre axes :

1. la sémantique du protocole, qui s'appuie des arrangements d'opérations très simples dans des requêtes composées ;
2. la gestion des espaces de nommage exportés qui permet à NFSv4.0 de s'affranchir du Mount Protocol qui disparaît ;
3. la prise en compte des états (fichiers ouverts, verrous...,), contrairement à NFSv3 qui avait une logique sans état, NFSv4.0 est fondamentalement stateful et intègre tous les mécanismes indispensables à la gestion de ces états ;
4. les verrous bloquants, NFSv4.0 s'affranchit du protocole auxiliaire NLM et fournit toutes les primitives qui permettent l'implémentation de tels états spéciaux. Il ajoute aussi le mécanisme de délégation, souvent adossé à celui de verrou, même si les deux notions ne se recouvrent pas.

D.1 .Les requêtes composées

Les fonctions du protocole NFSv4 sont décrites dans le tableau D.1. Il est en apparence étonnement simple en comparaison de NFSv3 et NFSv2 décrits dans les tableaux B.3 et B.1.

	Nom de la fonction	Action réalisée
0	NFSPROC4_NULL	Action nulle
1	NFSPROC4_COMPOUND	Requête composée

Table D.1 – Appels du protocole NFS Version 4

A première vue, ce protocole se résume à "ne rien faire" (avec NFSPROC4_NULL()) et "faire quelque chose" (via NFSPROC4_COMPOUND()à), mais la situation est plus complexe.

Une requête composée est en fait une liste d'actions à effectuer, chacune décrite par un "opération" dont la liste exhaustive est décrite dans le tableau D.2. Cette liste est longue et elle diffère profondément de la sémantique de NFSv3.

Nom de l'opération	Action réalisée
OP_ACCESS	Test si une action est possible
OP_CLOSE	Ferme le fichier courant ouvert
OP_COMMIT	Flush les données en stockage permanent
OP_CREATE	Crée un nouveau fichier
OP_DELEGPURGE	Purge une délégation
OP_DELEGRETURN	Retourne une délégation
OP_GETATTR	Obtient les attributs du FH courant
OP_GETFH	Retourne le handle courant
OP_LINK	Crée un nouveau lien
OP_LOCK	Pose un verrou sur un fichier
OP_LOCKT	Teste une action sur un verrou
OP_LOCKU	Relâche un verrou
OP_LOOKUP	Effectue un lookup depuis le FH courant
OP_LOOKUPP	Obtient le FH parent du FH courant
OP_NVERIFY	Vérifie si des attributs sont différents
OP_OPEN	Ouvre un fichier
OP_OPENATTR	Obtient le FH du "répertoire d'attributs"
OP_OPEN_CONFIRM	Confirme un "lock owner"
OP_OPEN_DOWNGRADE	Dégrade le mode d'ouverture d'un fichier
OP_PUTFH	Positionne le FH courant
OP_PUTPUBFH	Positionne le "Public FH"
OP_PUTROOTFH	Place le handle de la racine dans le FH courant
OP_READ	Lecture
OP_READDIR	Lecture du contenu d'un répertoire
OP_READLINK	Lecture du contenu d'un lien symbolique
OP_REMOVE	Détruit une entrée (tous types)
OP_RENAME	Renomme une entrée (tous types)
OP_RENEW	Renouvelle une lease adossée à un état
OP_RESTOREFH	Le "Saved FH" devient le FH courant
OP_SAVEFH	Recopie le FH courant dans le "Saved FH"
OP_SECINFO	Permet d'obtenir les options de sécurité RPC
OP_SETATTR	Positionne les attributs sur le FH courant
OP_SETCLIENTID	Positionne la Client ID
OP_SETCLIENTID_CONFIRM	Confirme une Client ID
OP_VERIFY	Vérifie si des attributs sont identiques
OP_WRITE	Écriture

Table D.2 – Opérations du protocole NFS Version 4.0

Pour bien comprendre le fonctionnement de ces opérations, il faut considérer que chaque requête composée est exécutée dans un contexte ou existent

plusieurs variables :

- **CURRENT_FH** est le fhandle courant
- **SAVED_FH** est le fhandle sauvegardé

Les opérations viennent agir sur ces variables : par exemple, OP_LOOKUP cherchera à résoudre un nom à partir du fhandle contenu dans CURRENT_FH, et stockera le résultat dans CURRENT_FH si cette opération est un succès, de même OP_CREATE créera une nouvelle entrée dans le répertoire pointé par CURRENT_FH et stockera le résultat dans CURRENT_FH.

Les autres variables sont utilisés par d'autres opérations, ainsi OP_LINK créera un lien dans le répertoire cible dont le file handle est contenu dans SAVED_FH, OP_RENAME désignera aussi son répertoire destination grâce à SAVED_FH.

Le CURRENT_FH peut être recopié dans le SAVED_FH par l'opération OP_SAVEFH et être restauré par OP_RESTOREFH. Enfin, l'opération OP_PUTFH permet de positionner le CURRENT_FH tandis que OP_GETFH permet de lire sa valeur (par exemple pour la retourner vers le client).

Les requêtes composées sont des tableaux d'opérations qui listent les opérations à effectuer pour obtenir le résultat souhaité. Par exemple, le LOOKUP de NFSv2 sera équivalent à la requête suivante :

```
PUTFH   (directory filehandle)
LOOKUP  (entry name)
GETFH
```

Pour le LOOKUP de NFSv3, qui retourne en plus les attributs du fichier afin de peupler le cache de métadonnées du client, la requête sera alors celle-ci

```
PUTFH   (directory filehandle)
LOOKUP  (entry name)
GETATTR
GETFH
```

La logique des requêtes composantes est extrêmement puissante et flexible, elle permet en particulier de s'affranchir de la sémantique forte de POSIX, même s'il est très simple de reproduire celui-ci en transcrivant, dans ce formalisme, le comportement des requêtes du protocole NFSv3. NFSv4 est en effet conçu pour exister en dehors du monde Unix et Linux. Par exemple, les systèmes de fichiers du monde Linux, conformes à POSIX, n'ont pas la même sémantique que ceux du monde Windows. Ceux-ci ne peuvent pas être exportés par NFSv3, mais cela est parfaitement envisageable avec NFSv4.

Lister en détail les opérations de NFSv4.0 serait un exercice des plus fastidieux, et qui n'apporterait que peu de nouveaux éléments. Pour le lecteur désormais familier avec NFSv2, NFSv3 et le formalisme de la FSAL, il est désormais évident, à titre d'exemple, que READLINK va lire le contenu d'un lien

symbolique. Dans la suite, on se contentera donc de faire la lumière sur certaines fonctionnalités nouvelles de NFSv4.0

D.2 .Les opérations de validation

NFSv4 s'inscrit en droite ligne de NFSv3, le serveur considère que le client va en particulier cacher des métadonnées pour optimiser certains traitements et le protocole lui offre la possibilité de faire cela de manière encore plus efficace par deux opérations, VERIFY et NVERIFY. Toutes deux présentes des attributs en arguments et utilisent le CURRENT_FH. VERIFY va vérifier si les attributs de l'entrée pointée par CURRENT_FH sont les mêmes que ceux présentés en arguments tandis que VERIFY va vérifier s'ils sont différents. NVERIFY est donc un "not VERIFY". Des erreurs spécifiques du protocole, NFS4ERR_NOT_SAME et NFS4ERR_SAME sont dédiés à ces deux opérations.

Par exemple, OP_VERIFY peut être utilisé dans la requête suivante qui va venir s'assurer que l'entrée que l'on est sur le point de supprimer est bien la bonne[1] :

```
PUTFH (directory filehandle)
LOOKUP (file name)
VERIFY (filehandle == fh)
PUTFH (directory filehandle)
REMOVE (file name)
```

Si cette requête rentre en concurrence avec une autre qui remplace le fichier par un autre (par exemple en enchainant un REMOVE et un CREATE), cette requête détectera l'anomalie et sera à même de produire une erreur pour en informer le client. Dans le cas de la requête ci-dessus, si cette situation se produit, l'erreur retournée par VERIFY à la troisième ligne arrêtera le traitement de la requête et le fichier ne sera pas effacé.

D.3 .Les traversées de jonctions

Les jonctions sont un mécanisme qui permet de "chaîner" différents serveurs NFS les uns derrière les autres. Supposons qu'un espace de nommage contienne un répertoire /remote/nfs dont chaque répertoire est un point de montage d'un autre serveur NFS, par exemple le répertoire /remote/nfs/serveur1_home est le montage NFS de /home sur la machine1. En NFSv3, le fait de faire un lookup depuis le répertoire /remote/nfs vers /remote/nfs/serveur1_home va soit produire une erreur soit forcer le premier serveur NFS a produire des requêtes NFS vers la machine serveur1. Cette machine verra des requêtes qui

1. NFSv4 compte le file handle parmi les attributs d'une entrée, le fhandle peut donc être exploité par VERIFY et NVERIFY

émanent du premier serveur NFS, pas du client.

NFSv4 gère la situation plus proprement en introduisant la notion de jonction qui est empruntée au système de fichier distribué Andrew File System (AFS) [100], très populaire dans les années 90. Si l'on reprend l'exemple précédent, la traversée de /remote/nfs vers /remote/nfs/serveur1_home va déclencher une erreur spéciale, NFS4ERR_MOVED. Le client va comprendre qu'il est en train de traverser une jonction et va faire une requête contenant GETATTR sur l'attribut spécial *fs_locations*. Le serveur va alors retourner une structure qui permettra au client de discuter avec le second serveur NFS, donc la machine serveur1 et de faire toutes les négociations client/serveur nécessaires. Par la suite, le client enverra ses requêtes directement vers serveur1 sans surcharger inutilement le premier serveur NFS.

D.4 . La gestion des exports et la disparition du Mount Protocol

L'utilisation du Mount Protocol de NFSv2 et NFSv3 est source de bien des difficultés pour les administrateurs systèmes. En effet, si NFS utilise toujours le même port, normalisé dans la description du protocole, et qui porte le numéro 2049. Le Mount Protocol, que ce soit dans sa version v1 ou v3, s'appuiera sur un port éphémère, le daemon qui implémente ces protocoles utilisera le premier port disponible à partir de 1024. Si le trafic doit traverser un parefeu, il est facile de discriminer le trafic associé à NFS mais pas celui du Mount Protocol. Le "montage NFS" qui associe ces deux protocoles est alors difficile à réaliser. Par ailleurs, Mount Protocol est un protocole peu sécurisé, ce qui soulève des problèmes liés à la sécurité informatique. NFSv4 apporte une solution radicale à ce problème : il supprime le Mount Protocol (et l'ensemble des protocoles auxiliaires).

Comment est opérée la gestion des exports dans ce cas-là ? Afin d'exposer le mécanisme mis en jeu, on prendra l'exemple d'un serveur NFS qui exporte deux espaces de nommage, */home* et */backups*. Dans le cas de NFSv3, le client appellera la fonction MOUNT du Mount Protocole en lui donnant un chemin en argument, par exemple, /home/users/deniel, le serveur le comparera aux exports qu'il sait gérer (donc /home et /backups) et verra qu'il s'agit d'un sous-répertoire de /home, il autorisera ou non l'accès et construira le file handle qui correspond et le retournera au client qui pourra commencer à faire des requêtes purement NFSv3 à partir de ce point.

NFSv4 va fonctionner différemment, il va construire un *pseudo système de fichiers*, noté pseudofs dans la suite du document. Celui-ci sera en lecture seul et ne contiendra que des répertoires. Chacun de ces répertoires se comportera comme une jonction décrite au paragraphe précédent, quand on les traversera on aboutira à l'intérieur du véritable espace de nommage exporté. Revenons à notre exemple, dans lequel un client cherche à monter /home/users/deniel sur un serveur qui exporte /home et /backups : le client va

construire une requête qui commencera par l'opération PUTROOTFH, celle-ci va stocker dans le CURRENT_FH le file handle qui adresse la racine du pseudofs, il va construire une série de lookups à partir de ce point. Par exemple, le client pourra réaliser la requête composée suivante :

```
PUTROOTFH            -- Obtention de la racine
                        du pseudofs
LOOKUP (home)        -- Passage de jonction /home
LOOKUP (users)       -- deux lookups successif
                        dans l'export
LOOKUP (deniel)
GETFH                -- Retour du file handle vers
                        le client
```

Ce mode de fonctionnement s'appuie totalement dans la logique des requêtes composées d'opérations et est par construction très flexible.

D.5 .Prise en compte de RPCSEC_GSS dans NFSv4

Dans le cas de NFSv4, la prise en compte des différents types de sécurité est bien plus simple et générique que pour NfSv3 puisque le Mount Protocol a disparu et qu'un seul protocole rentre en jeu. En théorie, il est possible de gérer cette distinction au niveau même des entrées du système de fichier. Ainsi, rien n'empêche, si l'on considère deux fichiers dans un même répertoire, de rendre obligatoire l'utilisation de RPCSEC_GSS pour le premier et de rendre celle-ci inutile ou optionnelle pour le second. C'est la fonction OP_SECINFO, présente dans toutes les versions mineures de NFSv4 qui va, sur la base d'un fhandle envoyé en argument, permettre à un client de savoir quel type de RPC, sécurisées ou non, doivent être mise en œuvre.

En pratique, les choses sont bien plus simples et aucun serveur n'exploite actuellement la versatilité autorisée par les spécifications de NFSv4. le schéma retenu sera identique à celui qui prévaut pour NFSv3 et l'opération SECINFO ne sera réalisée que sur le file handle du répertoire racine de l'espace de nommage, à l'emplacement de la jonction entre le pseudo système de fichiers au travers duquel NFSv4 gère ses montages et les systèmes de fichier exporté.

D.6 .La gestion des états

NFSv4.0 est un protocole fondamentalement *stateful* qui intègre la gestion des états contrairement à NFSv2 et dans une moindre mesure NFSv3. Ces états sont de trois types :

- les réservations partagées, qui correspondent à des fichiers ouverts,
- les verrous de fichiers,
- les délégations.

Pour gérer ces états, il faut identifier le propriétaire de l'état, lequel va au final identifier le thread d'un processus sur une machine client qui effectue une action sur le montage NFS. Plusieurs structures sont utilisées pour cela dans le protocole NFSv4.0 :

- la **clientid** qui identifie la machine client;
- la **stateid** qui identifie l'état sur la machine client;
- la **seqid** ou identifiant de séquence, qui permet de suivre l'évolution d'un état.

Un état est donc identifié par un couple `(clientid, stateid)` et il lui est associé une `seqid` qui est incrémentée à chaque évolution de l'état. Ce dernier identifiant les transitions d'un état dans les requêtes NFS et permet de traiter celles-ci dans l'ordre. En effet, pour des raisons de contention réseau, certaines transitions pourraient arriver dans le mauvais ordre.

La clientid va être négocié entre chaque client et le serveur, au travers des opérations SETCLIENTID et SETCLIENTID_CONFIRM.

L'opération SETCLIENTID_CONFIRM permet de gérer les cas où le client viendrait à réémettre l'opération OPEN, en cas de congestion réseau par exemple. La négociation de clientid, initiée par SETCLIENTID doit donc être confirmée par SETCLIENTID_CONFIRM. On construit un schéma de négociation en trois étapes qui est, dans l'idée, très similaire du *three ways handshake* à la base de l'établissement d'une connexion dans le protocole TCP.

La stateid est un argument sur toutes opérations qui modifient l'état d'un fichier, cela inclut en particulier les I/O effectuées au travers de OP_WRITE et OP_READ. En effet, si la notion de "fichier ouvert" est absente de NFSv2 et NFSv3 (elle n'existe que sur le client, via la VFS), celle-ci est omniprésente sur le serveur NFSv4.0. Il faut noter que les opérations REMOVE et RENAME sont instrumentées par des stateids [2], de même que SETATTR qui peut tronquer un fichier (en positionnant l'attribut "taille de fichier") ce qui constitue une I/O.

La clientid et la stateid sont des identifiants opaques pour le client qui doit se contenter de les présenter au serveur. Le serveur est en revanche capable de comprendre le contenu de ces identifiants. La seqid est un simple entier non signé sur 32 bits et dont la valeur initiale est toujours 0.

Les notions de clientid, stateid et seqid ne sont pas seulement indispensables pour gérer les transitions d'états, elles sont particulièrement importantes pour gérer les situations de reprises sur panne ou *crash recovery*, lorsqu'un client ou un serveur s'arrête brutalement. Il s'agit pour chaque acteur de faire la différence entre le OP_DESTROY_SESSIONs états avant et après le crash. NFSv4.0 autorise l'implémentation de différentes stratégies. La gestion des situations de reprise sur panne est très complexe en NFSv4, particuliè-

2. c'est nécessaire pour traiter proprement la suppression d'un fichier, on rappelle que renommer un fichier en un nom existant provoque l'effacement du fichier qui possède déjà ce nom

rement en NFSv4.0, elle justifierait à elle seule la rédaction d'un document résumant les situations possibles, en application des règles décrites dans les "tables de la loi" un peu arrides que sont les RFCs. Ce sujet ne sera pas abordé plus avant dans ce mémoire.

la *shared reservation*

Cet état ne représente ni plus ni moins qu'un fichier ouvert par le client. Ce sont les opérations OPEN et OPEN_CONFIRM qui permettront la négociation de cet état entre un client et un serveur. L'opération OPEN_CONFIRM permet de négocier cette ouverture du fichier en trois étapes, pour gérer les cas où le client viendrait à réémettre des requêtes contenant OPEN.

La logique est ici parfaitement analogue à celle qui est exploitée dans la négociation d'une clientid [3]. Il est possible de dégrader un état de fichier ouvert avec l'opération OPEN_DOWNGRADE, par exemple un fichier ouvert en "lecture et écriture" peut être dégradé en "lecture seule". Les fichiers ouverts sont révoqués par l'opération CLOSE.

les verrous

Les verrous sont parfaitement analogues aux verrous POSIX. es verrous sont adossés, toujours adossés à un autre état qui représente un fichier ouvert, mais il s'agira d'états différents même s'ils sont implicitement associés. Un verrou est créé par l'opération LOCK, cette opération est complexe, car elle fait apparaître la notion, sous-jacente dans POSIX, de "possesseur de verrou" ou *lockowner*. Le lockowner sera présenté par le client, il se compose d'une clientid et d'un identifiant opaque pour le serveur, mais qui a du sens pour le client, par exemple, il pourra représenter un thread ou un processus.

Les verrous peuvent être positionnés sur l'ensemble d'un fichier ou sur une ou plusieurs sections du fichier, définies par des couples (offset, longueur), si le système de fichiers sous-jacents le permet. L'opération LOCKT permet de tester la présence d'un verrou et détecter des conflits potentiels, un verrou, désigné par la stateid correspondante, sera révoqué par l'opération LOCKU.

les délégations

Les délégations sont un mécanisme particulièrement novateur mais qui induit de réelles complexités lors de l'implémentation. L'idée qui est derrière ce concept est assez simple : NFS est avant tout un protocole conçu pour gérer un parc de stations de travail qui partage un espace de stockage de données commun, cet espace est souvent fractionné en différentes parties, chacun étant par exemple dédiée à un utilisateur présent sur une seule machine client. C'est typiquement ce qui se passe avec les "home directories" des

3. négociation faite avec SETCLIENTID et SETCLIENTID_CONFIRM

utilisateurs montés par NFS et qui représentent une grande majorité des utilisations de NFS. Dans ce contexte, on peut considérer que sur une période de temps assez longue, une partie des données exportées par le serveur NFS ne vont être accédées que par un seul client à la fois. Pour décharger le serveur, il suffit de "donner temporairement" cette partie de l'espace exportée au client qui fera moins de requêtes sur le serveur. Les délégations concernent des fichiers ouverts, comme les verrous, elles seront représentées par des états adossés à l'état "père" qui correspond à celui de la shared reservation correspondant au fichier ouvert, elles peuvent être en lecture seule ou en lecture et écriture.

Une délégation permet au client de mettre en œuvre des stratégies de cache particulièrement agressives, car il aura l'assurance qu'il est le seul à agir sur le fichier et que le serveur ne va pas avoir à gérer de concurrence d'accès dessus. Par exemple, supposons que le client vient régulièrement modifier les cent premiers octets d'un fichier (il peut s'agir d'un entête) : il cachera le contenu du fichier après avoir obtenu une délégation en lecture-écriture et mettra à jour les cent premiers octets sans faire de requêtes sur le serveur, celui-ci se "désintéresse" temporairement du fichier pendant la durée de la délégation. Quand la délégation se termine, le client vient faire les opérations d'écriture nécessaires pour indiquer au serveur le nouvel état du fichier.

Délégations avec *call-backs*

Dans le cadre d'une délégation, on peut choisir de donner régulièrement des informations au serveur quant aux évolutions de l'état du fichier. Pour ce faire, le mécanisme de *call-backs* sera mis en jeu. Celui-ci n'est pas trivial et met en œuvre un serveur transient : le rôle du client et du serveur vont en effet s'inverser, le client va se positionner en serveur et répondre aux requêtes émises par le serveur qui souhaite s'informer des évolutions sur le fichier. Ces requêtes seront réalisées selon le *Call Back Protocol* décrit dans la section 15 de RFC3010. On ne rentrera pas en détail sur cette fonctionnalité qui a été peu exploitée dans le monde NFS. En particulier, le serveur NFS-Ganesha n'implémente pas cette fonctionnalité.

D.6.1 . Le protocole NFSv4.1

Le protocole NFSv4.1 apparaît en 2010 avec RFC5661[73]. Il ne s'agit pas d'une nouvelle version de NFS, mais d'une version mineure de NFSv4, considérée comme la version numéro 4.1 .

Le protocole NFSv4.0 a subi différentes évolutions et la RFC3010 s'est vu amendé par RFC3530 [15]. En effet, les efforts pour créer des clients et des serveurs selon les règles de RFC3010 ont montré que certaines situations "aux limites" [4] n'étaient pas possibles à implémenter.

4. aussi appellées *corner cases* en anglais

NFSv4.1 a plusieurs objectifs : rendre le protocole NFSv4 plus simple à appréhender et introduire des fonctionnalités particulièrement innovantes telles que pNFS qui ajoute la notion de parallélisme à NFS.

D.6.2 . Les nouvelles opérations de NFSv4.1

NFSv4.1 dispose des mêmes opérations que NFS4.0, et il fonctionne avec le même mécanisme de requêtes composées. NFSv4.1 dispose d'opérations supplémentaires décrites dans le tableau D.3.

Nom de l'opération	Action réalisée
OP_BACKCHANNEL_CTL	Opère sur un backchannel
OP_BIND_CONN_TO_SESSION	Ajoute une connexion à une session
OP_EXCHANGE_ID	Création d'une clientid
OP_CREATE_SESSION	Création d'une session
OP_DESTROY_SESSION	Destruction d'une session
OP_FREE_STATEID	Libération d'une stateid
OP_GET_DIR_DELEGATION	Délégation de répertoire
OP_GETDEVICEINFO	pNFS : informations sur un device
OP_GETDEVICELIST	pNFS : iste des devices
OP_LAYOUTCOMMIT	pNFS : commit d'une layout
OP_LAYOUTGET	pNFS : obtention d'une layout
OP_LAYOUTRETURN	pNFS : révocation d'une layout
OP_SECINFO_NO_NAME	Complète l'opération SECINFO
OP_SEQUENCE	Indique le numéro de séquence
OP_SET_SSV	Change la SSV d'une clientid
OP_TEST_STATEID	Teste une stateid
OP_WANT_DELEGATION	Négociation d'une délégation
OP_DESTROY_CLIENTID	Destruction d'une clientid
OP_RECLAIM _COMPLETE	Fin de la récupération d'états
OP_ILLEGAL	Opération illégale

Table D.3 – Opérations spécifiques du protocole NFS Version 4.1

On peut regrouper ces nouvelles opérations sous différentes bannières

- Les opérations qui simplifie des modes de fonctionnement de NFSv4.0 jugés trop complexes
  - OP_SEQUENCE permet une gestion plus simple des séquences dans les transitions d'états
  - OP_EXCHANGE_ID, OP_DESTROY_CLIENTID, pour la gestion des clients id, OP_CREATE_SESSION, OP_DESTROY_SESSION, pour les sessions mas aussi OP_SET_SSV simplifient la gestion des clients et rendent obsolètes OP_SETCLIENTID et OP_SETCLIENTID_CONFIRM.
  - OP_BACKCHANNEL_CTL et OP_BIND_CONN_TO_SESSION facilite la gestion des connexions des clients, en particulier les back-channels uti-

lisés par les clients dans le cas où des délégations avec utilisation du Call Back Protocol sont mises en oeuvre ;
    - OP_RECLAIM _COMPLET, OP_FREE_STATEID et OP_TEST_STATEID simplifient la gestion des états
- Les opérations qui rajoutent des capacités à des fonctionnalités existants
    - OP_ILLEGAL permet à un client d'indiquer dans le retour d'une requête que l'une des opérations demandées étaient illégales, cela peut en particulier se produire dans la gestion des états
    - OP_SECINFO_NO_NAME permet de gérer plus finement les montages sécurisés (tels que ceux qui utilise RPCSEC_GSS
    - OP_WANT_DELEGATION donne plus de souplesse dans la déclaration des délégations de fichiers
    - OP_SEQUENCE permet de gérer les séquences de requêtes
- les opérations qui rajoutent des fonctionnalités totalement nouvelles
    - OP_GET_DIR_DELEGATION introduit le concept de délégation de répertoire
    - OP_GETDEVICEINFO, OP_GETDEVICELIST, pour la gestion des *devices* OP_LAYOUTCOMMIT, OP_LAYOUTGET et OP_LAYOUTRETURN, pour la gestion des *layouts* implémentent le sous-protocole NFSv4.1/pNFS qui introduit le parallélisme des I/O dans NFSv4

Parmi les nouvelles fonctionnalités, NFS-Ganesha n'intégrera jamais les délégations de répertoires, les délégations de fichiers, déjà présentes dans NFSv4.0 ne seront pas prises en considération plus.

Le rédacteur du présent document sera aux premières loges dans l'implémentation du support du nouveau protocole NFSv4.1, en particulier la nouvelle gestion des séquences et des clientids et les premières implémentations de pNFS.

NFSv4.0 utilisaient des couples de fonctions du type "(DO_SOMETHING, DO_SOMETHING_CONFIRM)" mais NFSv4.1 s'en affranchit d'une manière très élégante, en introduisant les notions de sessions et de séquence de requêtes. Une fois que le client a négocié une clientid avec EXCHANGE_ID, il va utiliser celle-ci pour établir des sessions grâce à CREATE_SESSION. Les sessions peuvent être multiples et être détruites sans qu'il soit nécessaire de toucher à la clientid. Une fois les sessions négociées, le client peut commencer à opérer sur le serveur, et à former une séquence au sein de la session.

Dans la sémantique de NFSv4.1, l'ensemble des requêtes composées émises par un client doit commencer par l'opération SEQUENCE, qui sera donc toujours la toute première opération de la requête. Cette opération permet d'indiquer au serveur le numéro d'ordre d'une requête au sein d'une session. Le numéro de séquence ne pouvant qu'augmenter. Ainsi, il est facile pour le serveur de trier les requêtes incidentes et de les gérer dans le bon ordre, même

si des congestions réseaux viennent retarder certains messages. Des codes d'erreurs spécifiques sont indiqués dans RFC5661[73] pour chacun des cas possibles, tout en indiquant quel comportement implémenter dans les parties client et serveur pour rétablir une situation correcte.

La fonctionnalité pNFS est, elle aussi, au cœur des nouveaux développements, elle est décrite en détail dans la section D.6.3.

D.6.3 . La fonctionnalité pNFS du protocole NFSv4.1

La fonctionnalité pNFS est probablement la fonctionnalité nouvelle majeure du protocole NFSv4.1. Elle permet d'introduire du parallélisme dans les accès à un montage NFS. La stratégie de parallélisme qui est ici mise en œuvre est très proche de celle exploitée dans l'architecture du système de fichiers parallèle Lustre[16].

Le schéma de fonctionnement de pNFS n'inclut pas simplement des clients et des serveurs, il compte des clients, des serveurs de métadonnées, ou *metadata servers*, désignés par l'acronyme, MDS et des serveurs de données, ou *data servers*, désignés par l'acronyme DS. On est alors en présence d'un schéma à trois acteurs, dit "en tierce partie", décrit par la figure D.1. Comme

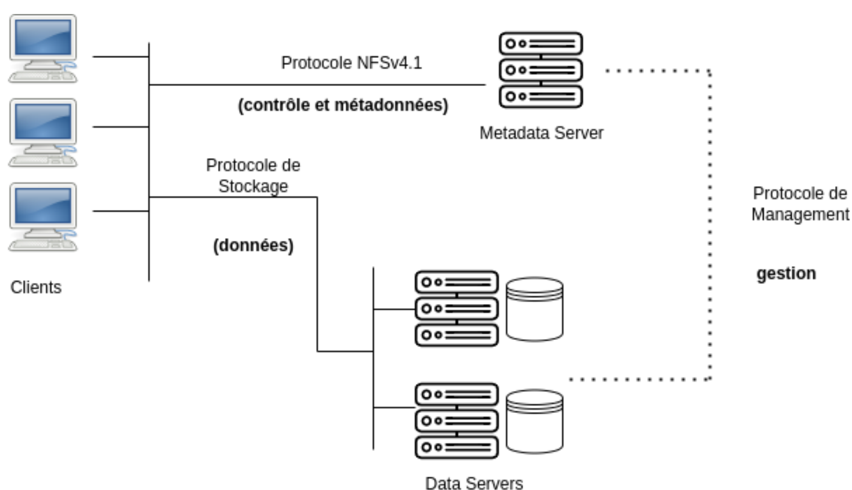


Figure D.1 – Schéma en tierce partie mis en jeu par NFSv4.1/pNFS

on peut le voir sur cette figure, il n'y a pas que le seul protocole NFSv4.1 qui soit mis en jeu, ce dernier n'est utilisé qu'entre le client et le MDS, celui qui sera monté comme un serveur NFS classique. Quand un client souhaite réaliser une IO, le serveur va retourner une structure spéciale, nommée *layout* et qui indique comment le client doit faire pour aller réaliser les IO, celles-ci n'étant pas faites sur MDS, mais sur les DS.

Les layouts contiennent différentes informations : des Uniform Resource Identifier (URI)s y décrivent comment accéder aux DS, ces URIs sont conformes

aux spécifications de RFC5665[39], mais aussi la manière dont les données doivent être distribués entre les différents MDS, on parle de *striping*. Cette approche est particulièrement intéressante, car elle sépare le chemin de métadonnées, qui est unique, des multiples chemins de données.

Ces derniers peuvent mettre en jeu des ressources différentes et cela permet d'optimiser celle-ci, mais aussi d'exploiter des protocoles différents pour les chemins de données entre les clients et les DS. Le protocole NFSv4.1 ne fait pas d'hypothèses strictes quant à la nature du protocole entre les clients et les DS, il suppose simplement que celui-ci dispose de fonctionnalités très simples, permettant de lire et d'écrire des données. Quelques protocoles sont proposés dans la description de pNFS, mais elles ne sont pas obligatoires :

- la **layout file** qui consiste en l'utilisation du protocole NFSv4 utilisé comme protocole de stockage.
- la **layout blocks** qui permet d'accéder des disques accessibles par des protocoles réseau, par exemple par le protocole iSCSI[102][19]. Elle est décrite plus précisément dans RFC5663[43].
- la **Object Based Layout** qui permet d'accéder aux *storage objects* définis par le protocole OSD-2[95]. Ce type de ressources est en particulier utilisé dans le système de fichiers opensource OSDFS[22] et dans le système de fichiers parallèle distribué PanFS[23] commercialisé par la société Panasas.

Il est tout à fait possible de mettre en place son propre protocole de données, pour peu que l'on implémente le client et le serveur, ainsi que le code source nécessaire pour les connecter avec les structures du protocole NFSv4.1/pNFS. Sur ce dernier point, les spécifications de pNFS sont suffisamment souples pour réaliser cette action sans point bloquant.

Du point de vue du protocole NFSv4.1, le fait d'accéder au contenu d'un fichier est similaire à une forme de délégation. Tandis que les clients et les DS interagissent, il peut être utile d'informer le MDS sur les changements effectués afin qu'il puisse toujours afficher une information cohérente et à jour. Dans le cas des délégations, cette cohérence est réalisée au travers du *CallBack Protocol*, avec pNFS, il est possible de définir un protocole de gestion qui permette aux DS et MDS d'échanger des informations afin de maintenir la cohérence des métadonnées affichées [5]. La plupart des implémentations de pNFS mettront en place un tel protocole de gestion, mais celui est totalement indépendant de pNFS, les personnes qui l'implémentent sont libres d'utiliser la solution la plus adaptée.

---

5. les métadonnées correspondantes aux attributs POSIX *mtime*, *space_used* et *last_offset* sont particulièrement sensibles

Gestion des états sur les différents acteurs

Dans l'esprit, l'obtention d'une layout pour accéder à un fichier, via pNFS, est très similaire à l'obtention d'une délégation en lecture ou en écriture. De même que les *shared reservations*, les verrous ou les délégations, il s'agira d'un nouveau type d'état maintenu entre les client et le MDS. Ce dernier imposera qu'une layout soit toujours adossée à une *shared reservation*, donc un fichier ouvert au niveau du client. Le MDS est le seul endroit où seront gérés les états propres aux layouts, ce qui permet d'assurer une complète cohérence avec les autres types d'états du protocole NFSv4.

Du point de vue des Data Servers, les choses sont bien plus simples, il n'est pas nécessaire de négocier un état entre client et DS, puisque toute communication entre ces deux types d'acteurs suppose qu'un layout a été négocié entre le client concerné et le MDS. L'existence de cet état sur le MDS garantit qu'il n'y a aucun risque de conflit dans les IO concernés, puisque cela reviendrait à dire qu'on aurait négocié des layouts en conflit, ce qui est interdit. Les protocoles utilisés entre clients et DS sont donc parfaitement sans état. Dans le cas de la*layout files*, qui exploite les opérations READ et WRITE du protocole NFSv4. Dans ce cas particulier, ces opérations seront utilisés sans stateid alors que cela est normalement obligatoire. Pour implémenter cela, une stateid spéciale, dont tous les bits sont à 1, est utilisée. Cette stateid générique ne requiert aucune négociation préalable.

D.6.4 . Le protocole NFSv4.2

Le protocole NFSv4.2 est décrit par RFC7862[91]. C'est un protocole encore relativement jeune même si ses spécifications datent de 2016. Au moment où le présent mémoire est rédigé, le support de NFSv4.2 dans le noyau Linux n'est pas encore finalisé, et la version utilisée par défaut dans la plupart des distributions de ce système d'exploitation est NFSv4.1. La rédaction des spécifications du protocole a été une opération de longue haleine, l'IETF souhaitant que le document final ne requiert pas de republications, ce qui suppose toujours la création d'une nouvelle RFC qui rend obsolète la précédente [6].

La présente section est incluse dans le mémoire par souci d'exhaustivité. NFSv4.2 n'est pas officiellement pris en charge par NFS-Ganesha même si la communauté NFS a réalisé quelques tests de ce protocole en exploitant NFS-Ganesha. On trouvera donc dans les sources du logiciel des traces de ces premières utilisations test.

NFSv4.2 apporte essentiellement des fonctionnalités supplémentaires à NFSv4.1. Il n'opère pas de réelle modification du fonctionnement du protocole lui-même. Ces nouvelles fonctionnalités sont les suivantes, elles seront détaillées dans les paragraphes suivantes :

6. Les RFCs qui décrivent NFSv4.0 et NFSv4.1 ont ainsi subi plusieurs de ces révisions dans des laps de temps assez courts à l'échelle des publications de l'IETF

1. support de la copie et du clonage de fichier par le serveur (*Server-Side Clone and Copy*)
2. support des pré-déclarations d'accès aux fichiers (*Application Input/Output (I/O) Advise*)
3. support des fichiers "à trous" (*Sparse Files*)
4. support de la réservation d'espace (*Space Reservation*)
5. support de ADB (*Application Data Block)*
6. support de *Labeled NFS*
7. amélioration de la gestion des layouts (*Layout Enhancements*)

Copie et du clonage de fichier par le serveur

L'une des actions les plus courantes d'un utilisateur sur un système de fichier est la copie de fichiers, souvent réalisée au travers de la commande Unix `cp`.

Que se passe-t-il lors d'une copie de fichier ? Le client va ouvrir la source en lecture et la destination en écriture, puis itérer sur tous les blocs lus dans la source pour écrire des blocs similaires dans la destination. Il en résulte une avalanche de requêtes READ et WRITE entremêlées.

NFSv4.2 introduit l'opération COPY qui permet à un client de demander au serveur de faire cette copie de manière interne, en une seule et unique requête. Cette fonctionnalité simple est une source importante d'optimisation et va contribuer à supprimer une charge réseau inutile.

Une question se pose alors : que faire lorsque la source et la destination d'une copie ne sont pas gérées par les mêmes serveurs, si on lit sur `serverA` pour écrire sur `serverB` ? Pour gérer cette situation, le protocole NFSv4.2 introduit l'opération CLONE qui voit les deux serveurs négocier entre eux pour réaliser cette copie sans impliquer le client, qui est simplement notifié de la fin de la copie.

La situation est toutefois complexe, car un schéma en tièrce-partie qui implique le client, la machine `serverA` et la machine `serverB`, ces deux derniers acteurs doivent donc négocier une connexion explicite pour échanger les données. Cette dernière situation devient encore plus complexe si des notions de sécurité sont introduites, par exemple si l'on veut assurer que les données transmises le sont au travers de liaisons chiffrées [7].

Support des pré-déclarations d'accès aux fichiers

NFSv4.2 introduit la nouvelle opération IO_ADVISE qui reproduit le comportement de la fonction POSIX *fadvise()* [8]. Cette fonction permet à l'utilisateur d'un système de fichiers de données des indications au système quant à

7. Dans ce cas, il sera fait usage de RPCSEC_GSS dans la configuration *krb5p*
8. Voir la manpage https ://linux.die.net/man/2/fadvise

la manière dont il va exploiter les données qu'il est sur le point d'écrire dans un fichier, afin de rendre possible des optimisations. Par exemple, on pourra indiquer que le fichier va être accédé de manière aléatoire (*random access*) ou au contraire de manière séquentielle, s'il s'agit par exemple d'un film vidéo, ou encore que le fichier va être écrit mais ne sera probablement pas relu, cas typique d'un fichier de reprise/redémarrage (*checkpoint-restart file*).

Ce type d'information "métier" donnée au système de fichiers est particulièrement utile, surtout lorsque le système de fichiers exploite différentes ressources, par exemple des *pools* construits avec des SSD et d'autres avec des HDD. Elles permettent en effet d'optimiser le placement des ressources de stockage. Cette problématique de *Data Placement* est au cœur des problématiques qui apparaissent dans les systèmes de classe exaflopique. Elle sera de nouveau évoquée dans le chapitre 3.

Support des fichiers "à tous"

POSIX n'impose pas aux fichiers d'être alloués d'un seul tenant. Par exemple, un fichier peut avoir été écrit entre les offsets 0 et 100, puis entre les offsets 200 et 300. Il ne contiendra rien entre 100 et 200, et jamais il ne lui aura été alloué de ressources de stockage pour cette partie du fichier. Un tel fichier est désigné sous le nom de *sparse file*. Lorsqu'un lecteur est effectué dans un "trou" d'un *sparse file*, POSIX indique de ne pas retourner une erreur, mais de transformer à la volée ces métadonnées, qui indiquent qu'on est dans un trou, en blocs contenant des zéros. Ainsi, quand on recopie un sparse file sur un système conforme à POSIX, la copie n'est pas un sparse file, les trous auront été comblés par des zéros, la copie sera donc plus grande que le fichier original [9].

NFS ignore les sparse files jusqu'à cette version NFSv4.2, et l'utilisateur ne voyait alors que des fichiers contigus [10] au travers des montages NFS.

Dans le cas de NFSv4.2, lorsque le serveur détecte un trou lors d'une opération de lecteur, il va retourner une erreur spécifique, *NFS4ERR_HOLE* dont le statut contient l'offset auquel il y a de nouveau des données à lire, qui délimite la fin du "trou". Outre l'optimisation évidente, puisque cette stratégie évite de faire des requêtes READ inutiles, surtout si le trou est grand, cette fonctionnalité donne au client une vision fidèle de la topologie du fichier qu'il accède. Les opérations SEEK et SEEK_HOLE sont également introduites pour permettre à un client de scanner un fichier sans le lire de manière explicite.

Pour des raisons de compatibilité, une nouvelle opération READ_PLUS est introduite en parallèle de l'opération READ qui conserve l'ancien comporte-

9. En termes de métadonnées POSIX, les deux fichiers auront les mêmes valeurs pour l'attribut *last_offset* mais les valeurs de l'attribut *space_used* seront différentes

10. Dans ce cas, l'attribut *last_offset* est égal à l'attribut *space_used* multiplié par la taille de bloc utilisée

ment et ignore donc les trous.

Support de la réservation d'espace

De même que l'opération IO_ADVISE vient reproduire le comportement de la fonction *fadvise()*, les nouvelles opérations ALLOCATE et DEALLOCATE de NFSv4.2 viennent reproduire au travers de NFS le comportement de la fonction *posix_fallocate()* [11] . Cette fonction permet à un client de déclarer à l'avance l'espace dont il va avoir besoin pour écrire et l'autoriser à effectuer une réservation de cet espace. De nombreuses optimisations sont rendues possibles par ce type de fonctionnalités.

Support de ADB

Cette fonctionnalité vise à optimiser les opérations produites par des clients qui stockent sur NFS des données qui sont susceptibles d'être écrites selon des motifs prévisibles et répétitifs : c'est par exemple le cas d'une application qui stocke des systèmes de fichiers entiers dans des fichiers [12] et qui sont amenés à formater ces fichiers. NFSv4.2 introduit ainsi l'opération WRITE_SAME qui permet d'ordonner au serveur décrire les mêmes blocs de données de manière répétées selon un schéma décrit en argument. L'objectif est clairement de décharger les liens réseaux entre clients et serveurs.

Support de *Labeled NFS*

Cette fonctionnalité est celle qui était la plus attendue dans NFSv4.2, elle fut aussi la plus compliquée à définir, ce qui causa de nombreux retards dans la publication des spécifications du protocole.

L'introduction du protocole RPCSEC_GSS[40] et la possibilité de l'utiliser en lieu et place de ONC/RPC depuis NFSv3 a rendu NFS populaire parmi les populations qui souhaitent partager des informations de manière sécurisée. RPCSEC_GSS permet en effet, via la GSSAPI, d'utiliser le protocole KRB5[71] pour garantir l'authentification des acteurs, leur permettre de négocier des secrets et d'utiliser ceux-ci pour réaliser du chiffrement point à point. L'utilisation de "NFS kerbérisé" répondait ainsi parfaitement aux besoins de cette classe particulière d'utilisateurs.

Le Diable est toutefois dans les détails et des études menées par l'agence de sécurité américaine National Security Agency (NSA), ont démontré les vulnérabilités de cette solution. Des documents, regroupées sous le sigle Security Enhanced NFS (SENFS), seront rédigés pour préciser les contours d'un nouveau protocole encore plus robuste[69]. Sans trop rentrer dans les détails, on utilisera, en plus de protocoles comme Krb5 qui agissent entre les acteurs,

11. Voir la manpage https ://linux.die.net/man/2/fadvise
12. Des fichiers qui seront par exemple montés comme des *lookpbacks*

des outils comme SELinux[99]. SELinx vient apposer des étiquettes aux ressources du système d'exploitation et à ses utilisateurs et permet de définir des politiques d'accès qui disent finement qui a le droit de faire quoi et avec quoi. La solution vient revisiter certains aspects de NFS qui devient dès lors *Labeled NFS* [81]. L'initiative SEENFS proposera également une relecture des RPC, qui succédera à RPCSEC_GSS et sera décrite par la RFC7861[2].

NFSv4.2 est la première version du protocole à implémenter et donc à rendre concret Labeled NFS, pour beaucoup cette fonctionnalité est considérée comme l'apport majeur de cette version du protocole.

Amélioration de la gestion des layouts

NFSv4.2 améliore la gestion de layouts introduites pour pNFS dans NFSv4.1, en particulier le traitement des erreurs du côté du client et la gestion des performances du côté du client.

Annexe E : Le protocole RPCSEC_GSS et la sécurité réseau

Le protocole RPCSEC_GSS est une extension du protocole ONCRPC qu'il enrichit avec la prise en compte de fonctionnalités associées à la sécurité. Ces appels s'appuient sur les services fournis par la Global Security Service Application Programming Interface (GSSAPI) qui repose sur le service kerberos5.

Les paragraphes suivants présentent un bref rappel sur les services proposés par kerberos5, puis la GSSAPI et sa mise en oeuvre, et enfin RPCSEC_GSS.

E.1 .kerberos 5

Kerberos5, qui sera désigné par l'acronyme Kerberos v5 (krb5) dans la suite du document, est un protocole d'authentification qui permet à différentes parties de s'assurer de l'identité l'une de l'autre, mais aussi de négocier des secrets communs qui pourront servir pour mettre en place des mécanismes de chiffrement entre les acteurs. Il est décrit par la RFC1510 [71].

Krb5 est un protocole d'authentification Authentication Authorization Accounting (AAA) [1], acronyme qui désigne les trois fonctionnalités attendues :

- **Authentication** : il s'agit d'être en mesure de s'assurer de l'identité d'un acteur impliqué dans un échange ;
- **Autorisation** : un acteur authentifié (cf point précédent) va demander à accéder à des ressources classifiées selon différents niveaux. Des politiques d'accès indiqueront qui pour faire qui sur quelles ressources ;
- **Traçabilité** [2] : le système doit garder une trace horodatée des actions et des demandes des acteurs.

Krb5 a été développé dans le cadre d'un projet "Athena", conduit au Massachusetts Institute of Technology (MIT), afin de développer un protocole respectant la triade AAA. Il devient très vite un standard sur les systèmes Unix puis Windows entre la fin des années 90 et le début des années 2000. Il faut noter que krb5 n'est pas d'un protocole réseau, il ne fait en particulier aucune hypothèse sur la façon dont les acteurs vont échanger des messages.

1. AAA = Authentication Autorisation Accounting
2. la traçabilité est désignée par le terme "Accounting" en anglais, ce qui constitue le dernier A de la triade AAA

E.2 .La GSSAPI

La GSSAPI [3], est une API qui vise à offrir une interface normalisée et unique pour accéder à un service d'authentification. Ses spécifications sont décrites par l' IETF dans différentes RFCs (RFC1508[58], RFC2078[60] et enfin RFC2743[61]), précisant les différents appels en C et les structures à mettre en place.

La GSSAPI est conçue pour être générique. Cette couche d'abstraction très formelle rend le code totalement indépendant du mécanisme sous-jacent, l'adhérence à celui-ci se trouve simplement déportée dans un fichier de configuration. Une implémentation de la GSSAPI par-dessus Krb5 existe conformément à la RFC1964[59].

D'autres implémentations seront envisagées pour des protocoles d'authentification via des échanges de clefs publiques, tels que SPKM[1], puis LIPKEY[37], mais elles seront peu utilisées, et l'industrie informatique se désintéressera peu à peu d'elles, lui préférant GSS/krb5.

De même que Krb5 , la GSSAPI n'est pas une API orientée réseau, elle se contente de fournir les appels qui permettent de construire les messages nécessaires à son fonctionnement. Elle est très rigide, et son utilisation doit suivre des schémas stricts. je renvoie à ce titre le lecteur à mon article sur le sujet paru en 2003 dans la revue Linux Magazine [28]

E.3 .RPCSEC_GSSv2

Si Krb5 et la GSSAPI ne sont pas des protocoles réseau, RPCSEC_GSS en est explicitement un. Il s'agit d'une extension de ONCRPCv2, qui se voit rajouter des appels l'authentification et le chiffrement.

E.4 .Prise en compte de RPCSEC_GSS dans NFSv3

NFSv3 est le premier protocole NFS à s'appuyer sur RPCSEC_GSS. Ce dernier est fondamentalement un protocole de transport à états, qui va s'appuyer sur un protocole connecté et incompatible avec des datagrammes. RPCSEC_GSS sera ainsi utilisé avec NFSv3/TCP et jamais avec NFSv2 qui a presque disparu à cette époque.

La mise en œuvre de RPCSEC_GSS avec NFSv3 suppose d'envisager tous les protocoles auxiliaires, en particulier le *Mount Protocol*. Cette démarche est décrite dans le document RFC2623 [36]. Ce dernier document décrit en particulier les contraintes et les solutions pour une prise en compte efficace des *automounters*[17]

Par ailleurs, la fonction NULL de l'ensemble des protocoles ne s'appuie

3. Global Security Service Application Programming Interface

jamais sur RPCSEC_GSS, il doit être possible de tester la présence du serveur [4].

E.5 .Montages NFS kerbérisés

Krb5 est difficilement dissociable de son application sur les systèmes de fichiers parallèles et distribués. La fonctionnalité "NFS kerbérisées" s'appuie sur le remplacement des ONCRPCv2 par RPCSEC_GSS, qui s'appuie sur le formalisme de la GSSAPI, laquelle est mise en œuvre grâce aux services fournis par krb5.

E.6 .Interaction avec le client knfs via RPCSEC_GSS

NFS-Ganesha est une implémentation d'un serveur NFSv4 et à ce titre n'implémente que la partie serveur. Il est conçu pour opérer avec le client présent dans le noyau Linux, ou Kernel NFS client (knfs). Durant cette période petascale, on assiste à une nette domination des machines sous Linux.

Le client knfs sera donc la cible prioritaire de NFS-Ganesha, et l'on prendra les choix techniques destinés à favoriser l'interopérabilité avec ce type de client.

Dans le cadre du knfs, et du Kernel NFS server (knfsd), l'intégration avec kerberos et la GSSAPI est complexe car ce sont des modules qui tournent au sein du noyau, tandis que les implémentations de kerberos fonctionnent dans le user space. Une passerelle est donc construite dans le système d'exploitation Linux via deux composants :

- le daemon *rpc.gssd* qui vient négocier des structures de la GSSAPI pour des acteurs du noyau depuis le user space ;
- un point de montage spécial, le *rpc_pipefs* qui permet de véhiculer des informations entre le user space et le kernel space.

Le déroulement d'une négociation est le suivant, décrit par la figure E.1 :

1. l'administrateur système client réalise un montage du serveur NFS-Ganesha, via la commande `mount` ;
2. le client sollicite le `rpc.gssd` pour que celui-ci négocie un contexte de sécurité avec la GSSAPI. Il effectue cette communication via la passerelle entre kernel space et user space implémentéé par le montage `rpc_pipefs` ;
3. cette négociation implique NFS-Ganesha, le `rpc.gssd` se connecte sur le serveur afin de pouvoir échanger avec lui. Côté serveur, on reçoit une seconde connexion de client ;
4. le daemon `rpc.gssd`, NFS-Ganesha et le service krb5 viennent négocier ensemble pour construire un contexte krb5 qui est enveloppé dans un contexte GSS.

4. de le *pinger* via la commande `rpcping` sans avoir à disposer de tickets kerberos

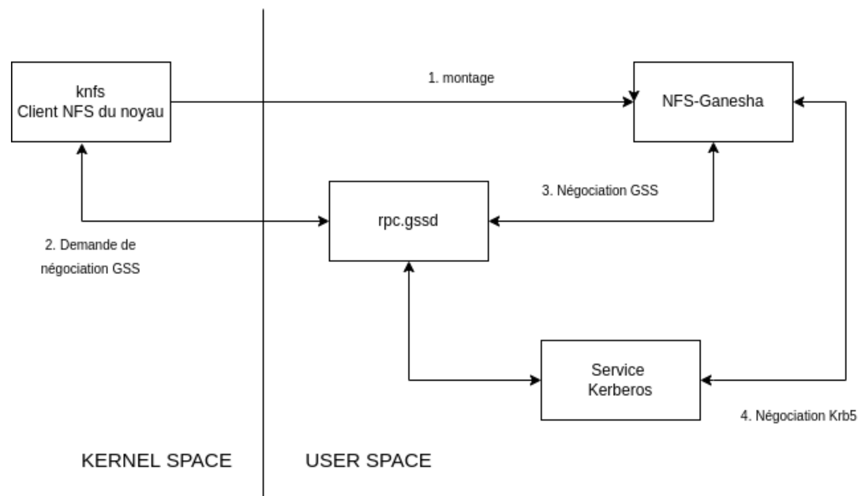


Figure E.1 – Négociation GSS via le rpc.gssd

Une fois la négociation effectuée, le `rpc.gssd` transmet la structure GSS négociée au client knfs qui la conserve. Ayant réalisé son travail, le `rpc.gssd` va fermer la connexion qu'il a établie avec NFS-Ganesha. Cette connexion n'a servi qu'à effectuer une négociation GSS, enveloppée dans des requêtes d'établissement de contexte de RPCSEC_GSS. En revanche, le client viendra ensuite produire des requêtes RPCSEC_GSS en utilisant la structure négociée par le service `rpc.gssd`. Du point de vue de NFS-Ganesha, il s'agit de maintenir les structures GSS négociées avec le `rpc.gssd` et qui seront utilisées par le `knfs`, on ne peut donc pas associer cette structure à une connexion cliente.

La structure de contexte GSS dispose d'un champ nommé *Object ID (OID)*, qui identifie la structure de manière unique. Cet identifiant est présenté en tant qu'alias dans les échanges par RPCSEC_GSS. Un cache similaire à celui du DRC décrit dans la section 1.5.3, adresse de manière associative les structures GSS négociées au moyen de cette OID. Ce cache sera rempli lors de la négociation avec le `rpc.gssd` puis exploité lorsque le client émettra des requêtes en exposant cette GSS OID.

Annexe F : Le protocole 9p2000.L

La première version du protocole 9p, adossée aux premières versions de Plan 9, permettait essentiellement d'accéder aux contenus de fichiers. La quatrième version, désignée sous l'acronyme 9p2000.L, permet de gérer l'ensemble des objets présents dans les systèmes de fichiers, incluant les répertoires, les liens standards et symboliques ainsi que les fichiers spéciaux. C'est cette version de 9p qui sera implémenté dans NFS-Ganesha pour mettre en place les IO-Proxies.

Le protocole 9P2000.L n'adopte pas les mêmes approches que les différentes versions du protocole NFS décrites dans le présent document dans les sections B.2.3, D et D.6.1. Son schéma est bien plus simple que NFS, qui s'appuie sur RPC, lequel s'appuie sur XDR et sur la paire TCP/IP et UDP/IP. En particulier, les échanges entre clients et serveurs ne sont pas encodés dans des structures complexes qu'il faut encoder grâce à XDR, 9p décrit comment doivent être construit les messages en s'appuyant sur des types de bases très simples : entiers signés et non signés sur 8, 16, 32 ou 64 octets, ainsi que des chaines de caractères. 9p suit une logique fondamentale *stateful* et suppose qu'il s'appuie sur un protocole réseau connecté, sans perte. C'est le cas des protocoles TPC/IP et RDMA qui sont implémentés dans le support de 9p implémenté dans le noyau Linux, ce seront également les couches de transport qui seront implémentées dans NFS-Ganesha dans sa déclinaison sous la forme d'IO-Proxies.

Les messages de 9p2000.L se regroupent en 3 classes simples :

1. les requêtes, dont les noms sont préfixés par la lettre **T** ;
2. les réponses aux requêtes, dont les noms sont préfixés par la lettre **R** ;
3. les messages d'erreur, dont le nom est préfixé par la lettre **L**, le seul message à appartenir à cette classe est le message LERROR.

Les messages du protocole 9p2000.L sont définis dans le tableau F.1. Ce tableau ne liste pas les requêtes et les messages de réponse, par souci de compacité. Chacune des lignes de ce tableau correspond donc à une paire message/réponse, à l'exception du message ERROR qui est toujours une réponse, mais jamais une requête.

On notera que certains appels semblent redondants : on trouve ainsi un appel RENAME et un appel RENAMEAT, ou un appel REMOVE et un appel UNLINKAT qui paraissent faire les mêmes actions sur les systèmes de fichiers. Ils correspondent aux différentes versions du protocole 9p. Les appels RENAMEAT et UNLINKAT ont été définis dans la quatrième version de Plan 9, dans 9p2000.L tandis que RENAME et REMOVE sont d'anciens appels obsolètes.

L'une des notions sous-jacentes à la logique du protocole 9p2000.L est l'idée d'activité : une activité est un thread d'un processus sur une machine

Nom de l'opération	Action réalisée
ERROR	Message d'erreur
STATFS	Information de statut sur le FS
LOPEN	Ouverture d'un fichier
LCREATE	Création d'un fichier
SYMLINK	Création d'un lien symbolique
MKNOD	Création d'un fichier spécial
RENAME	Renommer une entrée (obsolète)
READLINK	Lire un lien symbolique
GETATTR	Obtenir les attributs de l'entrée courante
SETATTR	Positionner les attributs de CurrentFH
XATTRWALK	Parcourir les attributs étendus de CurrentFH
XATTRCREATE	Créer un attribut étendu
READDIR	Lire le contenu d'un répertoire
FSYNC	Synchroniser une entrée
LOCK	Agir sur un verrou
GETLOCK	Obtenir le statut d'un verrou
LINK	Création d'un lien standard
MKDIR	Création d'un répertoire
RENAMEAT	Renommer une entrée
UNLINKAT	Détruire une entrée
VERSION	Obtenir la version du protocole 9p
AUTH	Gérer l'authentification d'une activité
ATTACH	Création d'une nouvelle activité
FLUSH	Flusher toutes les informations cachées
WALK	Descendre dans une arborescence
OPEN	Ouvrir une entrée
CREATE	Créer une entrée
READ	Lire dans un fichier
WRITE	Écrire dans un fichier
CLUNK	Détruire une entrée
REMOVE	Détruire une entrée
STAT	Opération obsolète
WSTAT	Opération obsolète

Table F.1 – Messages du protocole 9p2000.L

client qui va venir agir sur le système de fichiers. En particulier, un client n'enverra pas un nouveau message pour une activité donnée tant qu'un message de réponse n'a pas été émis. Contrairement à NFS, 9p2000.L ne s'appuie pas sur une logique de handle. Une activité embarque un état, similaire à un curseur, et qui décrit la position de l'activité dans l'espace de nommage exporté. Ce curseur peut être modifié avec le message WALK : connaissant le nom d'une entrée, on descendra le curseur sur l'entrée correspondante. Un mes-

sage, par exemple, pour créer ou détruire une entrée, sera toujours effectuée, pour une activité donnée, à la position courante du curseur de l'activité concernée.

Les curseurs des activités du client sont identifiées par un identifiant, appelé *fid*. Un fid [1] est un entier non signé sur 4 octets. Un fid, associé au client qui l'a créé, représente littéralement un curseur positionné pour une activité dans le système de fichiers. Ce sont les opérations ATTACH, WALK et CLUNK sont au centre de la gestion des fids.

- **ATTACH** est analogue à une action de montage dans NFS, et elle permet d'obtenir le premier fid à partir duquel on commencera à parcourir l'arborescence. En revanche, cette fid initiale devra être négociée pour chaque utilisateur.
- **WALK** permet de descendre dans l'arborescence et est effectuée depuis un fid qui représente un répertoire. Une nouvelle fid sera créée par WALK, elle pointera sur l'entrée dont le nom est donné en argument de l'opération. Elle pourra par exemple être ouverte avec l'opération LOPEN s'il s'agit d'un fichier ou permettre de continuer la descente dans l'arborescence (en effectuant un autre appel à WALK) s'il s'agit d'un répertoire.
- **CLUNK** détruit une fid, le numéro correspondant peut être réutilisé par le client pour un nouvel appel à ATTACH ou WALK.

Ce sera toujours le client qui présentera au serveur un numéro de fid. En termes d'implémentation, le fid est un index qui référence des items au sein d'un simple tableau. Client et serveur maintiendront ces tableaux cohérents au fur et à mesure de leurs échanges. Dans le protocole 9p, c'est le client qui choisit un identifiant fid disponible au moment de la requête. Les fids n'évoluent donc pas de façon monotone.

Une autre notion qui intervient dans les échanges selon le protocole 9p2000.L est la *qid*. Un qid est un buffer de 13 octets, opaque du point de vue du client et retourné par le serveur. Un qid représente une entrée dans le système de fichiers distribué et est totalement analogue à un fhandle NFS ou une inode dans un système de fichiers traditionnel. Les qids sont essentiellement exploitées pour permettre aux clients de cacher les informations retournées par le serveur.

1. Attention à ne pas confondre ce terme avec l'identifiant homonyme que l'on trouve en particulier dans Lustre (un fid dans Lustre identifie un fichier)

Annexe G : Spécifications du protocole LIOP

Cette annexe décrit en détail le protocole LIOP. Ce protocole a été introduit dans le cadre du nouveau protocole p9p. Il permet d'injecter du parallélisme dans le protocole 9p2000.L.

G.1 .Notations utilisées pour décrire les appels du protocole

Dans les sections qui suivent, on décrira les structures et les prototypes des opérations dans un langage proche de Remote Procedure Call Language (RPCL)[21], mais en utilisant les types élémentaires exploités par le protocole 9p2000.L. Dans cette notation, on suit les règles suivantes :

1. toutes les valeurs numériques sont en format little endian ;
2. toutes les valeurs numériques sont non signées ;
3. il n'existe que des valeurs entières, aucune valeur *float* ;
4. **u8** représente un octet, soit 8 bits ;
5. **u16** représente un entier court (short integer), soit 16 bits ;
6. **u32** représente un entier, soit 32 bits ;
7. **u64** représente un entier long, soit 64 bits ;
8. **uchar** représente un caractère dans une chaîne de caractères, c'est un type qui est donc équivalent à **u8** ;
9. **bool** est une valeur booléenne dont la valeur est *true* (1) ou *false* (0), il sera encodé sous la forme d'un **u8** / **uchar** ;
10. les types de bases peuvent être regroupés en tableau délimités par "[" et "]", par exemple **uchar[128]** représente une chaîne de 128 caractères ;
11. un buffer sera décrit comme un tableau d'octets, donc un tableau de **u8**
12. une châine de caractères est encodée comme un tableau de **uchar** et sa longueur stockée dans un entier standard non signé **u32** ;

On représentera les champs des différentes structures selon leur taille en octets. Un champ *my_id* décrit comme un *u64* seront donc écrite comme `my_id[8]`. Prenons par exemple la structure en C suivante :

```
cool_protocol_structure {
u32 cool_id;
uchar * cool_name_buffer;
u32 cool_name_len)
}
```

Cette structure très simple comprend donc un identifiant sur 32 bits et une chaîne de caractères. Dans notre notation, nous décrirons la structure de la manière suivante :

```
cool_protocol structure {  cool_id[4] cool_name(coll_name_len[2]) }
```

G.2 .Les spécifications du protocole LIOP

LIOP est conçu pour être un Data Server, c'est donc un protocole fondamentalement sans état, ou *stateless*. IL s'appuie sur un adressage des fichiers qui est toujours un adressage direct, par le biais d'un handle. On est là dans un logique qui est très similaire à celle qui est à l'œuvre dans les différentes versions du protocole NFS et des FSALs du produit NFS-Ganesha.

Initialement, LIOP devait s'interfacer avec Lustre, et donc les objets manipulés, qui sont toujours des fichiers, seront identifiés par des *fid*s, les structures qui représentent les fichiers dans Lustre. Dans LIOP, on va manipuler les structures décrites dans les paragraphes suivants.

G.2.1 . Lustre_fid

Cette structure représente un fichier dans Lustre. Elle est extraite des sources du code du système de fichiers Lustre.

```
lustre_fid {
 u64 f_seq
 u32 f_oid
 u32 f_version
}

lustre_fid { f_seq[8] f_iod[4] f_version[4] }
```

G.2.2 . ucred

Cette structure représente les credentials d'un utilisateur. On notera que, contrairement aux services d'authentification les plus simples de ONCRPCv2, on ne prend pas en compte les groupes secondaires d'un utilisateur, mais simplement son groupe principal.

En effet, on considère que le client du protocole LIOP aura préalablement choisi quel groupe est le plus pertinent pour réaliser cette opération. Par exemple, il pourra s'appuyer sur l'environnement de l'utilisateur final, potentiellement altéré par la commande en ligne `newgrp`.

```
ucred {
 u32 uid
 u32 gid
}

ucred { uid[4] gid[4] }
```

Dans le cadre du développement, il est envisagé d'étendre le protocole pour lui permettre de supporter Security Enhanced Linux (SELinux). On étendra donc la structure en lui ajoutant un contexte au sens de SELInux.

```
ucred {
 u32 uid
 u32 gid
 uchar scontext(u16 scontext_len)
}

ucred { uid[4] gid[4] scontent(scontext_len[2])
```

G.2.3 . version

Cette structure décrit la version du protocole LIOP, pour supporter concurremment différentes instanciations du protocole.

```
version {
 u32 major
 u32 minor
}

version { major[4] minor[4] }
```

G.2.4 . Codes d'erreurs

Le protocole LIOP retourne des erreurs sous la forme d'un *u32*. Les valeurs sont celles des erreurs standards de la norme POSIX, usuellement retournée par la variable *errno* de la LibC. Elles sont décrites dans le tableau G.1.

G.3 .Les fonctions du protocole LIOP

Les fonctions du protocole LIOP sont les suivantes :

- **STATUS** permet de vérifier l'état du service qui implémente le protocole LIOP ;
- **READ** effectue la lecture d'un buffer dans un fichier à un *offset* donné ;
- **WRITE** effectue l'écriture d'un buffer dans un fichier à un *offset* donné ;

Valeur	ERRNO	SIgnification
0	SUCCESS	Operation successful
1	EPERM	Operation not permitted
2	ENOENT	No such file or directory
4	EINTR	Interrupted system call
5	EIO	I/O error
6	ENXIO	No such device or address
9	EBADF	Bad file number
11	EAGAIN	Try again
12	ENOMEM	Out of memory
13	EACCES	Permission denied
14	EFAULT	Bad address
16	EBUSY	Device or resource busy
17	EEXIST	File exists
18	EXDEV	Cross-device link
19	ENODEV	No such device
20	ENOTDIR	Not a directory
21	EISDIR	Is a directory
22	EINVAL	Invalid argument
24	EMFILE	Too many open files
25	ENOTTY	Not a typewriter
27	EFBIG	File too large
28	ENOSPC	No space left on device
29	ESPIPE	Illegal seek
30	EROFS	Read-only file system

Table G.1 – Codes d'erreur du protocole LIOP

- **RELEASE** indique au serveur que le client ne s'intéressera plus au fichier concerné ;
- **GETFID** converti un chemin complet en *Lustre fid* ;
- **ERROR** produit une erreur ;

Voyons à présent chacune des fonctions une à une, avec les structures associées.

G.3.1 . ERROR

De même que pour le protocole 9p2000.L, les erreurs sont retournées par un message spécifique, qui n'est donc utilisé que par le serveur. La structure retournée est la suivante :

```
error_out {
u32 errno
}
```

```
error_out { errno[4] }
```

G.3.2 . STATUS

L’opération STATUS permet de vérifier si le service qui implémente LIOP est fonctionnel. Les structures sont les suivantes.

```
status_in {
 uchar lustre_fsname(fsnamelen[2])
 u64 flags
}

status_in { lustre_fsname(fsnamelen[2]) flags[8] }

status_out {
 u32 status
}

status_out { status[4] }
```

La fonction STATUS utilise un chemin comme argument, qui représente un point de montage, c’est-à-dire un espace de nommage géré par l’ensemble formé par le Metadata Server et les Data Servers.

G.3.3 . READ

L’opération READ permet d’effectuer une lecture "aléatoire" dans un fichier, c’est à dire une lecteur à un offet précis.

```
read_in {
 uchar lustre_fsname(fsnamelen)
 lustre_fid fid
 ucred user_cred
 u64 offset
 u32 length
}

read_in { lustre_fsname(fsnamelen[2]) lustre_fid[16]
 ucred[8] offset[8] length[4] }

read_out {
u8 data_read(length[4]) // implicitely length is u32
 u32 status
 bool eof_reached
```

```
 u64 hole_offset
}

read_out { status[4] eof_reached[4] hole_offset[8]
data_read(length[4]) }
```

Le champ *lustre_fsname* indique le point de montage sur lequel agir, couplé avec le champ *fid*, on identifie totalement le fichier. Le champ *cred* identifie l'utlisateur et le couple *(offset, length)* caractérise l'opération de lecture.

On notera que le protocole est conçu pour gérer les *sparse files*, ou "fichiers à trous". Si un trou est détecté dans l'opération de lecture, celle-ci s'arrête et la structure de sortie indique à quel offset reprendre la lecture.

Pour le traitement de cette situation, une erreur supplémentaire, retournée uniquement par l'opération READ, est ajoutée aux erreurs décrites dans le tableau G.1 : ERRHOLE dont la valeur est 1001. Le buffer *data_read* s'arrêtera à la bordure "gauche" du trou détecté, le champ *hole_offset* indique la bordure "droite" du trou.

G.3.4 . WRITE

La fonction WRITE permet d'effectuer une opération d'écriture. De même que pour READ les champs *lustre_fsname* et *fid* identifie le fichier, *cred* identifique l'utlisateur et le couple *(offset, length)* caractérise l'opération d'écriture.

```
write_in {
 uchar lustre_fsname(fsnamelen)
 lustre_fid fid
 ucred user_cred
 u64 offset
 u8 data_write(length[4])
}

write_in { lustre_fsname(fsnamelen[2]) lustre_fid[16]
ucred[8] offset[8] data_write(length[4]) }

write_out {
 uchar lustre_fsname(fsnamelen)
 u64 written_lenghth
 u32 status
}

write_out { written_length[4] status[4] }
```

G.3.5 . RELEASE

La fonction RELEASE permet au serveur de dire qu'il ne va plus utiliser un fichier. Ce fichier es défini par le couple *lustre_fsname* et *fid*. Le protocole LIOP est fondamentalement stateless, cette fonction permet au serveur de gérer ses couches de cache et potentiellement de libérer des ressources probablement allouées. Le drapeau *is_read* indique si les ressources concernées sont relatives à la lecture ou à l'écriture.

La fonction RELEASE n'est pas obligatoire et n'est pas un équivalent d'un *close()* de la LibC ou d'une opération OP_CLOSE de NFSv4. Il s'agit simplement d'une indication qui permet au client d'optimiser la gestion du protocole du côté du serveur.

```
release_in {
 uchar lustre_fsname(fsnamelen[2])
 lustre_fid fid
 bool is_read
}

release_in { lustre_fsname(fsnamelen[2]) lustre_fid[16] is_read[2] }

release_out {
 u32 status
}

release_out { status[4] }
```

G.3.6 . GETFID

La fonction GETFID permet simplement de convertir un chemin complet, dans un point de montage, en *Lustre fid*.

```
getfid_in {
 uchar lustre_fsname(fsnamelen[2])
 uchar filepath(pathlen[2])
}

getfid_in { lustre_fsname(fsnamelen[2]) filepath(pathlen[2])}

getfid_out {
 lustre_fid[16]
}

getfid_out { lustre_fid[16] }
```

Annexe H : Un exemple de fichier de configuration de KVSNS

Un fichier de configuration de la bibliothèque KVSNS assez générique. Celui-ci exploite le key-value store REDIS et l'object store RADOS inclus dans le logiciel Ceph.

```
[kvsns]
    extstore_lib = /usr/lib64/libextstore_rados.so
    kvsal_lib = /usr/lib64/libkvsal_redis.so

    # Values are LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR,
    # LOG_WARNING, LOG_NOTICE, LOG_INFO and LOG_DEBUG
    # if missing, default value is LOG_CRIT
    Log_level = LOG_CRIT

[kvsal_redis]
    server = localhost
    port = 6379

[rados]
    pool = kvsns
    cluster = ceph
    user = client.admin
    config = /etc/ceph/ceph.conf
```

Ce fichier est conforme à la syntaxe INI [1], basé sur des sections qui débutent par des balises entre crochets et nommés *stanzas*. La lecture de ce fichier de configuration est assuré par la bibliothèque open-source *ini-config* [2].

1. Voir la page https ://fr.wikipedia.org/wiki/Fichier_INI
2. Voir la page https ://github.com/topics/ini-config

Annexe I : Les mécanismes du KVSNS pour gérer la sémantique CRUD

Cette annexe décrit les outils qui permettent de faire fonctionner un object store qui fonctionne selon la sémantique CRUD

I.1 .Accès séquentiel et sémantique CRUD

Les types d'accès dans les gestionnaires de stockage objet sont diverses. La sémantique CRUD ne précise pas de manière explicite comment sont effectuées les accès aux contenus des objets. D'une manière synthétique, on peut voir deux méthodes d'accès, que l'on retrouve dans le formalisme de l'API KVSNS :

- l'accès dit **séquentiel** qui permet de lire et d'écrire à l'intérieur d'un objet, lequel est vu comme un tableau unidimensionnel d'octets, l'approche est ici similaire aux accès qui sont réalisés dans un fichier dans l'approche POSIX classique. On trouvera donc des appels similaires dans leurs formes aux appels `pread()` et `pwrite()` de la LibC ;
- l'accès dit **global** qui ne permet que de lire ou d'écrire la totalité d'un objet. On utilise ici des appels `GET` et `PUT`, très proches de ce que l'on peut trouver dans les protocoles S3[55], porté par Amazon et Swift[80] porté par le projet Openstack[79].

Certains *object stores* permettent de réaliser des accès séquentiels dans les objets, c'est en particulier le cas de RADOS [1], inclus dans le projet Ceph[78], ou CORTX/MOTR[84] [2] Cependant, une majorité des object stores n'implémentent qu'une méthode d'accès global, c'est notamment le cas de l'object store Phobos[18], porté par le CEA et capable de gérer des bandes magnétiques. Dans le cas de ces derniers, KVSNS retournera l'erreur `-ENOTSUP` pour `kvsns_write()` et `kvsns_read()`, mais implémentera les appels `kvsns_cp_from()` et `kvsns_cp_to()`.

Gérer la sémantique CRUD avec KVSNS

SI KVSNS permet d'exporter via NFS, au travers de FSAL_KVSFS et NFS-Ganesha, les object stores capables de faire des accès séquentiels, un problème se pose pour ceux dont la sémantique ne permet que les accès globaux. Pour répondre à cette question, le crud_cache a été mis en place.

D'une manière assez générique, si l'object store auquel on s'intéresse ne dispose que d'un accès global aux contenus des objets, il suffit de recopier ces

1. Voir la page https ://ceph.io/en/
2. MOTR permet en fait des accès séquentiels alignés sur les frontières de blocs, mais il est simple de reconstruire des appels conformes au comportement de `pread()` et `pwrite()` sur cette base.

contenus, via des opérations GET, dans un système de fichiers local au serveur dans lequel il sera possible de faire des accès séquentiels. Les données seront par la suite mises dans l'object store par l'utilisation de requêtes PUT. Les algorithmes qui permettent de gérer de tels caches ne sont pas nouveaux, ils sont en particulier utilisés dans la fonctionnalité *Binding HSM* du système de fichiers Lustre[97][87]. Le détail de ce fonctionnement est décrit dans l'annexe I.

la sous-couche OBJSTORE

La fonctionnalité crud_cache repose sur une version modifiée de la sous-couche EXTSTORE de KVSNS, qui est elle-même une bibliothèque autonome. Cette couche implémente la machine à états décrite en annexe, qui vient gérer la cohérence et les mouvements de données entre l'object store et le cache sur un système de fichiers persistant.

À ce niveau, la couche EXTSTORE permet de faire ressortir une nouvelle couche d'abstraction, qui enrobe la seule gestion des mouvements de données, la couche OBJSTORE. Celle-ci dispose de quelques appels très simples (PUT, GET et DELETE) qui reprennent la sémantique CRUD. OBJSTORE vient dissimuler l'object store, lequel pourrait être Parallel Heterogeneous OBject Store, un logiciel développé au CEA, ou n'importe quel object store qui dispose par exemple de l'interface Amazon S3. Cette couche devient à son tour une nouvelle bibliothèque de fonctions dynamique, chargée au démarrage, et indiquée par un fichier de configuration.

Les mécanismes derrière le crud_cache sont les suivants :

- un *object store* qui ne supporte que les accès globaux ;
- un système de fichiers local destiné à être utilisé comme un cache ;
- un outil de copie, désigné sous l'acronyme *copytool* dans le formalisme de la fonctionnalité de "binding Lustre/HSM", capable de recopier l'intégralité du contenu entre le cache et l'object store, ce type d'outil utilisera notoirement les requêtes PUT et GET de l'object store ;
- la mise en œuvre d'une machine à états simple permettant d'assurer la cohérence des données cachées ;
- la mise en oeuvre d'un gestionnaire de politique tel que Robinhood[57][56][26] pour gérer le remplissage du cache et renvoyer, via le copytool, les données peu utilisées et en cohérence vers l'object store et les supprimer du cache.

I.2 .La machine à états du crud_cache

Le crud_cache s'appuie sur une logique à états très simple, elle reprend exactement celle mise en œuvre dans le binding Lustre/HSM présente dans le produit dès la version Lustre 2.5, cette logique est résumée dans la figure

I.1. Il faut noter que cette gestion de cache est elle-même très proche de celle qui a été décrite dans la partie sur la période terascale avec le cache SHERPA décrit dans la section 1.3.

Dans ce mécanisme, le contenu d'un objet peut être dans les états suivants :

- **NEW** : le contenu vient d'être créé, il est encore vide ;
- **CACHED** : le contenu dans le cache est plus récent que celui dans l'object store ;
- **DUPLICATED** : le contenu est le même dans le cache et dans l'object store
- **RELEASED** : le contenu n'existe que dans l'object store et n'est pas présent dans le cache.

Les transitions entre les états sont les suivantes, résumées dans la matrice de transitions suivante et le schéma I.1 :

	NEW	CACHED	DUPLICATED	RELEASED
NEW		INTERDIT	INTERDIT	INTERDIT
CACHED	write		write	INTERDIT
DUPLICATED	INTERDIT	archive		restore
RELEASED	INTERDIT	INTERDIT	release	

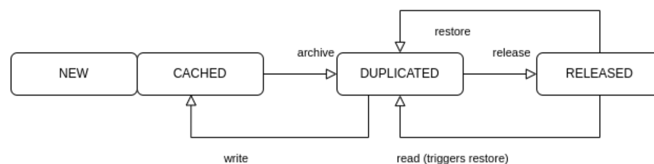


Figure I.1 – Transitions d'états dans le fonctionnement du crud_cache

Revoyons une à une chacune des transitions possibles :

- transitions depuis l'état **NEW**
  - vers l'état **CACHED** : il suffit d'écrire des données dans le contenu ;
  - vers l'état **DUPLICATED** : *TRANSITION ILLICITE*
  - vers l'état textbfRELEASED : *TRANSITION ILLICITE*
- transitions depuis l'état **CACHED**
  - vers l'état **NEW** : *TRANSITION ILLICITE* on ne peut pas "rajeunir" un contenu qui a hébergé des données et le ramener dans un état où il n'a pas encore recueilli d'informations ;
  - vers l'état **DUPLICATED** : c'est l'action *archive*, déclenchée par l'appel `extstore_archive()` invoqué lui-même par `kvsns_archive()` qui permet cette transition. Elle déclenche une action au niveau du copytool qui va faire une opération PUT dans l'object store pour y écrire le contenu le plus récent, présent dans le cache ;

- vers l'état **RELEASED** : *TRANSITION ILLICITE* cela reviendrait à perdre le contenu plus récent qui est dans le cache, il faut d'abord passer par l'état intermédiaire **DUPLICATED**;
- transitions depuis l'état **DUPLICATED**
  - vers l'état **NEW** : *TRANSITION ILLICITE* comme indiqué plus haut il n'est pas possible de "rajeunir" un contenu;
  - vers l'état **CACHED** : en écrivant de nouvelles données dans une entrée de type **DUPLICATED**, donc synchrone avec l'object store, rend le contenu dans le cache plus récent;
  - vers l'état **RELEASED** : c'est l'action *release*, qui permet cette transition. Elle déclenche, outre le changement d'état, l'effacement du contenu dans le cache, lequel est ici identique à celui dans l'object store;
- transitions depuis l'état **RELEASED**
  - vers l'état **NEW** : *TRANSITION ILLICITE* comme indiqué plus haut il n'est pas possible de "rajeunir" un contenu;
  - vers l'état **CACHED** : *TRANSITION ILLICITE* cela reviendrait à dire que des informations sont apparues par magie dans le cache, celui-ci doit être peuplé par une *restore* qui va passer l'entrée dans l'état **DUPLICATED**;
  - vers l'état **DUPLICATED** : c'est l'action *restore*, déclenchée par l'appel `extstore_restore()` invoqué lui-même par `kvsns_restore()` qui permet cette transition. Elle déclenche une action au niveau du copytool qui utilise alors une opération GET pour recopier dans le cache le contenu présent dans l'object store, celui-ci est donc synchrone avec l'object store. Il faut noter, comme nous le verrons plus tard, que la première tentative via crud_cache de lire dans un contenu dans l'état **RELEASED** déclenche une opération automatique de "restore".

I.3 .Implémentation de la machine à états

Le crud_cache est essentiellement implémentée au sein de la couche EXTSTORE. Le lecteur aura remarqué que des fonctions similaires existent dans la couche KVSNS, elles ne servent qu'à exposer ces fonctionnalités à l'utilisateur sans exposer la ouche EXTSTORE ce qui viendrait à briser l'architecture logicielle en couches utilisée.

L'implémentation de la machine à états suppose une conservation des états des différents objets. Ces états seront conservés sous la forme d'une association clef-valeur gérée par l'API KVSAL. La clef qui sera utilisée aura le format ``%llu.state''(kvsns_ino_t)''.

Les transitions entre les états décrits ci-dessus sont pilotées par les 3 appels suivants, ces appels reproduisent le même type de fonctionnement que celui exploité par la fonctionnalité Lustre/HSM, mais là où Lustre va implémenter les choses en *kernel space*, crud_cache les implémentera en *user space*.

Chacun de ces appels agit sur un objet à la fois :

- **extstore_archive()** : écrit dans l'object store les contenus plus récents déjà présents dans le cache ;
- **extstore_release()** : supprime du cache un contenu qui a été recopié dans le stockage objet;
- **extstore_restore()** : repeuple le cache avec du contenu conservé dans le stockage objet.

Les appels `extstore_write()` et `extstore_read()` sont directement impactés puisqu'ils vont venir agit sur des contenus ramenés sur le cache local pour permettre d'exposer un comportement qui permet un accès séquentiel. Cette opération n'est réalisable que si l'objet n'est pas dans l'état **RELEASED**. C'est la raison pour laquelle une opération de lecture (via `extstore_read()` sur un objet RELEASED va déclencher une action *restore* pour ramener les informations dans le cache.

Il est important de voir que le crud_cache est uniquement implémenté au niveau de la couche EXTSTORE, la couche KVSNS qui en exploite les services ne voit rien de cette gestion du cache et n'a rien à faire pour la prendre en compte.

I.4 .Déplacement des données : la couche OBJSTORE

La machine à états suppose de disposer d'un `copytool` capable de déplacer des contenus entre le gestionnaire de stockage objet et le cache local. De manière à rendre le code du crud_cache aussi indépendamment du stockage objet que possible, une nouvelle couche d'abstraction est introduite : la couche OBJSTORE. Celle-ci ne va exposer que les deux opérations CREATE, PUT, GET et DELETE, ce qui reproduit la sémantique CRUD. La couche OBJSTORE sera compilée comme une librairie dynamique indépendante du reste, chargée au démarrage par un appel à `dlopen()`. L'api très simple de cette couche est décrite par le tableau I.1.

Nom de l'appel	Nature
objstore_init()	Initialisation de la bibliothèque de fonctions
objstore_fini()	Fermeture de la bibliothèque de fonctions
objstore_get()	Lecture complète du contenu d'un objet
objstore_put()	Écriture complète du contenu d'un objet
objstore_del()	Effacement d'un objet

Table I.1 – Les Appels de l'API OBJSTORE

Une implémentation simple va réaliser les mouvements de données au sein des appels `objstore_get()` et `objstore_put()`, mais cela peut s'avérer

problématique si l'on replace l'outil en perspective de l'ensemble de la solution intégrée au serveur NFS-Ganesha. Typiquement, une requête NFS de lecture d'un fichier peut déclencher in fine une action "GET" dans OBJSTORE. Si ce dernier est un object store comme Phobos, qui gère des bandes magnétiques, on est face à une opération très longue au regard du reste des traitements effectués par le serveur. Au niveau du serveur NFS-Ganesha, cela signifie qu'un thread du serveur va commencer de la ressource CPU pendant un long moment pour réaliser ce déplacement. Cette centralisation de l'exécution de l'action n'est pas souhaitable dans une architecture que l'on souhaite aussi distribuée que possible. Pour cette raison, le produit GRH a été développé.

I.5 .Le GRH

GRH est l'acronyme de *Ganesha Request Handler*. Il est destiné à recevoir des requêtes de mouvements de données initiés par la couche crud_cache, depuis ou vers un gestionnaire de stockage objet. Le GRH permet de coordonner ces requêtes et de gérer la reprise sur pannes.

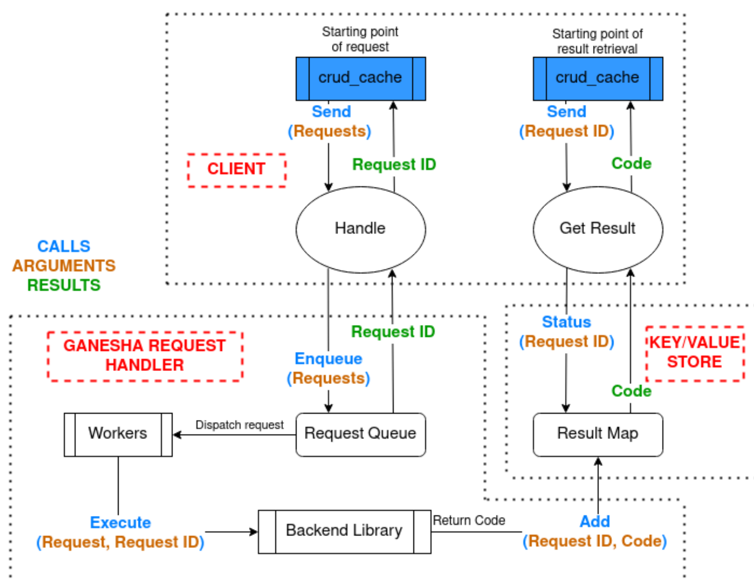


Figure I.2 – Fonctionnement des requêtes dans l'outil GRH

Le fonctionnement du GRH peut se résumer de la manière suivante, résumé dans la figure I.2

1. un acteur utilisant la couche crud_cache déclenche une action qui implique un mouvement de données ;
2. l'acteur crée une requête cliente asynchrone , un identifiant de cette

requête est alors produit. Cet identifiant permet de connaître le statut en cours de la requête;

3. le GRH va désigner un de ses *workers* et lui attribuer la requête qui est mise en fil d'attente. Son exécution sera réalisé en utilisant des fonctions exposées par la *Backend Library*;
4. lorsqu'une requête est terminée, un enregistrement de type "statut" est créé. Le client pourra obtenir cet enregistrement en présentant l'identifiant de requête.

Le fonctionnement du GRH est fondamentalement asynchrone, une fois que le client a obtenu un identifiant de requête il peut vaquer à ses propres occupations et aller vérifier à intervalles réguliers les états des requêtes actuellement en cours. La bibliothèque client permet également d'attendre de manière active, en implémentant une boucle de *polling* qui permet de réaliser un comportement synchrone au niveau du client.

Ce mode de fonctionnement présente différents avantages :

- une garantie de passage à l'échelle, ou *scalabilité*, il suffit d'ajouter plus de workers. Par ailleurs la gestion des requêtes et des résultats s'appuient sur des KVS qui passent, eux aussi, à l'échelle;
- un fort parallélisme et la possibilité d'exploiter tous les liens réseaux et toute la bande passante réseau disponible;
- une forte adaptabilité, si une machine faisant fonctionner une partie du GRH venait à dysfonctionner, il est possible de réduire la configuration, voire de la modifier pour lui faire prendre en compte des machines différentes;
- une forte flexibilité, car il est simple de mettre en œuvre différents *backends* en écrivant simplement un greffon sous la forme d'une nouvelle *Backed Library*;

Gestion des requêtes dans la GRH

Les requêtes du GRH ne sont que de trois types, calqués sur la sémantique CRUD, à savoir PUT[3], GET et DEL.

Les requêtes sont représentées par des enregistrements dans un *Key-Value Store*. ELles peuvent exister dans les états suivants :

- **PENDING** : la requête a été soumise par un client mais elle n'est pas encore exécutée;
- **RUNNING** : la requête a été affectée à un *worker* qui est en train de l'exécuter;
- **COMPLETED** : la requête est terminée et elle s'est réalisée sans produire d'erreurs;

3. La création qui constitue le "C" de l'acronyme CRUD, sera réalisée par la première requête PUT, les suivantes réalisant des "updates" qui correspondent à la lettre "U" CRUD

- **FAILED** : la reqûete est terminée mais elle s'est achevée sur une erreur ;

Gestion des requêtes redondantes

Le GRH implémente un mécanisme de gestion de requêtes dupliquées qui garantit que les utilisateurs ne vont pas saturer le système en émettant sans cesse les mêmes requêtes. Dans le principe, on est très proche du fonctionnement du DRC implémenté dans NFS-Ganesha et exposé dans la section 1.5.3. On conservera ainsi une liste des requêtes précédemment gérées ainsi que leurs résultats. Ceci seront simplement réémis. Ce DRC évitera en particulier de faire plusieurs fois le même mouvement de données.

Implémentation

Le GRH est essentiellement écrit en Python 3. Il s'appuie sur le logiciel Celery [4] qui fournit un gestionnaire de tâches distribués et hautement scalable.

Les *Backend Libraries* disponibles inclus le support de MOTR[84] et d'object store disposant d'une interface basée sur le protocole S3[55]. Le support d'un backend exploitant un système de fichiers POSIX est également disponible.

L'implémentation actuelle de la gestion des états s'appuie sur le Key-Value Store REDIS [5]. Celui-ci est géré nativement par le logiciel Celery. Ce dernier rend possible l'utilisation d'autre outils dont RabbitMQ, mais aussi MongoDB[20], Amazon SQS[48], CouchDB[5] de manière expérimentale.

Le serveur GRH expose une interface de type REST[27], implémentée avec la bibliothèque Flask [6]. L'utilisation de WSGI [7] qui semble présenter une meilleure scalabilité et permet de gérer plus facilement le support de services d'authentification comme Kerberos.

4. Voir la page https ://github.com/celery/celery
5. Voir la page https ://redis.io/
6. Voir la page https ://flask.palletsprojects.com/en/2.2.x/
7. Voir la page https ://wsgi.readthedocs.io/en/latest/what.html

Annexe J : Gestion de la sécurité dans les IO Proxies

Le concept d'IO-Proxy vient, par construction, segmenter à la fois les données, mais aussi les utilisateurs. En effet, un IO-Proxy est dédié à un job de calcul, et il ne fonctionnera que vis-à-vis de lui. On s'adresse donc à une sous-population qui n'exploite qu'une sous-partie bien identifiée des données. Cette considération permet la mise en œuvre de modèles de sécurité plus spécialisés.

J.1 .Les conteneurs chiffrés

Les conteneurs chiffrés exploitent la segmentation appliquée aux données. Celles-ci sont regroupées sous la forme d'ensemble aux contours clairement définis nommés *datasets* ou conteneurs. Ces derniers, qui regroupent des données destinées à être utilisées conjointement, peuvent requérir d'être protégées. Le sujet a été étudié par le CEA et la société ATOS dans le cadre de leur convention commune de R&D. Ainsi, la fonctionnalité de *conteneurs chiffrés* a été étudiée et mise en place.

Dans les grandes lignes, il s'agit d'utiliser la fonctionnalité Linux Unified Key Setup (LUKS) [6] du noyau Linux. Cette dernière permet de chiffrer de manière robuste un disque [1]. Le chiffrement s'appuie classiquement sur la connaissance d'un secret dont la connaissance permet d'accéder aux informations. Dans le cas de LUKS, il s'agit d'un mot de passe [2] qui doit être tapé par l'utilisateur. Le cas d'usage typique est celui du disque d'un ordinateur portable dont l'un des disques, lequel peut être le disque système de la machine, qui doit être déchiffré au démarrage de celle-ci. La gestion du secret sur laquelle est basé le chiffrement est donc géré de façon fondamentalement interactive. Le travail conduit avec ATOS se découpe en deux volets :

- LUKS sera modifié de façon à permettre de déverrouiller le chiffrement non pas grâce à un mot de passe, mais grâce à un keytab, abréviation de *key table*. Il s'agit d'un fichier dont l'exposition du contenu permet de résoudre le secret. On pourrait dans une certaine mesure comparer un *keytab* avec un badge dont la possession permet d'accéder à un local ou un bâtiment.
- le serveur NFS-Ganesha, utilisé pour implémenter les Io-Proxies, sera modifié de telle sorte à être capable de référencer les *keytabs* vis-à-vis

1. Ce dernier peut être un disque virtuel ou une partition exposée par un contrôleur RAID par exemple

2. Il est intéressant de noter que LUKS supporte les mots de passe multiples, ce qui permet de spécialiser ceux-ci en fonction des populations

des conteneurs chiffrés et d'ouvrir ceux-ci. Le serveur utilisera la fonctionnalité décrite au point précédent pour déchiffrer le conteneur par le biais d'une FSAL spécifique. Dans ce cas précis, il s'agira d'une *FSAL empilable* (ou *stackable FSAL*) dont le principe est décrit dans la section 2.8.4

Le fichier de configuration du serveur NFS-Ganesha exploité comme IO-Proxy décrira, dans le fichier de configuration, les caractéristiques du conteneur chiffré. On trouvera ainsi le chemin vers le conteneur, lequel est implémenté sous la forme d'un disque virtuel, donc d'un fichier, et les ressources permettant d'utiliser le keytab requis et qui sont dépendantes des fonctionnalités du gestionnaire de sécurité exploité.

Initialement développée dans le cadre d'une exposition des données selon la logique des systèmes de fichiers, la fonctionnalité de conteneurs chiffrés pourrait trouver une nouvelle utilisation avec l'émergence actuelle du stockage objet.

J.2 .9p/krb5

Le travail évoqué ici fait écho à l'utilisation de krb5 pour chiffrer le trafic NFS. NFS-Ganesha implémente cette fonctionnalité en totalité, dans toutes ses déclinaisons[50]. Les IO-Proxies s'appuient sur le protocole 9p2000.L qui ne supporte pas krb5 a priori. Il serait toutefois intéressant de disposer de cette fonctionnalité pour garantir que seuls les membres de la population disposant des bonnes autorisations peut accéder au serveur.

Implémenter la fonctionnalité dans NFS-Ganesha est toutefois simple et peut se faire sans modifier le protocole lui-même, ce dernier dispose en effet de tout ce qui est nécessaire pour mettre krb5 en œuvre. Pour ce faire, on utilisera la requête 9p `ATTACH` mais en lui adossant un argument construit à partir de la requête `AUTH`. Les détails de l'implémentation sont décrits dans le paragraph suivant.

J'ai réalisé cette implémentation à titre de test et de "preuve de concept" en modifiant le code des requêtes `ATTACH` et `AUTH` dans le code de NFS-Ganesha et dans le code du client 9p présent dans le noyau Linux. Si elle se montre fonctionnellement viable, cette fonctionnalité suppose de recopier des tampons mémoires (afin d'encoder et de décoder les messages du protocole) avec un impact désastreux sur les performances. En particulier, les optimisations basées sur le paradigme *zero-copy*[3] deviennent inefficaces. Elle sera abandonnée, le support de Kerberos 5 ne sera plus utilisé qu'au travers du seul protocole NFSv4, qui ne souffre moins de l'injection des fonctionnalités de sécurité.

3. favorisées par RDMA

J.3 .Gestion de la sécurité dans 9p

La gestion de la sécurité est possible dans le protocole 9p2000.L, les requêtes ATTACH et AUTH disposent en effet de tous les outils nécessaires pour intégrer un gestionnaire de sécurité tels que le support de Kerberos 5.

La requête ATTACH du protocole 9p2000.L peut être décrite de la manière suivante :

```
size[4] Tattach tag[2] fid[4] afid[4] uname[s]
                aname[s] n_uname[4]
size[4] Rattach tag[2] qid[13]
```

Les arguments en sont les suivants :

- **tag** est l'étiquette de la requête. Les requêtes qui présentent le même tag sont généralement utilisées ensemble ;
- **fid** est la nouvelle fid qui doit être construite par cette requête, elle est proposée par le client ;
  uname est une chaine de caractères qui permet d'identifier la ressource que l'on souhaite accéder. Dans le cas d'une exportation d'une ressource adossée à la VFS il s'agira d'un chemin vers un système de fichiers ou une sous-arborescence d'un système de fichiers sur le serveur ;
- **aname** désigne l'utilisateur qui souhaite créer la nouvelle fid avec cette requête ;
- **afid** est une fid à partir de laquelle on va construire cette nouvelle fid. Il peut s'agir d'une fid précédemment négociée ou de la constante P9_NOFID qui dénote l'absence de fid.

L'argument afid mérite une attention particulière. En effet, il peut être construit par le truchement de la requête AUTH. Le protocole de celle-ci est le suivant :

```
size[4] Tauth tag[2] afid[4] uname[s]
              aname[s] n_uname[4]
size[4] Rauth tag[2] aqid[13]
```

En première approche, cette requête est très similaire, pour ne pas dire identique à la précédente, puisqu'elle prend presque les mêmes arguments et semble fournir le même type de résultats. Si ATTACH est dédiée au fonctionnement des systèmes de fichiers, en fournissant une première fid à partir de laquelle commencer à parcourir une arborescence, AUTH permettra de négocier, du point de vue d'une fonctionnalité comme krb5, une fid qui sera le résultat final de cette négociation. Elle sera par la suite exposée comme l'argument *afid* d'un ATTACH.

Un montage NFS kerberisé, dans sa modalité la plus simple, permet de garantir que l'acteur qui souhaite réaliser le montage NFS est bien celui qu'il prétend être. On passera les *credentials* nécessaires comme argument aname

de AUTH afin d'entamer ce processus. Dans le cas d'un montage 9p/krb5 ainsi implémenté, les requêtes AUTH qui ne présentent pas une afid viable (ce qui inclut P9_NOFID) seront rejetées.

Il serait techniquement possible d'implémenter les fonctionnalités de garantie d'intégrité (krb5i) et de chiffrement (krb5p). Comme décrit dans la section F, 9p et 9p2000.L sont des protocoles basés sur la gestion de buffers. Il serait parfaitement faisable de chiffrer et de déchiffrer ceux-ci avant et après leur transmission sur le réseau, ou de leur adosser une somme de contrôle pour garantir l'intégrité du message. Cette fonctionnalité serait a priori compatible avec la fonctionnalité 9p/RDMA décrite dans la section 2.10.2.

Annexe K : Distributions logicielles

Ce manuscrit mentionne plusieurs logiciels, dont NFS-Ganesha. Cette section recense les emplacements où trouver et télécharger ces logiciels. Ces derniers sont fournis sous la forme de dépôt GIT, hébergés sur le site `github.com`

Site Communautaire NFS-Ganesha : https://github.com/nfs-ganesha

Ce site est le site de la communauté NFS-Ganesha. C'est une "organisation", au sens de GitHub, qui regroupe le produit NFS-Ganesha et plusieurs produits connexes (tels que la bibliothèque `ntirpc`). C'est la base de référence que doivent utiliser les développeurs collaborant autour du produit NFS-Ganesha. Il dispose en outre d'un wiki assez exhaustif pour quiconque veut se familiariser avec le produit.

Site du projet IO-SEA : https://github.com/io-sea

Il s'agit de l'organisation (au sens de GitHub) qui regroupe l'ensemble de produits exploités dans le projet IO-SEA décrit à la section 3.11. Il s'agit de la base de référence affichée par le projet pour ses distributions logicielles.

Mon propre dépôt NFS-Ganesha : https://github.com/phdeniel/nfs-ganesha

Il s'agit là de mon propre dépôt NFS-Ganesha. En plus d'être le "dépôt" originel à partir duquel a été créé celui de la communauté NFS-Ganesha, c'est également l'endroit qui regroupe les modifications que j'apporte moi-même au produit.

Mon propre dépôt KVSNS : https://github.com/phdeniel/kvsns

Ce dépôt est l'endroit où est née la bibliothèque KVSNS, avant d'être répliqué et exploité dans le projet IO-SEA. Comme pour les autres logiciels, il s'agit de la zone de travail dans laquelle figure mes modifications les plus récentes du produit.