Rapport d'étude du système de stockage distribué CEPH

Mélody BALLOUARD, Elliot BARTHOLME,
Pierre BATHELLIER, Mathilde LONGUET,
Louise NAUDIN, Virgile VANÇON

SR05 - P17

Résumé

L'objet d'une étude proposée en *Algorithmique et Systèmes Répartis* à l'Université de Technologie de Compiègne concerne le système de stockage distribué libre et open source Ceph, initialement inventé par Sage Weil. Application directe de l'enseignement théorique dispensé pendant le semestre, ce projet en autonomie réalisé à six est rendu compte au travers de ce document.

Il présentera les motivations qui ont conduit à produire cette solution, ainsi que ces principaux concurrents. Son architecture et les rôles de ses composants seront ensuite expliqués, tout comme le fonctionnement de ses différents algorithmes. L'étude sera conclue par une présentation de l'installation sur des machines de l'école puis d'une soutenance orale.

Table des matières

1 Introduction

Ceph est un système de stockage distribué à forte résilience, haute disponibilité et durabilité des données, qui est polyvalent et ne nécessite pas de matériel spécialisé. Performant et auto-gérant, il automatise toutes ses tâches afin de réduire les coûts d'exploitation et fonctionner de manière autonome ; c'est aujourd'hui une vraie solution informatique pour répondre aux problématiques de stockage en masse distribué.

Il est le précurseur des solutions Software Defined Storage (SDS) en open source, des systèmes qui se basent sur la séparation du stockage physique et concret des données de l'intelligence liée à leur gestion. Ces derniers offrent de grandes capacités de dimensionnement puisqu'ils permettent de virtualiser des ensembles de noeuds physiques en *cluster*, et bénéficient d'autre part d'une forte résilience puisque les données sont sauvegardées à différents endroits et que lors de pannes de disque la performance de ses algorithmes et de son fonctionnement lui permettent de se reconstruire.

Fruit du travail de thèse de Sage Weil, soutenue en 2004, la première version stable de Ceph n'a pu être sortie qu'en 2012, après plusieurs années de développement à temps plein. En 2014, c'est Red Hat qui reprend la plupart des développements du système. Depuis 2012, ce n'est pas moins de 10 versions majeures qui ont été construites et publiées.

Supporté par la plupart des systèmes Linux, Ceph agit comme une interface de bas niveau avec des supports de stockage classiques en les rendant *intelligents*. Ils forment des entités appelées *Object Storage Devices OSD*, qui organisent les données dans des conteneurs et utilisent des identifiants pour s'affranchir de la connaissance de leur emplacement physique, et offrir une interaction unique avec un système de stockage entièrement distribué.

2 Motivations et concurrents

2.1 Principes sous-jacents

Il est important de comprendre les problématiques qui ont permis à Ceph de voir le jour. Les systèmes de fichiers distribués récents ont des architectures qui reposent pour la plupart sur le stockage d'objets. Dans ces systèmes, les disques durs sont remplacés par des OSD qui sont *intelligents* : ils rassemblent un processeur, une interface et enfin un disque dur ou un RAID. Les clients interagissent la plupart du temps avec un serveur de méta-données (cf. 2) pour les opérations globales et la localisation des données, et interagissent directement avec les OSDs pour des lectures et des écritures. Ces systèmes ont une capacité d'évolution limitée car la charge de travail des serveurs de méta-données n'est pas répartie.

Ceph repose sur une nouvelle approche, qui offre des performances bien meilleures en termes de rapidité, de fiabilité et de capacité d'évolution grâce à la répartition de l'intelligence du système entre les différents OSDs. En effet, il ne repose pas sur des méta-données gardant la trace de chaque élément stocké, mais sur un algorithme de placement.

Mais pourquoi de tels systèmes de stockage ?

Aujourd'hui, l'humanité produit autant d'informations en 2 jours qu'elle ne l'a fait en 2 millions d'années. Alors que les données étaient encore stockées sur des disques durs de quelques méga-octets il y a vingt ans, les besoins de certaines entreprises se comptent aujourd'hui en péta-octets. Il y a donc de nouvelles demandes réelles de stockage en masse, auxquelles de nouvelles problématiques se sont associées.

Lorsque ces besoins sont apparus, il a fallu augmenter la capacité de stockage des machines. La première idée a été de relier des multitudes de disques durs à une même machine, afin d'augmenter sa capacité (l'accès à plusieurs disques se fait donc via le même ordinateur). Quand plusieurs clients veulent y accéder, la machine doit alors gérer les interactions avec chaque client ainsi qu'avec les données. Les ordinateurs classiques ne possèdent pas la puissance de calcul nécessaire.

Ces machines ont alors été remplacées par des super-calculateurs, extrêmement puissants et rapides mais également très onéreux. Cela pose deux problèmes : le coût de mise en place et la fragilité de ces systèmes, car leur intégralité dépend de la machine centrale. C'est à ce moment qu'ont été introduits les systèmes distribués et décentralisés, qui permettent de répartir les données et la charge de travail sur un ensemble de machines moins puissantes. Cette solution offre à la fois robustesse et fiabilité, ainsi que des coûts d'installation bien plus faibles.

Le problème est alors que ces systèmes sont basés la plupart du temps sur du matériel propriétaire qui nécessitent des efforts de maintenance importants, ce qui se révèle souvent onéreux. Prenons l'exemple d'EMC, qui est une entreprise d'informatique américaine leader des solutions de stockage distribué, basées sur du matériel et du logiciel propriétaire. En 2012, 34% de son chiffre d'affaire (c'est-à-dire 5.2 milliards de dollars) provenait de la maintenance et du support technique apporté à ses clients, tandis que 1.1 milliards de dollars était dépensé en recherche et développement sur leurs logiciels.

Ceph propose une nouvelle approche au monde du stockage distribué : une couche

logiciel libre et open source reposant sur du matériel standard. En étant open source et surtout *community-focused*, Ceph est développé en collaboration avec ses utilisateurs et les membres de la communauté des logiciels libres : les dépenses de l'entreprise en RD sont alors peu élevées.

Ceph a été développé pour répondre à plusieurs critères : être complètement open source, être *community-focused*, pouvoir évoluer facilement, ne présenter aucun point central critique afin d'être un système fiable et robuste, et également être auto-gérant. Ce dernier point important est né du constat que face à des ensembles de machines aussi importants, le système doit pouvoir faire face à des pannes matérielles de manière autonome.

Ceph a tout d'abord été utilisé dans le monde de la recherche. La transition pour les entreprises peut être plus lente car elles sont très vigilantes sur les environnements et les architectures de leurs systèmes de stockage. La procédure de validation pour changer d'environnement est longue et souvent coûteuse, ce qui ralentit la transition vers des systèmes plus modernes comme Ceph.

2.2 Concurrents

L'objectif de cette partie est de décrire de manière concise quelques uns des concurrents de Ceph sur le marché. Ces présentations permettent de prendre du recul sur le système Ceph en observant d'autres architectures et d'autres façons de penser le stockage distribué. Dans ce rapport, trois solutions de stockage distribué seront présentées en détails : GlusterFS, Lustre et StorPool. Ils ne constituent évidemment pas une liste exhaustive, mais leurs spécificités nous permettront de mieux apprécier les qualités et les défauts de Ceph.

GlusterFS

Contrairement à Ceph qui est un système de stockage d'objets étendu aux *blocs* (pour le stockage de machines virtuelles par exemple) et aux fichiers, GlusterFS (pour Gluster File System) est un système de stockage open source distribué de fichiers étendu aux objets. Il est capable de stocker une quantité très importante de données, jusqu'à plusieurs pétaoctets (10^{15} octets). Tout comme Ceph, il est développé par Red Hat, qui le décrit comme étant très résistant, performant, et pouvant facilement évoluer. Son architecture est intéressante, car contrairement à la plupart des systèmes de stockage il ne possède pas de serveur ou de fichier central gardant une trace de la localisation de chaque objet (comme Ceph). Ce premier point rend Gluster résistant, car il évite d'avoir un point central très fragile qui paralyserait tout le système en cas de panne. Le parallèle avec Ceph est évident : Gluster localise les fichiers qu'il stocke de manière algorithmique, en utilisant une fonction de hachage particulière, ce qui n'est pas sans rappeler l'algorithme CRUSH. Il ne possède donc pas de serveur central de méta-données, ce qui lui assure d'être performant et sûr.

Gluster utilise donc le stockage par blocs, qui distribue un ensemble de données hachées sur des ordinateurs Linux connectés. Il permet de stocker d'énormes quantités de données de manière sécurisée, tout en les gardant très accessible. Une étude menée en avril 2014 par IOP Science a montré que Gluster surpasse Ceph en termes de performances, mais

possède encore des instabilités entraînant des pertes de données partielles ou totales [1].

GlusterFS est donc une solution à privilégier pour stocker de gros volumes de données qui n'ont pas vocation à changer au cours du temps, alors que Ceph sera une meilleure solution pour le stockage rapide et facile d'un ensemble de données présentant beaucoup d'évolutions.

Lustre

Lustre (réunion de *Linux* et *Cluster*) est un système de fichiers distribué libre, généralement utilisé pour de très grands ensembles de serveurs car il est très performant. Il a été conçu pour fonctionner sur des centaines de noeuds différents, sans altérer la sécurité ou la vitesse du système. Initialement développé par la société Cluster File System, il a été racheté par Sun Microsystems puis par Oracle. Lustre est désormais maintenu par la communauté open source ainsi que certaines entreprises spécialisées.

L'objectif de Lustre est de gérer très rapidement des ensembles de données très importants : c'est pourquoi 8 des des 10 super-ordinateurs les plus rapides du monde (et 70% des 100 premiers) dépendent de Lustre pour le stockage de leurs données. La principale qualité de Lustre est donc sa rapidité, que Ceph ne peut même pas approcher. En revanche, il possède un serveur central de méta-données qui représente un gros point faible du système, car il est chargé de vérifier l'authentification des clients qui demandent des accès aux fichiers stockés puis de leur indiquer la localisation des fichiers en question. En cas de panne, tout le système est paralysé [2].

StorPool

StorPool est un logiciel de stockage intelligent très performant. C'est une pure solution logicielle, qui peut utiliser n'importe quel serveur ou système de stockage. Il rassemble de manière dynamique des serveurs au sein d'un cluster (qui peuvent être ajoutés ou retirés à tout moment), et permet d'utiliser les mêmes serveurs pour le stockage et les calculs. Si StorPool a été initialement développé comme un système de stockage d'objets, il a désormais évolué pour devenir un système de stockage unifié, c'est-à-dire pouvant stocker à la fois des objets, des blocs et des fichiers. La technologie StorPool, découpe les données entrantes en blocs de taille fixe et les distribue entre les différents nœuds. Chaque donnée est répliquée entre deux et cinq fois selon les préférences de l'administrateur, de façon à protéger le cluster contre la défaillance d'un ou plusieurs disques ou d'un nœud complet. Le logiciel embarque également un API REST permettant de faciliter son intégration avec les technologies Web.

Le principal avantage de StorPool est sa rapidité et ses performances, meilleures que celles de Ceph. Le système se voulant moins évolutif, il se débarrasse de certaines couches logicielles complexes de Ceph pour ne garder qu'un noyau simple et particulièrement efficace.

Il existe bien-sûr de nombreuses autres solutions de stockage distribué, comme Quobyte, ScaleIO, et bien d'autres encore. La comparaison avec d'autres systèmes est importante pour apprécier les particularités de Ceph mais aussi pour identifier des axes d'amélioration. Chaque système est conçu en général pour répondre à des besoins spéci-

fiques, comme la vitesse d'accès au données, la capacité de stockage, la disponibilité des données, etc... Si Ceph est très fiable et parfaitement adapté à des ensembles de données très changeant, d'autres solutions existent et le choix de la technologie à adopter dépend entièrement des besoins de l'utilisateur.

3 Architecture de Ceph

L'organisation du système avec la présentation de ses composants sera présentée dans cette partie.

Ceph repose sur un ensemble de noeuds serveurs qui constitue un *magasin intelligent* de stockage d'objets : Rados (*Reliable Autonomic Distributed Object Store*). Rados abstrait le stockage physique en unifiant différents supports de stockage physique sous la forme d'un seul cluster virtuel. Il permet de cacher la complexité du stockage et de la répartition des données en proposant une interface client d'accès plutôt simple et en offrant une forte échelonnabilité : il est possible d'ajouter facilement un grand nombre de périphériques de stockage.

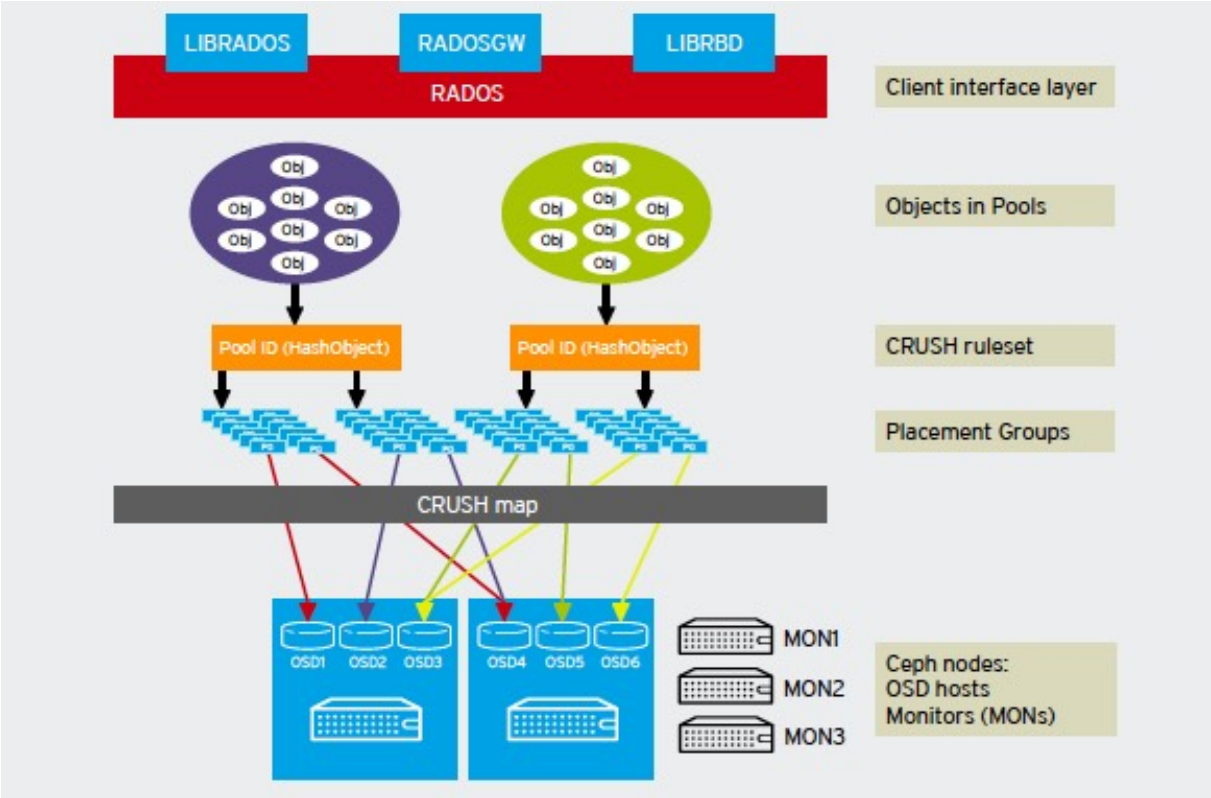


FIGURE 1 – Architecture de Ceph

Les noeuds serveurs qui composent Rados peuvent être de trois type : OSD, Monitor ou MDS.

Ceph OSD : comme ils ont été introduits, les *Object Storage Devices* sont des ordinateurs possédant un disque physique formaté avec un système de fichier (e.g : *xfs*) couplé à un système d'exploitation qui gère des processeurs permettant de faire des calculs répartis. Dans Ceph, à chaque disque physique est associé un daemon appelés parfois aussi OSD (*Ceph Object Storage Daemon*), la distinction n'étant pas toujours bien explicite. Ce daemon est chargé de stocker les objets sur le disque, de les répliquer, et de la redistribution des données en cas de panne.

Chaque OSD vérifie périodiquement l'état de santé des autres et fournit ces informations aux moniteurs. C'est ce que l'on appelle le battement de coeur (*heartbeat*). Chaque

daemon vérifie le battement de coeur des autres OSD toutes les six secondes en envoyant des messages (*Request To Peer*) ; si au bout de vingt secondes il n'a pas reçu le signal d'un de ses voisins il le considérera comme en panne, et le signalera à un moniteur qui pourra alors mettre à jour la carte du cluster.

Monitor : les noeuds moniteurs sont donc indispensables dans le cluster. Leur rôle est de maintenir à jour la carte du système dont le consensus sur la version commune entre les noeuds est défini grâce à l'algorithme *Paxos*. Il est préférable d'utiliser un nombre impair de moniteurs pour faciliter ce consensus ; avoir plus d'un moniteur est de même fortement recommandé afin de s'assurer de la disponibilité du cluster en cas de défaillance unitaire.

MDS : le *MetaData Server* n'est utile que lorsqu'on souhaite déployer un système de fichiers partagés (*CephFS*). Abordé en détail dans la partie 3.1.2, *CephFS* permet d'abstraire la représentation des zones de stockage d'un cluster en proposant un accès aux objets aux normes Posix. Le serveur MDS est chargé de stocker dans Rados les méta-données et la hiérarchie. Attention, il ne fournit pas au client les données stockées dans le *file system*, mais simplement ses méta-données.

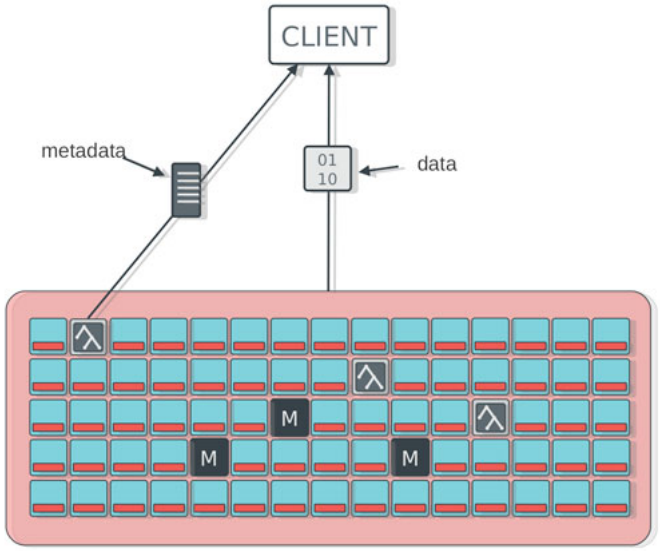


FIGURE 2 – Fonctionnement d'un Metadata Server

Les Pools : le cluster stocke les objets dans des groupes logiques appelés *pools*. Les pools peuvent permettent de définir un ensemble de règles et de propriétés sur les objets qui y sont stockés. Son paramètre principal est la résilience ou disponibilité des données, qui définit le nombre de fois que chaque donnée doit être répliquée. Elle permet de définir le nombre de noeuds serveurs pouvant tomber en panne tout en gardant un accès à l'objet en lecture ou en écriture. Ainsi si on réplique n fois une donnée sur n noeuds, le nombre de noeuds serveur pouvant tomber en panne est égal à $n - 1$. De manière un peu plus brutale, si un ordinateur disparaît, la donnée pourra toujours être récupérée sur une autre machine puisqu'elle a été dupliquée (facteur de durabilité).

Placement groups PG : les *placement groups* (groupes de placement) sont des conteneurs logiques qui ne possèdent pas d'équivalents dans le stockage physique. Ils sont la plus petite unité de stockage de Ceph et sont regroupés dans les pools, sachant qu'un placement group est indivisible physiquement (un PG appartient à un OSD seulement),

contrairement aux pools qui sont répartis sur plusieurs OSD (voir figure 1). Ils permettent donc de segmenter les objets pour répartir équitablement les données entre chaque noeud grâce à l'algorithme CRUSH (4.1).

Le nombre de *placement groups* par pool doit être défini à la création, sachant que le nombre idéal (arrondi à le puissance de 2 la plus proche) est en général :

$$n_{PGs} = \frac{100 * N_{OSD}}{nb_{replicate}}$$

Ce découpage et cette distribution des rôles assure à Ceph un fonctionnement optimal grâce à une coordination et des décisions performantes assurées grâce aux algorithmes. L'avantage de ce système est qu'il ne nécessite pas de matériel spécifique et peut être déployé simplement sur des machines physiques qui sont gérées intelligemment via le couplage à un daemon et l'association aux moniteurs.

3.1 Modes d'accès à Ceph

L'interface d'accès à Rados implémentée nativement offre des possibilités d'entrée basiques qui ne satisfont pas des besoins tels que l'accès en *cloud* par exemple. D'autres interfaces ont donc été développées dans l'optique d'adapter un système réparti basé sur Ceph dans des environnements différents.

3.1.1 Librados

Il est nécessaire avant de s'intéresser aux modes d'accès comme *CephFS*, *RBD* et *RadosGW*, d'introduire Librados. Librados est une API de bas niveau, native, qui fournit des outils de connexion à Rados en C, C++, Java, Python, Ruby et PHP. Elle permet concrètement d'interagir avec deux types de daemon du cluster : Les moniteurs et les OSD.

Bien qu'il soit possible de manipuler le cluster grâce à Librados, d'autres modes d'accès de plus haut niveau ont été mis en place dans le but de faciliter l'utilisation de Ceph et cacher la complexité de la gestion pour l'utilisateur. Tout ces modes d'accès utilisent les outils fournis par Librados et il est possible d'intégrer sur le même cluster plusieurs modes d'accès hétérogènes.

3.1.2 CephFS

CephFS ou Ceph FileSystem se base sur un concept couramment utilisé : les systèmes de fichiers. L'immensité d'un disque dur pose un problème de complexité pour l'utilisateur lorsqu'il veut localiser des informations ; le système de fichiers est donc un formatage logique du disque qui permet d'abstraire à l'utilisateur la manipulation des ses entités et des unités de stockage physique. Chacun possède, selon le système d'exploitation et ses spécificités, des conventions de nommage et une arborescence propre.

Dans Ceph, l'implémentation d'un système de fichiers est un moyen de simplifier l'accès à Rados avec une interface plutôt classique de navigation dans une arborescence.

L'approche est décentralisée pour améliorer la gestion des méta-données et équilibre des charges de travail. Les clients interagissent avec un *Metadata Merver* (MDS, 2) afin d'effectuer des opérationsn de renommage, changement de propriétaires... Cette approche permet uniquement de pré-visualiser du contenu. Afin d'effectuer des opérations de lecture/écriture (I/O) les clients communiquent directement avec les OSD. Cette séparation des rôles augmente l'échelonnabilité de manière significative.

CephFS pallie justement à ce problème, grâce à son approche décentralisée[3]. Ceph décuple les opérations sur les données et les méta-données en éliminant les fichiers de tables d'allocations et en les remplaçant par les fonctions CRUSH. Cela permet à Ceph de mettre à profit l'intelligence des OSD et de leur déléguer non seulement l'accès aux données, mais aussi la mise à jour, la sérialisation, les réplications, etc.

CephFS se base sur trois principales entités [4] :

Côté serveur de méta-données : le MDS sert en tant qu'index sur la carte des OSD afin de faciliter la lecture et l'écriture. Lorsqu'une mise à jour est faite sur un MDS, comme une création de fichier, le MDS stocke cette mise à jour dans les méta-données.

Il parcourt la hiérarchie d'un système de fichiers afin de traduire le nom d'un ficher en un *inode* de fichier. Un fichier est composé de différents objets et Ceph le généralise en une *map* reliant données du fichier et séquence d'objets.

Côté client : lorsque plusieurs utilisateurs ouvrent un même fichier en écriture en même temps, le MDS supprime toute possibilité de lecture avec le cache, afin que la lecture ne se base pas sur des données déjà dépassées. Il limite aussi les possibilités d'écriture bufferisée, afin de ne pas stocker les données temporairement en mémoire vive avant de les écrire par blocs sur les OSD. Ces deux actions forceront les I/O des clients à être synchrones.

Côté OSD : comme cela a été dit, Ceph délègue la responsabilité de la migration des données, de leur réplication et de leur détection d'erreurs aux OSD, qui stockent les données et offrent aux clients et au serveur de méta-données une seule adresse logique pour les retrouver.

CephFS n'est disponible que depuis la version Jewel, et a apporté un nouveau type d'accès encore plus intuitif au stockage des objets dans Rados.

3.1.3 Radosgw

Radosgw est aussi une interface de stockage d'objets dans un cluster Ceph. Contrairement au stockage en mode fichier qui s'appuie sur une recherche dans des répertoires hiérarchisés, le stockage en mode objet gère les données comme des objets ou *bucket*.

Chaque objet contient des données, des méta-données et un identifiant unique grâce auquel on peut le nommer, et le retrouver. Basé sur Apache et FastCGI, c'est concrètement un serveur HTTP, le *Ceph Objet Gateway Daemon*, qui sert de porte d'entrée au cluster avec une architecture REST (URIs, requêtes *http*...)

Conjointement aux serveurs web, cette interface fonctionne bien avec des APIs compatibles avec Amazon S3 ou OpenStack Swift et permet d'associer par exemple un *cloud* entier avec un système de stockage Ceph.

3.1.4 RBD

Le Ceph's **R**ados **B**lock **D**evice permet un mode de stockage en mode bloc (images contenant des objets créées sur disque). Il consiste à stocker les données dans des volumes appelés blocs, qui sont tous contrôlés par un serveur de gestion. Cela permet en pratique de monter un volume d'accès à Rados comme disque local sur un client Linux.

Des intégrations sont fournies pour permettre un stockage en mode bloc à des machines virtuelles. La gestion des images disque est présentée plus en detail dans la partie 4.2.

Ceph propose donc plusieurs interfaces et modes d'accès à Rados, le coeur du stockage. Il est important de mentionner les différents algorithmes qui concourent à la vivacité et la sûreté du système ; c'est l'objet de la partie suivante.

4 Algorithmes répartis

Ceph est un système de stockage distribué ne possédant pas de serveur central : comme les différentes ressources du systèmes sont réparties, plusieurs algorithmes (eux-mêmes répartis) ont été développés. Les trois algorithmes principaux de Ceph qui seront présentés ici sont l'algorithme CRUSH de placement de données, l'algorithme RBD pour la réalisation de snapshots, et enfin l'algorithme PAXOS chargé des consensus entre moniteurs. Le fonctionnement de la détection et remontées de pannes sera également présentée.

4.1 CRUSH : algorithme de répartition des données

4.1.1 Présentation

CRUSH (abréviation de « Controlled Replication Under Scalable Hashing »)[5], est l'algorithme chargé de déterminer le placement des données dans un cluster Ceph, que ce soit en lecture ou en écriture. Il est sans doute l'algorithme implémenté par Ceph le plus important, rendant ce système de stockage à la fois unique, puissant et surtout autogérable. En effet, en cas de panne d'un composant du cluster, CRUSH va se charger, sans aucune intervention extérieure, de déterminer les conséquences de cette panne ainsi que de récupérer les données éventuellement perdues. Grâce à CRUSH, Ceph est un système de stockage extrêmement fiable, qui ne comporte aucun point central de défaillance.

Lorsque le cluster Ceph reçoit une demande d'écriture, CRUSH va déterminer les différents emplacements sur lesquels les données doivent être stockées. Il va ensuite transmettre ces emplacements à Rados, qui se chargera lui d'écrire les données sur les différents OSD. C'est un algorithme pseudo-aléatoire (c'est-à-dire que ses résultats s'approchent d'un aléa statistiquement parfait) de distribution de données, qui distribue des réplicas d'objets dans un cluster structuré et composé d'éléments hétérogènes.

CRUSH est une fonction déterministe, qui associe à une valeur d'entrée (typiquement l'identifiant d'un objet ou d'un groupe d'objet) une liste de machines sur lesquelles les données doivent être stockées. CRUSH n'a besoin que d'une description courte et hiérarchique des disques constituant le cluster, et des règles de placement des réplicas. Cette approche a deux avantages majeurs : premièrement elle est complètement distribuée, c'est-à-dire que n'importe quelle entité du système peut calculer de manière indépendante la localisation de n'importe quel objet du cluster ; deuxièmement, elle ne nécessite que très peu de méta-données, qui seront dans tous les cas stockées de manière statique et qui ne changeront que lorsque la carte du cluster sera modifiée, *i.e.* lorsque qu'un disque sera ajouté ou enlevé. Il n'y a donc pas de serveur ou de fichier central qui garde une trace de l'emplacement de chaque donnée : CRUSH peut les retrouver lui-même.

L'algorithme est conçu pour distribuer de manière optimale et uniforme les données sur les différentes ressources disponibles. Il les réorganise lorsque des disques sont ajoutés ou enlevés du cluster, et impose des contraintes paramétrables sur le placement des réplicas d'objets, maximisant la sûreté des données en cas de panne matérielle. Différents mécanismes de sûreté des données sont supportés (différentes techniques de stockage),

comme la « n-way replication », l'erasure coding, etc... CRUSH a donc été conçu pour gérer la distribution d'objets dans des systèmes de stockage importants, dans lesquels l'évolution, la performance et la fiabilité sont d'une importance critique.

4.1.2 Comparaison avec les systèmes existants

Plusieurs systèmes sont, comme Ceph, basés sur le stockage d'objets. Ils utilisent pour la majorité d'entre eux des approches semi-aléatoires ou basées sur des heuristiques pour placer leurs nouvelles données sur les différents disques de sauvegarde. Cependant, ces données sont rarement déplacées lors de l'ajout ou du retrait d'un disque, ce qui n'assure pas une distribution équilibrée sur l'ensemble des disques.

Le plus gros inconvénient de ces systèmes est que, contrairement à CRUSH, ils localisent toutes leurs données via une sorte de répertoire central de méta-données, qui constitue alors un point de défaillance vulnérable. Pour retrouver ses données, CRUSH ne repose que sur la description du cluster et sur une fonction de mapping déterministe. Cet avantage est particulièrement intéressant lors de l'écriture de données, puisque n'importe quel système utilisant l'algorithme peut calculer la localisation d'une nouvelle donnée sans avoir à consulter un quelconque fichier central.

4.1.3 L'algorithme CRUSH en détail

L'algorithme CRUSH distribue les données au sein d'un cluster en assurant une distribution uniforme des données sur les différentes machines. Chacune possède un *poids*, qui représente la quantité de données que cette machine pourra stocker. La distribution est contrôlée par une carte du cluster dite hiérarchique, et que l'on pourrait représenter sous la forme d'un arbre n-aire. Le placement des données est défini en fonction de règles de placement, qui spécifient combien de réplicas les données doivent avoir et quelles restrictions sont imposées sur le placement de ceux-ci (par exemple, certaines règles imposent que les réplicas soient placés dans des armoires physiques différentes pour qu'ils ne partagent pas le même circuit électrique et ainsi éviter des pertes en série en cas de panne).

Étant donnée une seule valeur d'entrée entière x, CRUSH retournera une liste ordonnée $\overrightarrow{R}$ de n unités de stockages. L'algorithme utilise tout d'abord une puissante fonction de hachage prenant également x en entrée et qui est chargée de le hacher en différentes parties qui seront stockées dans les *placement groups*. Le mapping est complètement déterministe et indépendamment calculable par n'importe quelle entité du système possédant la carte à jour du cluster, les règles de placement et bien sûr x. CRUSH est répétable, c'est-à-dire que le même input donnera toujours les mêmes outputs, et stable car il assure un déplacement minimal des données en cas de changement(s) dans la carte du cluster.

a) Carte hiérarchique du cluster

La carte hiérarchique du cluster est composée de machines et de ce qu'on appelle des buckets (i.e. groupes), possédant tous deux des identifiants numériques et des poids

associés. Elle a pour but de représenter les différentes entités du système sous la forme d'un arbre *n-aire*, et permet à CRUSH d'identifier les zones à risque du cluster.

Les buckets peuvent contenir un nombre quelconque de machines mais également d'autres buckets, de façon à former des nœuds dans un stockage hiérarchique qui peut donc être représenté comme un arbre, avec les machines toujours placées aux extrémités. Les poids des différentes machines sont affectées par l'administrateur ; ils servent à contrôler la quantité de données que cette machine peut stocker.

Le poids de chaque machine est obtenu à partir de ses capacités, et le poids des buckets est obtenu en additionnant le poids de tous ses composants. Au sein d'un cluster de grande taille, il peut exister des machines avec des capacités et des performances différentes, et l'utilisation de chaque machine est statistiquement corrélée avec la charge de travail du cluster : la charge de cahcune est donc en moyenne proportionnelle à la quantité de données stockées, ce qui évite la surcharge de travail de certains ordinateurs.

Un bucket est donc une entité logique composée d'autres entités logiques ou de machines physiques. Ces machines physiques peuvent par exemple commencer par être regroupées dans un premier bucket selon leurs caractéristiques, puis on peut ensuite regrouper des buckets partageant le même circuit électrique d'alimentation, etc... Cette construction apportant à la fin une carte hiérarchique.

CRUSH est basé sur quatre types de bucket différents, possédant chacun un algorithme de sélection particulier, servant à traiter le mouvement des données inhérent à l'ajout ou au retrait d'un à plusieurs OSD. C'est également ce qui différencie CRUSH de la plupart des autres techniques de *hashing*, dans lesquelles un changement du nombre de machines entraîne une réorganisation massive des données.

b) Les différents types de "bucket" :

CRUSH a été conçu pour répondre notamment à deux problématiques : tout d'abord être efficace et pouvoir évoluer facilement, mais également assurer un déplacement minimum des données en cas de changement dans la carte du cluster (après l'ajout ou la suppression d'un ou plusieurs OSD).

C'est dans ce but que CRUSH implémente 4 types de buckets différents, chargés de représenter les nœuds logiques dans l'arbre modélisant la carte hiérarchique du cluster. Ce sont les bucket uniformes, les « list buckets », les « tree buckets » et les « straw buckets ». Chacun de ces types de bucket implémente des structures de données différentes, mais surtout des fonctions c(r, x) différentes. Ces fonctions sont chargées de retourner, de manière pseudo-aléatoire, un élément interne au bucket lors du processus de placement des réplicas.

c) Choix du type de bucket

Si les buckets n'ont pas vocation à être modifiés, alors le type uniforme est le plus approprié. Dans ce type de bucket, toutes les machines possèdent le même poids. Si l'on s'attend à ce que les buckets s'agrandissent, alors les *list buckets* sont les plus adaptés car ils offrent un déplacement optimal de données en cas d'ajout de machine(s). Si l'on s'attend au contraire à ce que les buckets diminuent, les *straw buckets* constituent un bon choix, car ils offrent une migration optimale entre les sous-arbres du bucket. Enfin, les *tree*

buckets offrent une solution intermédiaire aux trois autres, offrant une bonne performance quel que soit la configuration.

Action	Uniform	List	Tree	Straw
Speed	O(1)	O(n)	O(log n)	O(n)
Additions	poor	optimal	good	optimal
Removals	poor	poor	good	optimal

FIGURE 3 – Performances des différents types de bucket

CRUSH a été conçu pour distribuer des données de manière uniforme sur un ensemble de machines pondérées, afin de maintenir une utilisation statistiquement équilibrée de chacune et ainsi prévenir au maximum les pannes. Le placement des réplicas a évidemment une importance capitale sur la sûreté des données en cas de panne matérielle.

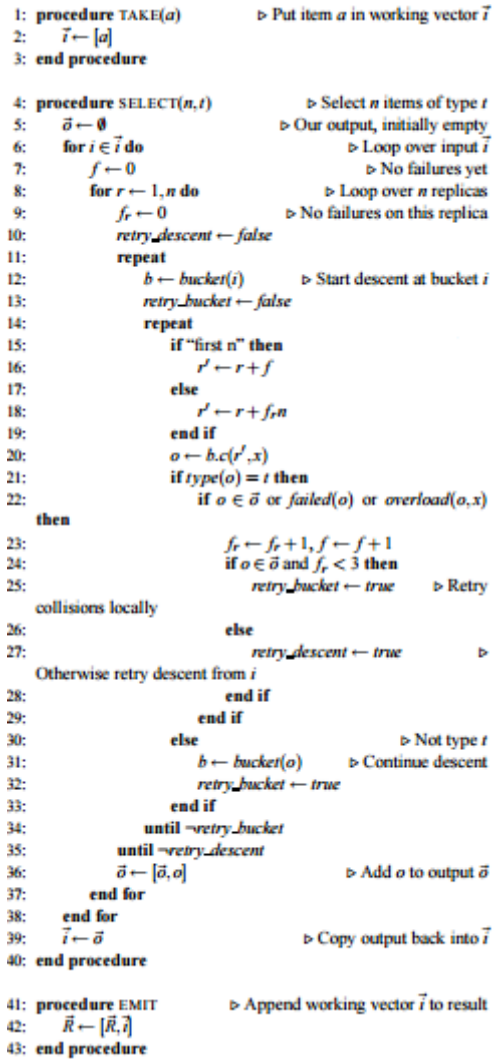
CRUSH analyse la carte du cluster qu'il reçoit comme paramètre d'entrée pour en déduire les origines des pannes potentielles, comme par exemple la proximité physique, le partage de la même source d'alimentation, du même réseau, etc... En incorporant ces informations dans la carte hiérarchique du cluster, les règles de placement de CRUSH peuvent placer les réplicas d'un objet dans des domaines de défaillance et de panne différents, le tout en maintenant la distribution souhaitée par l'utilisateur.

d) Analyse de l'algorithme

L'algorithme est présenté dans la figure ci-dessous. Il prend comme entrée un entier x représentant par exemple l'identifiant d'un objet ou d'un groupe d'objet dont les réplicas seront placés sur les mêmes machines, et deux paramètres : n qui est le nombre de réplicas souhaités pour cet élément, et t qui est le type de machine souhaité. Il retourne une liste ordonnée d'OSD sur lesquelles les données vont être placées (après avoir été hachées).

Tout d'abord, la procédure $TAKE(a)$ sélectionne un élément de la carte hiérarchique du cluster (typiquement un bucket) et le place dans $\overrightarrow{i}$, qui servira d'élément de base pour la procédure suivante (ce sera le point de départ pour le parcours de la carte : tous les éléments fils de ce nœud seront explorés).

La procédure $SELECT(n, t)$ itère sur chacun élément i de $\overrightarrow{i}$. Chaque i peut donc être lui-même un bucket ou alors directement un OSD. La fonction va ensuite choisir choisir n éléments en parcourant de manière récursive le sous-arbre i, descendant dans les buckets intermédiaire puis en sélectionnant finalement un élément de manière pseudo-aléatoire grâce à la fonction $c(r, x)$ présentée précédemment (et qui est, rappelons le, définie pour chaque type de bucket), et ce jusqu'à trouver une machine de type t.



```
1: procedure TAKE(a)                          ▷ Put item a in working vector i
2:     i ← [a]
3: end procedure

4: procedure SELECT(n,t)                      ▷ Select n items of type t
5:     o ← ∅                                  ▷ Our output, initially empty
6:     for i ∈ i do                           ▷ Loop over input i
7:         f ← 0                              ▷ No failures yet
8:         for r ← 1,n do                     ▷ Loop over n replicas
9:             f_r ← 0                        ▷ No failures on this replica
10:            retry_descent ← false
11:            repeat
12:                b ← bucket(i)              ▷ Start descent at bucket i
13:                retry_bucket ← false
14:                repeat
15:                    if "first n" then
16:                        r' ← r + f
17:                    else
18:                        r' ← r + f_r·n
19:                    end if
20:                    o ← b.c(r',x)
21:                    if type(o) = t then
22:                        if o ∈ o or failed(o) or overload(o,x)
   then
23:                            f_r ← f_r + 1, f ← f + 1
24:                            if o ∈ o and f_r < 3 then
25:                                retry_bucket ← true      ▷ Retry
   collisions locally
26:                            else
27:                                retry_descent ← true     ▷
   Otherwise retry descent from i
28:                            end if
29:                        end if
30:                    else                                 ▷ Not type t
31:                        b ← bucket(o)          ▷ Continue descent
32:                        retry_bucket ← true
33:                    end if
34:                until ¬retry_bucket
35:            until ¬retry_descent
36:            o ← [o, o]                         ▷ Add o to output o
37:        end for
38:    end for
39:    i ← o                                      ▷ Copy output back into i
40: end procedure

41: procedure EMIT                            ▷ Append working vector i to result
42:     R ← [R, i]
43: end procedure
```

FIGURE 4 – Algorithme de placement CRUSH pour un objet x [5]

La procédure $SELECT(n,t)$ va parcourir un certain nombre de niveau de la carte hiérarchique du cluster afin de récupérer n éléments de type t dans le sous arbre i. Au cours de ce parcours, certains items peuvent être rejetés puis re-sélectionnés en utilisant un nouvel input r', et cela pour différentes raisons : en cas de collision (si cet item a déjà été sélectionné au paravent), si la machine n'est plus opérationnelle ou si elle est déjà surchargée. Les machines qui ne sont plus opérationnelles ou qui sont déjà surchargés sont marquées comme telles dans la carte du cluster, mais elles sont laissées dans la carte hiérarchique afin d'éviter un déplacement inutile des données. Dans le cas de machines plus opérationnelles ou surchargées, CRUSH redistribue les données (toujours de manière uniforme) en recommençant la récursivité de la procédure $SELECT(n,t)$ du début. En cas de collision, un nouvel input r' est choisi afin de parcourir les sous-arbres intérieurs où la probabilité de collision est plus faible (la taille de ces buckets est en général plus grande que n).

Les pools dites *répliquées* et les pools avec *erasure coding* ont des règles de placements différentes. En effet, les pools à réplication n'attachent pas d'importance à l'ordre des OSD, puisque chacun contient des réplicas ou des données primaires complètes. Lorsqu'en

cas de panne une donnée primaire est perdue (i.e. la donnée d'origine), un des réplicas doit prendre sa place et devenir primaire. Dans ce cas, CRUSH sélectionne les n premières cibles en utilisant la règle $r' = r + f$, où f est le nombre de tentatives de placement ratées pour la procédure en cours, c'est-à-dire que l'OSD défaillant sera enlevé de la liste et qu'un nouveau sera ajouté à la fin de la liste.

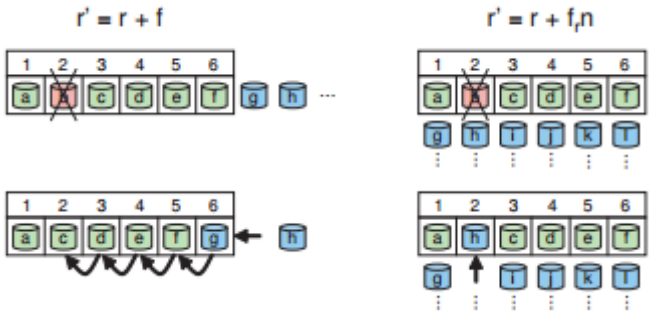


FIGURE 5 – Mécanisme de resélection d'un disque par CRUSH en cas de rejet, pour une procédeure $SELECT(6, disk)$ et pour les pools à réplication (en haut) et à erasure coding (en bas)

Dans le cas des pools avec *erasure coding*, l'ordre des machines est important, puisque chaque machine va stocker des bits différents de l'objet en question. Si l'un des OSD tombe, il doit donc être remplacé et son remplaçant doit prendre la même place dans la liste de retour de l'algorithme, afin que l'ordre des autres machines reste inchangé. Pour cela, CRUSH re-sélectionne une cible en utilisant une autre règle : $r' = r = frn$, avec frn qui est le nombre de tentatives ratées pour ce réplica, et assure ainsi la conservation de l'ordre des machines. Une fois que les n machines ont étés sélectionnées, CRUSH les transmets à Rados qui se chargera d'effectuer les opérations nécessaires.

Le développement de systèmes de stockages distribués imposent la prise en compte de nombreuses problématiques, comme la capacité d'évolution du système ou sa fiabilité. CRUSH répond à ces problématiques en rendant le système auto-gérant d'une part, et en analysant la réalité physique du réseau d'autre part pour assurer une sécurité maximale aux données.

4.2 Rados Block Device : algorithme incrémentiel d'instantanés

Cette partie est consacrée à l'utilisation de Rados Block Device dans Ceph. Les fonctionnalités au niveau de la gestion des données qu'il implémente, ainsi que son historique des modifications effectuées pour la répartition des données seront étudiées.

4.2.1 Contexte et définition des entités

Il est important de commencer par préciser certains termes qui seront utilisés par la suite. Une *image* est l'état d'un emplacement du cluster Ceph, c'est à dire l'état d'un ou

plusieurs OSDs par exemple. Cette image est en constante activité et évolue au cours du temps. A partir de ces images, il est possible de générer un *snapshot* d'une partie du système Ceph. Ces snapshots permettent d'introduire une notion d'historique de l'évolution du système.

Il n'est pas possible d'éditer et de modifier directement un snapshot une fois qu'il a été réalisé. Cependant, RBD permet de faire ce que l'on appelle des *copy-on-write* : ce sont des clones de snapshots qui sont éditables. Dans la suite de cette partie, le *child* désignera l'image obtenue par clonage d'un snapshot, qui sera lui appelé le *parent*.

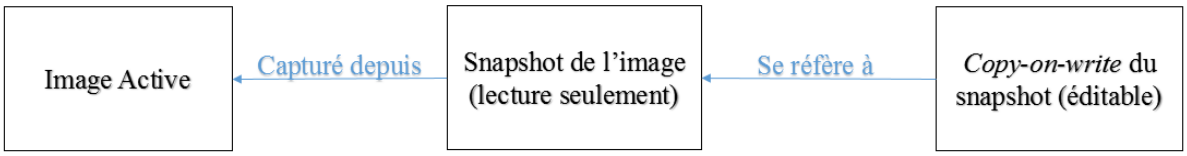


FIGURE 6 – Lien entre image active, snapshot et image clonée

Chaque child contient une référence au snapshot *parent* à partir duquel il a été créé, ce qui permet de faire le lien entre les deux entités. Ce lien autorise le child à ouvrir le snapshot père et à le lire. Un child possède une structure de données décrivant certains éléments clés, comme :

- l'identifiant du *snapshot* parent
- l'identifiant de *pool* du père
- l'identifiant de l'image à partir de laquelle le *snapshot* parent a été réalisé.
- l'identifiant de la *pool* où sera placé le *child*
- le nom du clone

Les identifiants des pools permettent de mettre en place une procédure : l'une des applications possible peut être de stocker dans une pool les snapshots et les images de type *read-only*, et dans une autre pool uniquement les clones. Cela permet de bien distinguer les pools où les images sont éditables et celles où elles ne le sont pas. Le clone obtenu va pouvoir agir de la même manière que les images classiques, à partir desquelles les snapshots ont été construits, c'est-à-dire qu'il est possible d'ouvrir le clone, de le lire, d'écrire dedans, de le cloner à nouveau, etc...

4.2.2 Fonctionnalités liées au snapshot

L'un des points sensibles du clonage est la protection du snapshot avant et pendant l'opération de clonage afin d'éviter une modification non-voulue. En effet, puisque l'image clonée contient une référence du snapshot père, il ne doit y avoir aucune altération de cette image.

Le processus se décompose en 4 grandes étapes : Tout d'abord, la création d'une image du disque, qui sera ensuite utilisée pour créer le snapshot. La deuxième étape consiste en la création du snapshot à partir de l'image précédemment créée. Un identifiant de pool lui est alors attribué. La troisième étape va consister à protéger le snapshot, en empêchant d'éventuelles modifications qui interviendrait au cours du clonage et ainsi prévenir de

possibles pertes de données. Enfin, la dernière étape est bien-sûr le clonage à proprement parler du snapshot.

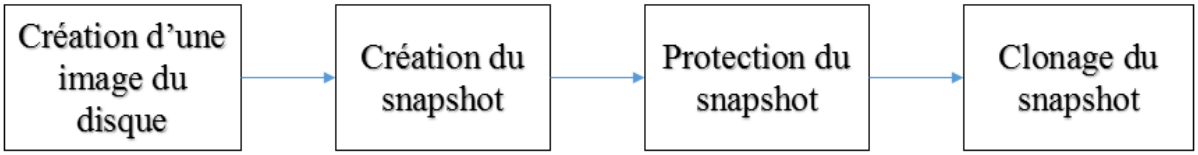


FIGURE 7 – Création d'une image clonée

L'un des points forts de Ceph est de permettre de faire des *snapshots incrémentaux*. Ils consistent en la production des images clonées en un temps d'exécution relativement faible. Dans le cas d'une défaillance, la manipulation d'images permet une récupération de l'ensemble des données. Les snapshots incrémentaux remplissent alors parfaitement ce rôle.

Le principe est de lancer deux clusters simultanément dans deux localisations différentes. Au lieu de faire de nouvelles copies des snapshots à chaque itération, un fichier est créé contenant les changements effectués entre les deux prises de snapshots. Ce fichier binaire est créé à la suite de l'utilisation de la commande *export*. Il contient le nom du snapshot initial, la terminaison du nom du second snapshot, la taille du second snapshot et les différences entre les deux snapshots.

Cet export peut soit être sauvegardé hors du système Ceph, soit être importé et greffé au niveau de l'image déjà existante sur un cluster Ceph distant. Son contenu va alors être écrit au niveau de l'image de sauvegarde. Un snapshot est créé avec le nom précédemment utilisé pour la terminaison, et dans le cas où un fichier avec le même nom existerait déjà, le processus serait interrompu et les données ne seraient pas écrasées. C'est cette incrémentation d'informations sur l'image existante qui justifie le terme de snapshots incrémentaux.

Il est possible de synthétiser ce fonctionnement par le schéma ci-dessous :

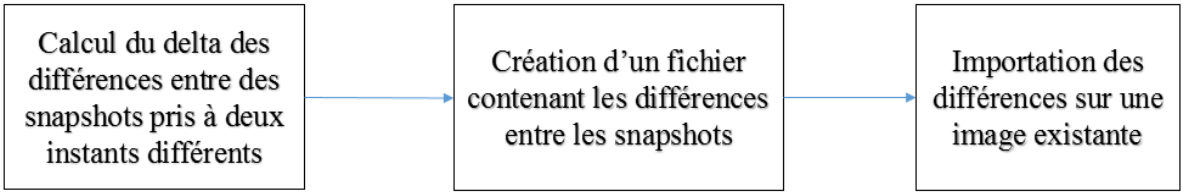


FIGURE 8 – Fonctionnement de l'incrémentation de snapshot

Les snapshots incrémentaux présentent néanmoins certaines limites. Tout d'abord, il n'y a aucune certitude que l'importation du fichier de différences se fasse sur une image dans le même état que celle à partir duquel ce fichier a été exporté. De plus, il n'y a pas de moyen de contrôler le contenu du fichier contenant les différences. Cela signifie qu'il faut utiliser le nom du fichier pour savoir à quel snapshot il fait référence.

Le rollback de snapshot permet de récupérer des fichiers qui auraient été supprimés lors de l'étape précédente. Cette opération consiste à écraser la version actuelle d'une

image par un snapshot, qui aura été pris en amont de l'image. Plus la taille de l'image est grande, plus le temps d'exécution de ce retour en arrière sera long.

Pour retourner à un état pré-existant, il est préférable de cloner un snapshot et d'utiliser cette image clonée, plutôt que d'effectuer un*rollback* d'une image avec un snapshot.

4.3 Détection d'une panne

La détection d'une panne est un élément primordial dans Ceph. En effet lorsqu'un noeud-serveur tombe, le système doit impérativement être avertis afin de remplacer les données perdues et ainsi conserver un nombre de réplicas cohérent. Le placement des données récupérées est bien-sûr déterminé par CRUSH.

4.3.1 Les battements de coeur

Comme présenté dans l'architecture, les daemon de stockage détectent les pannes d'autres OSD grâce aux battements de coeurs. A intervalle de temps régulier chaque site vérifie le battement de coeur de ses voisin. Par défaut, il est de 6 secondes (ce délai peut être paramétré). En réponse l'autre daemon lui renverra son battement de coeur, attestant ainsi de sa *bonne santé*.

Si un site ne reçoit pas de signe de vie d'un autre site pendant plus de 20 secondes alors il le considère en panne et le signale à un moniteur. Par défaut, un moniteur considère qu'une machine est en panne au bout de 3 signalements, mais ce seuil peut être paramétré par l'utilisateur.

On présente ci-dessous l'algorithme de détection d'une panne d'un OSD, permettant de mettre à jour la carte du cluster. L'algorithme PAXOS qui va ensuite être aborder permet d'obtenir un consensus, entre les moniteurs, sur la version à jour de la carte du cluster.

1 Initialisation OSD //Sur tous les OSDs

$IntervalleVerification \leftarrow 3$ //Intervalle en seconde auquel on vérifie l'état des autres sites
$SeuilPanne \leftarrow 20$ //Seuil en seconde à partir du quel on considère un autre OSD comme en panne
Démarrer un timer à IntervalleVerification pour chaque voisin de l'OSD

2 Initialisation Moniteur

$Tab_i[k] \leftarrow (OK, 0) \forall k \in 1, \ldots, N$ //Tab_i est un tableau de N couples (état, nbsignal), où état prend les valeur OK ou Panne, nbsignal est le nombre d'alerte reçu par le moniteur pour ce site
$NbSignalMax \leftarrow 3$ //nombre de signalement que doit recevoir un moniteur pour déclarer un site en panne

3 Expiration du Timer IntervalleVerification d'un OSD_j

Démarrer un timer à SeuilPanne pour OSD_j

4 Expiration du Timer SeuilPanne d'unOSD_j

Envoyer([Panne] j) au Moniteur

5 *Monitor* **: Réception d'un message [Panne] d'un OSD**

Recevoir ([Panne], i) de OSD_j
$Tab[i].nbsignal \leftarrow Tab[i].nbsignal + 1$
Si $Tab[i].nbsignal \leftarrow NbSignalMax$ **alors**
  $Tab[i].etat \leftarrow PANNE$
FinSi

Ce mécanisme de mise à jour dynamique de la carte du cluster est primordial au bon fonctionnement de Ceph. En effet, en plus d'assurer une constante sûreté des données, il permet à l'algorithme CRUSH de pouvoir toujours retrouver les données au sein du cluster en gardant constamment sa carte à jour.

4.4 PAXOS : algorithme de consensus

4.4.1 Introduction de l'algorithme

L'algorithme PAXOS a été proposé par Leslie Lamport en 1989. Il est aujourd'hui utilisé dans la plupart des systèmes informatiques du web et a permis le *cloud computing*. Ce protocole résout le problème de consensus, c'est-à-dire qu'il permet à un groupe de processus peu fiable de déterminer de manière déterministe et sûre un consensus, si certaines conditions peuvent être satisfaites, tout en assurant que le groupe reste cohérent si les conditions ne peuvent être satisfaites. C'est un algorithme d'exclusion mutuelle et de détection de failles.

4.4.2 Contexte d'utilisation dans Ceph

Lorsqu'un client Ceph veut accéder aux données, en lecture ou en écriture, il commence par contacter un moniteur Ceph pour récupérer la plus récente copie de la carte des clusters (*cluster map*).

Un cluster peut ne contenir qu'un seul moniteur Ceph, mais la fiabilité et la tolérance aux pannes est alors mise à défaut : si l'unique moniteur tombe (en cas de panne par exemple) l'accès aux données est compromis et les clients Ceph ne peuvent alors plus accéder aux données.

Ceph prend en charge un groupe de moniteurs pour éviter ce point d'échec. Mais dans un groupe de moniteurs, la latence et d'autres défauts peuvent causer le décrochage d'un ou plusieurs moniteurs dans l'état actuel du cluster. Pour cette raison, Ceph doit obtenir une concordance de l'état actuel du cluster entre plusieurs moniteurs.

Pour établir ce consensus, Ceph utilise toujours la majorité des moniteurs et l'algorithme PAXOS. Plus simplement, tout changement dans la structure de données d'un moniteur, qu'il s'agisse de la *monitor map*, *OSD map*, *Placement Group map*, *MetaData Server map* ou *CRUSH map*, est réalisé par PAXOS (étant donné qu'une *cluster map* est un composite de ces cartes).

4.4.3 Objectifs de l'algorithme

Le problème que résout l'algorithme PAXOS est celui de la recherche d'un consensus. Supposons un ensemble de processus qui propose chacun une valeur. L'algorithme doit s'assurer qu'une seule valeur va être choisie et qu'elle soit communiquée à l'ensemble des processus. Si aucune valeur n'est proposée alors aucune valeur ne doit être choisie.

PAXOS satisfait les propriétés de sûreté suivantes :

— La valeur choisie doit faire partie des valeurs proposées par les processus ;
— Une seule valeur doit être choisie ;
— Un processus n'apprend jamais qu'une valeur a été choisie à moins qu'elle ne l'ait été.

L'algorithme garantie l'intégrité si moins de la moitié des sites sont en pannes. Dans ce cas, la propriété de vivacité est satisfaite, l'élection d'un *leader* est faite et la progression est assurée. Néanmoins dans certain cas (détaillés plus loin), l'algorithme pourrait ne jamais terminer. Ces cas sont rares car les conditions qui pourraient empêcher PAXOS d'avancer sont difficiles à créer.

L'algorithme est divisé en trois parties distinctes : une première partie proposition dans laquelle les processus proposent leur valeur ; une seconde partie élection dans laquelle une proposition sera choisie et une dernière partie apprentissage. Pour simplifier la compréhension de l'algorithme, il a été choisi de séparer les sites en fonction de ces trois comportements (proposition, élection, apprentissage) car leurs actions sont bien séparées et spécifiques à un type. En pratique, il n'y a aucune distinction entre les sites *proposer*, *acceptor* et *learner*, chaque site remplit les trois rôles.

L'algorithme suppose un système asynchrone. Les messages échangés peuvent être longs, se perdre ou se dupliquer.

4.4.4 Algorithme

6 Site *Proposer* **: Initialisation sur** P_i

$numero_i \leftarrow 0$ //Numéro de proposition
$valeur_i \leftarrow 0$ //Valeur proposée
$nbPromiseReceived_i \leftarrow 0$ //Nombre de message *promise* reçu

7 Site *Proposer* **: Début sur** P_i

$numero_i \leftarrow Random()$
Envoyer ([*prepare*], n_i) à une majorité de sites Acceptors
$nbPromiseReceived_i \leftarrow 0$ //Remise à zéro car il peut y avoir plusieurs propositions faites

8 Site *Proposer* : Réception d'un message *promise* sur P_i

Recevoir ([*promise*], n, v) de A_j
$nbPromiseReceived_i \leftarrow nbPromiseReceived_i + 1$ //Une promesse supplémentaire a été reçue
Si $valeur_i < v$ alors $valeur_i \leftarrow v$
Sinon $valeur_i \leftarrow Random()$ //Si aucune valeur n'a encore été proposée, le site est libre de choisir la valeur qu'il souhaite
Envoyer ([*accept*], $numero_i$, $valeur_i$) à une majorité de sites Acceptors

9 Site *Acceptor* : Initialisation sur A_i

$numeroRetenu_i \leftarrow 0$ //Plus haut numéro de proposition reçu
$valeurRetenue_i \leftarrow 0$ //Valeur retenue

10 Site *Acceptor* : Réception d'un message *prepare* sur A_i

Recevoir ([*prepare*], n) de P_j
Si $n > numeroRetenu_i$ alors
  $numeroRetenu_i \leftarrow n$ //Le numéro n devient le nouveau plus haut numéro de proposition reçu
  Envoyer ([*promise*], $numeroRetenu_i$, $valeurRetenue_i$) à P_j
FinSi
  //Sinon rien n'est fait (sauf en cas d'optimisation où un message "non retenu" peut-être envoyé à P_j

11 Site *Acceptor* : Réception d'un message *accept* sur A_i

Recevoir ([*accept*], n, v) de P_j
Si $numeroRetenu_i <= n$
  $valeurRetenue_i \leftarrow v$
  Envoyer ([*accepted*], $valeurRetenue_i$) à P_j et à chaque L_k //La valeur retenue est envoyée à tous les Learners
FinSi

12 Site *Learner* : Réception d'un message *accepted* sur L_i

Recevoir ([*accepted*], v) de P_j

4.4.5 Fonctionnement de l'algorithme

En pratique, il n'y a aucune distinction entre les sites *Proposer*, *Acceptor* et *Learner*. Chaque site remplit les trois rôles et possède ces trois comportements. La distinction facilite cependant la compréhension de l'algorithme, c'est pourquoi il a été choisi de le présenter ainsi. [6]

Construction de l'algorithme

Le but de l'algorithme est d'élire une valeur unique par une majorité de sites. Les sites proposent une valeur à partir d'un ensemble de valeurs, une d'entre elle sera alors choisie par une majorité de sites. Les sites effectuant la proposition sont appelés *Proposer*, ceux chargés de l'élection *Acceptor*. Afin de s'assurer de l'élection d'une valeur dans tous les cas possibles, dont celui où il n'y a qu'une seule proposition de valeur faite, il est nécessaire de suivre cette exigence :

$P1$: Un site *Acceptor* doit accepter la première valeur qu'il reçoit.

Cependant une valeur est choisie que lorsqu'elle est acceptée par une majorité de sites, ce qui en raison de l'exigence $P1$ peut poser problème (dans le cas où chaque site reçoit une valeur différente en premier). Pour répondre à ces deux exigences, un *Acceptor* doit toujours respecter $P1$ mais doit également pouvoir accepter d'autres valeurs. Pour garder une trace de ces différentes propositions de valeurs, ces dernières sont intégrées à un couple (*numero de proposition*, *valeur*), appelé proposition. Pour garantir que toutes les propositions choisies aient la même valeur v, une nouvelle exigence est ajoutée :

$P2$: Si une proposition de valeur v est acceptée, alors l'ensemble des propositions faites avec un numéro plus élevé doit avoir la même valeur v.

Pour remplir cette exigence, il faut que tous les sites *Proposer* effectuant une proposition avec un numéro n plus élevé aient connaissance des valeurs v_i acceptées par les sites *Acceptor*. Leur proposition contiendra la valeur v_i de celle au numéro le plus élevé parmi toutes les propositions numérotées moins de n, afin de tendre vers l'acceptation par tous d'une même valeur v.

Du côté des sites *Acceptor*, afin de satisfaire $P1$ tout en satisfaisant les conditions des propositions, il est décidé qu'il ne peut accepter une proposition faite avec un numéro inférieur à celles qu'il a déjà reçu.

Détail des étapes

Une requête, attendant une réponse, est faite par un Client aux systèmes distribués. Cette requête peut être, par exemple, une demande d'écriture sur un système de fichiers distribués. La requête va être poussée par un site *Proposer*, dont la charge est de convaincre les *Acceptor* de s'accorder. Le rôle du *Proposer* est celui de proposer une valeur mais également de coordonner l'élection en cas de conflit. Celui de l'*Acceptor* est d'élire une valeur et de servir de mémoire résistant aux pannes.

Dans un premier temps, un site *Proposer* choisit un nombre afin de préparer une proposition de valeur. Il envoie ce nombre à un quorum de sites *Acceptor* dans un message *prepare*. Lorsqu'un site *Acceptor* reçoit ce message *prepare* de P_j, si le numéro qu'il contient est le plus haut qu'il n'ait jamais reçu alors il le garde en mémoire et répond au site *Proposer* avec un message *promise* contenant le numéro accepté et la valeur retenue, s'il en a une. Cette valeur correspond à une ancienne proposition acceptée faite par un autre site *Proposer*. Le message *promise* est la promesse faite par un site *Acceptor* à un site *Proposer* qu'il n'acceptera aucun message *prepare* contenant un nombre inférieur au

numéro reçu. Dans le cas où le site *Acceptor* a déjà reçu un précédant message *prepare* d'un autre site avec un numéro plus élevé, ce message est oublié.

Lorsqu'un site *Proposer* reçoit suffisamment de réponses *promise*, il devient *Leader* et envoie un à une quorum de sites *Acceptor* un message *accept* contenant le numéro de la proposition faite et la valeur proposée. Cette valeur égale le maximum des valeurs récupérées dans les messages *promise* reçus, ou une valeur quelconque choisie par le site *Proposer* si aucune valeurs n'étaient retournées dans les réponses *promise*.

La réception d'un message *accept* par un site *Acceptor* entraîne la mise à jour de la valeur retenue par celle contenue dans le message *accept*, si le numéro de proposition correspondant est plus élevé que celui mémorisé par le site *Acceptor*. Un message de confirmation *accepted* contenant la nouvelle valeur retenue est ensuite répondu au site *Proposer* et envoyé à l'ensemble des sites *Learner*.

Les sites *Learner* s'occupent, une fois l'acceptation d'une requête Client faite, de l'exécution de cette requête et du retour de la réponse au Client.

Déroulement de PAXOS au premier tour fructueux

La figure ci-dessous résume le déroulement de l'algorithme PAXOS en représentant les échanges de messages entre les sites *Proposer*, *Acceptor* et *Learner*. Sur cet exemple, le premier tour est couronné de succès.

Information relative à la figure : V_n égale le maximum des V_a, V_b, V_c.

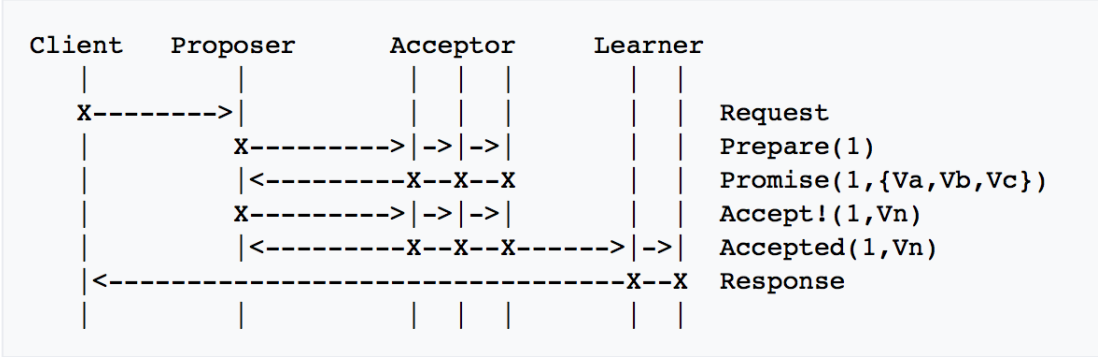


FIGURE 9 – Exemple des échanges de messages lors du déroulement de *PAXOS*

4.4.6 Robustesse : Gestion des crash

Si un *Leader* tombe en panne, un autre site peut être déclaré *Leader* et prendre le contrôle en faisant une proposition de valeur plus élevée. Dans le cas où le *Leader* original redeviendrait actif, les deux *Leaders* peuvent co-exister grâce aux règles d'accepter uniquement de propositions numérotées plus élevées et de n'engager que des valeurs précédemment acceptées.

De plus, PAXOS est tolérant aux partitions. Si deux partitions acceptent séparément une valeur, il n'y a pas d'état incohérent lors de leur fusion. La cohérence est assurée par la condition majoritaire : une partition ne peut arriver à un consensus sans avoir la majorité, et si une partition a la majorité et parvient à un consensus, alors la valeur doit également être acceptée par les sites des autres partitions.

Néanmoins, la terminaison de l'algorithme n'est pas toujours assurée. Dans le cas où deux *Leaders* renchérissent l'un à l'autre en proposant des numéros plus élevé, PAXOS pourrait ne jamais terminer. Même dans ce cas-ci où la propriété de vivacité n'est pas satisfaite, la propriété de sûreté l'est toujours,

Si l'étude théorique des algorithmes permet une compréhension en profondeur du système Ceph, la mise en place et l'utilisation d'un cluster est une étape nécessaire pour comprendre en pratique son fonctionnement. La dernière partie de ce rapport est ainsi consacrée à l'installation du système Ceph et à son utilisation.

5 Mise en place et installation d'un cluster

Nous aborderons dans cette partie le déploiement d'un cluster Ceph sur plusieurs machines physiques, comme nous avons pu l'expérimenter en salle informatique au bâtiment Pierre Guillaumat. L'entrée en matière sera ici plus subjective, avec une approche de type *compte rendu* et documentation d'une installation. Nous nous sommes basés presque uniquement sur la documentation officielle pour cette partie déploiement[9].

Avant tout chose, il est nécessaire de s'assurer de la compatibilité du système d'exploitation avec les exigences de Ceph, qui n'est disponible que sur des noyaux Linux (4.4), mais en général, toutes les nouvelles versions sont recommandées. Le système étant encore relativement récent et les mises à jour nombreuses, posséder un *OS* qui bénéficie encore d'un suivi et d'un soutien logiciel actif est préférable. Les kernels compatibles et versions de distributions Linux conseillées sont disponibles en ligne.

Six machines avec une partition de libre nous ont été proposées pour le déploiement d'un cluster. Au départ, inspirés par des systèmes sur Ubuntu, nous avons installé six Ubuntu Server ; cependant, la configuration semblait bien plus compliquée, et un flot d'erreurs que plusieurs autres groupes ont eues et qui n'ont pas été résolues nous ont dissuadés de continuer. Beaucoup de temps a été perdu lors de ces tentatives et il a donc été choisi de se tourner vers une distribution CentOS en version 7. Quasiment identique aux systèmes RedHat, sa simplicité d'utilisation et sa robustesse ont été appréciées.

De plus, chaque machine qui sera utilisée, soit en tant que client (accès au cluster) ou en tant que noeud (*Ceph Node*), doit être configurée un minimum. Cette étape, antérieure au déploiement et que l'on appelle familièrement *preflight* et plus communément *configuration step*, permet aux composants de pouvoir utiliser toutes les fonctionnalités d'un noyau Linux pour leur fonctionnement.

Elle consiste concrètement à préparer certains services utilisés par Ceph et à configurer des liaisons réseaux, un système d'utilisateur et de droits performant pour que le cluster puisse être déployé et utilisé sans préemption répétée. Les connexions SSH sont notamment très importantes puisqu'elles sont constamment utilisées par le programme d'installation *ceph-deploy* et permettent de tout *monitorer* depuis la même machine que l'on nommera couramment *admin-node*.

5.1 Configurer les dépôts et installer ceph-deploy

Comme il a été mentionné, *ceph-deploy* permet de lancer et configurer le déploiement d'un cluster via divers options et arguments. Programme très polyvalent basé sur des scripts Python, il sera nécessaire pour la plupart des étapes. Il faut donc ajouter à notre distribution le dépôt où il se trouve, avec CentOS, une simple commande *subscription-manager repos* puis un *yum install* permettent d'avoir le programme prêt.

5.2 Activer le protocole NTP

Acronyme de *Network Time Protocol*, *ntp* est un protocole permettant de distribuer l'heure sur un réseau informatique. Il possède deux côtés : un client *ntpdate* et un serveur *ntpd* sachant qu'une machine peut avoir le rôle des deux. Il permet à une machine de

synchroniser son horloge sur celle d'un serveur de plus haute précision, souvent centralisés par régions géographiques. Ces serveurs sont organisés en couches, des strates numérotées dans l'ordre croissant, dont la précision des horloges décroît progressivement (à partir des strates 0 et 1) afin de pouvoir diffuser l'heure à de nombreux clients sans surcharger les serveurs critiques qui sont les plus précis. L'objectif de ce fonctionnement pyramidal est de fournir une heure de qualité sans compromettre sa disponibilité.

On comprend bien l'importance de ce protocole pour l'exécution de Ceph pour que les différentes entités (notamment les moniteurs) du cluster possèdent la même horloge afin synchroniser leurs envois de messages et leurs états.

5.3 Communications SSH et utilisateurs

Remplaçant du programme Telnet, le Secure Shell (SSH) est un programme et protocole de communication TCP à chiffrement asymétrique, entre deux machines. Il permet d'obtenir à distance un *shell* ou une ligne de commande, ce qui permet d'utiliser des machines informatiques de manière décentralisée avec le réseau. Dans le cadre de Ceph, SSH sera utile pour administrer depuis un seul ordinateur (*admin-node*) l'ensemble des machines qui composent le cluster. Il est de plus utilisé par le programme *ceph-deploy* qui ne nécessite d'être lancé à distance depuis qu'une seule machine, afin d'exécuter du code d'installation sur chaque noeud.

Par ailleurs, cet accès SSH doit être rendu possible sans mot de passe pour ne pas être sans arrêt réquisitionné, pour cela on génère une paire de clés (une privée et une publique) au chiffrement RSA dont la publique est distribuée sur les noeuds. Lors d'une tentative de connexion ultérieure à la machine, notre clé privée suffira à l'authentification et aucune action de l'utilisateur ne sera nécessaire pour l'ouverture d'une session. On configure aussi le fichier *config* pour pouvoir définir avec quel utilisateur se connecter par défaut sur chaque machine, ainsi que le fichier */etc/hosts* hosts qui donne un moyen d'assurer la résolution de noms : identifier les hôtes avec des noms au lieu d'adresses IP.

Finalement, les accès aux noeuds doivent pouvoir se faire avec un accès super-utilisateur (*sudo*) afin de pouvoir y faire la configuration nécessaire. Pour cela, un nouvel utilisateur est créé sur chaque machine, auquel les accès *sudo* sans mot de passe sont donnés en écrivant dans un fichier de configuration à son nom la ligne suivante :

```
{username} ALL = (root) NOPASSWD:ALL > /etc/sudoers.d/{username}
```

Cela permet à *ceph-deploy* de pouvoir s'exécuter en autonomie sans intervention, à condition qu'il n'y ait pas d'erreurs.

La combinaison ces paramétrages de droits permet de travailler plus efficacement et de donner le contrôle total à l'installation d'un cluster Ceph. Ils ne doivent pas être négligés car ils représentent un réel gain de temps et préviennent bon nombre d'erreurs de configuration.

5.4 Configuration réseau

Toutes les communications au sein d'un cluster Ceph se réalisent grâce au réseau. Il est donc primordial de posséder un paramétrage satisfaisant de toutes ses composantes

pour que les performances soient exploitées au maximum.

Tout d'abord, le réseau doit être activé au démarrage des machines automatiquement afin de gagner du temps et d'éviter tout oubli.

```
cd /etc/sysconfig/network-scripts/
sed -i -e 's@^ONBOOT="no@ONBOOT="yes@' ifcfg-enps025
```

Ensuite, différents ports doivent être ouverts pour pouvoir communiquer sans restriction entre les noeuds. Les moniteurs utilisent notamment le port 6789 par défaut, tandis que les OSD utilisent la plage 6800 :7300. En fonction du système Linux utilisé, certaines configurations par défaut ont un pare-feu très strict qui peut compromettre les communications entre clients du réseau et les noeuds de Ceph, il est dans ce cas nécessaire de le paramétrer.

Remarquons finalement que lorsque l'on déploie un très gros cluster, il est recommandé d'utiliser un deuxième réseau. En effet par défaut Ceph n'en considère qu'un seul, le réseau frontal public (*front-side public network*). Lorsque les charges de travail du système réparti sont très importantes, les performances peuvent être significativement améliorées grâce à l'ajout d'un réseau *arrière* (*back-end cluster network*) qui est utilisé uniquement par les OSD pour répliquer des données et transmettre leurs battements de coeur (*heartbeats*).

Cela soulage le réseau public de communication des clients, ainsi que les moniteurs et éventuels serveurs de méta-données (*MDS*) comme en témoigne le schéma suivant.

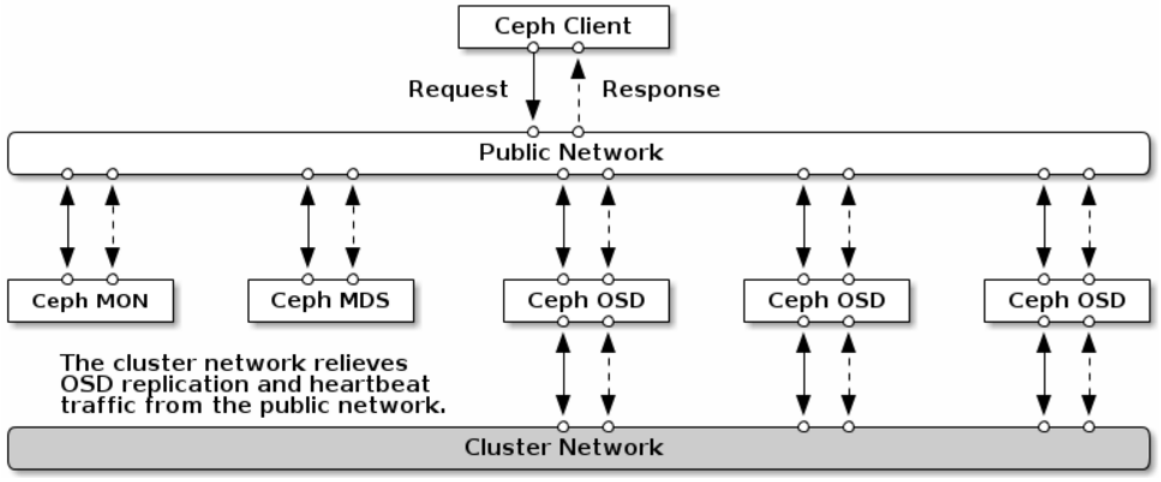


FIGURE 10 – Ajout d'un réseau secondaire dans le cluster

Cette séparation ajoute finalement une amélioration de la sécurité en cas de blocage volontaire du réseau par des attaques de déni de service (*DoS*), puique les noeuds de stockage ne seront pas perturbés pour communiquer entre eux.

Cette configuration n'a cependant pas nécessité d'être abordée dans notre étude.

5.5 Déploiement

Dans notre configuration, nous disposons de six machines que nous avons nommées $node_1, node_2 ... node_6$. Le noeud depuis lequel s'effectue le déploiement est le numéro 6 : $noeud_6 = admin\text{-}node$.

On commence par définir le premier moniteur du cluster :

```
ceph-deploy new [initial-monitor-node(s)]
```

Le $node_5$ est le premier moniteur défini pour le cluster, mais nous en avons ensuite créé deux autres afin de s'approcher du paramétrage requis en *production* par la documentation, ce pour permettre à un moniteur d'être arrêté sans échec du système.

On crée maintenant les OSD : afin de simplifier l'installation il a été décidé d'utiliser des répertoires au lieu d'un disque de stockage entier :

```
ceph-deploy osd prepare [ceph-node]:[dir-path]
```

Les noeuds 5,4,3,2 ont été utilisé pour le stockage donc déclarés en tant qu'OSD avec des chemins d'accès uniformes : $dir\text{-}path = node_i : /var/local/osd_i$.

Le schéma suivant explique plus en détail l'organisation des noeuds. Même si cela n'est pas forcément recommandé en situation de production, il est très bien possible de regrouper sous une même machines plusieurs rôles (moniteur, *storage device*...) :

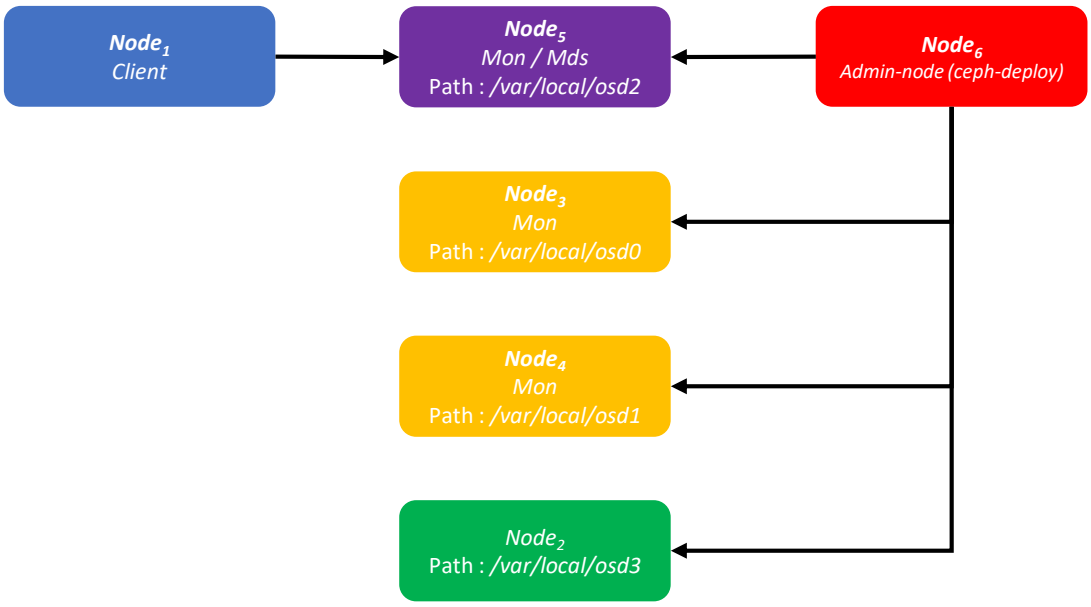


FIGURE 11 – Architecture du cluster de l'étude

$Node_1$ a été défini comme le client : l'ordinateur depuis lequel les utilisateurs accèdent théoriquement au cluster (un utilisateur *lambda* n'a normalement aucune raison d'opérer depuis une des machines qui composent les noeuds de stockage de Ceph).

Il faut ensuite les activer :

```
ceph-deploy osd activate [ceph-node]:[dir-path]
```

Il faut finalement copier les fichiers de configuration qui ont été créés sur les noeuds, ainsi que copier la clé d'administrateur du cluster qui permet d'utiliser la ligne de commande de Ceph sans spécifier les adresses des moniteurs avant d'exécuter une quelconque action :

```
ceph-deploy admin [admin-node][ceph-node...]
```

Le déploiement est alors terminé et le cluster est automatiquement démarré, on peut vérifier son état avec la commande :

```
ceph health
```

5.6 Utilisation et surveillance

Une fois le cluster déployé, il faut apprendre à s'en servir et à le superviser. Tout comme *ceph-deploy*, *ceph* est une commande dont le code est écrit en Python, et qui permet d'exécuter une multitude d'actions. Couplée à la gestion des services Linux lancés par le noyau avec systemctl on peut tout administrer.

Il est possible d'afficher des arbres du cluster, son état de santé, l'état de chacune des entités qui la composent, suivre en temps réel les communications ou encore ajouter des OSD, des moniteurs...

Pour finir, le stockage de données distribué étant l'objectif principal de Ceph, c'est donc tout naturellement dessus que cette étude sera conclue. Les possibilités sont nombreuses et nous n'avons pu réellement exploiter ses capacités bien que nous en ayons eu une approche très positive.

Le programme d'interaction avec un cluster de stockage Ceph et éponyme : *rados*. Il prend beaucoup d'arguments et permet de réellement tirer partie des possibilitées offertes par le système de stockage. On peut stocker des données, en retrouver, prendre un snapshot de l'état des *pools*... Par exemple, la commande principale pour sauvegarder un objet dans Rados est :

```
rados put {object-name} {file-path} --pool=data
```

On peut le retrouver avec un mapping de son *pool* et de son nom :

```
ceph osd map {pool-name} {object-name}
```

Il existe beaucoup d'associations avec les commandes UNIX de manipulation de fichiers :

```
rados -p data ls       |       rados rm test-object-1 --pool=data
```

Globalement, *ceph -w, ceph -s, ceph health, ceph tree* etc. sont les commandes les plus utilisées pour surveiller un cluster, et elles permettront d'appuyer de manière claire la démonstration effectuée le 28 Juin.

6 Conclusion

Ceph est un système de stockage distribué particulièrement adapté aux nouvelles façons de travailler : décentralisées, partagées, instantanées, et riches en données. Créé pour fonctionner sur du matériel standard, géré par des couches logicielles open source, sa configuration et son déploiement sont accessibles pour la plupart des utilisateurs de Linux.

L'intelligence du stockage apportée par l'algorithmique répartie du système ainsi que par son architecture novatrice, permet à Ceph de s'auto-gérer et d'affranchir ses utilisateurs des problématiques de gestion de la durabilité et de la disponibilité des données. Sa forte résilience réduit les coûts liés aux opérations de maintenance des supports de stockage ainsi que les besoins d'intervention extérieure.

Par son architecture et son algorithmique distribuée, Ceph offre un outil concret d'étude des concepts introduits dans l'UV *Algorithmes et Systèmes Répartis* enseignée par Bertrand Ducourthial à l'Université de Technologie de Compiègne.

Projet de groupe en totale autonomie, cette étude s'est basée d'une part sur de nombreuses recherches documentaires, et d'autre part sur des travaux pratiques conclus par le déploiement d'un cluster dans un environnement Linux ; elle apporte un fort enrichissement dans le cursus d'ingénieur en informatique.

Table des figures

Références

[1] Tamara Scott. Big data storage wars : Ceph vs gluster, 2017.

[2] Torben Kling Petersen. Inside the lustre file system, 2015.

[3] Joab Jackson. Converging storage : Cephfs now production ready, 2016.

[4] Murat. Ceph : A scalable, high-performance distributed file system, 2011.

[5] Ethan L. Miller Carlos Maltzahn Sage A. Weil, Scott A. Brandt. Crush : Controlled, scalable, decentralized placement of replicated data.

[6] Leslie Lamport. Paxos made simple. 2001.

[7] Harry Brundage. Neat algorithms - paxos, 2014.

[8] Angus MacDonald. Paxos by example, 2012.

[9] Ceph documentation.