Sécurité et disponibilité des données stockées dans les nuages

Théodore Jean Richard Relaza

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *12/02/2016* **par :**

Théodore Jean Richard RELAZA

Sécurité et Disponibilité des Données Stockées dans les Nuages

---

JURY

Christine MORIN	Directrice de recherches, INRIA	Rapporteur
Yves DENNEULIN	Professeur, ENSIMAG	Rapporteur
Pierre SENS	Professeur, Univ. Paris 6	Examinateur
Abdelaziz M'ZOUGHI	Professeur, Univ. Toulouse 3	Examinateur
Jacques JORDA	Maître de conférences HDR, Univ. Toulouse 3	Examinateur
Aurélien ORTIZ	Ingénieur, APX	Examinateur

École doctorale et spécialité :

MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture

Unité de Recherche :

Institut de Recherche en Informatique de Toulouse (UMR 5505)

Rapporteurs :

Christine MORIN et *Yves DENNEULIN*

Directeurs de Thèse (co-encadrants) :

Abdelaziz M'ZOUGHI et *Jacques JORDA*

Remerciements

Tout d'abord, je tiens à remercier mes directeurs de thèse Abdelaziz M'zoughi et Jacques Jorda pour la confiance qu'ils m'ont accordée en acceptant d'encadrer ce travail de thèse et pour leurs précieux conseils. J'ai été sensible à leurs qualités humaines d'écoute et de compréhension tout au long de ce travail de thèse.

Je remercie Mme Christine Morin et M. Yves Denneulin qui m'ont fait l'honneur d'être rapporteurs de ma thèse. Je les remercie également de l'intérêt qu'ils ont porté à ce travail à travers un regard critique, juste, avisé et constructif.

Je tiens aussi à remercier MM. Pierre Sens et Aurélien Ortiz d'avoir accepté d'examiner ce travail.

Je suis très reconnaissant à tous les membres de ma famille, tout particulièrement mes parents, mes sœurs et mon petit frère pour leur confiance inconditionnelle, pour leur soutien et pour leurs encouragements, depuis toujours source de réconfort. Tout ce que je pourrai dire ne suffira jamais à exprimer tout ce que je leur dois et combien ils ont été importants pour faire de moi ce que je suis aujourd'hui. Cette thèse est aussi la vôtre.

Je souhaite également remercier tous mes amis pour leur soutien et leur disponibilité à n'importe quelle heure du jour et de la nuit.

Enfin, mes remerciements vont également à tous ceux qui ont contribué de près ou de loin, à l'aboutissement de ce travail.

Résumé

Avec le développement de l'Internet, l'informatique s'est basée essentiellement sur les communications entre serveurs, postes utilisateurs, réseaux et data centers. Au début des années 2000, les deux tendances à savoir *la mise à disposition d'applications* et *la virtualisation de l'infrastructure* ont vu le jour. La convergence de ces deux tendances a donné naissance à un concept fédérateur qu'est le *Cloud Computing* (informatique en nuage). Le stockage des données apparaît alors comme un élément central de la problématique liée à la mise dans le nuage des processus et des ressources. Qu'il s'agisse d'une simple externalisation du stockage à des fins de sauvegarde, de l'utilisation de services logiciels hébergés ou de la virtualisation chez un fournisseur tiers de l'infrastructure informatique de l'entreprise, la sécurité des données est cruciale. Cette sécurité se décline selon trois axes : la disponibilité, l'intégrité et la confidentialité des données.

Le contexte de nos travaux concerne la virtualisation du stockage dédiée à l'informatique en nuage (Cloud Computing). Ces travaux se font dans le cadre du projet SVC (Secured Virtual Cloud) financé par le Fond National pour la Société Numérique "Investissement d'avenir". Ils ont conduit au développement d'un intergiciel de virtualisation du stockage, nommé CloViS (Cloud Virtualized Storage), qui entre dans une phase de valorisation portée par la SATT Toulouse-Tech-Transfer. CloViS est un intergiciel de gestion de données développé au sein du laboratoire IRIT, qui permet la virtualisation de ressources de stockage hétérogènes et distribuées, accessibles d'une manière uniforme et transparente. CloViS possède la particularité de mettre en adéquation les besoins des utilisateurs et les disponibilités du système par le biais de qualités de service définies sur des volumes virtuels.

Notre contribution à ce domaine concerne les techniques de distribution des données afin d'améliorer leur disponibilité et la fiabilité des opérations d'entrées/sorties dans CloViS. En effet, face à l'explosion du volume des données, l'utilisation de la réplication ne peut constituer une solution pérenne. L'utilisation de codes correcteurs ou de schémas de seuil apparaît alors comme une alternative valable pour maîtriser les volumes de stockage. Néanmoins aucun protocole de maintien de la cohérence

des données n'est, à ce jour, adapté à ces nouvelles méthodes de distribution. Nous proposons pour cela des protocoles de cohérence des données adaptés à ces différentes techniques de distribution des données. Nous analysons ensuite ces protocoles pour mettre en exergue leurs avantages et inconvénients respectifs. En effet, le choix d'une technique de distribution de données et d'un protocole de cohérence des données associé se base sur des critères de performance notamment la disponibilité en écriture et lecture, l'utilisation des ressources système (comme l'espace de stockage utilisé) ou le nombre moyen de messages échangés durant les opérations de lecture et écriture.

Mots-clés : Disponibilités des données, cohérence, gestion des replicas, code d'erreur, distribution des données, quorum.

Abstract

With the development of Internet, Information Technology was essentially based on communications between servers, user stations, networks and data centers. Both trends *"making application available"* and *"infrastructure virtualization"* have emerged in the early 2000s. The convergence of these two trends has resulted in a federator concept, which is the *Cloud Computing*. Data storage appears as a central component of the problematic associated with the move of processes and resources in the cloud. Whether it is a simple storage externalization for backup purposes, use of hosted software services or virtualization in a third-party provider of the company computing infrastructure, data security is crucial. This security declines according to three axes : data availability, integrity and confidentiality.

The context of our work concerns the storage virtualization dedicated to Cloud Computing. This work is carried out under the aegis of SVC (Secured Virtual Cloud) project, financed by the National Found for Digital Society *"Investment for the future"*. This led to the development of a storage virtualization middleware, named CloViS (Cloud Virtualized Storage), which is entering a valorization phase driven by SATT Toulouse-Tech-Transfer. CloViS is a data management middleware developped within the IRIT laboratory. It allows virtualizing of distributed and heterogeneous storage resources, with uniform and seamless access. CloViS aligns user needs and system availabilities through qualities of service defined on virtual volumes.

Our contribution in this field concerns data distribution techniques to improve their availability and the reliability of I/O operations in CloViS. Indeed, faced with the explosion in the amount of data, the use of replication can not be a permanent solution. The use of *"Erasure Resilient Code"* or *"Threshold Schemes"* appears as a valid alternative to control storage volumes. However, no data consistency protocol is, to date, adapted to these new data distribution methods. For this reason, we propose to adapt these different data distribution techniques. We then analyse these new protocols, highlighting their respective advantages and disadvantages. Indeed, the choice of a data distribution technique and the associated data consistency protocol is based on performance criteria, especially the availability and the number of messages exchanged during the read and write operations or the use of system resources (such as storage space used).

Keywords : Data availability, consistency, replica management, error codes, data distribution, quorum.

Table des matières

Table des figures

Introduction

Présentation du sujet

De nos jours, le domaine des technologies de l'information et de la communication est devenu l'un des piliers de la société moderne. Le développement du concept de *Cloud Computing* (informatique en nuage) a constitué une avancée majeure vers la démocratisation de ce domaine. Le stockage des données devient alors l'un des éléments cruciaux de l'infrastructure, que ce soit au niveau de la performance ou de la sécurité. En termes de sécurité des données, plusieurs critères peuvent être énumérés, tels que la disponibilité, l'intégrité ou encore la confidentialité. Ces travaux de thèse s'inscrivent dans le cadre du projet SVC (***S**ecured **V**irtual **C**loud*) financé par le Fond National pour la Société Numérique *"Investissement d'avenir"* et qui entre dans une phase de valorisation portée par la SATT Toulouse-Tech-Transfer. Nous travaillons sur CloViS qui fait partie du sous-projet *"stockage sécurisé"* du projet SVC. CloViS est un intergiciel de gestion des données développé au sein du laboratoire IRIT (***I**nstitut de **R**echerche en **I**nformatique de **T**oulouse*), qui permet la virtualisation de ressources de stockage hétérogènes et distribuées, accessibles d'une manière sécurisée, uniforme et transparente.

Parmi les critères de sécurité des données énoncés, nous nous sommes intéressés à la problématique de la disponibilité des données. Une solution couramment utilisée pour améliorer à la fois la disponibilité et les performances de lecture des données dans un environnement distribué est la réplication totale. Cependant cette technique présente le double inconvénient d'être pénalisante pour les écritures et de gréver de manière trop importante les capacités de stockage. Elle n'est donc pas viable à long terme face à l'explosion du volume des données. Ainsi, de nouvelles méthodes (dans le contexte du stockage) telles que les codes correcteurs ou les schémas à seuil permettent de trouver un compromis entre les surcoûts générés, le temps d'accès aux données et leur disponibilité. Comme dans le cas de la réplication totale, l'utilisation de ces techniques nécessite un protocole de gestion de la cohérence pour maintenir la cohérence de différentes répliques d'une donnée qui sont éparpillées dans plusieurs ressources de stockage. Par exemple, les protocoles de cohérence dits *à quorums* (appelés souvent *protocole ou système à quorums*) permettent de garantir la cohérence des données dans un environnement distribué lorsque la réplication est utilisée comme technique de distribution des données.

Un protocole à quorums forme une structure logique des répliques en plusieurs sous-ensembles appelés quorums. Dans ce contexte, un quorum est l'ensemble minimum de répliques garantissant la cohérence des données lors d'une opération bien définie (par exemple une opération d'écriture ou de lecture). Or, à ce jour, les protocoles de gestion de la cohérence existant ne sont adaptés ni aux codes correcteurs ni aux schémas à seuil. Nos travaux, présentés dans ce manuscrit, adressent cette problématique de manière théorique et pratique.

Contributions

Notre contribution concerne les techniques de distribution des données afin d'améliorer la disponibilité des données et la fiabilité des opérations d'entrées/sorties dans CloViS. D'abord, après avoir étudié les différentes solutions proposées dans la littérature sur les techniques de distribution des données, nous avons proposé l'utilisation de mécanismes tels que les codes correcteurs ou les schémas à seuil dans CloViS, qui permettent d'améliorer la disponibilité des données et de maîtriser les volumes de stockage.

Ensuite, nous avons proposé une adaptation des quatre protocoles de gestion des répliques (à savoir, *le quorum majoritaire*, *le quorum en grille*, *le quorum en arbre* et *le quorum en trapèze*) au contexte de ces techniques de distribution des données choisies pour améliorer la fiabilité des opérations d'entrées/sorties dans CloViS. En effet, aucun protocole de gestion de répliques publié à ce jour n'est adapté à ces techniques de distribution des données. Par exemple, lors d'une opération de lecture, ces protocoles de gestion de répliques ne garantissent qu'une seule réplique à jour alors qu'avec les codes correcteurs ou les schémas à seuil, plusieurs répliques à jour peuvent être requises.

Enfin, nous avons effectué des évaluations numériques de ces nouveaux protocoles de gestion de répliques pour mettre en exergue leurs avantages et leurs inconvénients respectifs. Ces évaluations numériques se basent sur des critères de performance spécifiques : la disponibilité en écriture et en lecture et l'utilisation des ressources système. Par ailleurs, une infrastructure est actuellement en cours d'élaboration pour la mise en œuvre de ces solutions et l'évaluation de leurs performances en environnement réel. Dans cette optique, nous avons commencé par améliorer ClovisFS (*système gestion de fichiers de CloViS*). Ce nouveau système de gestion de fichiers est actuellement opérationnel. Nous avons également implémenté l'algorithme CRUSH (***C**ontrolled **R**eplication **U**nder **S**calable **H**ashing*) [110] permettant l'optimisation du placement des données dans l'ensemble des ressources de stockage physique, afin de mener des comparaisons avec les systèmes concurrents tels que Ceph.

Organisation du document

Ce manuscrit est organisé en deux grandes parties : *"la virtualisation CloViS"* et *"distribution et cohérence des données"*. La première partie est constituée de deux cha-

pitres dont le premier présente le contexte général du *Cloud Computing* tandis que le deuxième introduit le projet CloViS et détaille l'architecture de l'intergiciel CloViS et le rôle de chacun de ses composants.

La deuxième partie est composée de quatre chapitres principaux. Nous exposons l'état de l'art des techniques usuelles de distributions des données dans le premier chapitre et celui de maintien de la cohérence des données dans le deuxième chapitre. Nous détaillons ensuite dans le troisième chapitre les propositions de gestion de la cohérence des données dans CloViS. Nous effectuons également dans le même chapitre des évaluations numériques de chaque protocole de cohérence des données proposé. Enfin, nous concluons et nous donnons des perspectives sur le projet CloViS dans le dernier chapitre.

Première partie

La virtualisation CloViS

Chapitre 1

Le cloud computing

Sommaire

1.1 Définition

L'informatique en nuage (Cloud Computing) fait référence à l'externalisation des processus informatiques d'une entreprise chez un fournisseur prenant en charge

les ressources matérielles et logicielles nécessaires au fonctionnement du système informatique de son client. Il existe autant de définitions que de points de vue, et les spécialistes donneront à l'informatique en nuage des définitions diverses selon qu'ils viennent de la sécurité, du web, de l'infrastructure, etc. Le National Institute of Standards and Technology (NIST) propose une définition qui se veut exhaustive :
"*Le Cloud Computing est un modèle qui permet d'accéder rapidement à un pool de ressources informatiques mutualisées, à la demande (serveurs, stockage, applications, bande passante, etc.), sans forte interaction avec le fournisseur de service [...]*" [72].
Selon cet organisme, l'informatique en nuage s'appuie sur cinq caractéristiques essentielles :

- **Un accès libre à la demande :** l'utilisateur consomme des services (capacité de stockage, de calcul, plateforme de développement, etc.) dans le cloud selon ses besoins et ceci de façon automatique c'est à dire sans nécessiter d'interaction humaine avec le fournisseur de services ;
- **Le service doit être accessible via un réseau :** tous les services proposés aux utilisateurs doivent être disponibles sur le réseau et accessibles via des mécanismes standards favorisant l'utilisation des plateformes hétérogènes (par exemple téléphones mobiles, tablettes, ordinateurs portable, poste de travail, etc.) ;
- **Mutualisation des ressources :** les ressources informatiques du fournisseur sont agrégées et mises à disposition des consommateurs sur un modèle multi-locataires, avec une attribution dynamique des ressources physiques et virtuelles en fonction de la demande. Les consommateurs n'ont généralement aucune connaissance ni aucun contrôle de l'endroit exact où sont stockés les ressources fournies. Toutefois, ils peuvent parfois imposer l'emplacement à un niveau plus haut d'abstraction (par exemple le pays, le Datacenter, etc.) ;
- **Redimensionnement rapide :** les consommateurs peuvent rapidement ajouter ou enlever des ressources informatiques en fonction de leurs besoins. Pour le consommateur, les ressources informatiques mises à disposition par le fournisseur semblent illimitées et peuvent être consommées en quantité à tout moment ;
- **Le service doit être mesurable :** les systèmes contrôlent et optimisent de façon automatique l'utilisation des ressources par rapport à une moyenne estimée du service consommé.

La figure 1.1 montre une vue globale de l'informatique en nuage. Elle comporte deux grandes parties : La partie interne (le fournisseur) et la partie externe (les consommateurs). L'accès se fait par le biais d'un pare-feu chargé de filtrer les accès pour prévenir les attaques. Une fois passé ce pare-feu, l'utilisateur atteint un réseau virtuel interne ; il a alors accès aux services gérés pour lui par son fournisseur. Ces services peuvent prendre la forme de services logiciels consommés à la demande (par exemple des services de messagerie ou de base de données) ou, de manière plus complète, des machines virtuelles remplaçant les serveurs du client. Dans ce dernier cas, ces machines virtuelles fonctionnent sur des serveurs physiques hébergés par le fournisseur, parfois conjointement avec des machines virtuelles appartenant à d'autres clients. Ces services manipulent des données (fichiers conventionnels ou fichiers disques de machines virtuelles). Ces données sont stockées sur des nœuds de stockage (serveurs,

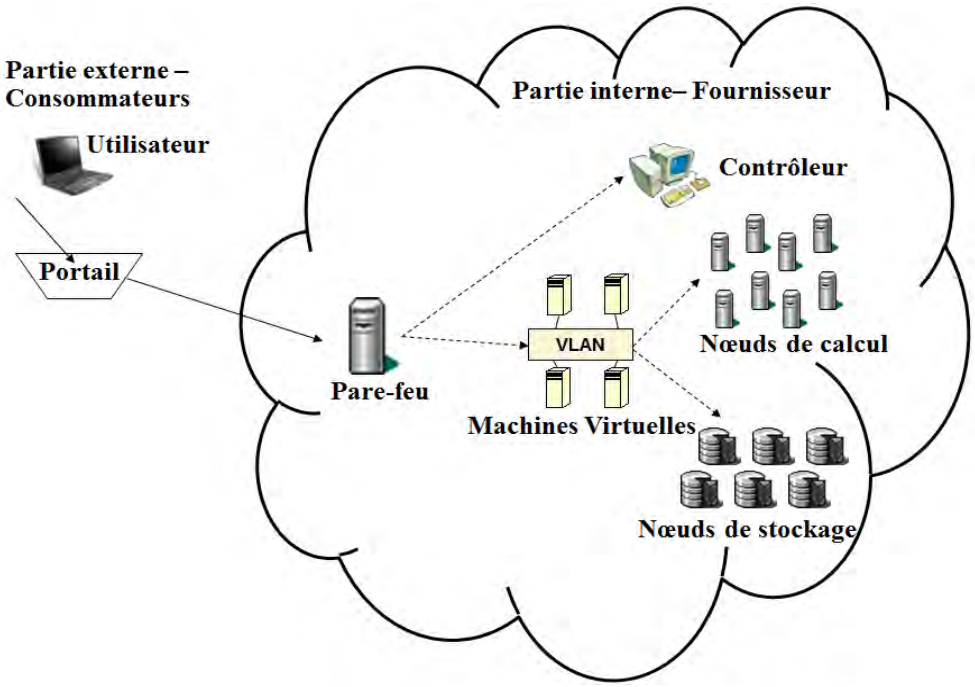


FIGURE 1.1 – Une vue globale de l'informatique en nuage

baies, SAN, NAS) et doivent être sécurisées pour garantir la confidentialité, y compris vis-à-vis des administrateurs du fournisseur.

Dans le concept d'informatique dans le nuage, la puissance de calcul et de stockage de l'information sont proposés comme des services par des entreprises spécialisées et facturés d'après l'utilisation réelle. De ce fait, les clients de ces entreprises n'ont en théorie plus besoin de serveurs dédiés.

1.2 Les modèles de déploiement du Cloud Computing

Le Cloud Computing repose sur des ressources physiques qui peuvent être situées chez le client ou chez un prestataire, être partagées ou non. Ainsi, les utilisateurs du cloud peuvent choisir entre se construire leurs propres infrastructures ou en louer une chez un fournisseur de service spécialisé, bénéficier de services plus ou moins étendus proposés par ces fournisseurs ou encore combiner ces deux options. Ainsi, selon les approches des entreprises se distinguent trois formes ou typologies de Cloud Computing :

1.2.1 Les clouds privés

Les clouds privés mettent l'ensemble des ressources à la disposition exclusive d'une seule entreprise. Les services de ces clouds sont dédiés aux besoins propres de l'entre-

prise cliente. Ce type de cloud offre aux entreprises des services qui leur permettront d'optimiser et de mieux maîtriser l'utilisation de leur ressources informatiques. Les problèmes d'intégration et les exigences de sécurité pour les données et applications critiques sont parmi les raisons qui poussent les utilisateurs à adopter les clouds privés. Avec les clouds privés, les utilisateurs ont plus de contrôle de leurs applications et données que les autres typologies de cloud.

Ils sont souvent gérés en interne par une entreprise pour ses besoins. Dans ce cas, les ressources informatiques sont géographiquement situées dans le périmètre de l'entreprise. Le service informatique est alors vu comme un fournisseur de services et les entités de la société comme les clients de ce service. Mais ils peuvent aussi être gérés par un prestataire externe qui met à disposition du client un ensemble de services à la demande de ce dernier. Les clients louent ces services à la demande, selon leurs besoins. On parle dans ce cas de *clouds privés virtuels*. Dans ce dernier cas, une partie des services externalisés est pris en charge par un prestataire de confiance. Par ailleurs, une même infrastructure d'un prestataire peut contenir plusieurs clouds privés virtuels appartenant à différents clients. Chacun d'eux peut accéder uniquement à son cloud privé via son propre réseau. Avec le cloud privé virtuel, le lieu d'hébergement est connu par l'entreprise cliente et se trouve souvent dans le même pays que cette dernière. Cela permet surtout d'éviter des litiges juridiques dus aux éventuelles différences de législation en vigueur dans chaque pays hébergeur.

Cependant, la mise en place d'un cloud privé exige un investissement initial et des coûts de maintenance pour les clients. Elle nécessite également des compétences et des locaux adaptés. Même si les préoccupations en matière de sécurité et de contrôle de données sont dissipés avec les clouds privés (puisque le client conserve le contrôle total des données et des applications), le cloud privé demande des efforts considérables en cas d'évolution des besoins.

1.2.2 Les clouds publics

Les clouds publics mettent l'ensemble des ressources à disposition des clients via l'Internet public. Les ressources informatiques des différents clients peuvent être hébergées dans n'importe quel datacenter du prestataire voire passer d'un datacenter à l'autre pour garantir ainsi la qualité de service requise contractuelle ou optimiser l'utilisation des ressources du prestataire. Les clients consomment ainsi les services clouds sans se soucier des emplacements où sont hébergées leurs données. Ces emplacements sont gérés par des entreprises spécialisés qui louent leurs services ; la puissance de calcul et de stockage est alors mutualisé entre les différents clients. C'est ce type de cloud qui est utilisé pour la fourniture d'outils de bureautiques en ligne comme *Google Apps* ou *Office 365*, pour la la mise à disposition d'infrastructures virtuelles comme *OVH* ou *CloudWatt*, etc.

La mise en place d'un cloud public demande de lourds investissements uniquement du côté fournisseur de services. Les clients n'ont pas besoin d'investissement initial pour utiliser les clouds publics. Ils sont uniquement facturés pour les ressources informatiques

et/ou services qu'ils consomment. Cette catégorie de clouds offre une flexibilité et une facilité d'utilisation qui constituent la solution idéale pour les utilisateurs qui ont besoin d'infrastructures susceptibles de s'adapter à une charge fluctuante. Elle permet de surcroit une économie d'échelle significative et permet de développer rapidement des applications en évitant de provisionner un investissement dans les ressources informatiques et en même temps de limiter les risques.

Néanmoins, des inquiétudes persistent concernant la sécurité et la protection de la confidentialité des données dans un cloud public. En effet, ce cloud s'appuie et communique selon les politiques de sécurité du fournisseur pour garantir la sécurité des données de ses clients. Le cloud public offre un maximum de flexibilité et est souvent moins cher qu'un cloud privé pour le même service mais c'est un modèle moins sécurisé que celui du cloud privé. "*Sous la pression de l'Union Européenne et de la CNIL (**C**ommission **N**ationale de l'**I**nformatique et des **L**ibertés), les prestataires de Clouds publics assurent désormais plus de traçabilité sur l'emplacement des ressources mises à disposition en différenciant des grandes zones : Europe, Amérique et Asie*" [9]. Il peut aussi y avoir des litiges juridiques dus aux éventuelles différences de législation en vigueur si le fournisseur dispose de centres de données situés dans différents pays.

1.2.3 Les clouds hybrides

Les clouds hybrides mêlent les deux premiers concepts. Par exemple un cloud pour les applications et un autre pour les données. Ils constituent une solution alternative aux coûteuses infrastructures redondantes destinées à la continuité de l'activité en cas de défaillance, de sinistre, etc. Une entreprise possédant un cloud privé peut faire appel à un cloud public lors d'une montée en charge brutale ou alterner les clouds public et privé en fonction de la conjoncture. Les ressources du cloud public louées au fournisseur seront vues comme appartenant au cloud privé par les clients.

Les clouds hybrides permettent d'allier les avantages des clouds publics et privés. Ils permettent de répartir le traitement des données selon une politique préalablement définie par l'utilisateur, selon que le données soient stratégiques ou non. Les utilisateurs peuvent par exemple mettre les données et applications sensibles dans leur clouds privés et les applications qui demandent plus de flexibilité dans les clouds publics, plus rentables et plus performants. La partie publique du cloud peut être utilisée pour tester le bon fonctionnement d'une application nouvellement développée avant son lancement réel.

Cependant, les clouds hybrides héritent aussi les inconvénients des clouds publics et privés, tels que l'exigence d'investissement initial et des coûts de maintenance pour les clients, les problématiques de sécurité et de protection des données confidentielles, etc. Par ailleurs, la lourdeur de gestion de deux ou plusieurs clouds peut s'avérer plus contraignante pour les clients. Le fait d'unir clouds publics et clouds privés peut aussi poser des problèmes supplémentaires comme la difficulté d'intégration des services, la différence de politique de sécurité, etc.

1.3 Les niveaux de services du cloud computing

Le cloud computing permet aux entreprises de consommer des services informatiques à la demande et avec facturation à l'usage. Ces services peuvent se présenter sous plusieurs formes : sous forme d'application exploitée par l'utilisateur, sous forme de plate-forme qui exécute l'application et enfin sous forme d'infrastructure qui fournit les ressources matérielles nécessaire à la plate-forme d'exécution des applications Ce concept est alors décliné en trois offres, à savoir, l'*infrastructure en tant que service* (IaaS), la *plate-forme en tant que service* (PaaS) et le *logiciel en tant que service* (SaaS) que nous allons détailler par la suite.

1.3.1 IaaS (Infrastructure as a Service)

L'IaaS (ou l'infrastructure en tant que service) met à disposition des utilisateurs une infrastructure avec des ressources informatiques prêtes à l'emploi telles que des serveurs, des équipements réseau, de l'espace de stockage, etc. Les matériels et ressources informatiques des clients sont dématérialisés. L'IaaS est vu comme une abstraction d'un centre de données sur laquelle les consommateurs déposent leur environnement de production et leurs applications. Les clients disposent donc d'un environnement de production (dont ils en sont pas propriétaires) disposant de capacités matérielles qui peuvent s'adapter à tout moment de l'évolution des besoins. Du point de vue du client, ces ressources sont donc infinies, de nouvelles ressources pouvant être mises à leur disposition à la demande (et facturées à l'usage) en cas d'accroissement de la demande. Inversement, elles peuvent être retirées si elles ne sont plus nécessaires.

Les clients peuvent démarrer ou arrêter à la demande des serveurs virtuels dans ces centres de données, sans avoir à se soucier des machines physiques sous-jacentes, et des coûts de gestion qui sont liés (remplacement de matériel, climatisation, électricité etc.). Cela permet aux clients de se concentrer davantage sur le développement de leurs applications sans avoir à se préoccuper de l'achat de leurs propres serveurs ou de la gestion de l'infrastructure. Toutefois cela n'enlève pas toutes les responsabilités aux entreprises utilisatrices. Cette approche a l'avantage d'être très flexible mais demande de maintenir les mêmes compétences informatiques au sein de l'entreprise que pour les solutions serveur classiques sur site. Les utilisateurs conservent la responsabilité de gérer leurs environnements systèmes et les couches logicielles associées. Ils doivent ainsi disposer de ressources humaines suffisamment expertes pour gérer eux-même leurs systèmes et veiller au bon fonctionnement de leurs applications installées sur les machines virtuelles dans le cloud.

Les doutes sur la sécurité, sur la localisation géographiques des données et des machines virtuelles constituent les principales barrières à l'adoption des infrastructures en tant que service. Mais ces doutes commencent à se dissiper grâce aux contraintes qu'imposent certains organismes de réglementation aux fournisseurs de clouds publics [9], à la multiplication des fournisseurs locaux et à la maturité des offres. Une autre barrière à l'entrée est la non compatibilité des différentes technologies utilisées par

chaque fournisseur de clouds. Par exemple, l'utilisation d'Amazon impose la préparation de machines virtuelles au format Amazon sous l'hyperviseur open source Xen tandis que terremark et Orange Business Services utilisent VMware ESX.

1.3.2 PaaS (Platform as a Service)

La PaaS (ou la plate-forme en tant que service) est un modèle de cloud computing où le fournisseur de services met à disposition de ses utilisateurs des plate-formes d'exécution, de déploiement et de développement d'applications qui sont prêtes à l'emploi et fonctionnelles. Pour cela, PaaS fournit un niveau d'abstraction supplémentaire par rapport à IaaS : en plus de l'infrastructre matérielle, l'hébergement et le framework d'application sont dématérialisés. Les clients "poussent" leurs applications existantes dans le Cloud, ou développent de nouvelles applications avec les outils proposés par le fournisseurs tels que *Google App Engine* (plate-forme de conception et d'hébergement d'applications web basée sur les serveurs Google). Les utilisateurs ont à leur disposition des systèmes informatiques performants, configurés et maintenus par le fournisseur cloud pour y développer leurs propres applications. Les développeurs ne se préoccupent plus du déploiement, de la mise à jour et licence des logiciels ou de la nature des technologies pour exécuter leurs applications. La gestion des systèmes d'exploitation sous-jacents, des serveurs d'applicationsou des serveurs web sont aussi à la charge du fournisseur de services. Ils se concentrent uniquement sur l'architecture et le codage de leurs applications, permettant ainsi d'améliorer nettement la productivité. Cette plate-forme favorise également la collaboration entre les salariés du client.

La PaaS a cependant également quelques inconvénients. L'un des points faibles du PaaS est la limitation aux technologies disponibles chez le fournisseur de services pour le développement des applications. Par exemple, *Python* ou *Java* pour Google App Engine, *.NET* pour Microsoft Azure. Ainsi, les clients doivent réécrire toutes leurs applications s'ils veulent changer de fournisseur PaaS. La migration d'applications existantes vers ce cloud peut s'avérer compliquée selon le fournisseur de PaaS visé. En effet, il faut les réécrire si le fournisseur ne propose pas la plateforme d'exécution initialement utilisé. Or, ces plateformes sont parfois propriétaires. Par exemple le langage *Apex* de Saleforces est utilisé par la plate-forme de développement baptisée *Force.com* ; on est limité à ce langage et aux modules logiciels disponibles sur cette plate-forme tels que les bases de données, les frameworks, etc. Cela crée une forte dépendance vis-à-vis du fournisseur qui peut favoriser une certaine instabilité. En effet les clients n'ont aucun contrôle sur les évolutions de services virtuels de leurs fournisseurs, ce qui peut poser des problème si le fournisseur décide de mettre fin au développement de son service pour diverses raisons. Par ailleurs, le fait de ne pas avoir le contrôle sur les machines virtuelles sous-jacentes pourrait constituer aussi un inconvénient pour le développeur puisqu'il est impossible pour le client de surveiller les opérations effectuées par son fournisseur.

1.3.3 SaaS (Software as a Service)

Le SaaS (ou le logiciel en tant que service) est un modèle du cloud computing où le matériel, l'hébergement, le framework d'application et le logiciel sont dématérialisés. Le fournisseur SaaS s'occupe de la gestion des ressources nécessaires à l'infrastructure matérielle et à la plate-forme logicielle, de l'installation et de la configuration de tous les serveurs et des applications qui s'y exécutent (par exemple, serveur web, serveur de messagerie, intégration logicielle, mise à jour, etc.). Il veille également à la continuité de service en assurant la surveillance et les interventions en cas de pannes. Les clients accèdent à une application en mode hébergé sans avoir à se préoccuper ni de l'infrastructure matérielle, ni de la plate-forme logicielle. Ils n'achètent plus le logiciel mais le consomment à la demande, en payant son usage réel. Le logiciel est hébergé chez le fournisseur, dans son propre centre de données, comme c'est par exemple le cas pour les outils de messagerie et bureautique tels que *Google Apps, YahooMail, etc.*, *office 365*(outil collaboratif), *PostFiles* (envoi de fichiers lourds). Les clients se déchargent complètement de la partie opérationnelle de l'environnement système qui leur permet de se concentrer davantage sur leur cœur de métier ou sur des projets innovants à haute valeur ajoutée. L'utilisation de ce modèle de cloud computing permet aux entreprises d'avoir un bilan concret de leurs résultats. En effet, le coût de ces services devient une charge immédiatement déductible du résultat alors qu'avec les solutions serveurs classiques sur site, un investissement (par exemple, l'achat de matériel ou de licences logicielles) doit être déduit du résultat par le biais d'amortissements étalés sur plusieurs années. Par ailleurs, les applications sont accessibles et peuvent être utilisées partout et n'importe quand, il suffit juste d'une simple connexion Internet et d'un ordinateur.

Toutefois, le modèle SaaS présente également des inconvénients majeurs qui peuvent parfois constituer des barrières à l'entrée pour certaines entreprises utilisatrices. Le fait de confier les données sur des machine virtuelles appartenant à un fournisseur de services soulève des problèmes de confidentialité et de sécurité supplémentaires. De plus, les clients sont limités aux services proposés par le fournisseur de services. En effet, les services et les fonctionnalités de l'application sont souvent prédéterminés par le fournisseur de services. Ainsi, toute spécificité non prévue ne peut être développée que par le fournisseur. Par ailleurs, le changement de fournisseur SaaS est complexe car outre le transfert des données, il faut s'assurer que l'ensemble des fonctionnalités utilisées sont bien présentes dans le nouveau contexte applicatif. Enfin, ce modèle pose également de vraies questions sur la pérennité du fournisseur, notamment lorsque le service proposé en mode SaaS n'est pas disponible sous sa forme traditionnelle parce que la disparition du fournisseur entraine souvent la fin immédiate du service, ce qui n'est pas le cas dans un monde de licence traditionnel.

La figure 1.2 synthétise les caractéristiques d'un cloud selon son modèle de déploiement et son niveau de service. D'une part, le niveau d'abstraction et la flexibilité dépendent du niveau de service proposé. Le modèle SaaS représente plus d'abstraction mais moins de flexibilité par rapport aux deux autres niveaux de services (ce qui implique une forte dépendance du client vis-à-vis du fournisseur SaaS) tandis que le modèle IaaS représente moins d'abstraction et plus de flexibilité. D'autre part, le contrôle des

données et des applications par l'utilisateur dépend du modèle de déploiement choisi. L'utilisateur dispose plus de contrôle sur ses données et applications avec le cloud privé que les deux autres. Mais tout cela a un prix : l'investissement initial à réaliser du coté client est sans commune mesure.

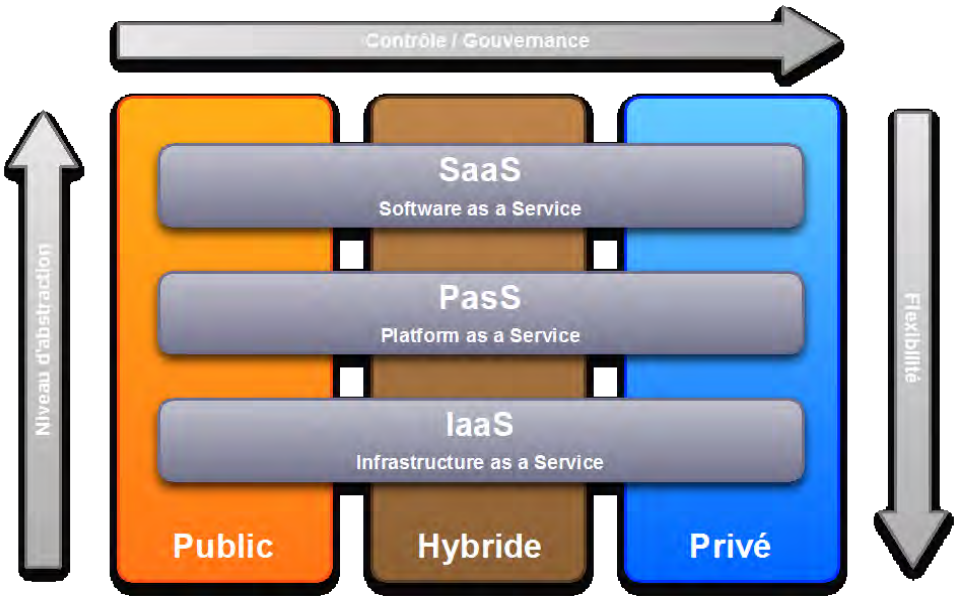


FIGURE 1.2 – Différent types de Cloud Computing

1.4 Les différents types de stockage dans le cloud

Depuis la généralisation de l'Internet à haut débit, les besoins en espace de stockage ne cessent d'augmenter et le stockage a suivi l'évolution de la puissance de calcul des ordinateurs. Historiquement, les premiers systèmes de stockage informatiques étaient de type attachement direct ou DAS (***D**irect **A**ttached **S**ystem*) : un ensemble de disques était physiquement lié à un serveur ou à une station de travail. Mais avec ce type d'architectures, les données deviennent inaccessibles lors d'une panne de la machine, de plus, les espaces disque des machines sont figés ou faiblement évolutifs. Ces problèmes ont conduit au développement des technologies de stockage en réseau telles que le NAS (***N**etwork **A**ttached **S**torage*) et le SAN (***S**torage **A**rea **N**etwork).*

Le stockage NAS attache les ressources de stockage au réseau LAN de l'entreprise, le plus souvent un réseau IP sur Ethernet. Un serveur NAS possède son propre système de gestion de fichiers et il est indépendant du système d'exploitation des serveurs qui s'adressent à lui. Les ressources sont accessibles via le réseau en utilisant un protocole réseau comme TCP/IP et des protocoles applicatifs tels que NFS (***N**etwork **F**ile **S**ystem*), CIFS (***C**ommon **I**nternet **F**ile **S**ystem* aussi nommé SMB : ***S**erver **M**essage **B**lock*) et AFP (***A**pple **F**iling **P**rotocol*). Les serveurs de stockage supportent plusieurs types de systèmes de fichiers et permettent ainsi le partage de fichiers entre plusieurs

serveurs et clients hétérogènes (par exemple sous Windows, Apple ou encore les systèmes d'exploitation basés sur UNIX). On peut également y ajouter à chaud de l'espace de stockage à la demande et sans éteindre le serveur de stockage NAS. Le stockage NAS permet une administration simplifiée via une interface WEB. Toutefois, un des inconvénients majeurs du stockage NAS est qu'il est gourmand en terme de bande passante. Il n'est donc pas particulièrement adapté aux flux multimédia (applications de *streaming multimédia*) ni aux traitements *temps réel*. Il n'est pas non plus optimisé pour le stockage de masse [81].

Le SAN est une méthode de stockage en réseau qui se présente sous forme d'un réseau secondaire intégralement dédié au stockage afin de désengorger le réseau de l'entreprise. Le réseau SAN, généralement basé sur le protocole FC (***F****iber* ***C****hannel*), interconnecte les serveurs entre eux et les périphériques de stockages. Il existe trois topologies pour ce réseau, à savoir :

- ☞ **la topologie point-à-point** : Elle relie un serveur et un périphérique de stockage. Ces deux entités reliées disposent donc de la totalité de la bande passante. Mais l'augmentation des débits a favorisé l'utilisation de deux autres topologies ;
- ☞ **la topologie en boucle** : les nœuds (serveurs et périphériques de stockages) sont situés sur une boucle. Si un nœud veut communiquer avec un autre, il doit passer par un de ses voisins immédiat. Donc, si un nœud est défaillant, la boucle est cassée et les nœuds de la boucle sont hors circuit. Pour pallier à ce problème, on utilise actuellement un HUB qui est relié à tous les nœuds. Ce HUB permet alors de mettre hors circuit un composant défaillant sans casser la boucle ;
- ☞ **la topologie "fabric"** : les nœuds du réseau sont raccordés aux autres via un ou plusieurs switchs FC. Cette topologie offre un débit nettement amélioré par rapport aux deux autres mais elle est très onéreuse.

La communication entre le serveur et le périphérique de stockage utilise le réseau SAN alors que celle entre le client et le serveur utilise le réseau LAN (***L****ocal* ***A****rea* ***N****etwork*). En effet, le réseau SAN est conçu pour s'adapter à la transmission des données de masse, alors que le réseau LAN est plus flexible et convient aux échanges de données entre le client et le serveur [81]. A la différence du stockage NAS, les systèmes de gestion de fichiers sont localisés sur les clients. Donc, les clients n'accèdent plus à des fichiers mais directement aux données brutes situées sur les périphériques, en mode « *bloc* ». Cela permet un gain de performance car l'accès aux données devient rapide. La technique de stockage SAN permet le partage des ressources de stockage entre un nombre très important de systèmes de traitements et d'utilisateurs. Elle soulage également le LAN de l'entreprise des charges induites par le transfert massif des données. Toutefois, la technique de stockage en réseau SAN possède également quelques inconvénients. Le réseaux dédié permet d'excellents performance mais peut se révéler très onéreux pour l'entreprise. Le réseau SAN devient dépendant du système d'exploitation des serveurs car des problèmes peuvent survenir si des serveurs ayant de systèmes de gestion de fichiers différents s'adressent au même disque. Mais ce dernier pourrait être résolu en utilisant les « *SAN file systems* ».

Dans le monde du cloud computing, les solutions de stockage doivent répondre à un

certain nombre d'impératifs. Ainsi, les deux solutions de stockage (DAS et NAS) citées précédemment ne conviennent pas au contexte du cloud computing. En effet, pour la technologie DAS, le taux de panne élevé, l'énorme gaspillage d'espace disque et surtout son approche locale constituent des défauts rédhibitoires. Pour la solution NAS, c'est surtout son problème de performance qui constitue un blocage : elle supporte mal, par exemple, des applications demandant de grosses performances disques comme des bases de données. Ainsi, l'implémentation du stockage dans les nuages est principalement fait en utilisant soit la solution SAN, soit les serveurs de stockages dédiés. La première solution est la plus simple à utiliser mais son coût est prohibitif. Ce point constitue souvent un frein à l'adoption de cette solution car les fournisseurs de stockage de masse dans un environnement cloud doivent allier à la fois performances et coûts. Quand à la seconde solution, elle est principalement utilisée pour fournir un stockage de données à des fins de sauvegarde ainsi que pour SaaS, PaaS et IaaS. Pour être efficace et compétitifs, les serveurs utilisés sont bâtis avec des processeurs de faible puissance, une mémoire RAM limitée mais avec une forte intégration de disque (jusqu'à 24 disques dans un serveur 2U). Le problème est alors de fournir une abstraction des ressources physiques à la couche logicielle et être capable de gérer ce nombre potentiellement très important de disques à travers un ensemble d'outils dédiés.

Dans un environnement de cloud computing, les offres de services de stockage peuvent se présenter sous plusieurs formes selon la frontière d'intervention du prestataire du cloud : le logiciel en tant que service (SaaS), la plate-forme en tant que service (PaaS) et l'infrastructure en tant que service (IaaS). Il y a également des systèmes de base de données dits NoSQL (***N**ot **O**nly **SQL***) qui occupent une place importante dans les systèmes de stockage distribué. Nous allons énumérer quelques principaux outils de gestion de stockage distribué (une liste non exhaustive) pour chaque niveaux de service et également des systèmes de base de données dits NoSQL.

1.4.1 Type SaaS

Diverses offres de stockage SaaS ont vue le jour, pour couvrir aussi bien les problématiques de partage que d'archivage des données. Parmi ces offres, Dropbox est parfaitement représentative des caractéristiques communes de ces services.

Dropbox est un service propriétaire de stockage dans le cloud de type SaaS. Outre le simple hébergement des fichiers, Dropbox permet à plusieurs utilisateurs de partager, modifier et synchroniser des fichiers en ligne [106], ce qui permet donc une collaboration entre ses utilisateurs. A cette fin, il divise chaque fichier en plusieurs morceaux de taille égale à 4 mégaoctets. Lorsqu'un utilisateur ajoute un fichier dans son dossier Dropbox local, l'application locale du client Dropbox va calculer les valeurs de hachages de tous les morceaux du fichier en utilisant l'algorithme de hachage *SHA-256*. Ces valeurs de hachage sont alors envoyées au serveur et comparées à celles qui y sont déjà stockées. Le serveur demande ainsi au client de transférer uniquement les morceaux qui ne sont pas encore stockés côté serveur, afin d'optimiser le trafic réseau et le coût de stockage. L'établissement de connexions entre les clients et les serveurs Dropbox sont effectuées

avec TLS (***T**ransport **L**ayer **S**ecurity*) [94]. Un des points faibles de Dropbox est que l'adresse mail pour s'y inscrire n'est pas vérifiée [94].

1.4.2 Type PaaS

Le stockage dans le contexte PaaS se structure autour d'environnements intégrés dédiés à l'exécution d'applications développées spécifiquement. Quelques exemples sont présentés ci-dessous.

- ⇨ **ConPaaS** : C'est un environnement intégré d'exécution open source qui vise à simplifier le déploiement d'applications de cloud computing. Une application est définie comme une composition d'un ou de plusieurs services. Les principales caractéristiques qui distinguent ConPaaS des autres offres PaaS sont : son approche visant à automatiser le passage à l'échelle des applications et son interopérabilité avec plusieurs clouds computing [83]. ConPaaS dispose actuellement de neuf services [2] :
  - deux services d'hébergement respectivement pour les applications PHP et JSP (***J**ava**S**erver **P**age*) ;
  - un service de base de données MySQL ;
  - un service de base de données NoSQL basé sur scalarix ;
  - un service MapReduce ;
  - un service TaskFarming pour le traitement batch à haute performance ;
  - un service Selenium pour les tests fonctionnels des applications web ;
  - un service HTC (***H**igh-**T**hroughput **C**ondor*) ;
  - un service XtreemFS offrant un système de fichiers distribué et répliqué.

  Les applications ConPaaS peuvent être composées d'un ou de plusieurs services. Par exemple, une application bio-informatique peut stocker les données génomiques dans XtreemFS, utiliser un service PHP et MySQL pour mettre en ligne une application web frontale et relier cette application web frontale au service MapReduce pour effectuer des calculs génomiques haute performance à la demande [83]. ConPaaS se présente donc comme une plate-forme en tant que service dans le cloud : il nécessite une infrastructure de niveau inférieure pour s'exécuter. Ainsi, il supporte à ce jour deux IaaS : OpenNebula et EC2 (*Amazon Elastic Compute Cloud*) [83].
- ⇨ **FRIEDA** (***F**lexible **R**obust **I**ntelligent **E**lastic **DA**ta Management*) [46] est un framework de gestion des données de type PaaS. Il gère le cycle de vie des données, qui comprend la planification et le provisionnement de stockage, le placement des données et l'exécution d'applications scientifiques dans des environnements clouds computing [51]. FRIEDA fournit un mécanisme d'exécution à deux niveaux qui sépare le contrôle des données et la phase d'exécution. Cette séparation permet une implémentation flexible des différentes stratégies de gestion des données au sein du même framework selon les exigences de l'application et les besoins en ressources. Il supporte deux systèmes de gestion de données : l'un *pré-déterminée*

et l'autre permettant le *partitionnement des données en temps réel*. Ces schémas facilitent diverses optimisations et compromis liés à la robustesse, la sélection des ressources de stockage, l'élasticité, la performance et le coût nécessaire dans les environnements de cloud computing [46]. L'architecture de FRIEDA est structurée selon trois rôles fondamentaux : *controller*, *master* et *workers* [51]. Le *controller* prend en charge la mise en place et la configuration de l'environnement pour la gestion des données et l'exécution d'applications. Le *master* est responsable de l'exécution d'applications et de la distribution des données. Les *workers* reçoivent les données et les tâches du composant *master* et les exécutent. Lorsque tous les *workers* ont terminé leurs tâches, le framework peut collecter les données de tous les nœuds. Il gère le cycle de vie des données requis dans le cadre du HPC en s'appuyant sur d'autres outils pour fournir l'accès effectifs aux données depuis les machines virtuelles. Il supporte pour cela trois options de stockage : un mode fichiers, Cassandra et DynamoDB (Amazon).

- ⇨ **Google AppEngine** : C'est une plate-forme de conception et d'hébergement d'applications web basée sur les serveurs de Google. Elle supporte les applications écrites en *Python*, *Java* et *Go*. Elle fournit également plusieurs types de services de stockage de données (MySQL, NoSQL et stockage orienté objet). Elle dispose d'un Load Balancer qui a pour rôle de répartir la charge sur les différents clusters. Son inconvénient majeur est son manque de flexibilité. En effet, les applications AppEngine fonctionnent exclusivement dans le cloud de Google [83]. De surcroît, les développeurs doivent utiliser les API propriétaires d'AppEngine, ce qui restreint la portabilité des applications vers d'autres infrastructures.
- ⇨ **Elastic Beanstalk** : C'est une solution PaaS propriétaire de AWS (***A****mazone* ***W****eb* ***S****ervices*). Elle simplifie le déploiement et le passage à l'échelle des applications et services web développés avec *Java*, *.NET*, *PHP*, *Node.js*, *Python*, *Ruby*, *Go* et *Docker* sur des serveurs tels qu'Apache, Nginx, Passenger et IIS [1]. Elle supporte plusieurs options de stockage et de base de données tels qu'Amazon RDS, Amazon DynamoDB, Amazon SimpleDB, Microsoft SQL Server, Oracle, IBM DB2 ou Informix. Comme dans le cas de Google AppEngine, *Elastic Beanstalk* est uniquement disponible dans les clouds d'Amazon [83].

1.4.3 Type IaaS

Au niveau infrastructure, les outils existants ont pour but de fournir une abstraction du stockage pour une approche orientée virtualisation. Même si ces outils sont susceptibles d'être utilisés dans d'autres contextes (par exemple, Ceph peut être utilisé pour du SaaS), la cible principale reste la fourniture de service de stockage bas-niveau pour les hyperviseurs car il s'agit du domaine le plus exigeant en termes de performances.

- ☞ **Ceph** est un logiciel open source (sous Licence LGPL) de stockage distribué de type IaaS, qui fait suite aux travaux de Sage Weil durant sa thèse [109]. Il dispose d'un composant principal (RADOS : ***R****eliable* ***A****utonomic* ***D****istributed* ***O****bject* ***S****tore*) qui assure un accès de manière fiable et autonome aux objets

de stockage distribué. Ceph/RADOS utilise l'algorithme CRUSH (***C**ontrolled **R**eplication **U**nder **S**calable **H**ashing*) pour optimiser les placements de données dans un stockage distribué [110, 17].

Ceph fournit une gestion dynamique et distribuée des méta-données en utilisant un cluster de méta-données (MDS) et stocke les données et les méta-données dans un OSD (***O**bject **S**torage **D**evices*) [37]. Lors d'une opération d'écriture, Ceph sélectionne un groupe de placement (PG : *placement group*) en utilisant l'algorithme CRUSH et stocke les objets correspondant dans les différents OSDs afférents. Trois stratégies de réplication sont implémentées dans Ceph :

- *"primary-copy"* : le premier OSD dans le PG transfère les écritures dans les autres OSDs et l'opération de lecture ne sera autorisée que si le dernier OSD a envoyé son acquittement.
- *"chain"* : les écritures des objets sont effectuées séquentiellement et l'opération de lecture sera autorisée une fois que le dernier objet a été écrit dans l'OSD correspondant.
- *"splay replication"* : la moitié des objets sont écrits de façon séquentielle, le reste étant effectué en parallèle. L'opération de lecture ne sera autorisé que si tous les objets sont écrits avec succès.

Chaque OSD utilise un journal pour accélérer les opérations d'écriture en réunissant plusieurs petites écritures et en les envoyant de façon asynchrone vers le système de fichiers lorsque le journal est plein [50]. Le journal peut être un device, une partition ou un fichier. Ceph/RADOS permet à une application d'interagir directement avec ce stockage distribué et fournit trois services principaux aux utilisateurs [40] :

- ✓ un accès direct au stockage objet via RADOSGW, compatible avec une API de type RESTFul (service Amazon S3, SWIFT / Openstack, etc.) ;
- ✓ un accès de type bloc via RBD (***R**ADOS **B**lock **D**evices*) ;
- ✓ un accès de type système de fichiers en réseau en utilisant CephFS.

L'un des points forts de Ceph est sa forte versatilité. En effet, il peut être utilisé pour le stockage massif des données, mais aussi en remplacement direct de services de stockage traditionnels, quelle que soit l'ancienne technologie utilisée [40]. Bien que Ceph soit présenté comme un système de stockage efficace, robuste et évolutif, la confidentialité et l'intégrité des données n'y sont pas implémentés. Les messages ne sont pas chiffrés sur le réseau et aucun algorithme d'encodage n'est utilisé au niveau des objets stockés. Par ailleurs, en se concentrant uniquement sur la disponibilité et l'évolutivité de données, Ceph/RADOS omet l'optimisation d'énergie ;

☞ **Lustre** [6, 105, 7, 37] est un de système de fichiers distribué basé sur des objets, disponible sur Linux sous une licence GPL. Son nom vient de deux mots *Linux* et *Cluster*. Il ne fait pas de copie des données ni des méta-données [37]. Au lieu de cela, il stocke les méta-données dans un stockage partagé, nommé MDT (***M**eta**D**ata **T**arget*) attaché à deux MDS (***M**eta**D**ata **S**ervers*), offrant ainsi un mode de basculement active/passive. Ce sont les MDS qui gèrent les requêtes

de méta-données. Les données elles-mêmes sont gérées de la même façon. Elles sont divisées en objets et ces derniers sont distribués dans plusieurs OSTs (***O**bject **S**torage **T**arget*) partagés qui sont attachés sur plusieurs OSS (***O**bject **S**torage **S**ervers*) pour fournir un mode de basculement active/active [37]. Ce sont les OSS qui gèrent les requêtes d'entrées/sorties. Lustre utilise les inodes et les attributs étendus pour mapper le nom d'un objet fichier à ses OSTs correspondants. Il implémente un LDLM (***L**ustre Distributed Lock Manager*) pour gérer la cohérence des caches [105]. Ce mécanisme de verrouillage permet à Lustre de supporter les accès concurrents en lecture et écriture. A titre d'exemple, Lustre supporte le verrou write-back qui permet aux clients de mettre en cache un grand nombre de mises à jour dans une mémoire avant de passer à une écriture proprement dite sur disque plus tard, réduisant ainsi les transferts des données. Plus de détails sur les fonctionnalités de LDLM peuvent être trouvés dans [105].

Lustre fournit des outils permettant de monter l'ensemble du système en espace utilisateur grâce à FUSE (Filesystem in User SpacE). Contrairement aux autres systèmes de stockage distribués, Lustre détecte les fautes lors des opérations des clients (qu'il s'agisse des requêtes de méta-données ou des transferts de données) et non pas lors des communications entre les serveurs. Ainsi, lorsqu'un client veut accéder à un fichier, il contacte d'abord le MDS. Si ce dernier n'est pas disponible, le client va demander au serveur LDAP pour trouver d'autres MDS qu'il pourra contacter. Ensuite, le système effectue un basculement, le premier MDS est alors déclaré défaillant, et le second devient actif. Une défaillance dans un OST se fait de la même manière : si les clients envoient des données vers un OST et que ce dernier ne répond pas pendant un certain temps, il sera redirigé vers un autre OST [37, 17]. Lustre fournit également des outils simples d'équilibrage de charge : lorsqu'un OST est trop chargé, le MDS va choisir d'autres OSTs disposant de plus d'espace de stockage libre pour stocker les données. L'un des inconvénients de Lustre est la technique de mappage utilisée pour localiser les objets d'un fichier au lieu d'une fonction de hachage déterministe tels que RUSH [54, 55] ou CRUSH[111]. En effet, la technique de mappage fait augmenter la pression au niveau des serveurs dédiés [17].

☞ **XtreemFS** : XtreemFS fait partie du projet XtreemOS [29]. C'est un système de fichiers distribué orienté objet, de type IaaS et qui a été spécialement conçu pour les environnements de l'informatique en grille [58]. XtreemFS est composé des clients, d'OSDs (***O**bject **S**torage **D**evices*) et serveurs des méta-données MRC (***M**etadata and **R**eplica **C**atalog*). Les OSDs de XtreemFS fournissent aux clients un accès haute performance aux objets qu'ils stockent et sont responsables du stockage effectif sur leurs disques. Par ailleurs ils maintiennent un cache pour les objets récemment accédés. Quand un client accède une donnée dans un OSD pour la première fois, sa demande comprend des informations sur les emplacements des autres répliques de la donnée, ce qui permettra à l'OSD de coordonner les opérations d'une manière peer-to-peer [59]. Aucun composant central n'est impliqué dans ce processus, ce qui rend XtreemFS tolérant aux défaillances et apte au passage à l'échelle.

Un MRC stocke toutes les informations sur ses volumes hébergés dans une base de données intégrée, qui dispose d'un mécanisme pour répliquer ses données entre les hôtes. Les volumes peuvent être hébergés par plusieurs MRCs, permettant ainsi d'améliorer la disponibilité des volumes et la performance des accès. XtreemFS fournit trois politiques différentes pour stocker des données dans le système de fichiers [104] : la première ne fait pas de réplication, la deuxième utilise WaR1 (***W****rite all* ***R****ead one*) et la dernière s'appuie sur le quorum majoritaire. Contrairement à Ceph, un des inconvénients majeurs de XtreemFS est le fait qu'il ne supporte pas les approches de stockage hiérarchisé. Par ailleurs, même si le trafic réseau est sécurisé avec l'utilisation des connexions SSL (***S****ecure* ***S****ockets* ***L****ayer*), les données ne sont pas encore protégées dans les périphériques de stockage. En effet, l'encryptage de bout en bout n'est pas encore implémenté dans XtreemFS.

☞ **HDFS** (***H****adoop* ***D****istributed* ***F****ile* ***S****ystem*) est le composant système de fichiers de Hadoop sous licence Apache Open Source développé par *Apache Software Foundation* [93]. Il est conçu pour stocker de façon fiable de très gros fichiers dans un grand cluster. Les données et les méta-données sont stockées séparément : les méta-données dans un serveur dédié appelé *NameNode*, tandis que les données sont répliquées et distribuées dans d'autres serveurs appelés (*DataNodes*). Un *NameNode* secondaire fournit une copie persistante du *NameNode* principal afin d'éviter qu'il y ait un point unique de défaillance pour les méta-données [37]. Les données sont subdivisées en blocs, ces derniers étant ensuite répliqués et distribués dans plusieurs *DataNodes*. Aucun serveur *DataNode* ne détient plus d'une copie d'un bloc et un rack contient au maximum deux copies d'un même bloc. Le *NameNode* vérifie qu'à chaque bloc correspond un nombre souhaité de répliques. Si un bloc est sous répliqué, le *NameNode* crée une nouvelle réplique et la place dans une file d'attente prioritaire, qui est vérifiée de façon périodique. Le nouveau bloc sera stocké en conformité avec la politique de placement utilisée pour les données. Les données sont ainsi répliquées de façon asynchrone, mais des outils sont fournis pour supporter le mode synchrone.

Les serveurs HDFS échangent des messages pour détecter des défaillances éventuelles telles que des pannes réseau et serveurs. Par exemple, toutes les trois secondes les *DataNodes* envoient un message au *NameNode* pour indiquer à ce dernier qu'ils sont toujours disponibles. Un *DataNode* est déclaré défaillant par le *NameNode* si ce dernier ne reçoit pas de message durant dix minutes. Ces messages contiennent également des informations statistiques (la capacité de stockage, le nombre de transferts de données en cours, etc.) qui permettent au *NameNode* d'assurer l'équilibrage de charge. Les utilisateurs accèdent aux fichiers en utilisant leur URLs dans l'espace des noms et contactent le *NameNode* pour déterminer où sont stockés les blocs de données. Ils contactent ensuite directement les *DataNodes* pour demander le transfert des informations. Cela se fait de manière transparente en utilisant une API en C, Java ou REST [37]. HDFS offre également un module basé sur FUSE. HDFS est optimisé pour les lectures récurrentes mais il est sensiblement moins performant pour des écritures concurrentes. Même si l'architecture serveurs *NameNode* active/passive vient compenser le problème de point

unique de défaillance, la gestion centralisée des méta-données constitue l'un de ses inconvénients majeurs.

- ☞ **S3** (***S**imple **S**torage **S**ervice*) : C'est une solution IaaS orientée stockage incluse dans AWS (***A**mazon **W**eb **S**ervices*). S3 a comme objectif de fournir une solution stockage à faible coût, hautement disponible et avec un modèle de facturation de type *paiement à l'usage* (*"pay-as-you-go"*) [76]. Les données stockées dans S3 sont organisées sur deux niveaux d'espace de nommage. Au niveau le plus haut on trouve les *"buckets"*, qui sont similaires aux dossiers ou aux conteneurs. Les buckets contiennent un nom global unique et servent à plusieurs besoins : ils permettent aux utilisateurs d'organiser leurs données et ils servent de références pour les rapports d'audit [76]. Chaque bucket peut contenir un nombre illimité d'objets de données. Chaque objet a un nom et une taille qui peut aller jusqu'à 5 Go ; les méta-données peuvent avoir une taille allant jusqu'à 4 Ko. Renommer ou déplacer un objet vers un autre bucket nécessite la récupération de l'objet entier et une ré-écriture dans S3 avec son nouveau nom. S3 implémente trois services de stockage pour l'accès aux données : SOAP (***S**imple **O**bject **A**ccess **P**rotocol*), REST (***R**epresentational **S**tate **T**ransfer*) et BitTorrent [76]. L'interface S3 est utilisée par divers systèmes et applications, y compris les plate-formes de gestion de cloud computing tels qu'OpenStack et CloudStack [97].
- ☞ **GlusterFS** : C'est un système de fichiers libre distribué en parallèle, de type IaaS et qui repose sur un modèle client/serveur. Les serveurs sont déployés comme des briques de stockage tandis que les clients se connectent aux serveurs et regroupent les volumes distants en un unique volume. Les composants du client manipulent les volumes virtuels en utilisant des translateurs empilables et quelques fonctionnalités additionnelles (comme la distribution, la réplication, le stripping, etc.). Ainsi, GlusterFS peut implémenter un niveau élémentaire de confidentialité en utilisant les translateurs (même s'il ne propose pas un stockage entièrement sécurisé certaines extensions de translateurs de HekaFS sont en cours de portage) et les outils de réplications et stripping. GlusterFS ne fait pas de réplication au niveau du volume logique mais utilise des techniques de mise en miroir comparables au RAID 1 [37] : une ou plusieurs copie(s) du contenu d'un dispositif de stockage sont effectuées dans d'autres dispositifs de stockage appartenant au même volume logique en utilisant des écritures synchrones. Ainsi, lors d'une opération d'écriture dans un volume logique composé de plusieurs dispositifs de stockage, un premier dispositif est choisi pour le stockage initial ; l'écriture est ensuite propagée de façon synchrone vers les miroirs. GlusterFS ne gère pas les méta-données dans un serveur dédié et centralisé mais il utilise un algorithme EHA (***E**lastic **H**ashing **A**lgorithm*)[8, 37] pour localiser les fichiers. EHA utilise une fonction de hachage pour convertir l'url d'un fichier en un nom logique qui est transparent pour les utilisateurs. Ceci est fait de manière uniforme et permet à GlusterFS de garantir que chaque stockage logique stocke à peu prêt le même nombre de fichiers. Il gère de manière intelligente la répartition de la charge, la réplication des données et des méta-données permettant d'avoir un système de stockage hautement disponible et résistants à plusieurs pannes. Gluster utilise des connections TCP/IP et InfiniBand RDMA permettant ainsi de gérer plusieurs milliers de clients sur

le même lien. Toutefois, certaines limites méritent d'être mentionnées. Premièrement, l'authentification des clients se fait en se basant sur l'adresse IP, laissant la porte ouverte pour les utilisateurs malveillants. Deuxièmement, la réplication n'est pas tenable et il ne gère pas de cache côté client. Or, l'utilisation des caches permet d'optimiser le trafic réseaux et le CPU lors des requêtes répétées sur le même fichier [74]. Enfin, l'économie d'énergie n'est pas prise en compte et ne peut pas être implémentée en utilisant simplement ce système de fichiers.

1.4.4 Autres types de stockage

Face à une croissance de façon exponentielle des besoins en termes de charge et de volumétrie de données, une alternative aux SGBD a vu le jour : le NoSQL. En effet, un des problèmes récurrent des bases de données relationnelles est la perte des performances lorsque l'on doit traiter un gros volume de données. Le principal changement apporté par NoSQL par rapport aux SGBDR classiques est le fait de renoncer au modèle ACID (***A****tomicité* ***C****ohérence* ***I****solation* ***D****urabilité*). Les solutions NoSQL répondent au théorème CAP (***C****onsistency* ***A****vailability* ***P****artion-Tolerance*) énoncé par Eric Brewer [24]. Les trois propriétés CAP sont toutes les trois recherchées dans un système distribué mais le théorème dit qu'il est impossible pour un système distribué d'obtenir à la fois ces trois garanties [24]. Bogdan George Tudorica propose une comparaison de plusieurs bases de données NoSQL dans [102].

- ⇨ **Cassandra** [28, 65, 34] fait partie des systèmes de base de données dits NoSQL. Cassandra est une base de données NoSQL orientée colonnes qui respecte l'architecture distribuée Dynamo d'Amazon et le modèle BigTable de Google [65]. Elle est écrite en Java et distribuée sous licence Apache. Lors d'une mise à jour, les données ne sont pas écrites directement sur disque mais stockées dans une table en mémoire. Lorsque la table en mémoire est pleine, son contenu sera écrit sur le disque et la représentation du disque est périodiquement compacté. Elle effectue le partitionnement et la réplication des données et elle offre la possibilité de mettre en cluster plusieurs serveurs. Cassandra n'utilise pas de mécanisme de verrouillage et les mises à jour sont effectuées de façon asynchrone [28].
- ⇨ **SimpleDB** est un système de gestion de base de données non relationnel (NoSQL) écrit en Erlang par Amazon, distribué sous une licence propriétaire. Il fait partie des offres commerciales d'Amazon au même titre que *Elastic Compute Cloud* (EC2) et *Simple Storage Service* (S3) [28]. La stratégie de réplication utilisée est la réplication asynchrone. SimpleDB supporte le modèle de cohérence éventuelle (Eventual Consistency) et ne supporte pas la cohérence transactionnelle [28]. L'inconvénient majeur du modèle de cohérence éventuelle est que les requêtes de lecture peuvent renvoyer des données qui ne sont pas à jour [108], ce qui peut ne pas être acceptable pour certaines applications.
- ⇨ **Terrastore** [28] est le système de gestion des données NoSQL utilisé dans Terracotta, solution *Open Source* de *clustering* de JVM. Les données stockées sont accessibles directement via HTTP mais des APIs client en Java et Python sont aussi implémentées. Terrastore partitionne les données sur les serveurs de façon

automatique et peut automatiquement redistribuer les données lorsque des serveurs sont ajoutés ou supprimés. Elle supporte la réplication et assure la cohérence pour chaque document : une lecture retourne toujours la dernière version d'un document. Par contre, elle ne supporte pas la cohérence multi-documents, ne garantissant pas ainsi le passage à l'échelle [5].

Les systèmes de bases de données NoSQL permettent d'améliorer la performance et le passage à l'échelle par rapport aux SGBD classiques, mais en contrepartie ils ne garantissent plus la cohérence stricte. Par ailleurs, Dianna Gudu et al. indiquent dans [50] que les systèmes de bases de données NoSQL souffrent d'un manque d'interfaces standardisées, ce qui entrave souvent leur adoption.

1.4.5 Le système de stockage CloViS

L'objectif principal du projet est de couvrir tous les aspects avec un seul intergiciel, c'est à dire un intergiciel capable de faire à la fois du SaaS, PaaS et IaaS avec un choix complet de modalités d'accès aux données. Cela nous a poussé à l'implémentation d'une couche de virtualisation de stockage nommé CloViS. Cet intergiciel a été développé en s'appuyant sur l'expertise acquise dans la virtualisation du stockage appliquée aux grilles informatiques [11]. CloViS se présente comme un service de stockage complet, utilisable dans tous les environnements classiques du nuage (le détail de CloViS sera donné dans le chapitre 2 de cette partie). An niveau PaaS, il propose des méthodes d'accès standard aux blocs de données telles que le protocole iSCSI. Il peut donc être utilisé sans difficultés pour fournir du stockage bloc à une infrastructure, de manière transparente vis-à-vis des clients du stockage. En ce sens, il remplace aisément les systèmes SAN usuels. Au niveau IaaS, des éléments particuliers sont à noter qui guident les besoins en stockage. Le schéma suivant présente une architecture standard d'un IaaS :

- une API pour les accès utilisateurs et administrateurs ;
- un contrôleur et un orchestrateur qui gèrent la création, la modification et le placement des machines virtuelles ;
- des systèmes de virtualisation comme VMWare, XenServer, KVM et autres ;
- une infrastructure réseau qui peut-être elle aussi être virtualisée ;
- des systèmes de stockages.

Les besoins en stockage dans des clouds de type IaaS se répartissent en deux fonctions bien distinctes :

- ☞ le stockage des machines virtuelles qui est vu comme temporaire. En effet, un serveur virtuel dans une solution cloud constitue une ressource temporaire qui peut être supprimée à tout moment ;
- ☞ le stockage de type « *bloc* » qui doit stocker de façon sécurisée et pérenne les données utilisateur des machines virtuelles.

A ces deux fonctions il convient d'ajouter les services de stockages de type SaaS (Storage as a service) qui permettent la conservation des données de l'utilisateur de manière

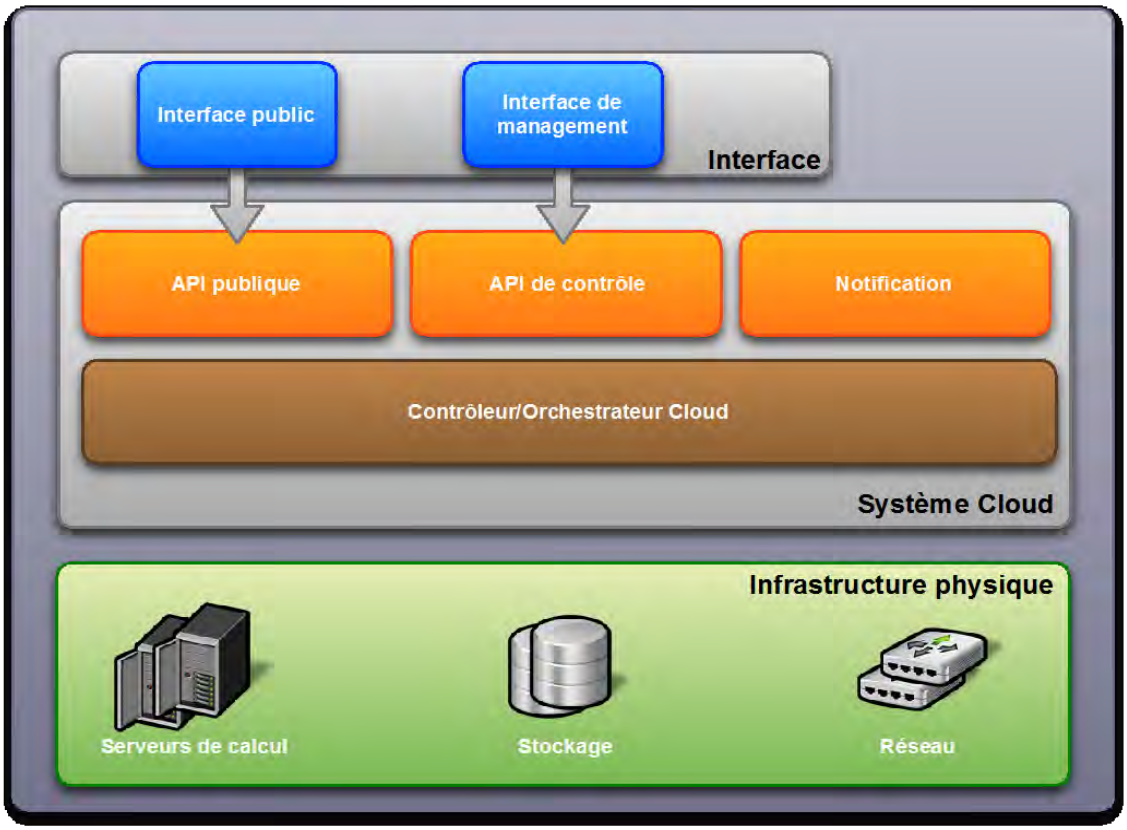


FIGURE 1.3 – Architecture standard d'un IaaS

sécurisée. Les données sont rendues accessibles à l'utilisateur grâce à une API de type RESTFul (service Amazon S3, Microsoft SkyDrive, etc.). Trois technologies différentes sont généralement utilisées selon les fonctions de stockage énumérées ci-dessus. Premièrement, pour le stockage des machines virtuelles, il s'agit généralement d'un système de gestion de fichier classique (ext4, nfs, etc.). Ce système de stockage peut être réparti entre les différents nœuds de virtualisation pour la migration à chaud des machines virtuelles et ainsi permettre de répondre à un besoin en haute disponibilité. Deuxièmement, pour le stockage des données utilisateur des VMs, on utilise le plus souvent un stockage de type bloc type iSCSI qui sera attaché de façon temporaire à la machine virtuelle. Il s'agit aussi du mode d'accès utilisé dans une contexte PaaS. Notons qu'un stockage de type système de gestion de fichier exporté depuis l'infrastructure physique et monté dans la machine virtuelle peut aussi être utilisé. Enfin, pour les services de type SaaS (Storage as a service), c'est une API REST qui est le plus souvent mise en œuvre. Ce choix est lié à la nature même des accès aux données : aucune gestion de la concurrence des données n'étant assurée ni souhaitable car trop lourde à supporter, la mise en œuvre d'un système de gestion de fichier classique n'est pas envisageable.

1.5 Sécurité des données stockées dans le cloud

Lorsque les données sont placées dans le nuage, la sécurité de leur stockage est critique. Parce que les données ne sont plus gérées par leur propriétaire, ce dernier doit être assuré que la sécurité est préservée. Cette sécurité est définie par trois paramètres principaux : la confidentialité, l'intégrité et la disponibilité. Ces trois paramètres permettent de caractériser les menaces susceptibles de peser sur les données.

La disponibilité des données est cruciale. Le fournisseur de stockage doit pouvoir garantir que les données seront accessibles quoi qu'il arrive, en s'engageant sur des délais de rétablissement du service sous astreinte financière. Des fournisseurs tels qu'Amazon avec son service S3 (Simple Storage Service) sont susceptibles d'être indisponibles pendant plusieurs heures. Dans un tel cas, l'infrastructure informatique du client devient inopérante partiellement (si quelques services seulement sont hébergés par le biais de solutions de type SaaS) ou totalement (dans le cas où toute l'infrastructure serveur du client est virtualisée et hébergée chez le fournisseur), entrainant des pertes importantes. Pour assurer la disponibilité des données en toutes circonstances, le fournisseur doit donc mettre en œuvre des solutions de redondance d'une part, et des mécanismes de sauvegarde d'autre part.

La problématique de l'intégrité des données concerne la garantie recherchée par le client que toutes ses données externalisées sont bien présentes chez son fournisseur et de manière non altérées. Des atteintes à l'intégrité des données peuvent résulter d'attaques de tiers malveillant, de défaillances des infrastructures de l'hébergeur, ou d'un choix délibéré de ce dernier de supprimer des données non accédées pour optimiser ses coûts. Le propriétaire des données doit donc pouvoir détecter une atteinte à l'intégrité des données sans nécessiter de conserver une copie locale de ses données (le but étant justement d'externaliser le stockage).

La confidentialité des données reste l'une des préoccupations majeures et le principal obstacle au développement des services dans le nuage. Cette confidentialité est exposée aux menaces classiques (attaques par injection, cross-site scripting, etc.) mais aussi à des menaces spécifiques à l'informatique en nuage (failles dans les hyperviseurs, gestion du périmètre de sécurité au sein d'une entreprise, confiance dans le fournisseur). A ce niveau, on peut distinguer des différences selon le degré d'externalisation de l'infrastructure du client chez le fournisseur :

- soit le client utilise l'hébergement d'application serveur (bases de données, web services, serveurs de messagerie, etc.), ces applications étant accédées par le client par le biais de connexions sécurisées. Cela suppose généralement une infrastructure minimale chez le client, car d'une part certaines applications ne peuvent pas être mises dans le nuage, et d'autre part les données des applications bureautiques restent au sein de l'infrastructure propre du client et nécessitent donc, à minima, un serveur d'authentification et un serveur de fichier. Dans ce cas, l'application étant gérée par le fournisseur, la protection des données est une problématique complexe ;

➜ soit le client virtualise ses serveurs (et le cas échéant, les postes de travail). Dans ce cas, toute l'infrastructure serveur du client est déplacée dans le nuage et toutes ses données s'y trouvent, hébergées sur des serveurs virtuels dédiés. Dans ce cas, le client dispose de moyens pour sécuriser ses données ou du moins, rendre leur accès ardu.

Le projet SVC a pour but d'apporter des réponses à ces préoccupations sécuritaires dans le cloud. L'objectif principal du projet est de disposer d'un Framework de sécurité permettant d'opérer dans le cloud des applications critiques du point de vue des traitements et des données sans craindre un piratage ou un examen par un organisme « curieux » ou malveillant. Le développement exponentiel du cloud ainsi que les innovations technologiques et économiques imposent une recomposition de la chaine de la valeur qui favorise l'émergence de grands opérateurs européens et mondiaux. Le Framework SVC est un middleware sécurité ouvert et disponible à tous garantissant la capacité d'innovation des acteurs économiques français et permettant de lutter à armes égales avec des entreprises étrangères. Ce Framework répond également à un enjeu de souveraineté par le maintien sur le territoire national de capacités de traitements d'informations stratégiques. Pour réaliser ce framework, plusieurs laboratoires et sociétés aux compétences complémentaires se sont structurés en consortium. Tous les acteurs impliqués sont des acteurs clés dans le domaine de la sécurité numérique et du cloud computing et sont extrêmement complémentaires pour construire une solution globale répondant en tous points aux exigences du marché. Ensemble, ils couvrent la totalité de la chaine de la valeur, de la recherche appliquée à la commercialisation :

→ Technologies innovantes issues de laboratoires de recherche de pointe : LAAS et IRIT ;
→ Éditeur de logiciel sécurité SaaS : ITrust, Secludit ;
→ Éditeur spécialisé dans les services à la demande opensource : BlueMind ;
→ Éditeur mode SaaS : Val Informatique ;
→ Opérateurs IaaS et SaaS : GOSIS, E-NEED, Bull ;
→ Société d'ingénierie et de service en accompagnement projet : Eurogiciel ;
→ Constructeur informatique et éditeur de logiciels d'infrastructure et infogéreur : Bull ;
→ Intégrateur et distributeurs du Framework : Bull, GOSIS, E-NEED, ITRUST.

Ainsi, le projet final qu'est SVC doit permettre à un ensemble de machines virtuelles, dans un cloud, de fonctionner ensemble comme dans un datacenter, avec un haut niveau de sécurité et avec une répartition des données sur plusieurs sites, de manière à ce qu'aucun site ne dispose sur ses espaces de stockage des données complètes et intelligibles. Un tel dispositif nécessite de :

- Sécuriser les accès au SVC (flux entrants, intégrité des VM) ;
- Assurer la confidentialité des informations échangées entre les applications hébergées dans le SVC (Routage et flux interVM et inter applications) ;
- Gérer les données de manière à les disséminer sur plusieurs sites tout en assurant une performance compatible avec l'exécution des applications (Segmentation et fragmentation) ;

- S'assurer de la non-fuite des données vers des sites non approuvés (prévention des pertes de données et filtrage) ;
- Détecter les comportements anormaux dans le SVC pour anticiper les dysfonctionnements ou failles de sécurité (Analyse comportementale et Surveillance temps réelle intelligente) ;
- Proposer des services à la demande.

CloViS fait partie du sous-projet « *Stockage sécurisé* » du projet SVC. C'est un intergiciel de gestion de données développé au sein du laboratoire IRIT, qui permet la virtualisation de ressources de stockage hétérogènes et distribuées, accessibles d'une manière uniforme et transparente.

1.6 Synthèse

Nous venons de présenter les différents modèles de déploiements et les niveaux de services du cloud computing. Le cloud computing offre un large choix de services informatiques à la demande et avec facturation à l'usage aux utilisateurs selon leurs besoins. Ces services peuvent se présenter sous plusieurs formes : logiciels, plates-formes et infrastructures. Trois modèles de déploiements sont possibles : le *cloud privé* où l'entreprise utilisatrice garde la charge de la gestion, le *cloud public* qui est géré par un prestataire externe et enfin le *cloud hybride* qui est un mélange d'un cloud privé et d'un cloud public.

Nous avons également vu dans ce chapitre que plusieurs solutions de stockage, tant commerciales que open source, sont disponibles dans le cloud computing. Qu'ils s'agissent de solutions de types SaaS, PaaS ou IaaS, il existe toujours un dilemme entre plusieurs paramètres tels que la performance, la disponibilité, le passage à l'échelle, la cohérence des données, etc. Nous avons également présenté le projet SVC dont l'objectif principal est de fournir un Framework afin de répondre aux préoccupations sécuritaires dans le cloud computing. CloViS, que nous allons détailler dans le chapitre suivant, est issu du projet SVC, et a pour ambition de couvrir tous les aspects de solutions de stockage dans le cloud computing (SaaS, PaaS et IaaS) avec un choix complet de modalités d'accès aux données.

Chapitre 2

Le projet CloViS

Sommaire

2.1 Définition et historique de CloViS

CloViS est l'acronyme de : **Clo**ud **Vi**rtualized **S**torage. CloViS est un intergiciel de gestion de données développé au sein du laboratoire IRIT (Institut de Recherche en Informatique de Toulouse), qui procède à la virtualisation de ressources de stockage hétérogènes et distribuées et permet un accès d'une manière uniforme et transparente. Il possède la particularité de mettre en adéquation les besoins des utilisateurs et les ressources du système par le biais de qualités de service définies sur des volumes virtuels. CloViS est le fruit de la tâche "*Stockage sécurisé*" du projet SVC (Secured Virtual Cloud) financé par le Font National pour la Société Numérique "Investissement d'avenir" et qui entre dans une phase de valorisation portée par la SATT Toulouse-Tech-Transfer.

CloViS est la suite logique du projet ViSaGe (***Vi****rtualisation du* ***S****tockage* ***a****ppliquée aux* ***G****rilles informatiqu****e****s*) qui est un intergiciel de grille offrant des services de stockage large-échelle en agrégeant des ressources physiques de stockage distribuées. ViSaGe a été développé dans le cadre du projet éponyme labellisé par le RNTL en 2005. Plusieurs

acteurs (issus de laboratoire de recherche de pointe et industriels) ont participé à ce projet RNTL tels que CS (Communications et Systèms), IRIT, SEANODES et EADS. Ce projet a pris fin officiellement en 2007 avec une première version opérationnelle. Les travaux ont ensuite continué au sein du laboratoire IRIT [75].

CloViS est un intergiciel qui s'installe sur des serveurs linux. Sa vocation est de permettre l'agrégation de ressources de stockage hétérogènes et distribuées sur différents nœuds pour offrir des espaces de stockage virtualisés, accessibles de manière transparente et garantissant des qualités de services définies. Ces ressources de stockage peuvent être différentes, que ce soit en ce qui concerne leurs tailles, leurs bandes passantes ou leurs latences. CloViS offre une vue unifiée de l'espace de stockage indépendamment des ressources de stockage utilisées. La figure 2.1 montre les différents niveaux d'abstraction opérés par CloViS. Le niveau d'abstraction le plus élevé pour

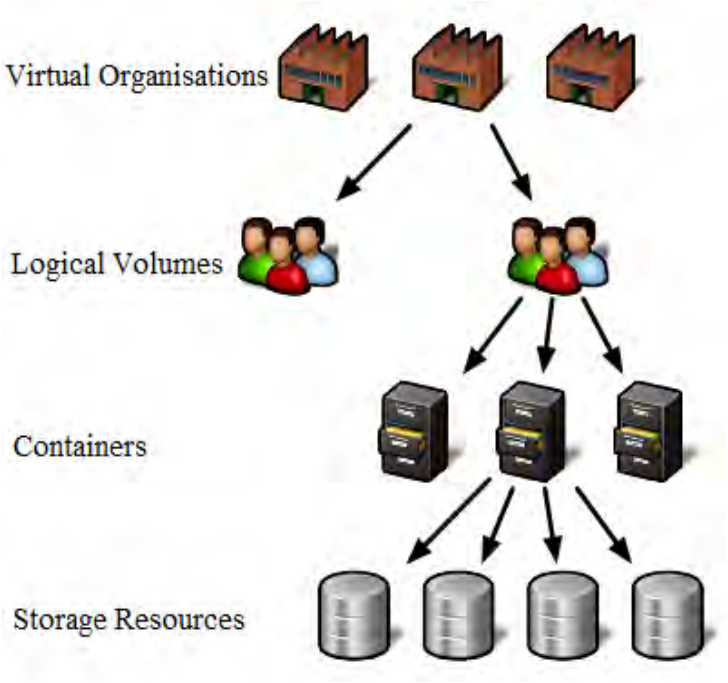


FIGURE 2.1 – Niveaux d'abstraction de CloViS

CloViS correspond à une organisation virtuelle (VO). L'espace de stockage est donc structuré en organisations virtuelles. Pour le fournisseur de stockage dans le nuage, une organisation virtuelle peut, par exemple, correspondre à un client particulier. De plus, un client peut avoir de nombreux départements et devrait être autorisé à assurer l'isolation entre ses espaces de stockage. Dans chaque organisation virtuelle, l'espace de stockage global dédié à un client peut donc être divisé en de nombreux volumes logiques (LV). Ces volumes logiques vont permettre la lecture et l'écriture des données par le biais d'une couche d'accès client (système de gestion de fichier compatible POSIX, bibliothèque d'entrées/sorties spécifique, etc.). Pour assurer la persistance des données, les volumes logiques s'appuient sur les ressources de stockage physiques partagées au niveau de chaque nœud. Ainsi, le fournisseur de stockage peut, au niveau de chaque

nœud de ses différentes grappes de serveurs (cluster), préciser le pourcentage d'espace disque à accorder pour chaque organisation virtuelle. La définition, par le fournisseur de stockage, des ressources physiques allouées à une organisation virtuelle peut ainsi prendre en compte la nature des disques sous-jacents (vitesse, usure) en fonction des demandes du client.

Le fournisseur de services cloud (CSP : ***C****loud* ***S****ervice* ***P****rovider*) doit permettre la différentiation de services. Par exemple, on peut avoir besoin d'un système de stockage très rapide pour exécuter les services qui consomment de la bande passante disque et un système de stockage un peu lent mais très fiable pour l'archivage des résultats produits par ces services. Le CSP peut donc allouer de l'espace de stockage sur SSD pour le premier cas et de l'espace sur de nombreux disques classiques pour le deuxième. En fonction des exigences du client, le CSP va affecter certaines ressources de stockage à la VO. Ainsi que nous l'avons souligné, différents clients peuvent avoir des besoins différents, et même un seul client peut avoir besoin de nombreuses caractéristiques d'espace de stockage. Ainsi, chaque volume logique peut utiliser de nombreuses qualités de services pour le stockage des données. Ces qualités de services sont implémentées dans les volumes logiques par le biais de conteneurs (CAN : ***C****ont****ai****n****er*). Ce niveau supplémentaire dans la hiérarchie du stockage de CloViS, masqué à l'utilisateur, est utilisé par le virtualiseur pour la mise en œuvre des politiques du fournisseur évoquées précédemment. Un volume logique est donc composé de plusieurs conteneurs possédant chacun des caractéristiques spécifiques en fonction des demandes de stockage effectuées par le composant d'accès client. Chaque volume logique a un conteneur par défaut qui est associé à une qualité de service spécifique. Chaque fois qu'une donnée est écrite, la qualité de service à lui attribuer doit être précise. Lorsque cette qualité de service est modifiée, CloViS cherche un conteneur qui implémente cette QoS sur ce volume logique. Si un tel conteneur ne peut être trouvé, un nouveau conteneur est crée en utilisant les ressources de stockage disponibles déclarées au niveau de l'organisation virtuelle. La donnée est alors déplacée du conteneur original vers le conteneur qui implémente la qualité de service requise.

L'utilisation d'un volume logique requiert donc les étapes suivantes :

1. Création d'une organisation virtuelle
2. Partage des ressources de stockage et affectation à une organisation virtuelle
3. Création d'un volume logique dans cette organisation virtuelle en précisant la QoS par défaut

Si l'accès doit se faire par le biais d'un système de gestion de fichiers, alors il faut aussi :

4. Formater le volume logique
5. Monter le système de gestion de fichier virtualisé sur les nœuds devant lire et/ou écrire dans le volume logique.

Les trois premières opérations sont de la responsabilité du fournisseur de service tandis que les deux dernières sont mises en œuvre par le client.

2.2 Architecture logicielle de CloViS

L'intergiciel CloViS a vocation à être installé sur des serveurs physiques ou virtuels fonctionnant sous Linux. Cette contrainte permet de s'abstraire des contraintes matérielles spécifiques, tant au niveau des disques que des interfaces réseau par exemple. Tous les composants de CloViS ont la même architecture : une partie serveur qui écoute les messages reçus en utilisant le service de communication et les gère, tandis que qu'une partie client interagit avec les autres composants (en utilisant leur partie serveur respective). La gestion des messages est réalisée au moyen de pool de threads afin de garder une architecture non bloquante. CloViS s'articule autour de quatre composants logiciels principaux :

- ☞ CAM (***C***lovis ***A***dministration and ***M***onitoring) : les outils d'administration et de monitoring qui permettent la gestion de l'intergiciel ;
- ☞ VRT (***V***i***RT***ualizer) : le virtualiseur qui est en charge de l'abstraction des ressources de stockage physiques en ressources de stockage virtuelles ;
- ☞ CCCC (***C***lovis ***C***oncurrency and ***C***oherency ***C***omponent) : un composant de gestion de la concurrence et de la cohérence des données ;
- ☞ CAL (***C***lient ***A***ccess ***L***ayer) : Une couche d'accès aux données. Cette couche peut être soit un système de gestion de fichiers compatible POSIX (nommé ClovisFS), soit un composant d'accès de type bloc nommé CBAL(***C****loViS* ***B****lock* ***A****ccess* ***L****ayer*), soit un composant d'accès sous forme d'une API RESTful.

A ces quatre composants de base il convient d'ajouter une infrastructure logicielle commune pour l'ensemble de ces composants (CAM, VRT, CCCC, CAL) avec des services associés.

La figure 2.2 représente l'architecture logicielle de CloViS. Les différentes applications utilisant CloViS vont procéder à des opérations de lecture et d'écriture. Ces opérations seront traitées par la couche d'accès aux données (*CAL*), qui contactera le VRT pour accéder aux objets de stockage qui correspondent, par exemple, aux fichiers cherchés. La couche d'accès aux données, par le biais du CCCC, va aussi gérer la concurrence des accès. Le VRT, pour sa part, est le responsable de la virtualisation ; c'est donc lui qui s'occupera de l'agrégation des ressources de stockage physique distribuées sur les différents nœuds dans une seule ressource de stockage logique, fournie à la couche d'accès aux données. Le VRT s'appuie aussi sur le composant CCCC pour le choix des ressources de stockages utilisées ou la sélection des ressources de stockages où sont stockées les données. Le cycle de vie de la virtualisation, comme le partage des ressources, la gestion de diverses politiques de placement et de migration des données, font appel au composant d'administration et de monitoring de CloViS, CAM. Il est situé au cœur de CloViS et est utilisé par tous les autres composants et par l'administrateur de l'infrastructure. Son rôle principal est d'exploiter les données collectées par son module de monitoring pour aider à améliorer les performances d'accès à ces données par le biais du module d'administration.

Tous les composants de CloViS partagent la même architecture logicielle. Ils sont constitués de deux parties. D'une part, une partie client qui interagit avec les autres

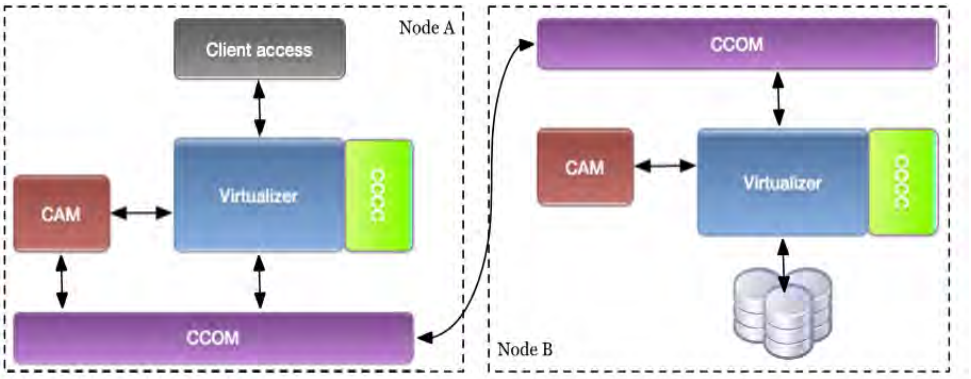


FIGURE 2.2 – Interaction des composants de CloViS

composants en leur envoyant des messages et d'autre part, une partie serveur qui reçoit les messages envoyés par les autres composants et les traite. Le traitement des messages par la partie serveur est pris en charge par un pool de thread commun à CloViS, permettant de tirer parti des performances des processeurs actuels.
Les messages échangés peuvent être de deux types :

- Des échanges **point-à-point**. Dans ce cas l'expéditeur indique l'identité précise du récipiendaire ;
- Des échanges **1 vers n** (messages multicast). Dans ce cas l'expéditeur précise le groupe des récipiendaires. Il appartient à chaque composant de s'inscrire au(x) groupe(s) susceptible(s) de le concerner.

Tous ces composants sont implémentés sous la forme d'une structure logicielle *component_t* au sein de CloViS. Ce type permet la définition d'une partie privée au composant (champ private) qui sert à stocker les informations spécifiques nécessaires aux fonctionnalités particulières de chaque composant. La partie commune contient les informations de base nécessaires au fonctionnement de cette infrastructure logicielle. On y trouve principalement :

- Un identifiant de serveur et de client qui permettent l'acheminement des messages point-à-point dans CloViS ;
- Un identifiant de journalisation pour les informations de trace et les messages d'erreur ;
- Les pointeurs vers trois fonctions de traitement implémentant la partie serveur du composant :
  - ☞ La fonction de traitement des messages simples ;
  - ☞ La fonction de traitement des messages multicast ;
  - ☞ La fonction de traitement des évènements périodiques.

Le service de communication est l'un des principaux services de l'infrastructure logicielle commune des composants de CloViS. Nommé CCOM (Clovis Communication), il permet l'échange de messages point-à-point et multicast entre composants. L'initialisation du service est masquée au développeur puisqu'elle s'effectue au sein de la partie commune de l'infrastructure des composants. Plusieurs fonctionnalités sont offertes

telles que l'envoi d'un message vers un destinataire, la réponse à un envoi précédant, le transfert d'un message à un nouveau destinataire, l'inscription à un groupe de diffusion de messages, l'envoi d'un message à un groupe de diffusion, etc. Les messages point-à-point peuvent être échangés entre composants situés sur des nœuds différents aussi bien qu'entre composants présents sur un même nœud. Dans le premier cas, un échange de messages par TCP est utilisé. Dans le second cas, tout dépend si les deux composants appartiennent à la même instance de CloViS ou non. Si les deux composants appartiennent à la même instance (cas le plus fréquent), un transfert des informations par une zone mémoire d'échange est utilisé. Dans le cas où les composants appartiennent à deux instances différentes (ce qui est le cas lorsque des commandes d'administration ou de monitoring sont exécutées depuis une console) un échange de message via TCP est mis en œuvre.

2.2.1 La gestion de la cohérence et de la concurrence (CCCC)

Dans un environnement de stockage distribué, un *modèle de cohérence* (aussi appelé *protocole de réplication*) est un ensemble des règles permettant de garantir la cohérence des répliques. Un *modèle de concurrence* est un ensemble des règles dictant l'ordre dans lequel les processus accèdent respectivement à leur section critique [75]. Le choix de séparer les notions de concurrence et de cohérence a été motivé par fait que les données répliquées ne sont pas les seuls objets partagés par les nœuds de l'intergiciel : il y a aussi les métadonnées de la couche d'accès aux données ainsi que les métadonnées propres à chaque composant [75]. Le composant CCCC (**C**loViS **C**oncurrency and **C**oherency **C**omponent) permet de maintenir la cohérence et la consistance des données. De fait, comme les métadonnées de CloViS sont stockées dans des volumes logiques gérés par CloViS lui-même, un tel service est requis indépendamment de la couche d'accès aux données cliente. Le composant CCCC utilise un algorithme totalement distribué, sûr et performant pour la gestion des verrous. Il en existe deux types dans CCCC :

- **READ** : verrou en lecture (partagé).
- **WRITE** : verrou en écriture (exclusif).

Chaque demande de verrou peut être associée à trois fonctions d'appel en retour. La première permet d'être prévenu lorsqu'un verrou non exclusif (READ) est requis sur une donnée déjà verrouillée avec un verrou identique. Cette fonction permet de transférer au demandeur la donnée éventuellement modifiée (gestion de cache). La deuxième fonction est utilisée lorsque un verrou est demandé sur une donnée déjà verrouillée avec un verrou incompatible. Cela permet de relâcher le niveau de verrou sur le nœud possédant le verrou le cas échéant. La troisième fonction permet d'être prévenu lorsque un verrou est demandé sur une donnée pour laquelle un verrou a précédemment été obtenu puis relâché. La gestion de la cohérence dépend du type de distribution des données mise en œuvre. Dans le contexte de nos travaux, elle sera assurée par des protocoles à base de quorums spécialement adaptés pour gérer les réplications par des codes correcteurs et schéma de seuil.

2.2.2 Le système d'administration et de monitoring (CAM)

Le composant CAM (**C**loViS **A**dministration and **M**onitoring) est le composant d'administration et de monitoring de CloViS. Il a la charge de la gestion de l'infrastructure de CloViS. Il est distribué sur l'ensemble des nœuds. Il est formé de deux modules : l'un d'administration et l'autre de monitoring. Le module administration maintient l'ensemble des métadonnées liées à CloViS : organisations virtuelles, volumes logiques, conteneurs et ressources de stockages. Par exemple, il est possible d'inscrire une nouvelle ressource de stockage qui pourra dès lors être utilisée pour la virtualisation du stockage. Les tâches d'administration de CloViS sont accessibles par ce module qui propage les actions vers les autres composants de CloViS distribués au sein de l'infrastructure. Les opérations courantes (créations, modifications et suppressions) sont propagées à l'ensemble des nœuds concernés. Il en va de même pour les opérations de montage des accès clients (système de fichiers ou accès blocs). Le module monitoring collecte, à intervalle régulier, l'état du système au niveau de chaque nœud pour permettre le maintien de la qualité de service requise sur chaque conteneur. Cet état du système concerne des informations globales (nœud actif ou non), des informations liées aux performances (bande passante et latence réseau, charge CPU et mémoire) ou des informations statistiques (données SMART des disques).
L'architecture adoptée par CAM lui permet d'une part de propager les informations collectées, et d'autre part d'intervenir, si nécessaire, pour améliorer la gestion du stockage et l'accès aux données en s'appuyant sur ces informations collectées. La communication entre les composants d'administration et les composants de monitoring est faite à travers CCOM. Afin d'établir une communication entre un utilisateur humain et CloViS, une interface en ligne de commande est disponible.

2.2.3 La virtualisation des ressources de stockage (VRT)

Le virtualiseur est un composant central dans l'architecture de CloViS. Il procède à l'agrégation des ressources de stockage physiques en volumes logiques. Il masque la complexité de l'architecture physique de stockage sous-jacente et offre une interface unifiée de type bloc pour le stockage des données. La couche d'accès client peut donc utiliser ces volumes logiques comme s'il s'agissait de disques locaux. Afin d'implémenter les différentes qualités de services requises au niveau du stockage, on a recours à des librairies à liaison dynamiques enfichables à chaud. Ces librairies de QoS sont utilisées par le virtualiseur pour déterminer le traitement à effectuer sur les données et la répartition des données sur les ressources de stockage physiques. Chaque librairie de QOS implémente une qualité de service spécifique (intégrité des données, confidentialité des données, etc.). Par exemple, un algorithme CRUSH peut être implémenté dans telle librairie permettant une optimisation des placements de données sur les ressources de stockage physiques.

Le virtualiseur est utilisé par la couche d'accès client pour les opérations de lecture, écriture ou suppression des blocs. Il implémente, pour cela, les opérations *vrt_read_block*, *vrt_write_block* et *vrt_delete_block*. Toutes ces opérations étant dédiées à la manipulation

de blocs, un identifiant global de bloc est requis pour chacune de ces opérations, ainsi qu'un *buffer* le cas échéant. Un identifiant global de bloc est représenté par les identifiants d'organisation virtuelle, de volume logique, de conteneur, d'inode et de bloc. Lorsqu'une telle demande est reçue, le virtualiseur utilise les identifiants d'organisation virtuelle, de volume logique et de conteneur pour retrouver le conteneur cible. A ce dernier est associé une qualité de service, constituée de deux champs :

- ☞ Un identifiant de librairie qui permet de sélectionner la librairie à utiliser parmi la liste de librairies disponibles ;
- ☞ Un identifiant interne, spécifique à la librairie, qui permet à cette dernière de déterminer comment implémenter les opérations requises.

Une fois la librairie sélectionnée, le virtualiseur invoque les opérations de lecture, écriture ou suppression de bloc en fournissant comme paramètre l'identifiant de bloc, le *buffer* si nécessaire, ainsi que quelques paramètres supplémentaires :

- ⇨ l'identifiant interne de librairie ;
- ⇨ la liste des ressources de stockage utilisées par le conteneur.

Grâce à ces paramètres, la librairie sélectionnée peut appliquer les traitements requis sur le bloc de données, et décider comment le distribuer sur les ressources de stockages disponibles pour le conteneur. Elle utilise alors une interface de programmation de bas-niveau du virtualiseur pour procéder à l'enregistrement sur les ressources de stockage physiques (*vrt_io_read*, *vrt_io_write*, *vrt_io_delete*). Il convient de noter que deux versions de cette interface de programmation existent : une **synchrone** lorsqu'une cohérence stricte des données est requise et une **asynchrone**. La figure 2.3 décrit les interactions entre le virtualizer CloViS et la librairie de qualité de service lors d'une opération (écriture, lecture ou suppression) sur un block.

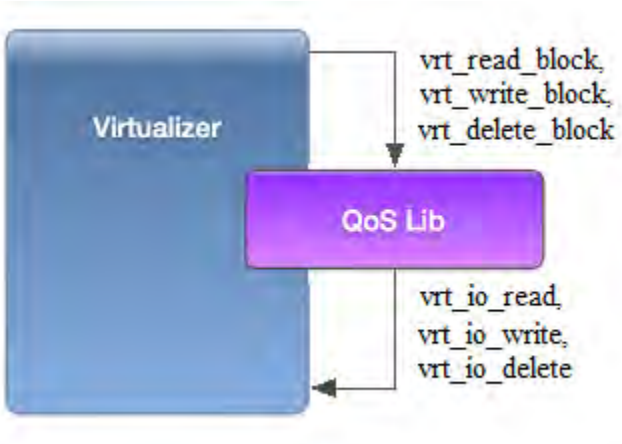


FIGURE 2.3 – Fonctionnement du virtualiseur de CloViS

Selon la qualité de service requise pour les données, la politique de placement privilégiera des ressources de stockage disposant d'une grande bande passante réseaux/disques, d'un certain degré de réplication ou bien d'un protocole de gestion de la cohérence

adapté à l'utilisation des données. Le choix d'un protocole de gestion de la cohérence est fait parmi un pool de protocoles fournis par la librairie du composant nommé CCCC de CloViS.

Des informations liées à la sémantique des données et leur placement effectif sur les ressources de stockage sont collectées au fur et à mesure durant l'exploitation et le traitement de ces données. Ces informations sont communiquées au VRT afin qu'une politique de placement adéquate soit mise en place au niveau du conteneur. Pendant l'exploitation des données, il se peut que la politique de placement choisie ne puisse plus satisfaire la qualité de service préconisée par l'application. Grâce au service CAM, le VRT peut être sollicité afin de lancer les actions permettant d'adapter la politique de placement. Ces actions peuvent être, par exemple, le changement de protocole de cohérence, la création d'une nouvelle réplique, etc.

2.2.4 La couche d'accès aux données

Plusieurs types de composants d'accès client sont fournis au niveau de CloViS, à savoir un système de gestion de fichiers compatible POSIX, une librairie d'accès par blocs et une API RESTful. Jusqu'à présent, deux de ces trois types de composants d'accès client sont implémentés et disponibles au sein de CloViS : un accès via un système de fichiers distribué compatible POSIX (basé sur FUSE) et un accès en mode bloc en utilisant le composant CBAL (qui exploite une zone mémoire partagée pour optimiser les performances d'accès). Cette dernière a vocation à servir à l'implémentation de pilotes blocs pour des virtualiseurs tels que QEMU. Mais un accès via une API RESTful peut être implémenté pour avoir un choix complet de modalités pour l'accès aux données.

Le système de gestion de fichiers de CloViS s'appelle ClovisFS (**CloViS F**ile**S**ystem). C'est un système de gestion de fichiers compatible POSIX basé sur FUSE [3]. Il permet l'accès à tout volume logique de n'importe quel conteneur, et prend en charge le changement de qualité de service sur les répertoires et les fichiers. ClovisFS permet de fournir aux utilisateurs du nuage (qu'il s'agisse d'utilisateurs physiques ou d'applications) une interface de type fichier pour l'accès aux données. Il assure les fonctionnalités de base d'un système de fichiers classique dans le contexte d'un environnement distribué à large échelle. Il est représenté par un module client, installé avec fuse, permettant d'accéder aux fichiers par le biais de l'interface POSIX. Le choix de la librairie fuse a été fait afin de permettre une intégration rapide au noyau linux tout en développant ClovisFS en espace utilisateur. En effet, traditionnellement, pour pouvoir implémenter et monter un système de fichiers, il faut soit être administrateur (le *root*), soit disposer d'un point de montage dans le fichier */etc/fstab*. FUSE est chargé comme un module pour le kernel et l'appartenance au groupe *fuse* (*/etc/group*) accorde la possibilité de montage. Ainsi, les développeurs peuvent développer leurs propres systèmes de fichiers sans toucher au kernel, et les utilisateurs peuvent accéder au système de fichiers à partir d'un point de montage sur sa machine sans avoir de privilèges particuliers. FUSE offre deux interfaces différentes pour développer un système de fichiers en espace utilisateur : une interface "*high-level*" et une interface "*low-level*". Nous avons choisi l'interface "*low-level*" parce

qu'elle permet d'avoir plus de contrôle sur les opérations du système de fichiers au prix d'une complexité d'implémentation plus importante. En effet, l'interface "*high-level*" est basée sur les chemins des objets et met en cache automatiquement les métadonnées dans le kernel alors que l'interface "*low-level*" est basée sur le numéro d'i-nœud. La conséquence est que le système de fichiers en espace utilisateur doit d'une part gérer les numéros d'i-nœud et d'autre part effectuer les translations de chemins d'accès (ainsi que la gestion des caches). ClovisFS utilise les librairies de QoS par le biais du virtualiseur afin d'effectuer le stockage distribué des données sur les différents nœuds avec une certaine qualité de service. Ces librairies utilisent le composant CCCC de CloViS afin, entre autre, d'assurer la gestion de la concurrence des métadonnées des fichiers.

ClovisFS assure la gestion des métadonnées et l'accès aux données sans faire de distinction entre données et métadonnées. Elles sont toutes considérées comme des objets du système de fichiers. Par contre, les données peuvent avoir une qualité de service différente de celle de leurs métadonnées. Avec ClovisFS, un fichier est découpé sous la forme d'objets comprenant un objet de type métadonnée et un ou plusieurs objets de type donnée. Ces objets sont identifiés par le quadruplet $< vo\_id, lv\_id, can\_id, ino\_id >$ qui est composé respectivement par l'identité de l'organisation virtuelle, l'identité du volume logique, l'identité du conteneur et le numéro d'i-nœud. La modification de qualité de service d'un objet implique la modification de l'identité de son conteneur qui se fait en utilisant les attributs étendus sous forme des paires $< clé, valeur >$. La lecture et l'écriture de ces attributs étendus se font respectivement via les appels systèmes *getxattr* et *setxattr* de l'API POSIX.

2.3 Synthèse sur le projet CloViS et problématique

CloViS est un intergiciel dédié au stockage dans un environnement distribué de type cloud. Il offre aux utilisateurs un accès homogène et transparent à des ressources de stockage hétérogènes et distribuées. Pour des raisons de performance, la structuration du stockage est prise en charge par les différents composants de CloViS qui doivent être déployés sur les différents nœuds participant à l'infrastructure, c'est-à-dire sur les nœuds qui partagent des ressources de stockage physiques. Cette architecture permet de décentraliser la gestion des services partout sur l'infrastructure, évitant ainsi les éventuels goulots d'étranglement induits autrement par une gestion centralisée des services.

L'utilisation d'un système de fichiers distribué nécessite un protocole de concurrence (requis pour gérer les verrouillages des objets lors des opérations telles que les écritures) et un protocole de cohérence (pour gérer les versions des répliques de chaque objet pour garantir leur cohérence à tout moment). C'est le composant CCCC qui assure ces deux mécanismes au sein de CloViS. Le service de gestion de la concurrence a été traité dans le cadre de la thèse de Aurélien Ortiz [75]. Il a proposé un mécanisme efficace et très fiable de verrouillage totalement distribué [10]. Nos travaux, présentés dans ce manuscrit, traitent du deuxième mécanisme du composant CCCC : la gestion de la cohérence de répliques des objets répartis dans les différentes ressources de stockage de CloViS.

Plus précisément, nous allons dans un premier temps faire un état de l'art sur les différentes techniques de distribution des données pour pouvoir choisir celles qui sont adaptées au stockage dans les nuages et permettent d'améliorer la disponibilité des données. Nous nous concentrerons sur les méthodes autorisant la définition de plusieurs niveaux de services en terme de disponibilité des données. La solution naturelle pour améliorer la disponibilité et la fiabilité des données est la réplication totale, qui offre la meilleure disponibilité possible. Néanmoins, elle engendre d'autres problèmes, principalement liés à l'utilisation excessive des ressources système (ressources de stockage, bande passante, etc.). Par conséquent, face à l'explosion du volume des données, l'utilisation de cette technique ne peut constituer une solution viable dans le cadre du stockage dans les nuages. Dans le cadre de cette thèse, pour trouver un compromis entre disponibilité des données et utilisation des ressources système, nous allons nous intéresser à d'autres techniques de distribution des données telles que les codes correcteurs et les schémas à seuil. Mais ces nouvelles techniques engendrent également d'autres problèmes liés à la cohérence des données, plus précisément la gestion des répliques. En effet, la plupart des protocoles de cohérence des données - largement étudiés actuellement - sont basés sur la réplication totale, c'est à dire qu'ils considèrent une copie identique pour tous les nœuds. Ce présupposé n'est pas respecté avec ces nouvelles techniques, car les nœuds n'ont forcément pas des copies identiques.

Par conséquent, dans un second temps, nous proposerons puis comparerons les performances des protocoles de gestion de la cohérence de répliques des données adaptés aux techniques de distribution des données choisies. Nous proposons pour cela des protocoles de cohérence des données adaptés à ces différentes techniques de distribution des données et nous analysons ensuite ces protocoles pour mettre en exergue leurs avantages et inconvénients respectifs. In fine, le choix d'une technique de distribution des données et d'un protocole de cohérence associé sera basé sur des critères de performances, notamment la disponibilité des opérations d'entrées/sorties, l'utilisation des ressources système ou encore le nombre de messages échangés durant les opérations d'entrées/sorties.

Deuxième partie

Distribution et Cohérence des données

Chapitre 1

Technique usuelle de distribution des données

Sommaire

La haute disponibilité des données et la haute performances des accès aux données ont été traditionnellement gérées par des équipements spécialisés destinés à des serveurs à très hautes performances. Bien que performant, face à l'explosion du prix des supercalculateurs et celui des logiciels afférents, cette solution s'avère très coûteuse. Dans le même temps, les microprocesseurs devenaient de plus en plus rapide et de moins en moins onéreux. Avec la démocratisation des PC (ordinateurs personnels) et la multiplication des logiciels et de systèmes d'exploitation libres (notamment les systèmes d'exploitation Linux [99]) ont émergé les clusters (grappe d'ordinateurs). Une des premières réalisations concrète de cluster date de 1995 avec le projet Beowulf de la NASA. Ce cluster a été utilisé pour la simulation de phénomènes physique et l'acquisition des données pour la science de la terre et de l'espace [99]. L'utilisation de clusters permet de bénéficier de plate-formes fiables et très performantes à des prix significativement moins onéreux par rapport à une solution de calcul hautes performances traditionnelle. Par exemple, *"le rapport prix /performance d'une grappe de PC est de 3 à 10 fois inférieur à celui des supercalculateurs traditionnels"* [35]. Le cluster mutualise les ressources d'un groupe de serveurs indépendants afin de fonctionner comme un seul et même système très performant. Mais cette mutualisation des ressources nécessite, pour les ressources de stockage, une technique de distribution des données pour augmenter la disponibilité des données stockées dans le système.

La solution triviale pour améliorer la disponibilité et la fiabilité des données est la

technique de réplication totale [112, 18, 60]. Cette technique de distribution des données permet d'obtenir la meilleure disponibilité possible pour les données mais cela implique une utilisation excessive des ressources du système [71]. Elle augmente également les coûts du système de 40% à 60% [47]. Par conséquent, face à l'explosion du volume des données, l'utilisation de la technique de réplication totale ne peut plus constituer une solution viable [107]. Dans la littérature, plusieurs travaux proposent d'autres techniques pour pallier ce problème [45, 79, 85, 39].

1.1 RAID

Le RAID (initialement "***R**edundant **A**rray of **I**nexpensive **D**isks*", c'est-à-dire regroupement redondant de disques peu onéreux) a été défini par l'Université de Berkeley [78, 79]. Actuellement, on considère plus adapté l'acronyme "***R**edundant **A**rray of **I**ndependant **D**isks*". Le RAID est une technique de distribution des données sur plusieurs disques durs afin d'améliorer les performances et/ou la tolérance aux pannes de l'infrastructure [77, 41, 92, 62]. Il existe sept niveaux standard de RAID (numérotées de 0 à 6) mais plusieurs types d'architectures sont possibles car ces sept niveaux standard peuvent se combiner entre eux.

Le RAID 0 (ou *entrelacement de disques*) subdivise les données en plusieurs morceaux de même taille et les stocke en les répartissant de façon équitable sur l'ensemble des disques de la grappe. Ce schéma offre la meilleure performance en écriture par rapport aux autres architectures de RAID car il n'a pas d'information redondante qui nécessite une mise à jour lors d'une opération d'écriture. Cette solution est largement utilisée dans des environnements de calcul intensif où la performance et la capacité plutôt que la tolérance aux pannes sont les principales préoccupations [32]. Avec le RAID 0, il est recommandé d'utiliser des disques de taille identique car sinon le disque de plus grande capacité ne sera pas pleinement exploité.

Le RAID 1 (ou *disques en miroir*) stocke les mêmes données sur l'ensemble de disques de la grappe. Ainsi, lors d'une opération de lecture, les données sont lues sur le disque qui optimise le délai et/ou l'utilisation des ressources [31]. Si la grappe utilise n disques alors ce schéma peut tolérer une défaillance jusqu'à $n-1$ disques. Cette solution est souvent utilisée dans les applications de base de données où la disponibilité et le taux de transactions sont plus importants que l'efficacité du stockage [49].

Le RAID 2 est désormais obsolète car cette technique utilise un contrôle d'erreur par code de Hamming (ECC : ***E**rror **C**orrection **C**ode*). Or ce dernier est désormais directement intégré dans les contrôleurs de disques durs. Ce schéma combine la méthode du RAID 0 et la technique de parité en écrivant sur des disques distincts les bits de contrôle ECC. Le RAID 2 offre un bon niveau de sécurité mais de mauvaises performances.

Le RAID 3 et RAID 4 utilisent le même principe, la seule différence se situant au niveau de la valeur du facteur d'entrelacement. Le RAID 3 travaille par octets alors que le RAID 4 travaille par blocs. Le fait de travailler par blocs permet au RAID 4

d'offrir des performances nettement supérieures par rapport au RAID 3. C'est pourquoi le RAID 3 tend à disparaître. Ces techniques utilisent n disques (avec $n \geq 3$) où les $n-1$ d'entre eux stockent les données tandis que le dernier disque stocke la parité. Ces schémas supportent la défaillance d'un seul disque, quel qu'il soit (y compris le disque de parité). Si le disque de parité tombe en panne, les $n-1$ permettent de reconstruire les données originales ; si l'un des disques de données tombe en panne, il est encore possible de reconstruire les données originales avec le contenu des disques de données restants et celui du disque de parité.

Le RAID 5 est similaire au RAID 4 mais la parité est répartie de façon circulaire sur l'ensemble des disques, éliminant ainsi le problème de goulot d'étranglement observé au niveau du disque de parité en RAID 4 et RAID 3. Ce schéma répartie également le travail sur l'ensemble des disques qui permet d'éviter l'usure prématurée de l'un des disques comme c'est souvent le cas avec le disque de parité en RAID 3 et RAID 4. Les données et la parité sont réparties sur tous les disques selon l'algorithme d'ordonnancement *round-robin* [25, 116]. Le RAID 5 offre, lors des opérations de lecture, des performances très proches de celles obtenues en RAID 0 tout en assurant une tolérance aux pannes plus élevée que ce dernier. Comme en RAID 3 et RAID 4, ce schéma supporte la défaillance de n'importe quel disque. Par ailleurs, il représente un surcoût faible en terme d'espace de stockage occupé (au maximum 33.33% de l'espace total utilisé). En effet, pour un système de n disques de capacité identiques, les $n-1$ disques sont utilisés pour stocker les données ($n \geq 3$). Ainsi, le surcoût est égal à l'espace de stockage total divisé par le nombre total de disques (soit $\frac{100}{n}\%$). Cependant, l'un des inconvénients majeurs de ce schéma est la pénalité de l'opération d'écriture car tous les disques sont requis pour pouvoir valider cette dernière.

Le RAID 6 est une évolution du RAID 5 en utilisant deux blocs de parité au lieu d'un. Cela permet au système de tolérer la défaillance simultanée de deux disques [19]. Par contre, cette augmentation de la tolérance aux pannes exige quelques concessions sur l'espace de stockage utilisé. Le surcoût en terme d'espace de stockage utilisé est doublé par rapport à celui du RAID 5. En effet, pour un système de n disques de capacité identiques ($n \geq 4$), seuls $n-2$ disques sont utilisés pour stocker les données. Ainsi, le RAID 6 présente un surcoût de $\frac{200}{n}\%$. Le temps de reconstruction des données en cas de défaillance simultanée de deux disques et celui de l'opération d'écriture sont assez long avec ce schéma à cause des calculs de redondance complexes.

A ces sept niveaux standards viennent s'ajouter les niveaux combinés et les niveaux spéciaux. Les niveaux de RAID combinés sont les résultats de la combinaison de deux niveaux de RAID classiques entre eux. Ces combinaisons offrent un large choix d'alternatives permettant d'obtenir un rapport performance/sécurité différent de celui obtenu avec les niveaux standards. Plusieurs combinaisons sont possibles. Par exemple, avec quatre disques, on peut faire deux RAID 1. La formation d'une pile RAID 0 à partir de ces deux piles RAID 1 donne naissance à un niveau de RAID combiné appelé RAID 10 (ou encore RAID 1+0), c'est-à-dire les éléments constitutifs du RAID 0 sont les deux RAID 1. Le RAID 10 présente une fiabilité assez élevée car il faut que tous les éléments d'une grappe (dans ce cas un RAID 1) soient défectueux pour entrainer une perte de données au niveau du système. Par ailleurs, en cas de défaillance d'un disque, la recons-

truction est assez rapide dans ce schéma car elle n'utilise que les disques d'une seule grappe et non la totalité. Quant aux niveaux de RAID spéciaux, ils sont les résultats de quelques améliorations effectuées sur les niveaux de RAID standards afin d'obtenir un critère de performance donné. Voici quelques exemples non exhaustifs :

- ☞ **RAID TP** (***T**riple **P**arity **RAID** technology*) : c'est une évolution de RAID 6 en augmentant à trois le nombre de blocs de parité, ce qui permet au système de tolérer la défaillance simultanée de trois disques ;
- ☞ **RAID DP** (***D**ual **P**arity*) : Ce type de RAID, souvent utilisé sur les serveurs de type NAS, est similaire au RAID 6 sauf que tous les blocs de parité sont stockés sur le même disque ;
- ☞ **RAID 5EE** : c'est une évolution de RAID 5 qui offre une meilleure performance en lecture grâce à l'utilisation d'un disque de plus, les données étant réparties sur les disques de secours pour une optimisation de l'accès en entrée/sortie.

1.2 Codes correcteurs

La théorie des codes correcteurs a été initialement abordée dans le contexte des communications pour corriger les erreurs de transmission d'une information [23, 82, 68]. Avec l'apparition des petits disques peu onéreux, les codes correcteurs ne se limitent plus aux communications classiques mais également aux support de stockage. Cela permet d'obtenir de nouvelles techniques de distribution des données dans un système constitué de plusieurs nœuds de stockage afin d'optimiser la bande passante et de résister à un certain nombre de pannes de nœuds de stockage. Dans ce contexte, les données sont subdivisées en plusieurs fragments, puis ces derniers sont encodés avec un certain niveau de redondances grâce à un *code correcteur* et ils sont stockés de manière disséminée à différents niveaux de la hiérarchie de (disques, nœuds de stockage, centres de données, etc.). Ces codes correcteurs sont utilisés dans les systèmes de stockage distribués afin d'augmenter la robustesse des données contre les défaillances éventuelles de certains serveurs [66]. Un code correcteur (n, k) encode les données (qui sont représentées comme k blocs ou fragments) en un mot de code de n blocs de façon à ce que n'importe quels k de ces n blocs permettent de reconstruire les données originales [61]. Le protocole génère ainsi $n - k$ blocs redondants en utilisant les k blocs de données [14]. Tant que le nombre de serveurs défaillants est en deçà du seuil de tolérance du code correcteur, les données originales peuvent-être récupérées à partir des blocs stockés dans les serveurs de stockage disponibles en utilisant le processus de décodage [67].

L'exemple suivant montre la robustesse d'un code correcteur $(4, 2)$ par rapport à la réplication totale [14]. Soient a et b les deux blocs de données. En utilisant ces deux blocs, le protocole génère les deux blocs redondants suivants : $a + b$ et $a - b$. Nous obtenons alors un mot de code composé de quatre blocs (a,b, $a+b$, $a-b$) où deux blocs choisis au hasard permettent de retrouver les blocs a et b (c'est-à-dire les blocs initiaux constituant les données originales). Par exemple, avec $a+b$ et b, nous pouvons retrouver le bloc a en soustrayant le bloc $a + b$ du bloc b. Par conséquent, nous obtenons bien

un $(4, 2)$ code correcteur, qui tolère la perte de deux blocs quelconques. Par contre, si nous dupliquons simplement les blocs a et b, on a un mot de code composé de quatre blocs : (a, b, a, b). Mais dans cette configuration, nous ne pourrons plus récupérer les données originales si les deux blocs a sont perdus. Cela montre la robustesse des codes correcteurs étant donné que ces deux techniques utilisent le même nombre de blocs (également de même taille).

Le *code Reed-Solomon* est l'un des codes correcteurs disposant d'une large gamme d'applications dans les communications numériques et le stockage. C'est un code correcteur dit "*parfait*" inventé par les mathématiciens *Irving S. Reed* et *Gustave Solomon*. Il fait partie d'une sous-famille des *codes BCH* (ces codes tiennent leur nom des initiales de leurs inventeurs : ***B**ose, Ray-**C**haudhuri et **H**ocquenghem*) [57]. Les *codes Reed-Solomon* sont construits et décodés en utilisant des arithmétiques dans un *corps fini* (noté **GF(q)** pour un corps fini de cardinal q). L'approche originale pour la construction de ces codes a été l'utilisation de polynômes pour encoder les données originales. Supposons que nous avons des données composées de k blocs, $\{b_0, b_1, \ldots, b_{k-2}, b_{k-1}\}$, appartenant chacun au corps fini **GF(q)**. Ces blocs sont utilisés pour construire un polynôme défini comme suit $P(x) = b_0 + b_1 x + \ldots + b_{k-2}x^{k-2} + b_{k-1}x^{k-1}$. Un mot de code c de *Reed-Solomon* est formé en évaluant le polynôme $P(x)$ à chacun de q éléments de **GF(q)**. Soit :

$$c = (c_0, c_1, \ldots, c_{q-1}) = \Big(P(0), P(\alpha), P(\alpha^2), \ldots, P(\alpha^{q-1})\Big) \tag{1.1}$$

où α est un *élément primitif* de **GF(q)**. Le nombre de blocs des données k est souvent appelé la *dimension* du code. Ce terme vient du fait que les mots de code de Reed-Solomon forment un espace vectoriel de dimension k sur **GF(q)**. Étant donné que chaque mot de code de Reed-Solomon est constitué de q éléments (voir équation (1.1)), on dit généralement que le code est de *longueur* $n = q$. C'est pourquoi un code Reed-Solomon est généralement noté $RS(n, k)$ (désigné par sa longueur n et sa dimension k). Chaque mot de code de Reed-Solomon défini dans le système d'équations (1.1) peut être exprimé sous la forme d'un système de q équations linéaires à k inconnus comme suit :

$$\left\{\begin{array}{rcl} P(0) & = & b_0 \\ P(\alpha) & = & b_0 + b_1\alpha + b_2\alpha^2 + \ldots + b_{k-1}\alpha^{k-1} \\ P(\alpha^2) & = & b_0 + b_1\alpha^2 + b_2\alpha^4 + \ldots + b_{k-1}\alpha^{2(k-1)} \\ \vdots & & \\ P(\alpha^{q-1}) & = & b_0 + b_1\alpha^{q-1} + b_2\alpha^{2(q-1)} + \ldots + b_{k-1}\alpha^{(k-1)(q-1)} \end{array}\right. \tag{1.2}$$

Cette équation peut être exprimée sous la forme matricielle comme suit :

$$M_1.\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{k-1} \end{pmatrix} = \begin{pmatrix} P(0) \\ P(\alpha) \\ P(\alpha^2) \\ \vdots \\ P(\alpha^{q-1}) \end{pmatrix} \tag{1.3}$$

Avec

$$M_1 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & \alpha & \alpha^2 & \dots & \alpha^{(k-1)} \\ 1 & \alpha^2 & \alpha^4 & \dots & \alpha^{2(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{(q-1)} & \alpha^{2(q-1)} & \dots & \alpha^{(q-1)(k-1)} \end{pmatrix}$$

Nous avons q équations pour k inconnus. En général, k équations sont suffisantes pour trouver les k inconnus. Si nous prenons par exemple, les k premières équations du système d'équations (1.2), nous avons :

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & \alpha & \alpha^2 & \dots & \alpha^{(k-1)} \\ 1 & \alpha^2 & \alpha^4 & \dots & \alpha^{2(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{(k-1)} & \alpha^{2(k-1)} & \dots & \alpha^{(k-1)(k-1)} \end{pmatrix} . \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{k-1} \end{pmatrix} = \begin{pmatrix} P(0) \\ P(\alpha) \\ P(\alpha^2) \\ \vdots \\ P(\alpha^{k-1}) \end{pmatrix} \tag{1.4}$$

Notons M_2 la matrice carrée suivante :

$$M_2 = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 1 & \alpha & \alpha^2 & \dots & \alpha^{(k-1)} \\ 1 & \alpha^2 & \alpha^4 & \dots & \alpha^{2(k-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{(k-1)} & \alpha^{2(k-1)} & \dots & \alpha^{(k-1)(k-1)} \end{pmatrix}$$

Alors l'équation (1.4) est équivalente à :

$$M_2 . \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{k-1} \end{pmatrix} = \begin{pmatrix} P(0) \\ P(\alpha) \\ P(\alpha^2) \\ \vdots \\ P(\alpha^{k-1}) \end{pmatrix} \tag{1.5}$$

La matrice M_2 est dite matrice de Vandermonde [103]. Comme α est un élément primitif de **GF(q)** alors $1, \alpha, \alpha^2, \dots, \alpha^{q-1}$ sont distincts (et $\alpha \neq 0$). Ainsi, $0, \alpha, \alpha^2, \dots, \alpha^{k-1}$ sont aussi distincts. Cela implique donc que la matrice de Vandermonde M_2 est inversible [103]. Par conséquent, l'équation (1.4) admet une solution unique pour les k blocs des données : $\{b_0, b_1, \dots, b_{k-2}, b_{k-1}\}$.
Soit $p \in \{2, 3, \dots, q\}$, alors la $p^{ième}$ ligne de la matrice M_1 est $\begin{pmatrix} 1 & \alpha^{(p-1)} & \alpha^{2(p-1)} & \dots & \alpha^{(p-1)(k-1)} \end{pmatrix}$ qui est égale à $\begin{pmatrix} 1 & (\alpha^{(p-1)}) & (\alpha^{(p-1)})^2 & \dots & (\alpha^{(p-1)})^{(k-1)} \end{pmatrix}$. Cette ligne correspond donc à une ligne d'une matrice de Vandermonde. Par conséquent, n'importe quel système d'équation formé par k équations du système d'équations (1.2) peut être transformé en :

$$M . \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{k-1} \end{pmatrix} = \begin{pmatrix} P(\lambda_0) \\ P(\lambda_1) \\ P(\lambda_2) \\ \vdots \\ P(\lambda_{k-1}) \end{pmatrix} \tag{1.6}$$

Où M est une matrice carrée de Vandermonde $k \times k$ et pour $i \in \{0, 1, 2, \ldots, k-1\}$, $\lambda_i \in \{0, \alpha, \alpha^2, \ldots, \alpha^{q-1}\}$, pour tout $i, j \in \{0, 1, 2, \ldots, k-1\}$, $i \neq j \Rightarrow \lambda_i \neq \lambda_j$. La matrice M est inversible (car les $0, \alpha, \alpha^2, \ldots, \alpha^{q-1}$ sont distincts) [103]. Cela implique donc que le système système d'équations (1.6) admet une solution unique. On obtient :

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{k-1} \end{pmatrix} = M^{-1}. \begin{pmatrix} P(\lambda_0) \\ P(\lambda_1) \\ P(\lambda_2) \\ \vdots \\ P(\lambda_{k-1}) \end{pmatrix} \tag{1.7}$$

Par conséquent, n'importe quelle combinaison de k lignes du système d'équations (1.2) permet de retrouver les blocs des données $\{b_0, b_1, \ldots, b_{k-2}, b_{k-1}\}$. Autrement dit, la connaissance de n'importe quels k éléments parmi $\{P(0), P(\alpha), P(\alpha^2), \ldots, P(\alpha^{q-1})\}$ permet de retrouver les données originales.

Des codes correcteurs ont été récemment implémentés dans certains systèmes de stockage distribués. Ils utilisent moins d'espace de stockage par rapport à la stratégie classique de trois répliques : par exemple, un code correcteur de Reed-Solomon avec quatre blocs redondants a seulement 40% de surcharge au niveau du stockage [88]. Cependant, presque toutes les études précédentes ont mis l'accent sur les effets des mises à jour, les entrées/sorties et l'optimisation de l'utilisation du réseau lors d'une défaillance d'un nœud. Pour atteindre ces objectifs, les données sont souvent considérées comme immuables (ou presque) : les mises à jour sont effectuées en écrivant de nouvelles données et les anciens blocs sont supprimés par un processus de ramasse-miettes en arrière-plan. En outre, soit la cohérence de données n'est pas discutée [88, 56], soit les protocoles utilisés ne sont pas stricts. Par exemple, une "*cohérence raisonnablement forte*" peut être utilisée [14], où une lecture concurrente avec des opérations d'écriture retourne une valeur non déterministe. A notre connaissance, la seule implémentation d'un protocole de cohérence forte bloquant les écritures lors d'une défaillance pour garantir la cohérence, est évoquée dans [26]. Cependant, ces solutions ne sont pas adéquates pour certaines applications. Par exemple, les disques de données de machines virtuelles des utilisateurs dans un contexte de virtualisation nécessitent une cohérence forte des fichiers sous-jacents afin de garantir la bonne exécution des applications. De même, le stockage de bases de données ne peut se contenter d'une cohérence relâchée lorsque plusieurs moteurs de requêtes accèdent simultanément aux données. Un protocole doit donc être implémenté afin d'obtenir une réelle cohérence forte entre les blocs dans un contexte de code correcteur. Plusieurs travaux ont été publiés sur la cohérence forte dans le contexte de réplication totale. Nous allons adapter certains de ces travaux dans le contexte de codes correcteurs.

1.3 Schéma à seuil

Considérons le problème suivant (cité dans [89]) : *"Onze scientifiques travaillent sur un projet. Ils veulent enfermer les documents dans une armoire de sorte que cette*

dernière peut être ouverte uniquement si au moins six scientifiques sont présents. Quel est le plus petit nombre de verrous nécessaire ?. Quel est le plus petit nombre de clés pour les verrous que chaque scientifique doit détenir ?"
Pour cela, la solution minimale utilise $\binom{11}{6} = 462$ verrous et $\binom{10}{5} = 252$ clés par scientifiques. Ces nombres ne sont pas tenables et ils croissent de façon exponentielle avec le nombre des scientifiques. Pour éviter ce genre de problème, Adi Shamir propose un schéma qu'il appelle *shéma à seuil* (n, k) en se basant sur l'interpolation polynomiale [89]. Il généralise le problème en considérant le secret comme une donnée D. Il divise D en n morceaux $\{D_1, D_2, \ldots, D_n\}$ de telle sorte que :

- ☞ la connaissance de k morceaux ou plus permet de reconstruire D ;
- ☞ la connaissance de $k-1$ morceaux ou moins rend D complètement indéterminée.

D'autres travaux proposent une généralisation du schéma à seuil ("*General Threshold Scheme*")[45, 113]. Un schéma à seuil général $p-m-n$ génère n blocs à partir des données (généralement par subdivision puis ajout d'informations de redondance) de façon à ce que n'importe quels m blocs permettent de reconstruire les données originales et moins de p blocs ne révèlent absolument aucune information sur les données originales. Les trois paramètres p, m et n doivent satisfaire la condition suivante : $1 \leq p \leq m \leq n$. Ces trois paramètres donnent aux utilisateurs un large choix selon leurs besoins (disponibilité, confidentialité, espace de stockage, ...). Un exemple de schéma à seuil spécifique est la réplication totale sur N nœuds. C'est un schéma à seuil $1-1-N$, où chaque bloc révèle des informations sur les données originales ($p = 1$), un seul bloc est requis pour reconstruire les données originales ($m = 1$) et il existe au total N blocs à disposition pour reconstruire les données originales ($n = N$). Le tableau 1.1 suivant résume les différentes possibilités du protocole selon le choix de valeur de chaque paramètre.

Schéma à seuil	**Description**
$1-1-n$	Réplication totale
$1-n-n$	Décimation (Striping)
$n-n-n$	Splitting (XORing)
$1-m-n$	Dissémination de l'information (Information Dispersal) avec $m < n$
$m-m-n$	Partage secret (Secret Sharing) avec $1 < m < n$
$p-m-n$	Ramp Scheme avec $1 < p < m \leq n$

TABLE 1.1 – **Liste de types de schéma à seuil (source : [45])**

Le schéma à seuil général $p-m-n$ permet de trouver un compromis entre la confidentialité, la disponibilité et l'espace utilisé dans un système de stockage. Quand la valeur du paramètre n augmente, la disponibilité des données augmente également mais cela demande davantage d'espace de stockage et la confidentialité diminue. L'espace de stockage occupé et la disponibilité des données sont inversement proportionnels au paramètre m (la taille d'un bloc étant proportionnelle à $\frac{1}{1+m-p}$). Par ailleurs, quand la valeur de m croît, chaque bloc contient moins d'information. Cela peut augmenter le nombre de blocs nécessaires qu'un intrus doit capturer pour pouvoir reconstruire une

portion utile des données originales. Quant au paramètre p, il est proportionnel à la confidentialité de données et à l'espace de stockage utilisé. Les schémas à seuil peuvent être utilisés à la place de techniques cryptographiques pour garantir la confidentialité des données mais les deux techniques peuvent aussi être combinées [45]. Une extension des schémas à seuil permet de vérifier l'authenticité des blocs [45]. Dans un schéma à seuil avec cette extension, les blocs sont construits de telle manière qu'un utilisateur puisse détecter, avec une probabilité élevée, si des blocs ont été modifiés (ne sont pas authentiques) [100]. Cette technique permet de garantir l'intégrité des données.

Avec cette flexibilité, le choix d'un schéma à seuil le plus approprié pour une situation donnée n'est pas trivial. Ainsi, pour aider les utilisateurs, certains travaux proposent une sélection automatique d'un schéma à seuil le plus approprié à leurs besoins [113]. Ils peuvent également paramétrer les interactions avec les nœuds de stockage contenant les différents blocs de données. Même si seulement m blocs sont nécessaires pour reconstruire les données originales, un utilisateur peut en demander plus (c'est-à-dire il peut envoyer une requête de lecture d'un certain nombre de blocs compris entre m et n). Cela permet de pallier le problème de lenteur éventuel de certain nœuds de stockage.

1.4 Synthèse

Nous avons présenté dans ce chapitre trois techniques usuelles de distribution des données, à savoir le *RAID*, le *schéma à seuil* et les *codes correcteurs*. Ce sont des solutions alternatives à la réplication totale pour réduire les surcharges relatives au niveau du stockage. Dans ces travaux, parmi les trois solutions proposées, nous avons choisi d'utiliser les codes correcteurs comme technique de distribution des données. Le choix a été motivé par les raisons suivantes :

- les implémentations de l'algorithme RAID 6 sont souvent limitées à tolérer au maximum deux défaillances simultanées de nœuds, à cause des calculs de redondance complexes. Bien que le RAID 1 puisse supporter plus de deux défaillances simultanées, il présente le même problème de surcharges au niveau du stockage que la réplication totale ;
- le schéma à seuil peut supporter à la fois la tolérance aux fautes et la confidentialité des données, mais par souci d'évolutivité dans CloViS, nous voulons séparer ces deux mécanismes.

Par ailleurs, Weatherspoon H. et al. ont montré, dans [107], que les systèmes utilisant les codes correcteurs présentent un temps moyen de bon fonctionnement (MTTF : ***M****ean* ***T****ime* ***T****o* ***F****ailure*) plus long que ceux avec la réplication totale avec les mêmes exigences en terme de stockage et de bande passante.

Chapitre 2

Technique usuelle de maintien de la cohérence des données

Sommaire

2.1 Les modèles de cohérence

L'utilisation des techniques de distribution des données est indispensable dans le cadre du stockage dans un environnement distribué [16, 38]. Cependant, cela nécessite un contrôle rigoureux des répliques pour éviter des incohérences de ces dernières qui peuvent souvent entrainer la perte partielle, voire totale, des données. Dans un système de stockage distribué, il existe deux grandes catégories d'approche : une *approche optimiste* et une *approche pessimiste*. L'approche pessimiste ne donne pas accès à une donnée tant que toutes ses répliques n'ont pas été synchronisées. Une donnée doit ainsi être verrouillée avant de la modifier. Il n'y a donc pas de mise à jour concurrente, évitant ainsi les confits à priori. A contrario, l'approche optimiste consiste à autoriser l'accès à une réplique sans synchronisation à priori avec les autres répliques. Les mises à jour concurrentes sont ainsi autorisées, ce qui implique que des conflits sont possibles et nécessitent donc d'une phase de réconciliation à posteriori.

Dans la littérature, plusieurs modèles de cohérence ont été définis pour les systèmes distribués afin de trouver un modèle performant et adapté aux applications [96, 22, 115, 101]. Un modèle de cohérence se présente comme un contrat entre les processus et le système de stockage. Une classification a été proposée par Tanenbaum et Van Steen [96].

- **Modèles centrés sur les données** : ces modèles considèrent que les données sont partagées par plusieurs utilisateurs. Ils concernent donc les opérations de lecture et d'écriture dans des données partagées tels que les systèmes de fichiers distribués, les mémoires partagées, les bases de données partagées, etc. Ces modèles se focalisent sur le processus de synchronisation entre les différentes répliques et l'ordonnancement des différentes opérations.
  - ⇨ **Avec synchronisation permanente** : ces modèles garantissent que seul un état cohérent des données peut être vu. Cela implique donc qu'une mise à jour doit être immédiatement propagée vers toutes les répliques. Il existe plusieurs modèles de cohérences avec synchronisation permanente. La *cohérence stricte* qui impose que chaque opération de lecture retourne la dernière valeur écrite dans une donnée, la *cohérence séquentielle* qui a été définie par Lamport en 1979 garantie que tous les processus voient toutes les opérations dans le même ordre, la *cohérence causale* assure que toutes les opérations d'écritures qui sont reliées par une relation de causalité sont vues par tous les processus dans le même ordre mais les écritures qui ne sont pas causalement liées peuvent être vues dans un ordre différent par des processus différents, etc. ;
  - ⇨ **Avec synchronisation uniquement à des moment précis**, appelés, *points de synchronisation* par exemple lors de la prise ou du relâchement du verrou. Ces modèles de cohérence ont pour objectifs d'améliorer la performance, le passage à l'échelle et d'optimiser le nombre d'échanges réseau lors d'une opération. Mais cela a un prix : il n'y a plus de garantie d'avoir le même état pour toutes les répliques. Ainsi, une opération de lecture peut retourner une donnée non à jour. C'est le cas des modèles de cohérence faible, de cohérence au relâchement, de cohérence à l'entrée, etc.
- **Modèles centrés client** : Il existe plusieurs applications dans lesquelles la concurrence apparaît dans une forme restreinte. Par exemple, les systèmes de bases de données, le DNS, le WWW (***W**orld **W**ide **W**eb*). Ces exemples tolèrent un degré relativement élevé d'incohérence [96]. Pour ces types d'applications, si aucune mise à jour n'a eu lieu pendant une longue période, toutes les répliques deviendront progressivement cohérentes. Ce type de cohérence est appelé cohérence éventuelle (*eventual consistency*). La cohérence éventuelle exige donc que les mises à jour soient propagées sur toutes les répliques. La propagation des mises à jour est faite de manière asynchrone [114]. Elle permet d'éviter le goulet d'étranglement induit par la synchronisation, mais reste ponctuellement source d'erreurs [90]. Par exemple, pendant la période d'incohérence (le temps de propagation des mises à jour sur toutes les répliques), on ne peut pas garantir que chaque processus accède à la même version d'une donnée. Ainsi, des problèmes surviennent lorsque l'on accède aux différentes répliques pendant une période relativement

courte. Un exemple illustrant ce problème est détaillé dans [96], en considérant un utilisateur mobile accédant à une base de données distribuée. Plusieurs approches ont été proposées pour améliorer ce modèle de cohérence. Marc Shapiro et al. proposent une nouvelle approche en utilisant des types de données partagées basés sur des conditions formelles simples qui sont suffisantes pour garantir la cohérence éventuelle [90, 91]. Ils nomment ces types de données partagées CRDT (***C**onvergent or Commutative **R**eplicated **D**ata **T**ypes*) [91]. Valter Balegas et al. proposent une alternative à la cohérence éventuelle, nommée, cohérence explicite (*Explicit Consistency*) [22]. D'autres approches introduisent les modèles de cohérence centrés client [96]. Ces modèles se basent surtout sur la cohérence d'un client individuel et ne considèrent pas le fait que les données peuvent être partagées par plusieurs utilisateurs [96]. L'hypothèse est alors la suivante : le client accède à différentes répliques au fil du temps, mais cela doit être transparent. Les modèles centrés client garantissent que lorsqu'un client accède à une nouvelle réplique, cette dernière est mise à jour avec les données manipulées par ce client auparavant, qui peuvent éventuellement résider dans d'autres sites.

Jerzy B. et al. proposent une formalisation des quatre modèles centrés clients dans [21]. Ils adoptent les notations suivantes :

- les opérations dans des objets partagés effectuées par le client C_i sont ordonnées par une relation $\overset{C_i}{\rightarrow}$;
- un serveur S_j effectue les opérations dans un ordre représenté par $\overset{S_j}{\rightarrowtail}$;
- les écritures et lectures dans des objets seront notés respectivement par w et r ;
- une opération de lecture et d'écriture effectuées par le serveur S_j sera notée respectivement par $r|_{S_j}$ et $w|_{S_j}$;
- $RW(r)$: est un ensemble d'écritures qui ont influencé l'état actuel des objets vus par la lecture r ;

⇨ **Monotonic Reads (MR)** : un protocole de cohérence est dit *Monotonic read* si la condition suivante est satisfaite *"Si un processus lit la valeur d'une donnée x, alors n'importe quelle opération de lecture sur x par ce processus retournera toujours la même valeur ou une autre valeur plus récente"* [96]. Autrement dit, le *monotonic read* garantit que si le processus a lu la valeur de x à l'instant t, il ne verra jamais une ancienne version de x à une date ultérieure. Selon Jerzy B. et al., un MR est défini comme suit [21] :

$$\forall C_i \forall S_j \left[r1 \overset{C_i}{\rightarrow} r2|_{S_j} \Rightarrow \forall w_k \in RW(r1) : w_k \overset{S_j}{\rightarrowtail} r2 \right]$$

⇨ **Monotonic Writes (MW)** : un protocole de cohérence est dit *Monotonic write* si la condition suivante est satisfaite *"Une opération d'écriture par un processus d'une donnée x est terminée avant toute autre succession d'opérations d'écriture de x par le même processus"* [96]. En d'autres termes, une opération d'écriture sur une réplique de x n'est autorisée que si cette réplique a été mise à jour vis à vis d'une précédente opération d'écriture, ce qui peut

avoir eu lieu sur d'autres répliques de x. Jerzy B. et al. définissent le MW comme suit [21] :

$$\forall C_i \forall S_j \left[w1 \xrightarrow{C_i} w2|_{S_j} \Rightarrow w_1 \overset{S_j}{\rightarrowtail} w2 \right]$$

- ⇨ **Read Your Writes (RYW)** : ce protocole vérifie la condition suivante *"L'effet d'une opération d'écriture d'une donnée x par un processus sera toujours vu par une succession d'opérations de lecture sur le même processus"* [96]. Cela veut donc dire qu'une opération d'écriture doit être terminée avant une succession d'opérations de lecture par le même processus, peu importe où cette opération de lecture a lieu. Jerzy B. et al. définissent le RYW comme suit [21] :

$$\forall C_i \forall S_j \left[w \xrightarrow{C_i} r|_{S_j} \Rightarrow w \overset{S_j}{\rightarrowtail} r \right]$$

- ⇨ **Writes Follows Reads (WFR)** : ce protocole vérifie la condition suivante *"Une opération d'écriture d'une donnée x par un processus suivant une précédente opération de lecture sur x par le même processus est garantie d'avoir lieu sur la même ou une valeur plus récente de x qui a été lue"* [96]. Cela veut dire donc toute succession d'opérations d'écriture par un processus d'une donnée x sera effectuée sur une réplique à jour de x avec la valeur la plus récente lue par ce processus. Selon Jerzy B. et al., un WFR est défini comme suit [21] :

$$\forall C_i \forall S_j \left[r \xrightarrow{C_i} w|_{S_j} \Rightarrow \forall w_k \in RW(r) : w_k \overset{S_j}{\rightarrowtail} w \right]$$

L'intergiciel CloViS a pour ambition de couvrir les trois types de solutions clouds : le SaaS, le PaaS et l'IaaS. Par exemple, CloViS doit être capable d'assurer la gestion des machines virtuelles et notamment le stockage des images, des données utilisateurs, leur migration d'un nœud vers un autre, etc. La conception de CloViS lui permet d'implémenter plusieurs modèles de cohérence et de les affecter selon l'exigence de l'application, certaines nécessitant un modèle de cohérence forte. Par ailleurs, le fait d'utiliser des techniques de distributions de données telles que les codes correcteurs exige une cohérence stricte pour un certain nombre de répliques. En effet, lors d'une opération de lecture, un code correcteur (n, k) nécessite k répliques à jour pour reconstruire la donnée originale. Or, comme nous l'avons vu dans la section 1.2, les implémentations actuelles font l'impasse sur la gestion de la cohérence, préférant empêcher l'accès le temps de la reconstruction des données d'un serveur plutôt que de permettre un accès en mode dégradé. Dans ces travaux, nous nous intéressons à une implémentation du modèle de cohérence forte : *les protocoles basés sur les quorums* [98, 12, 70, 69, 80], que nous allons détailler dans la section suivante.

2.2 Les protocoles de cohérence

De nombreux travaux ont été publiés sur les techniques de maintien de la cohérence des données dans le contexte de la réplication totale [15, 98, 48, 64, 95]. L'approche de base garantissant une cohérence stricte des données est le protocole ROWA (***R**ead **O**ne **W**rite **A**ll*) [15]. Lors d'une mise à jour d'une réplique, toutes les autres répliques sont également mis à jour au cours d'une opération atomique. Autrement dit, la validation d'une opération d'écriture ou de mise à jour d'une donnée requiert la validation de celle de toutes les répliques de cette donnée. Ce type de protocole est adapté aux applications où les données sont essentiellement en lecture seule [16]. Cela est dû au fait qu'avec ROWA, une seule réplique suffit pour lire les données et fournit ainsi un niveau de disponibilité élevé aux opérations de lecture. Néanmoins, ce protocole a des inconvénients majeurs tels que la lourdeur et surtout la non tolérance aux fautes des opérations d'écriture et de mise à jour [84]. L'écriture ou la mise à jour doit être propagée sur tous les fragments répliqués et ne peut pas être effectuée en présence d'une panne de partitions réseau ou même d'une seule réplique. Ces inconvénients majeurs constituent un frein sur l'utilisation de ROWA dans des environnements large échelle.

Les protocoles à quorums ont été introduits pour surmonter ces problèmes [98, 12, 70, 69, 80]. Un protocole à quorums forme une structure logique des répliques en plusieurs sous-ensembles appelés *quorums*. Dans ce contexte, un quorum est l'ensemble minimum de répliques garantissant la cohérence des données lors d'une opération bien définie. Dans cette thèse, nous nous intéressons particulièrement aux opérations les plus classiques telles que les opérations de lecture, d'écriture et/ou de mise à jour. Mais d'autres travaux s'intéressent également aux systèmes à quorums de types abstraits utilisant d'autres opérations que ces opérations de base [52, 53]. Le protocole à quorums permet de trouver un compromis entre le nombre de répliques accédées lors de l'opération de lecture et celui de l'opération d'écriture, les deux paramètres étant antagonistes. Par ailleurs, il existe plusieurs types de protocoles à quorums. La première implémentation d'un protocole à quorums a concerné le *quorum majoritaire* [98]. Dans le but de réduire la taille des quorums, d'autres protocoles à quorums (dit "*structurés*" [42]) s'appuient sur une organisation logique des nœuds. Plusieurs topologies sont utilisées dans la littérature des systèmes à quorums, comme l'arbre [12, 13], le triangle [30], la grille [33], la hiérarchie [63, 64, 36], l'hypercube [44], le losange [43] ou encore le trapèze [20, 95]. Ces organisations sont seulement logiques et n'ont aucun lien avec les structures physiques réelles de l'infrastructure contenant les nœuds qui stockent les différentes répliques [12].

Par définition, un quorum d'écriture (Q_w) est l'ensemble minimum de répliques sur lesquelles doit être effectuée une opération d'écriture de façon atomique et garantissant la cohérence des répliques des données. $|Q_w|$ dénote la taille du quorum d'écriture Q_w, c'est à dire le nombre de répliques appartenant à ce quorum. De façon analogue, un quorum de lecture (Q_r) est l'ensemble minimum de répliques sur lesquelles doit être effectuée une opération de lecture de façon à garantir la cohérence des répliques des données. $|Q_r|$ dénote la taille du quorum de lecture Q_r. Pour assurer cette cohérence,

les quorums d'écriture et de lecture sont soumis à certaines contraintes [27, 73] :

- ☞ Un quorum d'écriture Q_w et un quorum de lecture Q_r doivent avoir au moins un élément en commun :
$$Q_w \cap Q_r \neq \emptyset \tag{2.1}$$
- ☞ Deux quorums d'écriture Q_{w1} et Q_{w2} doivent avoir au moins un élément en commun :
$$Q_{w1} \cap Q_{w2} \neq \emptyset \tag{2.2}$$

La contrainte (2.1) garantit qu'un quorum de lecture contient au moins une réplique mise à jour (i.e. une réplique qui a été modifiée lors de la dernière opération d'écriture valide) alors que la contrainte (2.2) permet d'assurer que deux opérations d'écriture successives interviennent au moins sur une même réplique, garantissant ainsi l'incrémentation du numéro de version des données à chaque opération d'écriture.

Dans la suite de ce chapitre, nous allons faire une présentation succincte des différents protocoles à quorums dans le contexte de réplication totale. Ce sont les protocoles que nous allons adapter au contexte de nouvelles distributions des données (voir chapitre 3).

Notations Dans la suite :

- ☞ t représente la disponibilité des nœuds, c'est-à-dire la probabilité qu'un nœud donné soit disponible dans le système ;
- ☞ P_{write} désigne la disponibilité en écriture qui est définit comme la probabilité qu'une opération d'écriture puisse être réalisée avec succès dans le système ;
- ☞ P_{read} la disponibilité en lecture qui est définit comme la probabilité qu'une opération de lecture puisse être réalisée avec succès depuis le système ;
- ☞ $\Phi(i, j)$ représente la probabilité qu'au moins i nœuds parmi les j soient disponibles.

Sans perte de généralité [95], on suppose que :

- ⇨ la disponibilité de tous les nœuds du système est identique et est égale à t ;
- ⇨ les nœuds tombent en panne indépendamment les uns des autres ;
- ⇨ chaque nœud s'éteint en cas de panne. Ainsi, par abus de langage, nous utiliserons souvent *disponibilité de nœud* pour désigner la *disponibilité de réplique* des données qui y sont stockées ;
- ⇨ il n'y a pas de panne sur les liens de communication.

Alors l'expression de $\Phi(i, j)$ est :

$$\Phi(i, j) \equiv \sum_{k=i}^{k=j} \binom{j}{k} t^k (1-t)^{j-k} \tag{2.3}$$

2.2.1 Quorum majoritaire

Le quorum majoritaire consiste à valider une opération d'écriture si et seulement si une majorité de répliques ont pu être mises à jour [98]. Soit n le nombre totale de

répliques des données. Alors, avec le quorum majoritaire, les deux contraintes (2.1) et (2.2) peuvent être formulées différemment comme suit (2.4) et (2.5) :

$$|Q_w| + |Q_r| > n \tag{2.4}$$

$$2|Q_w| > n \tag{2.5}$$

La contrainte (2.4) garantit une intersection non vide entre un quorum de lecture et n'importe quel quorum d'écriture tandis que la contrainte (2.5) permet d'assurer une intersection non vide entre deux quorums d'écriture. La contrainte (2.5) impose donc qu'il faut au moins $\lfloor \frac{n}{2} \rfloor + 1$ répliques pour qu'une opération d'écriture soit validée. Par conséquent,

$$|Q_w| = \lfloor \frac{n}{2} \rfloor + 1 \tag{2.6}$$

Les relations 2.4) et 2.6) donnent $|Q_r| > n - \lfloor \frac{n}{2} \rfloor - 1$. Il faut donc au moins $\lceil \frac{n}{2} \rceil$ répliques pour pouvoir lire correctement les données. Donc,

$$|Q_r| = \lceil \frac{n}{2} \rceil \tag{2.7}$$

Les formules (2.6) et (2.7) donnant respectivement la taille des quorums d'écriture et de lecture respectivement vérifient les contraintes (2.4) et (2.5). La disponibilité en écriture est alors :

$$P_{write} = \Phi(\lfloor \frac{n}{2} \rfloor + 1, n) \tag{2.8}$$

La disponibilité en lecture est :

$$P_{read} = \Phi(\lceil \frac{n}{2} \rceil, n) \tag{2.9}$$

Ainsi, le quorum majoritaire peut résister aux pannes de $\lfloor \frac{n}{2} \rfloor$ répliques lors d'une opération d'écriture et de $\lceil \frac{n}{2} \rceil - 1$ répliques lors d'une opération lecture.

2.2.2 Quorum en grille

Avec ce protocole, les répliques sont réparties en une grille [33]. Un quorum d'écriture est obtenu en sélectionnant une colonne entière et une réplique dans chacune des autres colonnes restantes de la grille. Un quorum de lecture est obtenu en sélectionnant une réplique dans chaque colonne de la grille. Avec ces définitions, les quorums d'écriture et de lecture vérifient les contraintes (2.1) et (2.2) des systèmes à quorums. Notons N et M respectivement le nombre de répliques dans une colonne et dans une ligne de la grille. La taille d'un quorum d'écriture est alors égale à $N + M - 1$ tandis que celle d'un quorum de lecture est égale à M. Le protocole peut tolérer la défaillance de $N - 1$ répliques quelconques et jusqu'à $M(N - 1)$ répliques spécifiques. Une opération d'écriture sera validée si au moins une colonne dispose des N répliques disponibles et les colonnes restantes contiennent chacune au moins une réplique disponible. Pour une colonne donnée, la probabilité que les N répliques soient disponibles est égale à t^N et la

probabilité pour qu'il y ait au moins une réplique disponible et au plus $N-1$ répliques disponibles est égale à $1-(1-t)^N-t^N$. Donc la disponibilité en écriture est égale à :

$$P_{write} = \sum_{k=1}^{k=M} \binom{M}{k} \left(t^N\right)^k \left(1-(1-t)^N-t^N\right)^{M-k} \tag{2.10}$$

Pour une colonne donnée, la probabilité pour qu'il y ait au moins une réplique disponible est égale à $1-(1-t)^N$. Une opération de lecture sera validée si toutes les colonnes de la grille disposent chacune d'au moins une réplique disponible. La disponibilité en lecture est donc égale à :

$$P_{read} = \left(1-(1-t)^N\right)^M \tag{2.11}$$

La figure 2.1 montre un exemple d'une grille constituée de 15 répliques. Elles sont rangées dans une grille de 3 lignes et 5 colonnes (chaque colonne contient $N=3$ répliques et chaque ligne contient $M=5$ répliques). Sur cette figure, nous avons représenté deux

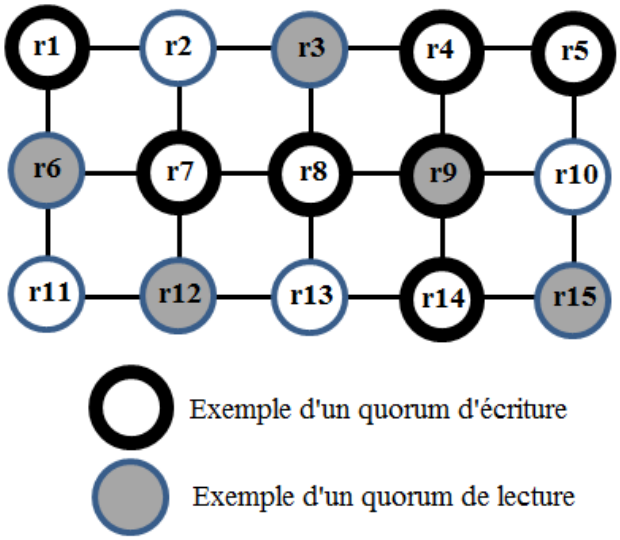


FIGURE 2.1 – Exemple de quorums en grille : $N=3$ et $M=5$

exemples de quorum : un quorum d'écriture et un quorum de lecture. Les nœuds {**r1, r7, r8, r4, r9, r14, r5**} constituent un quorum d'écriture tandis que les nœuds {**r6, r12, r3, r9, r15**} forment un quorum de lecture. La réplique **r9** est le point d'intersection de ces deux quorums.

2.2.3 Quorum en arbre

Les répliques sont organisées hiérarchiquement de façon à former un arbre. Dans cette section, par abus de langage, nous utiliserons souvent le nœud pour désigner la réplique de la donnée qui y est stocké. L'arbre est définit par les trois paramètres n, D et h, respectivement, le nombre de nœuds, le degré (nombre de nœuds fils attachés à chaque nœud de l'arbre sauf ceux qui sont situés dans la feuille de l'arbre) et la hauteur de

l'arbre. Il existe trois types d'approches permettant de constituer les quorums d'écriture et de lecture dans ce protocole [12, 13].

Première approche

Dans cette approche, on applique le quorum majoritaire dans chaque niveau de l'arbre. Ainsi, un quorum d'écriture est obtenu en sélectionnant la réplique du nœud au sommet et une majorité des répliques à tous les niveaux de l'arbre. Tandis qu'un quorum de lecture est obtenu en sélectionnant la réplique du nœud au sommet ou $\lceil \frac{D^l}{2} \rceil$ répliques dans un niveau l de l'arbre ($1 \leq l \leq h$). Les disponibilités d'écriture et de lecture s'expriment, dans ce contexte, par les formules suivantes [12, 20] :

$$P_{write} = \prod_{l=0}^{l=h} \Phi(\lfloor \frac{D^l}{2} \rfloor + 1, D^l) \tag{2.12}$$

$$P_{read} = 1 - \prod_{l=0}^{l=h} \left(1 - \Phi(\lceil \frac{D^l}{2} \rceil, D^l)\right) \tag{2.13}$$

La figure 2.2 représente un exemple d'un arbre constitué de $n = 13$ répliques, de degré $D = 3$ et de hauteur $h = 2$. Dans cet exemple, un quorum d'écriture est formé à partir

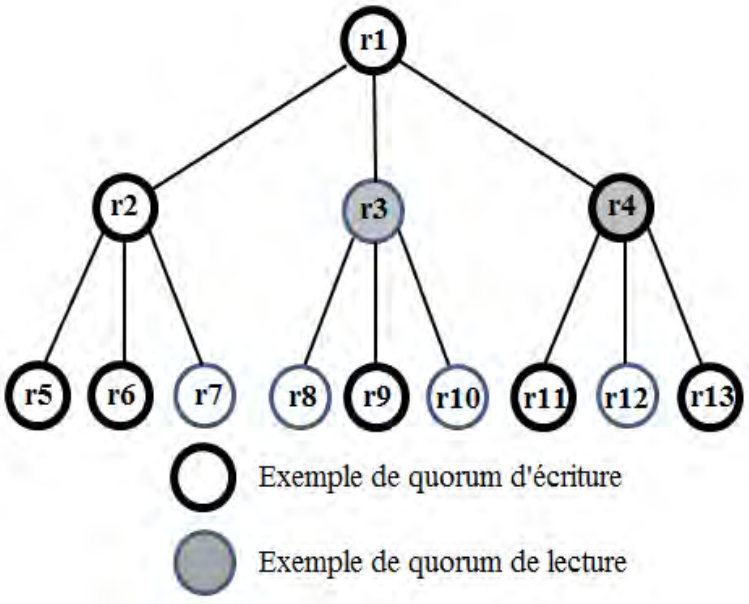


FIGURE 2.2 – Exemple de quorums en arbre : $n = 13$, $D = 3$ et $h = 2$ ($1^{ère}$ approche)

de n'importe quel ensemble de répliques contenant {**r1**}, 2 répliques du niveau $l = 1$ et 5 répliques du niveau $l = 2$ tandis qu'un quorum de lecture est obtenu en sélectionnant {**r1**} ou 2 répliques du niveau $l = 1$ ou 5 répliques du niveau $l = 2$. Ainsi, les répliques {**r1, r2, r4, r5, r6, r9, r11, r13**} forment un quorum d'écriture et les répliques {**r3, r4**} constituent un quorum de lecture. Il faut noter que la réplique {**r1**} seule forme un quorum de lecture et dans ce cas, lors de l'opération de lecture, on n'accède qu'à un seul nœud.

Deuxième approche

Pour une opération d'écriture, au lieu d'exiger une majorité à tous les niveaux de l'arbre, on sélectionne le nœud racine et, de manière récursive, une majorité des fils de chaque noeuds sélectionné jusqu'à ce qu'on atteigne les feuilles de l'arbre. Le nœud racine forme un quorum de lecture. En cas de défaillance d'un nœud (et à l'exception des nœuds feuilles de l'arbre), celui-ci peut être remplacé par une majorité de ses fils. Par exemple, si le nœud racine tombe en panne, une majorité des répliques situées au niveau $l = 1$ suffit pour lire les données. Sans perdre en généralité, supposons que $D = 2d + 1$ avec d entier naturel. La taille du quorum d'écriture est alors égale à $\sum_{l=0}^{l=h}(d+1)^h = \frac{(d+1)^l-1}{d}$. La taille du quorum de lecture varie de 1 jusqu'à $(d+1)^h$ selon le nombre de répliques défaillantes. Ce protocole peut donc tolérer la défaillance de n'importe quelles $\frac{(d+1)^l-1}{d} - 1$ répliques, et la défaillance de $n-1$ répliques particulieres [12]. On utilise une relation de récurrence pour évaluer la disponibilité en écriture et en lecture. Soient W_h et R_h respectivement la disponibilité en écriture et en lecture dans un arbre de degré h. Alors, la disponibilité en écriture et en lecture dans un arbre de degré $h + 1$ sont données par les formules suivantes (une démonstration plus détaillée est disponible dans [12]) :

$$W_{h+1} = t\left[\sum_{k=0}^{k=d}\binom{2d+1}{k}(W_h)^{d+k+1}(1-W_h)^{d-k}\right] \tag{2.14}$$

$$R_{h+1} = t + (1-t)\left[\sum_{k=0}^{k=d}\binom{2d+1}{k}(R_h)^{d+k+1}(1-R_h)^{d-k}\right] \tag{2.15}$$

Avec $W_0 = t$ et $R_0 = t$ respectivement la disponibilité en écriture et en lecture dans un arbre constitué uniquement d'un seul nœud.
Des exemples de quorum (en écriture et en lecture) sont donnés en figure 2.3, qui représente un arbre constitué de $n = 13$ répliques, de degré $D = 3$ (i.e., $d = 1$) et de hauteur $h = 2$. Dans cet exemple, selon la définition du protocole, n'importe quel quorum d'écriture doit contenir la réplique {**r1**} du nœud racine. De plus, il doit inclure une majorité des répliques du niveau $l = 1$ (**r2, r3 et r4**) et pour chaque nœud sélectionné, il doit inclure à son tour une majorité de répliques des nœuds fils. A titre d'exemple, les ensembles de répliques {**r1, r2, r5, r6, r4, r11, r13**} et {**r1, r3, r8, r10, r4, r12, r13**} forment chacun un quorum d'écriture. Quant au quorum de lecture, dans le meilleur des scénarios, il ne nécessite que la réplique {**r1**} du nœud racine. Par contre, quand la réplique du nœud racine n'est pas disponible lors d'une opération de lecture, un quorum de lecture peut être formé à partir de n'importe quelle majorité de répliques de ses nœuds fils, c'est-à-dire ({**r2, r3**} ou {**r2, r4**} ou {**r3, r4**}). Si aucune majorité ne peut être formée au niveau $l = 1$, une réplique dans ce niveau peut être remplacée par une majorité de répliques de ses nœuds fils. Par exemple, les répliques {**r5, r6**} ou {**r5, r7**} ou {**r6, r7**} peuvent substituer la réplique {**r2**} en cas d'indisponibilité de cette dernière. Ainsi, quand les répliques {**r1, r2, r4**} ne sont pas disponibles, l'ensemble de répliques {**r5, r6, r3**} peut être utilisé. Enfin, si les répliques {**r1, r2, r3, r4**} ne sont pas disponibles, alors une majorité de répliques de nœuds

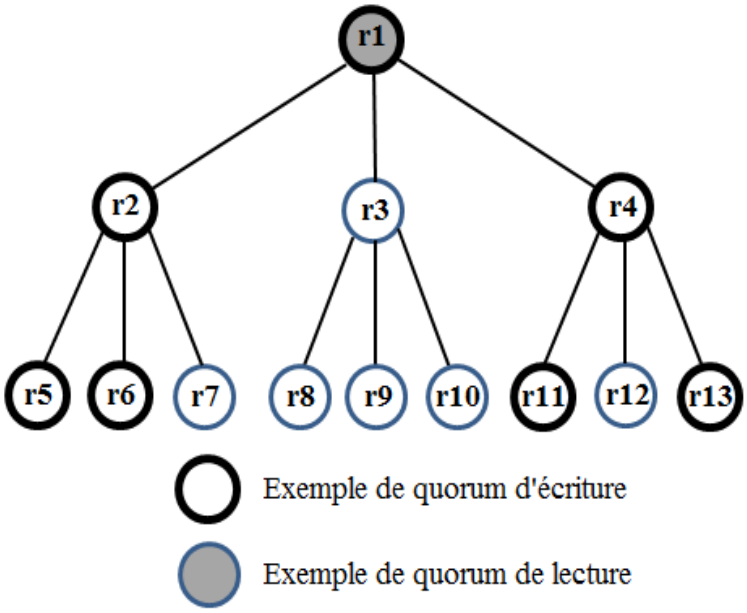


FIGURE 2.3 – Exemple de quorums en arbre : $n = 13$, $D = 3$ et $h = 2$ ($2^{ième}$ approche)

fils de n'importe quels deux nœuds au niveau $l = 1$ forme un quorum de lecture. Par exemple, les ensembles de répliques {**r5, r6, r8, r10**} ou encore {**r9, r10, r11, r13**} forment chacun un quorum de lecture. Par contre, si cette majorité de répliques de nœuds fils ne peut pas être trouvé pour au moins deux répliques parmi {**r2, r3, r4**}, alors l'opération de lecture échoue car les répliques se trouvant dans la feuille de l'arbre ne peuvent plus être remplacées par d'autres répliques en cas de défaillance.

Troisième approche

Le quorum d'écriture et celui de lecture ont la même définition avec cette approche. On utilise un arbre binaire pour organiser de manière logique les répliques des données. Un *chemin* dans l'arbre est une séquence de nœuds $\{s_0, s_1,..., s_h\}$ tel que pour tout $i \in \{0,...,h\}$, s_{i+1} est le fils de s_i. Si le protocole arrive à construire un chemin à partir du nœud racine jusqu'à n'importe quel nœud se trouvant au niveau des feuilles de l'arbre, alors l'ensemble de répliques dans ce chemin forme un quorum. Si un tel chemin ne peut pas être trouvé dû à une défaillance d'un nœud, disons s_l, alors ce nœud défaillant doit être remplacé par deux chemins : chaque chemin commence par le fils de s_l (les deux chemins ne doivent pas commencer sur le même nœud) et se termine vers un n'importe quel nœud se trouvant au niveau des feuilles de l'arbre. Dans ce protocole, le scénario correspondant au meilleur des cas permet d'obtenir un quorum dont la taille est égale à $h+1$: il s'agit du nombre de nœuds constituant un chemin depuis la racine de l'arbre vers une feuille de l'arbre. Dans le pire des cas, cette taille devient 2^h : il s'agit de tous les nœuds se trouvant au niveau des feuilles de l'arbre. On utilise une relation de récurrence pour évaluer la probabilité de trouver un quorum dans l'arbre binaire (c'est-à-dire la disponibilité en écriture et en lecture). Soit A_h la disponibilité en écriture et

en lecture dans un arbre binaire de hauteur h. Alors, la disponibilité en écriture et en lecture dans un arbre binaire de hauteur $h+1$ est égale à (une démonstration plus détaillée est disponible dans [13]) :

$$A_{h+1} = tA_h(1-A_h) + t(1-A_h)A_h + tA_h^2 + (1-t)A_h^2$$

c'est-à-dire,

$$A_{h+1} = 2tA_h + (1-2t)A_h^2 \tag{2.16}$$

Avec $A_0 = t$ représente la disponibilité en écriture et en lecture d'un arbre constitué d'un seul nœud.

La figure 2.4 montre un exemple d'un arbre binaire (c'est-à-dire de degré $D = 2$) constitué de $n = 15$ répliques et de hauteur $h = 3$. Avec cet exemple, selon la définition

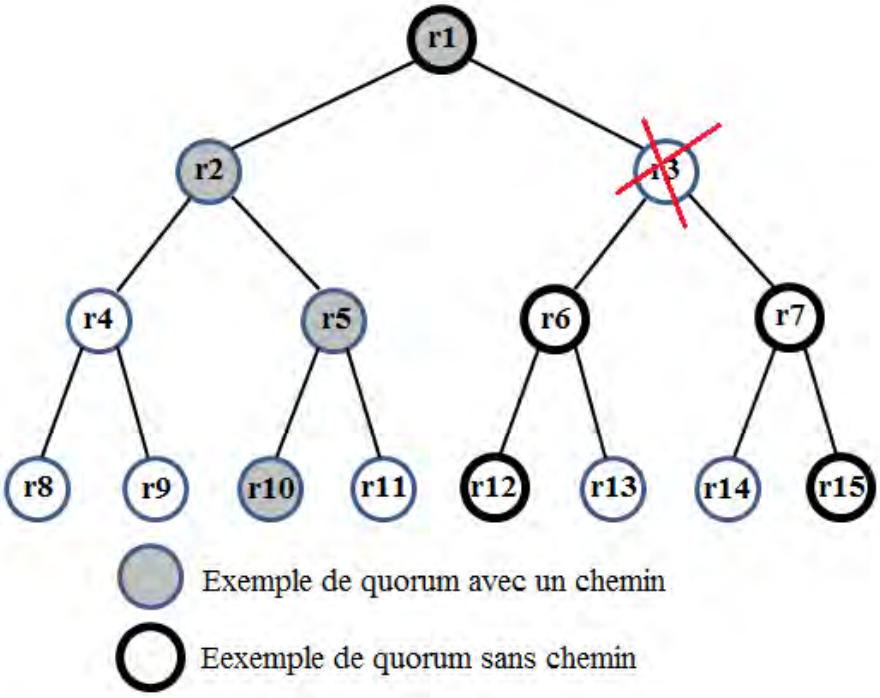


FIGURE 2.4 – Exemple de quorums en arbre : *n = 15*, *D = 2* et *h = 3* ($3^{ième}$ approche)

d'un quorum dans cet approche, l'ensemble de répliques {**r1, r2, r5, r10**} forme un quorum : il s'agit d'un chemin commençant par la racine {**r1**} de l'arbre et se terminant et se terminant par une feuille de l'arbre, en l'occurrence {**r10**}. Cet exemple montre également la construction d'un quorum sans pour autant avoir construit un chemin depuis la racine vers une feuille de l'arbre. Quand la réplique {**r3**} n'est pas disponible, le chemin commençant par {**r3**} vers une feuille de l'arbre peut être remplacé par deux chemins commençant par les deux nœuds des répliques {**r6**} et {**r7**} (en l'occurrence, les chemins {**r6, r12**} et {**r7, r15**}). Par conséquent, l'ensemble de répliques {**r1, r6, r12, r7, r15**} constitue un quorum en utilisant ce protocole.

2.2.4 Quorum en trapèze

Ce protocole organise de manière logique les répliques des données dans un trapèze de hauteur (h+1) [20]. Le trapèze contient h+1 niveaux et le niveau l contient $s_l = al+b$ répliques pour $0 \leq l \leq h$, avec :

- a est un nombre entier positif ou nul ;
- b est nombre entier strictement positif.

Avec ce protocole, les quorums d'écriture et de lecture sont définis comme suit pour garantir les cohérences des opérations d'écriture et de lecture :
Quorum d'écriture : Un quorum d'écriture est obtenu en sélectionnant une majorité des répliques se trouvant au niveau $l = 0$ et w répliques dans chaque niveau restant (c'est-à-dire du niveau $l = 1$ jusqu'au niveau $l = h$). Le paramètre w est un nombre entier choisi arbitrairement compris entre 1 et s_1 ($1 \leq w \leq s_1$). Il correspond au nombre minimum de répliques qui doivent-être écrites au niveau l (pour $1 \leq l \leq h$) lors d'une opération d'écriture pour que cette dernière puisse être validée.
Quorum de lecture : Un quorum de lecture est formé soit à partir d'une majorité des répliques résidant au niveau $l = 0$, soit à partir de $s_l - w + 1$ répliques résidant au niveau l ($1 \leq l \leq h$).

La probabilité de trouver une majorité au niveau $l = 0$ est égale à $\Phi(\lfloor \frac{b}{2} \rfloor + 1, s_0)$ et la probabilité de trouver au moins w répliques au niveau l (pour $1 \leq l \leq h$) est égale à $\Phi(w, s_l)$. Par conséquent, la disponibilité en écriture peut être exprimé comme suit :

$$P_{write} = \Phi(\lfloor \frac{b}{2} \rfloor + 1, s_0) \times \prod_{l=1}^{l=h} \Phi(w, s_l) \tag{2.17}$$

De façon similaire, la probabilité de trouver au moins $s_l - w + 1$ répliques au niveau l (pour $1 \leq l \leq h$) est égale à $P_l = \Phi(s_l - w + 1, s_l)$. Une opération de lecture ne peut pas être validée si le protocole ne trouve pas une majorité au niveau $l = 0$ et que tous les niveaux restant disposent chacun de moins de $s_l - w + 1$ répliques disponibles. Ainsi, la probabilité pour qu'une opération de lecture ne puisse pas être validée est égale à $\prod_{l=0}^{l=h}(1 - P_l)$ avec $P_0 = \Phi(\lfloor \frac{b}{2} \rfloor + 1, s_0)$. Par conséquent, la disponibilité en lecture est égale à :

$$P_{read} = 1 - \prod_{l=0}^{l=h}(1 - P_l) \tag{2.18}$$

La figure 2.5 représente un exemple de trapèze de paramètres $s_l = 2l + 3$, $a = 2$, $b = 3$ et $h = 2$. La valeur du paramètre w est fixée à 3. Dans cet exemple, selon la définition ci-dessus, un quorum d'écriture est un ensemble de 2 répliques (une majorité) au niveau $l = 0$, $w = 3$ répliques au niveau $l = 1$ et $w = 3$ répliques au niveau $l = 2$. Ainsi, l'ensemble de répliques {**r1, r3, r5, r7, r8, r9, r12, r14**} constitue un quorum d'écriture. Un quorum de lecture est un ensemble constitué de 2 répliques au niveau $l = 0$ ou 3 ($= s_1 - w + 1$) répliques au niveau $l = 1$ ou 5 ($= s_2 - w + 1$) répliques au niveau $l = 2$. Donc, les deux répliques {**r2, r3**} forment un quorum de lecture en utilisant ce protocole.

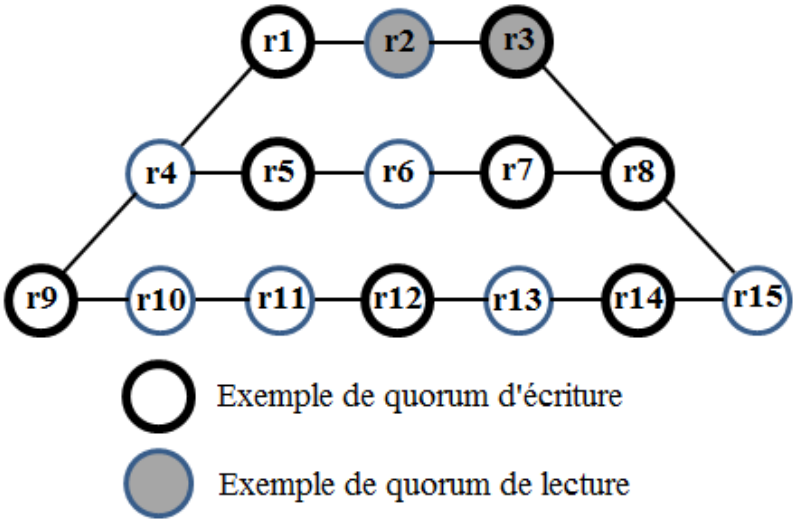


FIGURE 2.5 – Exemple de quorums en trapèze : $a = 2$, $b = 3$, $h = 2$ et $w = 3$

2.3 Synthèse

Nous avons présenté dans ce chapitre les différents modèles de cohérence dans un système de stockage distribué. Deux grandes catégories d'approches s'opposent. D'une part, les approches pessimistes, qui s'assurent d'abord que les répliques soient synchronisées avant d'autoriser les accès. La propagation des mises à jour de toutes les répliques d'une donnée est effectuée simultanément. Ces approches garantissent la cohérence des répliques des données et supportent ainsi les applications critiques. En contrepartie, la durée des mises à jour pourrait être relativement longue selon la performance de l'infrastructure utilisée. D'autre part, les approches optimistes propagent leur mises à jour en arrière-plan et découvrent éventuellement les conflits après qu'ils se soient produits. Elles autorisent aux utilisateurs l'accès à n'importe quelle réplique et à tout moment, en se basant sur l'hypothèse optimiste selon laquelle les conflits lors des mises à jour sont rares. Ces approches permettent d'améliorer la performance des opérations de mises à jour et de lecture. Par contre, elles ne peuvent pas garantir à tout moment la cohérence des données. Ainsi, la principale différence entre les approches optimistes et celles pessimistes réside sur la manière dont les mises à jour sont traitées. Nous avons vu également une autre classification des modèles de cohérence qui consiste à distinguer les modèles de cohérence centrés sur les données et ceux centrés sur le client. Les modèles centrés sur les données considèrent le fait que les données sont partagées entre plusieurs utilisateurs alors que les modèles centrés sur le client se basent sur la cohérence vis-à-vis d'un client individuel.

Nos travaux de thèse ont pour but de garantir la cohérence des répliques de données stockées dans CloViS avec un code correcteur (n, k) comme technique de distribution des données. Dans ce contexte, une opération de lecture doit trouver au moins k répliques à jour pour retrouver la donnée originale. Ainsi, une cohérence stricte entre au moins k répliques d'une donnée est nécessaire. Dans CloViS, nous avons choisi d'utiliser les

protocoles à base des quorums. Ces protocoles ayant été conçus dans le contexte de réplication totale, nous avons besoin de les adapter dans notre contexte et d'étudier leurs performances respectives. C'est ce que nous allons présenter dans le prochain chapitre.

Chapitre 3

Gestion de la cohérence des données dans CloViS

Sommaire

3.1 Contexte

Dans un environnement de stockage distribué, la solution naturelle pour améliorer la disponibilité et la fiabilité des données est le protocole de réplication totale. Cette technique de distribution des données permet d'obtenir la plus grande disponibilité possible mais entraîne une utilisation excessive des ressources système. Cette solution ne constitue pas une solution viable car l'un des défis d'un système de stockage distribué tel que CLoViS c'est de réduire l'utilisation des ressources système tout en gardant une disponibilité et une fiabilité élevées des données. Pour cela, CloViS permet l'utilisation de plusieurs techniques de distribution des données afin de pouvoir définir plusieurs niveaux de qualité de service en termes de stockage. Le choix de ces techniques de distribution des données est basé sur le(s) critère(s) prioritaire(s) aux yeux des utilisateurs tels que la fiabilité des opérations d'entrée/sortie, la disponibilité des données, la tolérance aux pannes, la performance, etc. Cependant l'utilisation des techniques de distribution des données nécessite un mécanisme de contrôle de la concurrence des accès aux données et un protocole de gestion de la cohérence des données (appelé souvent *protocole de réplication* dans le contexte de réplication des données). Le contrôle de la concurrence a été traité par Aurélien Ortiz dans le cadre de sa thèse effectuée au sein du laboratoire IRIT [10, 75]. Nos travaux, présentés dans ce manuscrit, traitent de la gestion de la cohérence des répliques des objets répartis dans les différentes ressources de stockage de CloViS.

De nombreux travaux ont été publiés dans la littérature sur la gestion des répliques des données dans le contexte de réplication totale. Quelques exemples non exhaustifs ont été donnés dans le chapitre 2 de la partie II. Mais ces protocoles ne sont pas adaptés aux autres techniques de distribution des données telles que le RAID, les schémas à seuil et les codes correcteurs. En effet ces protocoles de maintien de la cohérence, dans le cadre de réplication totale, ne garantissent qu'une seule réplique avec une bonne version lors

d'une opération de lecture alors que plusieurs répliques à jour sont souvent requises dans les autres techniques de distribution des données. Par exemple, avec un code correcteur de type (n, k) (avec $1 \leq k \leq n$), k répliques sont requises pour pouvoir reconstruire les données originales. Ainsi, nous avons décidé d'adapter ces protocoles de maintien de la cohérence pour supporter ces différentes techniques de distribution des données. Le nombre de répliques à jour exigées lors d'une opération de lecture va être paramétrable avec ces protocoles adaptés et peut aller de 1 jusqu'à n (nombre total de répliques utilisées pour stocker une donnée). Ensuite, nous analysons ces nouveaux protocoles afin de mettre en exergue leurs points forts et leurs points faibles respectifs. Les résultats de ces analyses constituent une aide à la décision au sein de CloViS pour le choix d'un protocole de maintien de la cohérence selon les critères de performance souhaités par les utilisateurs. Avec ces protocoles, plusieurs critères de performance peuvent être définis au sein de CloViS tels que la disponibilité des données, l'utilisation des ressources système, le nombre de messages échangés durant les opérations d'entrée/sortie, etc.

Dans le reste de ce manuscrit, nous allons adopter les notations suivantes :

- ☞ t : dénote la disponibilité des nœuds, c'est-à-dire la probabilité qu'un nœud donné dans le système soit disponible ;
- ☞ n : dénote le nombre total de blocs utilisés pour stocker une donnée ;
- ☞ P_{write} : représente la disponibilité en écriture, c'est à dire la probabilité que l'opération d'écriture d'une donnée dans le système réussisse ;
- ☞ P_{read} : représente la disponibilité en lecture, c'est à dire la probabilité pour qu'une opération de lecture d'une donnée depuis le système réussisse ;
- ☞ N_{write} : représente le nombre moyen de messages échangés durant une opération d'écriture ;
- ☞ N_{read} : représente le nombre moyen de messages échangés durant une opération de lecture.

Sans perte de généralité, pour l'évaluation de la disponibilité des opérations d'écriture et de lecture et le calcul du nombre moyen de messages échangés durant ces opérations, nous allons supposer dans tous ce qui suit que :

- ⇨ la disponibilité de tous les nœuds du système est identique et est égale à t ;
- ⇨ les nœuds tombent en panne indépendamment les uns des autres ;
- ⇨ chaque nœud s'éteint en cas de panne. Ainsi, par abus de langage, nous utiliserons souvent l'expression *disponibilité du nœud* pour désigner *la disponibilité des répliques des données qui y sont stockées* ;
- ⇨ il n'y a pas de panne sur les liens de communication ;
- ⇨ la réussite ou non des opérations d'écriture et de lecture d'un bloc sur un nœud dépend uniquement de la disponibilité de ce dernier ;
- ⇨ trois messages sont échangés quand une opération d'écriture d'un bloc réussit et uniquement un message si cette dernière échoue ;

  1. une requête d'écriture est envoyée pour récupérer l'ancien bloc ;
  2. le nœud concerné envoie une réponse pour autoriser l'opération d'écriture si il est disponible ;

3. s'il y a une réponse, le bloc à écrire sera envoyé vers le nœud concerné.

⇨ deux messages sont échangés quand une opération de lecture d'un bloc réussit et uniquement un message si cette dernière échoue ;

1. une requête de lecture d'un bloc est envoyée vers le nœud qui stocke ce bloc ;
2. le nœud concerné envoie alors le bloc demandé si il est disponible.

⇨ un nœud en défaillance n'émet pas de message lors des opérations d'écriture et de lecture.

Dans un souci de lisibilité, nous allons utiliser les expressions suivantes :

- ☞ $\Psi_N(t,i)$: dénote la probabilité que exactement i nœuds sur un total de N nœuds soient disponibles ;
- ☞ $\Phi_N(t,i,j)$: dénote la probabilité qu'au moins i nœuds et au plus j nœuds sur un total de N nœuds soient disponibles ;
- ☞ ξ_{n_u,n_r,n_w} : cette expression retourne 1 si l'évènement "*n_u nœuds parmi les n_r nœuds disponibles durant l'opération de lecture étaient également disponibles lors de la dernière opération d'écriture où au total n_w nœuds étaient disponibles*" est vrai, 0 sinon ;
- ☞ μ_{n_u,n_r,n_w} : cette expression retourne 1 si l'évènement "*n_u nœuds parmi les n_r nœuds disponibles durant l'opération de lecture étaient également disponibles lors de la dernière opération d'écriture où au total n_w nœuds étaient disponibles*" est vrai et permet de déterminer la version de la donnée, 0 sinon ;
- ☞ $T(n_u,t,w,N)$: représente la probabilité que exactement n_u nœuds parmi les nœuds disponibles durant une opération de lecture stockent des blocs à jour. En notant n_r ce nombre de nœuds, on a $n_r \geq N-w+1$, avec N et w respectivement le nombre total de nœuds et la taille d'un quorum d'écriture.

Avant d'utiliser ces expressions, nous allons d'abord déterminer leur formule mathématique. Par définition, $\Psi_N(t,i)$ et $\Phi_N(t,i,j)$ peuvent s'exprimer comme suit :

$$\Psi_N(t,i) = \binom{N}{i}.t^i.(1-t)^{(N-i)} \tag{3.1}$$

$$\Phi_N(t,i,j) = \sum_{k=i}^{k=j} \left[\binom{N}{k}.t^k.(1-t)^{(N-k)} \right] \tag{3.2}$$

Soient W, R et U trois variables aléatoires définies comme suit :

⇨ W : représente la liste des nœuds disponibles durant la dernière opération d'écriture valide ;

⇨ R : représente la liste des nœuds disponibles durant l'opération de lecture en cours ;

⇨ U : désigne l'intersection de W et R c'est à dire le nombre de nœuds disponibles à la fois durant l'opération de lecture en cours et la dernière opération d'écriture valide.

$|W|$, $|R|$ et $|U|$ désignent respectivement le nombre d'éléments de W, R et U. Soit **E** l'évènement suivant : "*n_u nœuds parmi les n_r nœuds disponibles durant l'opération de lecture étaient également disponibles lors de la dernière opération d'écriture où au total n_w nœuds étaient disponibles*". Alors l'expression ξ_{n_u,n_r,n_w} peut être exprimée en fonction de la probabilité de l'évènement **E**.

$$\xi_{n_u,n_r,n_w} = \begin{cases} 1 & \text{si } \mathbb{P}(E) \neq 0 \\ 0 & \text{sinon} \end{cases} \tag{3.3}$$

Or, l'évènement **E** est équivalent à l'évènement "$|W| = n_w$, $|R| = n_r$ et $|U| = n_u$". Donc, nous avons $\mathbb{P}(E) = \mathbb{P}(|W| = n_w, |R| = n_r, |U| = n_u)$.
Soit w la taille d'un quorum d'écriture. Par définition, W doit contenir au moins w éléments et au plus N éléments, soit $w \leq n_w \leq N$. Par ailleurs, $U = W \cap R$; le cardinal de W est donc supérieur ou égal à celui de U (soit $n_w \geq n_u$). Par conséquent, $max(w, n_u) \leq n_w \leq N$.
Soit $n_w \in [max(w, n_u), N]$. Pour que la probabilité $\mathbb{P}(|W| = n_w, |R| = n_r, |U| = n_u)$ soit non nulle, il faut que n_r vérifie les conditions suivantes : $n_r \geq n_u$ et $n_r \leq N + n_u - n_w$. En effet, la première condition vient du fait que $U = W \cap R$. Il en résulte que le cardinal de R est supérieur ou égal à celui de U. Quant à la deuxième condition, il faut que le cardinal de la réunion de W et de R soit inférieur ou égal au nombre total de nœuds impliqués dans le stockage d'une donnée, soit N. Alors $|W \cup R| \leq N$ c'est-à-dire $|W| + |R| - |U| \leq N \implies |R| \leq N + |U| - |W|$, soit $n_r \leq N + n_u - n_w$. D'où

$$\xi_{n_u,n_r,n_w} = \begin{cases} 1 & \text{si } n_w \in [max(w, n_u), N] \text{ et } n_r \in [n_u, N + n_u - n_w] \\ 0 & \text{sinon} \end{cases} \tag{3.4}$$

L'expression de μ_{n_u,n_r,n_w} peut être déduite de celle de ξ_{n_u,n_r,n_w}. En effet, si l'évènement **E** se produit (c'est-à-dire $\xi_{n_u,n_r,n_w} = 1$), la version de la donnée peut être déterminée si le nombre de nœuds disponibles durant cette opération de lecture est supérieur ou égal à $N - w + 1$ (autrement dit, $W \cap R \neq \emptyset$). Par conséquent,

$$\mu_{n_u,n_r,n_w} = \begin{cases} \xi_{n_u,n_r,n_w} & \text{si } n_r \geq N - w + 1 \\ 0 & \text{sinon} \end{cases} \tag{3.5}$$

L'expression $T(n_u, t, w, N)$ est la probabilité de trouver $|U| = n_u$ et $|R| \geq N - w + 1$ sachant que $|W| \geq w$. Alors,

$$\begin{aligned} T(n_u, t, w, N) &= \mathbb{P}(|U| = n_u, |R| \geq N - w + 1 || W| \geq w) \\ &= \frac{\mathbb{P}(|U| = n_u, |R| \geq N - w + 1, |W| \geq w)}{\mathbb{P}(|W| \geq w)} \\ &= \frac{\mathbb{P}(|U| = n_u, |R| \geq N - w + 1, |W| \geq w)}{\Phi_N(t, w, N)} \end{aligned} \tag{3.6}$$

Dans l'équation 3.6, il nous reste à déterminer $\mathbb{P}(|U| = n_u, |R| \geq N - w + 1, |W| \geq w)$. Nous avons,

$$\mathbb{P}(|U| = n_u, |R| \geq N - w + 1, |W| \geq w) = \sum_{k=w}^{k=N} \left[\sum_{i=N-w+1}^{i=N} \mathbb{P}(|U| = n_u, |R| = i, |W| = k) \right] \tag{3.7}$$

Pour $|W| = n_w \in [max(w, n_u), N]$ et $|R| = n_r \in [n_u, N + n_u - n_w]$ (autrement dit $\xi_{n_u,n_r,n_w} = 1$), parmi les n_r nœuds disponibles durant l'opération de lecture en cours, n_u nœuds étaient aussi disponibles durant la dernière opération d'écriture valide tandis que les $n_r - n_u$ nœuds restants ne l'étaient pas. Et, parmi les $N - n_r$ nœuds qui ne sont pas disponibles durant l'opération de lecture en cours, seuls $n_w - n_u$ nœuds étaient disponibles durant la dernière opération d'écriture valide. Ainsi,

$$\mathbb{P}(|U| = n_u, |R| = n_r, |W| = n_w) = \begin{cases} \Psi_N(t, n_r).\Psi_{n_r}(t, n_u).\Psi_{N-n_r}(t, n_w - n_u) & \text{si } \xi_{n_u,n_r,n_w} = 1 \\ 0 & \text{sinon} \end{cases} \quad (3.8)$$

Comme le paramètre ξ_{n_u,n_r,n_w} ne prend que deux valeurs (0 ou 1), alors l'expression dans la formule (3.8) est équivalente à :

$$\mathbb{P}(|U| = n_u, |R| = n_r, |W| = n_w) = \Psi_N(t, n_r).\Psi_{n_r}(t, n_u).\Psi_{N-n_r}(t, n_w - n_u).\xi_{n_u,n_r,n_w} \quad (3.9)$$

En utilisant les deux équations (3.7) et (3.8), nous avons l'expression suivante en conservant uniquement les bornes de sommations correspondantes à $\xi_{n_u,n_r,n_w} = 1$:

$$\mathbb{P}(|U| = n_u, |R| \geq N - w + 1, |W| \geq w) = \sum_{k=max(w,n_u)}^{k=N} \left[\sum_{i=n_u}^{i=N+n_u-k} \mathbb{P}(|U| = n_u, |R| = i, |W| = k) \right] \quad (3.10)$$

Donc,

$$\mathbb{P}(|U| = n_u, |R| \geq N - w + 1, |W| \geq w) = \sum_{k=max(w,n_u)}^{k=N} \left[\sum_{i=n_u}^{i=N+n_u-k} \Psi_N(t, i).\Psi_i(t, n_u).\Psi_{N-i}(t, k - n_u) \right] \quad (3.11)$$

Par conséquent, l'expression de $T(n_u, t, w, N)$ peut être déduite des équations (3.6) et (3.11) :

$$T(n_u, t, w, N) = \frac{\sum_{k=max(w,n_u)}^{k=N} \left[\sum_{i=n_u}^{i=N+n_u-k} \Psi_N(t, i).\Psi_i(t, n_u).\Psi_{N-i}(t, k - n_u) \right]}{\Phi_N(t, w, N)} \quad (3.12)$$

Ainsi, les expressions $\Psi_N(t, i)$, $\Phi_N(t, i, j)$, ξ_{n_u,n_r,n_w}, μ_{n_u,n_r,n_w} et $T(n_u, t, w, N)$ sont données respectivement par les formules (3.1), (3.2), (3.4), (3.5) et (3.12). Nous allons utiliser ces expressions dans ce qui suit.

3.2 Quorum majoritaire général

3.2.1 Définition

Le protocole *quorum majoritaire général*, que nous avons présenté dans [86], est une adaptation du quorum majoritaire classique pour pouvoir supporter plusieurs types de techniques de distribution des données telles que la réplication totale, les schémas à seuil, les codes correcteurs, etc. Pour cela, il doit prendre en considération le nombre de blocs à jour requis pour la reconstruction des données originales lors d'une opération de lecture. En effet, ce nombre de blocs à jour requis pour la reconstruction des données originales peut varier de 1 à n. Nous noterons w et r respectivement la taille du quorum d'écriture et le nombre de blocs à jour requis pour la reconstruction des données originales en utilisant ce nouveau protocole. Par exemple, en utilisant un *code correcteur* (n, k) comme technique de distribution des données, le paramètre r doit être supérieur ou égal à k et avec un *schéma à seuil général* $(p - m - n)$, il doit être supérieur ou égal à m. Avec le quorum majoritaire classique, il faut pouvoir écrire au moins $\lfloor \frac{n}{2} \rfloor + 1$ blocs sur un total de n blocs pour valider une opération d'écriture tandis qu'avec le quorum majoritaire général, en plus de cette condition, il faut aussi que ce nombre de blocs écrit soit supérieur ou égal à r. On a donc :

$$w = \max(r, \lfloor \frac{n}{2} \rfloor + 1) \tag{3.13}$$

Avec ce protocole, il faut au moins $n - w + 1$ blocs parmi les n blocs d'une donnée pour pouvoir déterminer le numéro version de cette dernière. Ensuite, le protocole doit sélectionner r répliques à jour pour valider une opération de lecture. Nous allons évaluer, pour ce protocole, la disponibilité des opérations d'écriture et de lecture et le nombre moyen de messages échangés durant ces opérations respectives.

3.2.2 Disponibilité en écriture

En utilisant ce protocole, une opération d'écriture d'une donnée sera uniquement validée si parmi les n blocs de cette donnée, au moins w d'entre eux sont écrits avec succès.

$$P_{write} = \Phi_n(t, w, n) \tag{3.14}$$

Pour $r \leq \lfloor \frac{n}{2} \rfloor + 1$, nous avons $w = \lfloor \frac{n}{2} \rfloor + 1$, ce qui correspond également à la taille d'un quorum d'écriture dans le quorum majoritaire classique. En revanche, pour $r > \lfloor \frac{n}{2} \rfloor + 1$, nous avons $w = r$. Ainsi, la différence de disponibilité en écriture ente le *quorum majoritaire général* et *quorum majoritaire classique* ne se produit que lorsque $r > \lfloor \frac{n}{2} \rfloor + 1$.

Nous proposons dans l'algorithme 1, page 71, une procédure pour former un quorum d'écriture avec le protocole quorum majoritaire général. La procédure d'écriture d'une donnée est définie dans l'algorithme 9 page 105.

Algorithm 1 Procédure de recherche d'un quorum d'écriture

```
1: Données
2: x                                              ▷ Identité de la donnée à écrire
3: (n, r)                                         ▷ Paramètres du protocole de distribution utilisé
4: P ← {P_1, P_2, ..., P_n}                       ▷ n nœuds utilisés pour le stockage de x

5: procedure getWriteQuorum(x, P)                 ▷ Recherche d'un quorum d'écriture
6:     Q ← ∅                                      ▷ Quorum de lecture
7:     v ← 0                                      ▷ Numéro de version de la donnée
8:     w ← max(r, ⌊n/2⌋ + 1)                      ▷ Taille d'un quorum d'écriture
9:     for all q ∈ P do
10:        v_i ← requête d'accès (x, q)           ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
11:        if estValide(v_i) then                 ▷ estValide(v_i) retourne 1 si v_i est un numéro de version valide et 0 sinon
12:            Q ← Q ∪ {q}
13:            if v_i > v then
14:                v ← v_i
15:            end if
16:        end if
17:        if |Q| = w then                        ▷ Un quorum d'écriture est obtenu
18:            break
19:        end if
20:    end for
21:    if |Q| = w then
22:        return [v + 1, Q]                      ▷ Version de la donnée à écrire et un quorum d'écriture
23:    else
24:        return NULL                            ▷ La recherche d'un quorum d'écriture échoue
25:    end if
26: end procedure
```

3.2.3 Disponibilité en lecture

Avec le *quorum majoritaire général*, une opération de lecture doit d'abord déterminer la dernière version de la donnée, puis sélectionner r blocs à jour afin de reconstruire les données originales. Si le protocole n'arrive pas à obtenir la dernière version de la donnée, l'opération de lecture échoue. Cette dernière échoue également si le protocole n'arrive pas sélectionner r blocs à jour même si la version de la donnée a été obtenue. Ces deux conditions peuvent être exprimées comme suit :

- ☞ parmi les n nœuds qui stockent les blocs de la donnée, au moins $n - w + 1$ nœuds doivent être disponibles durant cette opération de lecture (condition requise pour obtenir la version de la donnée) ;
- ☞ parmi les nœuds disponibles durant cette opération de lecture, au moins r d'entre eux l'étaient aussi lors de la dernière opération d'écriture valide de la donnée en question.

Ainsi, la disponibilité en lecture est égale à $\mathbb{P}(|U| \geq r, |R| \geq n - w + 1 \,|\, |W| \geq w)$ (avec W, R et U sont trois variables aléatoires définies dans la section 3.1 page 67). Or cette expression peut être exprimée en fonction de T définit dans l'équation (3.12). En effet,

$$\begin{aligned} P_{read} &= \mathbb{P}(|U| \geq r, |R| \geq n - w + 1 \,|\, |W| \geq w) \\ &= \sum_{k=r}^{k=n} \mathbb{P}(|U| = k, |R| \geq n - w + 1 \,|\, |W| \geq w) \\ &= \sum_{k=r}^{k=n} T(k, t, w, n) \end{aligned}$$

Ainsi,

$$P_{read} = \sum_{k=r}^{k=n} T(k,t,w,n) \tag{3.15}$$

Nous proposons, dans l'algorithme 2 page 72, une procédure pour former un quorum de lecture avec le protocole quorum majoritaire général. La procédure de lecture d'une donnée est définie dans l'algorithme 10 page 106.

Algorithm 2 Procédure de recherche d'un quorum de lecture

```
1: Données
2: x                                                          ▷ Identité de la donnée à lire
3: (n, r)                                                     ▷ Paramètres du protocole de distribution utilisé
4: P ← {P_1, P_2, ..., P_n}                                   ▷ n nœuds stockant les morceaux de x
5: procedure GETREADQUORUM(x, P)                              ▷ Recherche d'un quorum de lecture
6:     Q ← ∅                                                  ▷ Quorum de lecture
7:     v ← 0                                                  ▷ Numéro de version de la donnée
8:     compteur ← 0                                           ▷ Nombre de nœuds accédés
9:     w ← max(r, ⌊n/2⌋ + 1)
10:    for all q ∈ P do
11:        v_i ← requête d'accès (x, q)        ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
12:        if estValide(v_i) then              ▷ estValide(v_i) retourne 1 si v_i est un numéro de version valide et 0 sinon
13:            compteur ← compteur + 1
14:            if v_i > v then
15:                Q ← {q}
16:                v ← v_i
17:            else if v_i = v then
18:                if |Q| < r then
19:                    Q ← Q ∪ {q}
20:                end if
21:            end if
22:        end if
23:        if |Q| = r and compteur >= n − w + 1 then          ▷ Un quorum de lecture est obtenu
24:            return [v, Q]
25:        end if
26:    end for
27:    return NULL                                            ▷ La recherche d'un quorum de lecture échoue
28: end procedure
```

3.2.4 Nombre moyen de messages échangés

Nous allons évaluer le nombre moyen de messages échangés lors d'une tentative de lecture ou d'écriture en utilisant ce protocole. Le calcul de ce nombre moyen de messages échangés doit tenir compte de tous les scénarios possibles. En effet, même si une opération d'écriture ou de lecture échoue, plusieurs messages sont générés, par exemple les messages pour la requête de lecture ou d'écriture. Soit v_k le nombre de messages échangés lors d'une tentative d'écriture d'une donnée où k nœuds sur un total de n nœuds sont disponibles. Deux scénarios sont alors possibles :

- ⇨ $0 \leq k < w$: l'opération d'écriture échoue mais il y a déjà des messages échangés. On comptabilise n messages de requêtes d'écriture et k messages d'autorisation (provenant des k nœuds disponibles) ;
- ⇨ $w \leq k \leq n$: l'opération d'écriture réussit. Il y a n messages de requête d'écriture, k messages d'autorisation et k autres messages pour l'envoie des blocs vers les nœuds disponibles.

Ainsi, $n + k$ messages sont échangés lors d'une opération d'écriture qui a échoué alors que $n + 2k$ messages circulent quand cette opération d'écriture réussit. Donc, v_k peut être exprimé comme suit :

$$v_k = \begin{cases} n + k & \text{si } 0 \leq k < w \\ n + 2k & \text{si } w \leq k \leq n \end{cases} \tag{3.16}$$

Soit X, la variable aléatoire qui, à chaque événement élémentaire (opération d'écriture ou opération de lecture), associe le nombre de nœuds disponibles parmi les n nœuds utilisés pour le stockage de la donnée. Lors d'une tentative d'écriture ou de lecture, la variable aléatoire associe à chaque nombre de nœuds disponibles un nombre de messages échangés. Alors l'ensemble des valeurs que peut prendre la variable aléatoire X sur $\{0, 1, \ldots, n\}$ est $\{v_0, v_1, \ldots, v_n\}$ et pour $0 \leq k \leq n$, nous avons $\mathbb{P}(X = k) = \Psi_n(t, k)$. Le nombre moyen de messages échangés pour l'opération d'écriture est défini comme l'espérance mathématique de la variable aléatoire X. Ainsi, par définition,

$$N_{write} = \sum_{k=0}^{k=n} v_k.\mathbb{P}(X = k)$$

Par conséquent,

$$N_{write} = \sum_{k=0}^{k=n} v_k.\Psi_n(t, k) \tag{3.17}$$

Supposons que k nœuds soient disponibles lors d'une opération de lecture ($0 \leq k \leq n$). Alors $n + k$ messages sont échangés durant cette opération de lecture (n messages de requête de lecture et k messages de réponse). Donc,

$$\begin{aligned} N_{read} &= \sum_{k=0}^{k=n} (n + k).\mathbb{P}(X = k) \\ &= (n + k).\Psi_n(t, k) \end{aligned} \tag{3.18}$$

3.3 Quorum en grille général

3.3.1 Définition

Le *quorum en grille général* est une évolution du quorum en grille classique afin de pouvoir utiliser plusieurs types de techniques de distribution des données. Comme dans le quorum en grille classique, il organise les répliques d'une donnée de façon logique, dans une grille de largeur N et de longueur M. Chaque point de la grille représente une réplique de la donnée comme montre la figure 3.1 avec $N = 3$ et $M = 5$. Il existe alors M colonnes dont chacune contient N répliques c'est-à-dire au total $n = N \times M$ répliques. Soient w et r respectivement la taille du quorum d'écriture et le nombre de blocs à jour requis pour la reconstruction des données originales en utilisant ce nouveau protocole. Un quorum d'écriture est construit en sélectionnant $max(r, N + M - 1)$

répliques dont N répliques dans une colonne et au moins une réplique dans chacune des colonnes restantes de la grille. Ainsi,

$$w = \max(r, N + M - 1) \tag{3.19}$$

Pour $r \leq N + M - 1$, et en particulier pour $r = 1$ (dans le contexte de la réplication totale), nous retrouvons la définition d'un quorum d'écriture dans le *quorum en grille classique*. En effet, les N répliques d'une colonne entière et une réplique dans chacune des autres colonnes restantes de la grille permettent de valider une opération d'écriture (c'est-à-dire $N+M-1$ répliques). En utilisant ce protocole, les M répliques obtenues en sélectionnant une réplique dans chaque colonne de la grille permettent de déterminer la dernière version de la donnée. Connaissant la dernière version de la donnée, le protocole n'a plus qu'à sélectionner r répliques avec cette version pour valider une opération de lecture. La figure 3.1 représente un exemple d'une grille de largeur $N = 3$ et de longueur $M = 5$. Nous allons donner quelques exemples de quorum d'écriture en utilisant la

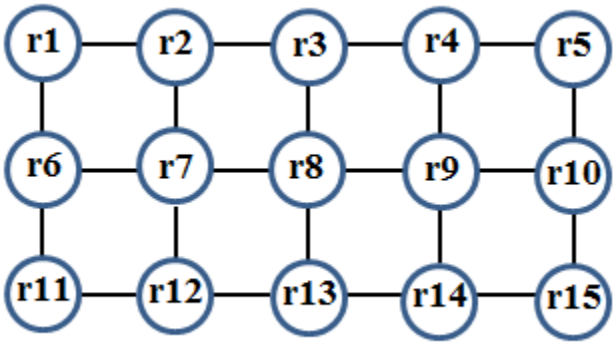


FIGURE 3.1 – Exemple de quorums en grille : $N = 3$ et $M = 5$

grille de la figure 3.1. Dans cet exemple, la taille d'un quorum d'écriture est égale à $w = \max(r, 7)$ car $N + M - 1 = 7$. Ainsi, deux cas seront à distinguer ($r \leq 7$ et $r > 7$) :

- ☞ $r = 4$ (cas $r \leq N + M - 1$) : ce cas est obtenu en utilisant par exemple un *code correcteur* $(15, 4)$ ou un *schéma à seuil* $1 - 4 - 15$ comme technique de distribution de données. La taille d'un quorum d'écriture est égale à $w = 7$. Ainsi, les 3 répliques de la $2^{ième}$ colonne et une réplique dans chacune des colonnes restantes ($1^{ière}$, $3^{ième}$, $4^{ième}$ et $5^{ième}$) forment un quorum d'écriture (par exemple **{r11, r2, r7, r12, r8, r14, r5}**, **{r6, r2, r7, r12, r13, r9, r10}**, etc.) ;
- ☞ $r = 9$ (cas $r > N + M - 1$) : ce cas est obtenu en utilisant par exemple un *code correcteur* $(15, 9)$ ou un *schéma à seuil* $1 - 9 - 15$ comme technique de distribution de données. La taille d'un quorum d'écriture est égale à $w = 9$. Ainsi, les 3 répliques de la $1^{ière}$ colonne et 6 répliques dans les colonnes restantes (dont chacune doit présenter au moins une réplique) forment un quorum d'écriture (par exemple **{r1, r6, r11, r2, r3, r8, r14, r10, r15}**).

Quant à l'opération de lecture dans cet exemple, le protocole va sélectionner $M = 5$ répliques à raison d'une réplique par colonne afin de déterminer la version de la donnée. Ensuite, il va sélectionner r répliques à jour pour reconstruire la donnée originale.

3.3.2 Disponibilité en écriture

En utilisant ce protocole, une opération d'écriture sera validée si le protocole arrive à sélectionner N nœuds disponibles dans une colonne et au moins un nœud disponible dans chacune des colonnes restantes, la somme du nombre de nœuds disponibles devant être supérieure ou égale à r. Supposons qu'il existe k colonnes contenant N nœuds disponibles durant une opération d'écriture (avec $1 \leq k \leq M$). Alors les $M-k$ colonnes restantes doivent contenir chacune au moins un nœud disponible. Soient ε_1, ε_2, ..., ε_{M-k} les positions de ces $M-k$ colonnes dans la grille[1] avec $1 \leq \varepsilon_i \leq M$ pour tout $1 \leq i \leq M-k$. Soient d_{ε_1}, d_{ε_2}, ..., $d_{\varepsilon_{M-k}}$ le nombre de nœuds disponibles dans les colonnes de positions respectives $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_{M-k}$ durant une opération d'écriture. Alors $1 \leq d_{\varepsilon_i} \leq N-1$ pour tout $1 \leq i \leq M-k$. En effet, nous avons supposé que seules k colonnes contiennent N nœuds disponibles, les $M-k$ restantes n'en contenant qu'au plus $N-1$. Par ailleurs, pour que l'opération d'écriture puisse se poursuivre, il faut que chacune de ces $M-k$ colonnes contienne au moins un nœud disponible. Donc, le nombre total de nœuds disponibles durant cette opération d'écriture est égal à $N.k + \sum_{i=1}^{i=M-k} d_{\varepsilon_i}$ et il doit être supérieur ou égal à r. Ainsi, pour qu'une opération d'écriture réussisse, il faut :

- ☞ k colonnes de la grille contenant chacune N nœuds disponibles avec $1 \leq k \leq M$;
- ☞ au moins un nœud disponible dans chacune des $M-i$ colonnes restantes ($\forall i \in \{0, 1, \ldots, M-k\}$, on a $1 \leq d_{\varepsilon_i} \leq N-1$) ;
- ☞ que le nombre total de nœuds disponibles soit supérieur ou égal à r (c'est-à-dire $N.k + \sum_{i=1}^{i=M-k} d_{\varepsilon_i} \geq r$).

Par conséquent, pour une colonne donnée de la grille, il y a deux cas possibles concernant le nombre de nœuds qui y sont disponibles durant une opération d'écriture valide, à savoir :

- ☞ tous les N nœuds sont disponibles, avec une probabilité de t^N ;
- ☞ d_{ε_i} nœuds sont disponibles où $1 \leq i \leq M-k$ et $1 \leq d_{\varepsilon_i} \leq N-1$, avec une probabilité de $\Psi_N(t, d_{\varepsilon_i})$.

Soit $\upsilon_{d_{\varepsilon_1},\ldots,d_{\varepsilon_{M-k}}}$ un paramètre qui prend la valeur 1 si le nombre total de nœuds disponibles est supérieur ou égal à r, la valeur 0 sinon.

$$\upsilon_{d_{\varepsilon_1},\ldots,d_{\varepsilon_{M-k}}} = \begin{cases} 1 & \text{si } \sum_{i=1}^{i=M-k} d_{\varepsilon_i} \geq r - N.k \\ 0 & \text{sinon} \end{cases} \tag{3.20}$$

1. Par exemple la position ε_2 veut dire la $\varepsilon_2^{ième}$ colonne de la grille

Pour tout $k \in \{1, 2, \ldots, M\}$, soit P_k la probabilité de trouver dans la grille exactement k colonnes ayant leurs N nœuds tous disponibles et les $M-k$ colonnes restantes contiennant chacune au moins un nœud disponible lors d'une opération d'écriture. Alors P_k est égal à :

$$P_k = \binom{M}{k}.(t^N)^k.\left(\sum_{\substack{1\leq d_{\varepsilon_1}\leq N-1 \\ \vdots \\ 1\leq d_{\varepsilon_{M-k}}\leq N-1}} \left[\upsilon_{d_{\varepsilon_1},\ldots,d_{\varepsilon_{M-k}}}.\prod_{i=1}^{i=M-k}\Psi_N(t,d_{\varepsilon_i})\right]\right) \tag{3.21}$$

Le terme $\upsilon_{d_{\varepsilon_1},\ldots,d_{\varepsilon_{M-k}}}$ dans la formule (3.21) permet d'éliminer toutes les éventualités qui ne vérifient pas la condition $N.k+\sum_{i=1}^{i=M-k} d_{\varepsilon_i} \geq r$ requise par la technique de distribution des données pour la reconstruction des données originales. Alors la probabilité pour qu'une opération d'écriture réussisse peut être formulée en fonction de P_k :

$$P_{write} = \sum_{k=1}^{k=M} P_k \tag{3.22}$$

Par conséquent, en utilisant les formules (3.21) et (3.33), la disponibilité en écriture avec ce protocole peut s'exprimer comme suit :

$$P_{write} = \sum_{k=1}^{k=M}\left[\binom{M}{k}.(t^N)^k.\left(\sum_{\substack{1\leq d_{\varepsilon_1}\leq N-1 \\ \vdots \\ 1\leq d_{\varepsilon_{M-k}}\leq N-1}} \left[\upsilon_{d_{\varepsilon_1},\ldots,d_{\varepsilon_{M-k}}}.\prod_{i=1}^{i=M-k}\Psi_N(t,d_{\varepsilon_i})\right]\right)\right] \tag{3.23}$$

L'algorithme 3 page 77 présente une procédure pour former un quorum d'écriture avec le protocole quorum en grille général. La procédure d'écriture d'une donnée est définie dans l'algorithme 9 page 105.

3.3.3 Disponibilité en lecture

En utilisant le *quorum en grille général*, pour qu'une opération de lecture soit validée, deux conditions sont requises :

- ⇨ au moins un nœud est disponible dans chacune des M colonnes de la grille (cette condition est nécessaire pour déterminer la dernière version de la donnée) ;

Algorithm 3 Procédure de recherche d'un quorum d'écriture

```
1: Données
2: x                                              ▷ Identité de la donnée à écrire
3: N                                              ▷ Nombre de nœuds dans une colonne de la grille
4: M                                              ▷ Nombre de nœuds dans une ligne de la grille
5: P_c                                            ▷ Liste des nœuds résidant dans la ligne c de la grille
6: P ← {P_1, P_2, ..., P_M}                       ▷ n nœuds utilisés pour le stockage de x

7: procedure getWriteQuorum(x, P)                 ▷ Recherche d'un quorum d'écriture
8:     Q ← ∅
9:     v ← 0
10:    test_col_entiere ← 0                       ▷ Teste si une colonne avec N nœuds accessibles a été trouvée
11:    w ← max(r, N + M − 1)
12:    nb_sup ← w − (N + M − 1)
13:    for c ← 1, M do
14:        T ← ∅                                  ▷ Liste des nœuds accessibles dans une colonne
15:        for all q ∈ P_c do
16:            v_i ← requête d'accès (x, q)       ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
17:            if estValide(v_i) then             ▷ estValide(v_i) retourne 1 si v_i est un numéro de version valide et 0 sinon
18:                T ← T ∪ {q}
19:                if v_i > v then
20:                    v ← v_i
21:                end if
22:            end if
23:        end for
24:        if |T| = 0 then                        ▷ Une colonne qui ne contient pas de nœud accessible
25:            return NULL
26:        else if |T| = N and test_col_entiere = 0 then
27:            Q ← Q ∪ T
28:            test_col_entiere ← 1
29:        else if nb_sup >= |T| then
30:            nb_sup ← nb_sup − |T| + 1
31:            Q ← Q ∪ {T}
32:        else if nb_sup < |T| then
33:            Q ← Q ∪ {nb_sup + 1 élément(s) de T}
34:            nb_sup ← 0
35:        end if
36:    end for
37:    if test_col_entiere ≠ 0 and |Q| = w then
38:        return [v + 1, Q]
39:    else
40:        return NULL
41:    end if
42: end procedure
```

⇨ parmi les nœuds disponibles au moins r d'entre eux contiennent un bloc à jour de la donnée (cette condition est requise par le protocole de distribution des données pour la reconstruction de la donnée originale).

Nous présentons dans l'algorithme 4, page 78, une proposition de procédure pour former un quorum de lecture avec le protocole quorum en grille général. La procédure de lecture d'une donnée est définie dans l'algorithme 10 page 106.

Dans un souci de lisibilité, nous allons adopter les notations suivantes :

⇨ $n_{w,c}$: désigne le nombre de nœuds disponibles dans la $c^{ième}$ colonne de la grille durant la dernière opération d'écriture valide $(1 \leq c \leq M)$;
⇨ $n_{r,c}$: dénote le nombre de nœuds disponibles dans la $c^{ième}$ colonne de la grille durant l'opération de lecture en cours $(1 \leq c \leq M)$;
⇨ $n_{u,c}$: représente le nombre de nœuds disponibles dans la $c^{ième}$ colonne de la grille durant à la fois la dernière opération d'écriture valide et l'opération de lecture en cours $(1 \leq c \leq M)$. Il désigne également le nombre de blocs à jour de la donnée.

Algorithm 4 Procédure de recherche d'un quorum de lecture

Données
x ▷ Identité de la donnée à lire
N ▷ Nombre de nœuds dans une colonne de la grille
M ▷ Nombre de nœuds dans une ligne de la grille
P_l ▷ Liste des nœuds résidant dans la ligne c de la grille
$P \leftarrow \{P_1, P_2, \ldots, P_N\}$ ▷ n nœuds utilisés pour le stockage de x

procedure GETREADQUORUM(x, P) ▷ Recherche d'un quorum de lecture
  $Q \leftarrow \emptyset$
  $v \leftarrow 0$
  $t[c] \leftarrow 0$ pour tout $c \in \{1, \ldots, M\}$ ▷ Prend la valeur 1 si on trouve au moins un nœud disponible dans la colonne c et 0 sinon
  for $l \leftarrow 1, N$ **do**
    for all $q \leftarrow P_l$ **do**
      $v_i \leftarrow$ requête d'accès (x, q) ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
      if estValide(v_i) **then** ▷ estValide(v_i) retourne 1 si v_i est un numéro de version valide et 0 sinon
        $\exists c \in \{1, 2, \ldots, M\}/q$ réside dans la colonne c.
        $t[c] \leftarrow 1$
        if $v_i > v$ **then**
          $Q \leftarrow \{q\}$
        else if $v_i = v$ **and** $|Q| < r$ **then**
          $Q \leftarrow Q \cup \{q\}$
        end if
      end if
    end for
    if $|Q| = r$ **and** $\sum_{c=1}^{c=M} t[c] = M$ **then**
      return $[v, Q]$
    end if
  end for
  return $NULL$
end procedure

Soit Z une variable aléatoire qui représente les valeurs des paramètres $n_{w,c}$, $n_{r,c}$ et $n_{u,c}$ de toutes les colonnes de la grille. La variable aléatoire Z peut être exprimée sous la forme d'une matrice $3 \times M$ (une matrice de 3 lignes et de M colonnes) telle que :

- ⇨ la première ligne représente le nombre de nœuds disponibles dans chaque colonne de la grille durant la dernière opération d'écriture valide et l'opération de lecture en cours ;
- ⇨ la deuxième ligne représente le nombre de nœuds disponibles dans chaque colonne de la grille durant l'opération de lecture en cours ;
- ⇨ la troisième ligne désigne le nombre de nœuds disponibles dans chaque colonne de la grille durant la dernière opération d'écriture valide.

On a :

$$Z = \begin{pmatrix} n_{u,1} & n_{u,2} & \ldots & n_{u,M} \\ n_{r,1} & n_{r,2} & \ldots & n_{r,M} \\ n_{w,1} & n_{w,2} & \ldots & n_{w,M} \end{pmatrix}$$

Lors d'une tentative de lecture, pour tout $c \in \{1, 2, \ldots, M\}$, les paramètres $n_{w,c}$, $n_{r,c}$ et $n_{u,c}$ prennent chacun des valeurs dans $\{0, 1, \ldots, N\}$. Alors le vecteur constitué par la $c^{ième}$ colonne de la matrice Z a $(N+1)^3$ valeurs possibles. Comme la matrice Z contient M colonnes, elle a par conséquent $(N+1)^{3.M}$ valeurs possibles. Soient $\{z_1, z_2, \ldots, z_{(N+1)^{3.M}}\}$ toutes les valeurs possibles que peut prendre la matrice Z.
Pour tout $k \in \{1, 2, \ldots, (N+1)^{3.M}\}$, la probabilité pour que la variable aléatoire Z

prenne la valeur z_k est égale à :

$$\mathbb{P}(Z = z_k) = \prod_{c=1}^{c=M} \mathbb{P}\left(|U| = z_k(1,c), |R| = z_k(2,c), |W| = z_k(3,c)\right) \tag{3.24}$$

où U, R et W représentent les trois variables aléatoires définies dans la section 3.1 page 67). La probabilité $\mathbb{P}(Z = z_k)$ dans la formule (3.24) est connue. En effet, la probabilité $\mathbb{P}(|U| = z_k(1,c), |R| = z_k(2,c), |W| = z_k(3,c))$ a déjà été calculée précédemment (voir formule (3.8) dans la page 69).
Ainsi, pour obtenir la disponibilité en lecture avec ce protocole, il nous reste à identifier les z_k qui répondent aux conditions requises pour la validation d'une opération de lecture et à effectuer la somme de leur probabilités respectives. Pour tout $k \in \{1, 2, \ldots, (N+1)^{3.M}\}$, soit σ_{z_k} le produit du nombre de nœuds disponibles dans chaque colonne de la grille.

$$\sigma_{z_k} = \prod_{c=1}^{c=M} z_k(2,c) \tag{3.25}$$

Lorsque l'éventualité $Z = z_k$ se présente, pour que l'opération de lecture se poursuive, il faut que σ_{z_k} soit non nul. Cela garantit que chaque colonne de la grille contient au moins un nœud disponible lors de cette opération de lecture. Par ailleurs, le nombre total de nœuds contenant un bloc à jour de la donnée est égal à $\sum_{c=1}^{c=M} z_k(1,c)$. Ainsi, la deuxième condition requise par l'opération de lecture peut s'exprimer par $\sum_{c=1}^{c=M} z_k(1,c) \geq r$. Nous allons définir par la suite δ_{z_k} qui va permettre au protocole de décider si la donnée peut être reconstruite quand l'éventualité $Z = z_k$ se présente. Soit,

$$\delta_{z_k} = \begin{cases} 1 & \text{si } \sigma_{z_k} \neq 0 \text{ et } \sum_{c=1}^{c=M} z_k(1,c) \geq r \\ 0 & \text{sinon} \end{cases} \tag{3.26}$$

Ainsi, lorsque l'éventualité $Z = z_k$ se présente lors d'une opération de lecture, cette dernière est validée quand le paramètre δ_{z_k} prend la valeur 1 et elle échoue lorsque ce paramètre prend la valeur 0. Par conséquent, la disponibilité en lecture avec le *quorum en grille général* peut être exprimée comme suit :

$$P_{read} = \sum_{k=1}^{k=(N+1)^{3.M}} \delta_{z_k}.\mathbb{P}(Z = z_k) \tag{3.27}$$

En remplaçant $\mathbb{P}(Z = z_k)$ par son expression dans la formule (3.24), nous obtenons :

$$P_{read} = \sum_{k=1}^{k=(N+1)^{3.M}} \left[\delta_{z_k}. \prod_{c=1}^{c=M} \mathbb{P}\left(|U| = z_k(1,c), |R| = z_k(2,c), |W| = z_k(3,c)\right) \right] \tag{3.28}$$

Nous allons maintenant exprimer P_{read} en fonction des paramètres $n_{w,c}$, $n_{r,c}$ et $n_{u,c}$ pour tour $c \in \{1, \ldots, M\}$. D'après la formule (3.9) dans la page 69, nous avons :

$$\mathbb{P}\left(|U| = n_{u,c}, |R| = n_{r,c}, |W| = n_{w,c}\right) = \Psi_N(t, n_{r,c}).\Psi_{n_{r,c}}(t, n_{u,c}).\Psi_{N-n_{r,c}}(t, n_{w,c} - n_{u,c}).\xi_{n_{u,c},n_{r,c},n_{w,c}} \tag{3.29}$$

Or, lorsqu'une tentative de lecture se termine avec succès, dans la $c^{ième}$ colonne de la grille, l'évènement $\{|U| = n_{u,c}, |R| = n_{r,c}, |W| = n_{w,c}\}$ est possible si et seulement si les trois conditions suivantes sont remplies : $1 \leq n_{r,c} \leq N$, $1 \leq n_{w,c} \leq N$ et $0 \leq n_{u,c} \leq min(n_{r,c}, n_{w,c})$. Dans le cas où ces conditions sont remplies, l'évènement $\{|U| = n_{u,c}, |R| = n_{r,c}, |W| = n_{w,c}\}$ peut se produire, ce qui implique que $\xi_{n_{u,c},n_{r,c},n_{w,c}} = 1$(Cf. page 67). La formule (3.29) devient donc :

$$\mathbb{P}\left(|U| = n_{u,c}, |R| = n_{r,c}, |W| = n_{w,c}\right) = \Psi_N(t, n_{r,c}).\Psi_{n_{r,c}}(t, n_{u,c}).\Psi_{N-n_{r,c}}(t, n_{w,c}-n_{u,c}) \tag{3.30}$$

Nous allons ensuite définir les trois ensembles suivants :

- $E_u = \{(n_{u,1}, \ldots, n_{u,M}) \mid \forall c \in \{1, \ldots, M\}\ 0 \leq n_{u,c} \leq min(n_{r,c}, n_{w,c})\}$;
- $E_r = \{(n_{r,1}, \ldots, n_{r,M}) \mid \forall c \in \{1, \ldots, M\}\ 1 \leq n_{r,c} \leq N\}$;
- $E_w = \{(n_{w,1}, \ldots, n_{w,M}) \mid \forall c \in \{1, \ldots, M\}\ 1 \leq n_{w,c} \leq N\}$.

L'expression de P_{read} dans la formule (3.28) est alors équivalente à :

$$P_{read} = \sum_{\substack{(n_{u,1},\ldots,n_{u,M})\in E_u \\ (n_{r,1},\ldots,n_{r,M})\in E_r \\ (n_{w,1},\ldots,n_{w,M})\in E_w}} \left[\delta_x . \prod_{c=1}^{c=M} \mathbb{P}\left(|U| = n_{u,c}, |R| = n_{r,c}, |W| = n_{w,c}\right)\right] \tag{3.31}$$

Avec $x = \begin{pmatrix} n_{u,1} & n_{u,2} & \ldots & n_{u,M} \\ n_{r,1} & n_{r,2} & \ldots & n_{r,M} \\ n_{w,1} & n_{w,2} & \ldots & n_{w,M} \end{pmatrix}$. En remplaçant $\mathbb{P}\left(|U| = n_{u,c}, |R| = n_{r,c}, |W| = n_{w,c}\right)$ par son expression dans la formule (3.30), l'expression de la disponibilité en lecture dans la formule (3.31) devient :

$$P_{read} = \sum_{\substack{(n_{u,1},\ldots,n_{u,M})\in E_u \\ (n_{r,1},\ldots,n_{r,M})\in E_r \\ (n_{w,1},\ldots,n_{w,M})\in E_w}} \left[\delta_x . \prod_{c=1}^{c=M} \left[\Psi_N(t, n_{r,c}).\Psi_{n_{r,c}}(t, n_{u,c}).\Psi_{N-n_{r,c}}(t, n_{w,c} - n_{u,c})\right]\right] \tag{3.32}$$

3.3.4 Nombre moyen de messages échangés

Dans cette sous-section, nous allons calculer le nombre moyen de messages échangés lors des opérations d'écriture et de lecture. Pour cela, nous allons définir la variable aléatoire X qui à chaque évènement élémentaire (en l'occurrence, une opération d'écriture ou une opération de lecture) associe le M-uplet formé par le nombre de nœuds disponibles dans chaque colonne de la grille. Cette variable aléatoire X correspond donc au M-uplet $(n_{w,1}, n_{w,2}, \ldots, n_{w,M})$ lorsque l'évènement élémentaire est une opération d'écriture et au M-uplet $(n_{r,1}, n_{r,2}, \ldots, n_{r,M})$ lorsqu'il s'agit d'une opération de lecture.

Nous allons commencer pour le calcul du nombre moyen de messages échangés pour l'opération d'écriture. Pour tout $c \in \{1, 2, \ldots, M\}$, nous avons $0 \leq n_{w,c} \leq N$ c'est-à-dire $(N+1)$ valeurs possibles. Il en résulte que le cardinal de l'ensemble de valeurs que peut prendre la variable aléatoire X est égal à $(N+1)^M$. Soient $x_1, x_2, \ldots, x_{(N+1)^M}$ toutes les éventualités possibles de la variable aléatoire X. Pour tout $c \in \{1, 2, \ldots, M\}$ et $k \in \{1, 2, \ldots, (N+1)^M\}$, $x_k(c)$ désigne le $c^{ième}$ composant du vecteur x_k, c'est-à-dire la valeur correspondant à $n_{w,c}$. Alors, pour tout $k \in \{1, 2, \ldots, (N+1)^M\}$, l'éventualité $X = x_k$ se présente avec une probabilité égale à :

$$\begin{aligned} \mathbb{P}(X = x_k) &= \prod_{c=1}^{c=M} \left[\binom{N}{x_k(c)} . t^{x_k(c)} . (1-t)^{N - x_k(c)} \right] \\ &= t^{\left[\sum_{c=1}^{c=M} x_k(c)\right]} . (1-t)^{\left[N.M - \sum_{c=1}^{c=M} x_k(c)\right]} . \prod_{c=1}^{c=M} \binom{N}{x_k(c)} \end{aligned} \tag{3.33}$$

Soient $\alpha_{x_k,c}$ un paramètre qui indique si parmi les c premières colonnes, au moins une colonne ne contient aucun nœud disponible et v_k le nombre de messages échangés lors d'une tentative d'écriture lorsque l'éventualité $X = x_k$ se présente (avec $1 \leq k \leq (N+1)^M$ et $1 \leq c \leq M$). Le paramètre $\alpha_{x_k,c}$ prend la valeur 1 si les c premières colonnes contiennent chacune au moins un nœud disponible, la valeur 0 sinon. Il peut être exprimé comme suit :

$$\alpha_{x_k,c} = \begin{cases} 1 & \text{si } \prod_{i=1}^{i=c} x_k(i) \neq 0 \\ 0 & \text{sinon} \end{cases} \tag{3.34}$$

Considérons la $c^{ième}$ colonne visitée lors d'une opération d'écriture où l'évènement $X = x_k$ se présente. On a :

- ⇨ $\alpha_{x_k,c}.N$ messages de requête d'écriture pour récupérer les anciens blocs résidant dans cette colonne, c'est-à-dire que le protocole envoie les N messages de requête d'écriture si et seulement si toutes les $c-1$ colonnes visitées précédemment contiennent chacune au moins un nœud disponible (cela correspond à $\alpha_{x_k,c} = 1$). Dans le cas contraire, il arrête l'opération d'écriture ;
- ⇨ $\alpha_{x_k,c}.x_k(c)$ messages de réponse sont envoyés par les nœuds disponibles si ces derniers ont reçu une requête d'écriture ;
- ⇨ $\alpha_{x_k,M}.x_k(c)$ messages contenant les nouveaux blocs sont envoyés vers les nœuds disponibles si et seulement si les deux conditions suivantes sont satisfaites :
  - ☞ le nombre de total des nœuds disponibles dans la grille est supérieur ou égal à r (c'est-à-dire $\sum_{c=1}^{c=M} x_k(c) \geq r$) ;
  - ☞ au moins une colonne parmi les M colonnes de la grille contient N nœuds disponibles (c'est-à-dire $\prod_{c=1}^{c=M} (N - x_k(c)) = 0$).

Le paramètre $\alpha_{x_k,c}$ permet donc de garantir que lors d'une opération d'écriture, le protocole stoppe directement le processus et l'opération d'écriture échoue dès qu'on rencontre une colonne qui ne contient aucun nœud disponible. Ainsi, le paramètre v_k peut être formulé comme suit :

$$v_k = \begin{cases} \sum_{c=1}^{c=M} [\alpha_{x_k,c}.N + \alpha_{x_k,c}.x_k(c) + \alpha_{x_k,M}.x_k(c)] & \text{si } \sum_{c=1}^{c=M} x_k(c) \geq r \text{ et } \prod_{i=1}^{i=M} (N - x_k(i)) = 0 \\ \sum_{c=1}^{c=M} [\alpha_{x_k,c}.N + \alpha_{x_k,c}.x_k(c))] & \text{sinon} \end{cases}$$

Comme pour tout $c \in \{1, 2, \ldots, M\}$, $\alpha_{x_k,c} = 0$ implique $\alpha_{x_k,M} = 0$ et $\alpha_{x_k,M} = 1$ implique $\alpha_{x_k,c} = 1$, alors $\alpha_{x_k,c}.\alpha_{x_k,M} = \alpha_{x_k,M}$. Par conséquent, nous pouvons factoriser l'expression de v_k :

$$v_k = \begin{cases} \sum_{c=1}^{c=M} [\alpha_{x_k,c}.\,[N + (1 + \alpha_{x_k,M}).x_k(c)]] & \text{si } \sum_{c=1}^{c=M} x_k(c) \geq r \text{ et } \prod_{i=1}^{i=M} (N - x_k(i)) = 0 \\ \sum_{c=1}^{c=M} [\alpha_{x_k,c}.(N + x_k(c))] & \text{sinon} \end{cases} \tag{3.35}$$

Si nous posons :

$$\lambda_{x_k} = \begin{cases} 1 & \text{si } \sum_{c=1}^{c=M} x_k(c) \geq r \\ 0 & \text{sinon} \end{cases} \tag{3.36}$$

$$\beta_{x_k} = \begin{cases} 1 & \text{si } \prod_{c=1}^{c=M} (N - x_k(c)) = 0 \\ 0 & \text{sinon} \end{cases} \tag{3.37}$$

Alors l'expression de v_k dans la formule 3.35 est équivalente à :

$$v_k = \sum_{c=1}^{c=M} \Big[\alpha_{x_k,c}.\,[N + (1 + \lambda_{x_k}.\beta_{x_k}.\alpha_{x_k,M}).x_k(c)]\Big] \tag{3.38}$$

En effet, dès que l'un des termes β_{x_k} ou λ_{x_k} vaut zéro, le terme $\alpha_{x_k,M}.x_k(c)$ disparaît. Par définition, le nombre moyen de messages échangés lors d'une opération d'écriture est défini comme étant l'espérance mathématique de la variable aléatoire X.

$$N_{write} = \sum_{k=1}^{k=(N+1)^M} v_k.\mathbb{P}(X = x_k)$$

En utilisant la formule (3.33), nous obtenons :

$$N_{write} = \sum_{k=1}^{k=(N+1)^M} \left[v_k.t^{\left[\sum_{c=1}^{c=M} x_k(c)\right]}.(1-t)^{\left[N.M - \sum_{c=1}^{c=M} x_k(c)\right]}. \prod_{c=1}^{c=M} \binom{N}{x_k(c)} \right] \tag{3.39}$$

Or, pour tout $k \in \{1, 2, \ldots, (N+1)^M\}$ il existe un unique M-uplet $(n_{w,1}, \ldots, n_{w,M}) \in \{0, 1, \ldots, N\}^M$ tel que $x_k = (n_{w,1}, \ldots, n_{w,M})$. Inversement, pour tout $(n_{w,1}, \ldots, n_{w,M}) \in \{0, 1, \ldots, N\}^M$ il existe un unique $k \in \{1, 2, \ldots, (N+1)^M\}$ tel que $(n_{w,1}, \ldots, n_{w,M}) = x_k$. Cela veut dire donc que les deux ensembles sont identiques :

$$\{x_k \mid k \in \{1, 2, \ldots, (N+1)^M\} = \{(n_{w,1}, \ldots, n_{w,M}) \mid \forall c \in \{1, 2, \ldots, M\}, 0 \leq n_{w,c} \leq N\}$$

Ainsi, N_{write} peut être exprimée en fonction des $n_{w,c}$ (pour c allant de 1 à M) comme suit :

$$N_{write} = \sum_{\substack{0 \leq n_{w,1} \leq N \\ \vdots \\ 0 \leq n_{w,M} \leq N}} \left[v_k . t^{\left[\sum_{c=1}^{c=M} n_{w,c}\right]} . (1-t)^{\left[N.M - \sum_{c=1}^{c=M} n_{w,c}\right]} . \prod_{c=1}^{c=M} \binom{N}{n_{w,c}} \right] \tag{3.40}$$

En posant $x = (n_{w,1}, \ldots, n_{w,M})$ et en remplaçant v_k par son expression dans la formule (3.38), le nombre moyen de messages échangés durant l'opération d'écriture dans la formule (3.40) devient :

$$N_{write} = \sum_{\substack{0 \leq n_{w,1} \leq N \\ \vdots \\ 0 \leq n_{w,M} \leq N}} \left[\sum_{c=1}^{c=M} \left[\alpha_{x,c} . \left[N + (1 + \lambda_x . \beta_x . \alpha_{x,M}) . n_{w,c} \right] \right] . \right.$$
$$\left. t^{\left[\sum_{c=1}^{c=M} n_{w,c}\right]} . (1-t)^{\left[N.M - \sum_{c=1}^{c=M} n_{w,c}\right]} . \prod_{c=1}^{c=M} \binom{N}{n_{w,c}} \right] \tag{3.41}$$

Après avoir déterminé le nombre moyen de messages échangés lors d'une opération d'écriture, nous allons calculer ce nombre dans le cas d'une opération de lecture. Soit μ_k le nombre de messages échangés lors d'une opération de lecture lorsque l'éventualité $X = x_k$ se présente (avec $1 \leq k \leq (N+1)^M$). Considérons la $c^{ième}$ colonne visitée lors d'une opération de lecture où l'évènement $X = x_k$ se présente (avec $1 \leq c \leq M$). On a :

- ⇨ $\alpha_{x_k,c} . N$ messages de requête de lecture, le terme $\alpha_{x_k,c}$ signifiant que le protocole envoie les N messages de requête de lecture si et seulement si toutes les $c-1$ colonnes visitées précédemment contiennent chacune au moins un nœud disponible (cela correspond à $\alpha_{x_k,c} = 1$). Dans le cas contraire, il arrête l'opération de lecture ;
- ⇨ $\alpha_{x_k,c} . x_k(c)$ messages de réponse venant des nœuds disponibles.

Ainsi, l'expression de μ_k est égale à :

$$\begin{aligned} \mu_k &= \sum_{c=1}^{c=M} \left[\alpha_{x_k,c} . N + \alpha_{x_k,c} . x_k(c) \right] \\ &= \sum_{c=1}^{c=M} \alpha_{x_k,c} \left[N + x_k(c) \right] \end{aligned} \tag{3.42}$$

Par définition, le nombre de messages échangés lors d'une opération de lecture est égale à l'espérance mathématique de la variable aléatoire X.

$$N_{read} = \sum_{k=1}^{k=(N+1)^M} \mu_k.\mathbb{P}(X = x_k)$$

En remplaçant $\mathbb{P}(X = x_k)$ par son expression dans la formue (3.33) et μ_k par son expression dans la formule (3.42), nous obtenons :

$$N_{read} = \sum_{k=1}^{k=(N+1)^M} \left[\left[\sum_{c=1}^{c=M} \alpha_{x_k,c} \left[N + x_k(c) \right] \right] . t^{\left[\sum_{c=1}^{c=M} x_k(c) \right]} . (1-t)^{\left[N.M - \sum_{c=1}^{c=M} x_k(c) \right]} . \prod_{c=1}^{c=M} \binom{N}{x_k(c)} \right] \tag{3.43}$$

Suivant le raisonnement dans la détermination de N_{write}, nous pouvons exprimer N_{read} en fonction des $n_{r,c}$ (pour c allant de 1 à M) :

$$N_{read} = \sum_{\substack{0 \le n_{r,1} \le N \\ \vdots \\ 0 \le n_{r,M} \le N}} \left[\left[\sum_{c=1}^{c=M} \alpha_{x,c} \left[N + n_{r,c} \right] \right] . \right. \\ \left. t^{\left[\sum_{c=1}^{c=M} n_{r,c} \right]} . (1-t)^{\left[N.M - \sum_{c=1}^{c=M} n_{r,c} \right]} . \prod_{c=1}^{c=M} \binom{N}{n_{r,c}} \right] \tag{3.44}$$

avec $x = (n_{r,1}, \ldots, n_{r,M})$.

3.4 Quorum en arbre général

3.4.1 Définition

Le *quorum en arbre général* est une adaptation de la première approche du quorum en arbre classique (voir section 2.2.3 de la partie II dans la page 57) pour pouvoir supporter plusieurs types de technique de distribution des données. Les répliques des données sont organisées, de manière logique, de façon à former un arbre de degré D et de hauteur h. Le degré d'un arbre est le nombre de nœuds fils attachés à chaque nœud de l'arbre (non compris les nœuds feuilles qui n'ont pas de fils) tandis que le hauteur de l'arbre correspond à la distance entre un nœud feuille de l'arbre et la racine. Cela veut donc dire qu'un arbre de hauteur h contient au total $h + 1$ niveaux. Ces niveaux sont numérotés de 0 à h en partant de la racine jusqu'à la feuille de l'arbre et le niveau 0 (appelé souvent la racine de l'arbre) ne contient qu'un seul nœud. Chaque nœud de l'arbre représente une réplique de la donnée comme montre la figure 3.2 avec $D = 3$ et

$h = 2$. Comme l'arbre est de degré D, chaque nœud de l'arbre (sauf ceux qui sont situés au niveau des feuilles) a D nœuds fils. Ainsi, le niveau l de l'arbre contient D^l répliques avec $0 \leq l \leq h$, c'est-à-dire au total l'arbre contient $n = 1+D+D^2+\ldots+D^h = \frac{1-D^{(h+1)}}{1-D}$ répliques. Pour tout $l \in \{0, 1, \ldots, h\}$, on note w_l le nombre de nœuds disponibles requis au niveau l de l'arbre afin de valider une opération d'écriture (avec $1 \leq w_l \leq D^l$). Il en résulte que la taille du quorum d'écriture est égale à :

$$w = w_0 + w_1 + \ldots + w_h \tag{3.45}$$

Soit r le nombre de répliques à jour requises pour la reconstruction des données originales en utilisant ce nouveau protocole. La taille du quorum d'écriture doit être supérieure ou égale à r c'est-à-dire : $w_0 + w_1 + \ldots + w_h \geq r$. Cette condition garantie que lorsque le *quorum en arbre général* est utilisé, au moins r répliques doivent être écrites correctement pour qu'une opération d'écriture soit valide. Ainsi, dès la validation d'une opération d'écriture d'une donnée, le protocole pourra lire cette dernière - la reconstruction de la donnée originale étant possible. Avec cette définition de quorum d'écriture, n'importe quelles $D^l - w_l + 1$ répliques se trouvant au niveau l permettent de déterminer la dernière version de la donnée. Par exemple, le protocole peut déjà déterminer la dernière version de la donnée lorsque le nœud qui se trouve à la racine de l'arbre (niveau $l = 0$) est disponible. Une fois connue la dernière version de la donnée, le protocole peut ensuite sélectionner r répliques avec cette version pour pouvoir reconstruire la donnée originale pour l'opération de lecture.

La figure 3.2 représente un exemple d'un arbre de degré $D = 3$ et de hauteur $h = 2$. Si nous utilisons une technique de distribution des données avec $r = 8$ alors une

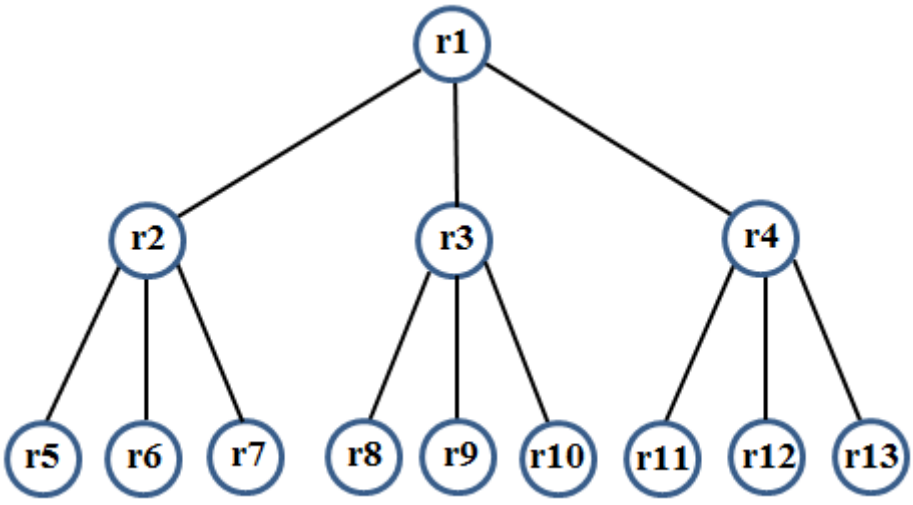


FIGURE 3.2 – Exemple de quorums en arbre : $D = 3$ et $h = 2$

réplique au niveau $l = 0$, deux répliques au niveau $l = 1$ et cinq répliques au niveau $l = 2$ (c'est-à-dire $w_0 = 1$, $w_1 = 2$ et $w_2 = 5$) forment un quorum d'écriture. Quant à l'opération de lecture dans cet exemple, avec cette définition de quorum d'écriture,

une $(D^0 - w_0 + 1)$ réplique au niveau $l = 0$ ou deux $(D^1 - w_1 + 1)$ répliques au niveau $l = 1$ ou cinq $(D^2 - w_2 + 1)$ répliques au niveau $l = 2$ permettent de déterminer la dernière version de la donnée. Ainsi, l'ensemble des répliques **{r1, r3, r4, r6, r8, r10, r11, r13}** forme un quorum d'écriture. De même, les ensembles de réplique(s) **{r1}**, **{r2, r4}** ou **{r5, r7, r9, r12, r13}** permettent chacun d'obtenir la dernière version de la donnée. Une fois cette version de la donnée connue, le protocole peut sélectionner r répliques à jour pour reconstruire la donnée originale.

3.4.2 Disponibilité en écriture

Avec ce protocole, une opération d'écriture sera validée uniquement si le protocole arrive à sélectionner $w = w_0 + w_1 + \ldots + w_h$ répliques (où w_l représente le nombre de répliques au niveau l pour tout $l \in \{0, 1, \ldots, h\}$). Pour $0 \leq l \leq h$, la probabilité pour qu'au moins w_l répliques soient disponibles au niveau l est égale à $\Phi_{D^l}(t, w_l, D^l)$ (Cf. la définition de Φ formule (3.2) page 67). Par conséquent, la disponibilité en écriture peut être exprimée comme suit :

$$\begin{aligned} P_{write} &= \prod_{l=0}^{l=h} \Phi_{D^l}(t, w_l, D^l) \\ &= \prod_{l=0}^{l=h} \left[\sum_{k=w_l}^{k=D^l} \left[\binom{D^l}{k} . t^k . (1-t)^{(D^l - k)} \right] \right] \end{aligned} \tag{3.46}$$

L'algorithme 5, dans la page 87, décrit une procédure pour former un quorum d'écriture avec le protocole quorum en arbre général. La procédure d'écriture d'une donnée est définie dans l'algorithme 9 page 105.

3.4.3 Disponibilité en lecture

En utilisant le *quorum en arbre général*, une opération de lecture est uniquement validée si les deux conditions suivantes sont satisfaites :

- ⇨ il existe un niveau l de l'arbre (avec $0 \leq l \leq h$) dans lequel au moins w_l nœuds sur un total de D^l nœuds sont disponibles (cette condition est nécessaire pour déterminer la dernière version de la donnée) ;
- ⇨ parmi les nœuds disponibles au moins r d'entre eux contiennent une réplique à jour de la donnée (cette condition est requise par le protocole de distribution des données pour la reconstruction de la donnée originale).

Une proposition de procédure pour former un quorum de lecture avec le protocole quorum en arbre général est présentée dans l'algorithme 6, page 88. La procédure de lecture d'une donnée est définie dans l'algorithme 10 page 106.

Nous allons adopter les notations suivantes :

Algorithm 5 Procédure de recherche d'un quorum d'écriture

```
1: Données
2: x                                                          ▷ Identité de la donnée à écrire
3: w_l                          ▷ Nombre de nœuds requis au niveau l lors d'une opération d'écriture
4: P_l                                                        ▷ Liste des nœuds se trouvant au niveau l
5: P ← {P_0, P_1, ..., P_h}                                   ▷ L'ensemble des nœuds utilisés pour le stockage de x

6: procedure getWriteQuorum(x, P)                             ▷ Recherche d'un quorum d'écriture
7:     Q ← ∅
8:     v ← 0
9:     test_version ← 0
10:    for l ← 0, h do
11:        compteur ← 0
12:        for all q ∈ P_l do
13:            v_i ← requête d'accès (x, q)        ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
14:            if estValide(v_i) then      ▷ estValide(v_i) retourne 1 si v_i est un numéro de version valide et 0 sinon
15:                if compteur < w_l then
16:                    Q ← Q ∪ {q}
17:                end if
18:                compteur ← compteur + 1
19:                if test_version = 0 then
20:                    if v_i > v then
21:                        v ← v_i
22:                    end if
23:                    if compteur = s_l − w_l + 1 then          ▷ La version de la donnée x est obtenue
24:                        test_version ← 1
25:                    end if
26:                end if
27:            end if
28:            if test_version ≠ 0 and |Q| = Σ_{j=0}^{j=l} w_j then          ▷ On passe au niveau suivant
29:                break
30:            end if
31:        end for
32:        if compteur < w_l then
33:            return NULL
34:        end if
35:    end for
36:    return [v + 1, Q]
37: end procedure
```

- ⇨ $n_{w,l}$: désigne le nombre de nœuds disponibles au niveau l de l'arbre durant la dernière opération d'écriture valide $(0 \leq l \leq h)$;
- ⇨ $n_{r,l}$: dénote le nombre de nœuds disponibles au niveau l de l'arbre durant l'opération de lecture en cours $(0 \leq l \leq h)$;
- ⇨ $n_{u,l}$: représente le nombre de nœuds disponibles au niveau l de l'arbre durant à la fois la dernière opération d'écriture valide et l'opération de lecture en cours $(0 \leq l \leq h)$. Il désigne également le nombre de répliques à jour de la donnée.

Soit la variable Z qui à chaque évènement élémentaire associe les valeurs des paramètres $n_{w,l}$, $n_{r,l}$ et $n_{u,l}$ de tous les niveaux de l'arbre. Cette variable aléatoire Z peut être présentée sous la forme d'une matrice $3 \times (h+1)$ (une matrice de 3 lignes et $h+1$ colonnes) où :

- ⇨ les éléments de la première ligne représentent le nombre de nœuds disponibles aux niveaux $l = 0, \ldots, l = h$ durant à la fois la dernière opération d'écriture valide et l'opération de lecture en cours ;
- ⇨ ceux de la deuxième ligne désignent le nombre de nœuds disponibles aux niveaux $l = 0, \ldots, l = h$ durant l'opération de lecture en cours ;
- ⇨ ceux de la troisième ligne représentent le nombre de nœuds disponibles aux niveaux $l = 0, \ldots, l = h$ durant la dernière opération d'écriture valide.

Algorithm 6 Procédure de recherche d'un quorum de lecture

```
1: Données
2: x                                                  ▷ Identité de la donnée à lire
3: h                                                  ▷ Hauteur de l'arbre
4: D^l                                                ▷ Nombre de nœuds résidant au niveau l de l'arbre
5: w_l                                                ▷ Nombre de nœuds requis au niveau l lors d'une opération d'écriture
6: P_l                                                ▷ Liste des nœuds se trouvant au niveau l
7: P ← {P_0, P_1, ..., P_h}                           ▷ L'ensemble des nœuds utilisés pour le stockage de x

8: procedure getReadQuorum(x, P)                      ▷ Recherche d'un quorum de lecture
9:     Q ← ∅
10:    v ← 0
11:    test_version ← 0
12:    for l ← 0, h do
13:        compteur ← 0
14:        for all q ← P_l do
15:            v_i ← requête d'accès (x, q)           ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
16:            if estValide(v_i) then                 ▷ estValide(v_i) retourne 1 si v_i est un numéro de version valide et 0 sinon
17:                compteur ← compteur + 1
18:                if test_version = 0 then
19:                    if compteur = D^l − w_l + 1 then   ▷ La version de la donnée est obtenue
20:                        test_version ← 1
21:                    end if
22:                    if v_i > v then
23:                        Q ← {q}
24:                    else if v_i = v and |Q| < r then
25:                        Q ← Q ∪ {q}
26:                    end if
27:                else if v_i = v and |Q| < r then
28:                    Q ← Q ∪ {q}
29:                end if
30:            end if
31:            if |Q| = r and test_version = 1 then   ▷ Le quorum de lecture est obtenu
32:                return [v, Q]
33:            end if
34:        end for
35:    end for
36:    return NULL
37: end procedure
```

On a donc :

$$Z = \begin{pmatrix} n_{u,0} & n_{u,1} & \dots & n_{u,h} \\ n_{r,0} & n_{r,1} & \dots & n_{r,h} \\ n_{w,0} & n_{w,1} & \dots & n_{w,h} \end{pmatrix} \tag{3.47}$$

Ainsi, pour tout $l \in \{0, 1, \dots, h\}$ nous avons : $Z(1, l+1) = n_{u,l}$, $Z(2, l+1) = n_{r,l}$ et $Z(3, l+1) = n_{w,l}$.

Lors d'une opération de lecture, pour tout $l \in \{0, 1, \dots, h\}$, les trois paramètres $n_{w,l}$, $n_{r,l}$ et $n_{u,l}$ peuvent prendre chacun des valeurs dans $\{0, 1, \dots, D^l\}$, c'est-à-dire que chaque paramètre a $1 + D^l$ valeurs possibles. Ainsi, le triplet $(n_{u,l}, n_{r,l}, n_{u,l})$ a $(1 + D^l)^3$ valeurs possibles. Comme l'arbre contient $h+1$ niveaux, alors la matrice Z peut prendre $\prod_{l=0}^{l=h}(1+D^l)^3$ valeurs possibles. Soient $z_1, z_2, \dots, z_{\prod_{l=0}^{l=h}(1+D^l)^3}$ toutes les valeurs possibles que peut prendre la matrice Z. Alors, pour tout $k \in \{1, 2, \dots, \prod_{l=0}^{l=h}(1 + D^l)^3\}$, la probabilité pour que la variable aléatoire Z prenne la valeur z est égale à :

$$\mathbb{P}(Z = z_k) = \prod_{l=0}^{l=h} \mathbb{P}\Big[|U| = z_k(1, l+1), |R| = z_k(2, l+1), |W| = z_k(3, l+1)\Big] \tag{3.48}$$

où U, R et W sont les trois variables aléatoires définies dans la section 3.1 page 67. La probabilité $\mathbb{P}(Z = z_k)$ dans la formule (3.48) est connue. En effet, la probabilité $\mathbb{P}\Big[|U| = z_k(1, l+1), |R| = z_k(2, l+1), |W| = z_k(3, l+1)\Big]$ a déjà été calculée précédemment (voir formule (3.8) dans la page 69). Ainsi, la disponibilité en lecture en utilisant ce protocole est égale à la somme des probabilités d'apparition des évènements $Z = z_k$ qui vérifient les conditions requises pour la validation d'une opération d'écriture. Pour tout $k \in \{1, 2, \ldots, \prod_{l=0}^{l=h}(1 + D^l)^3\}$, soit v_{z_k} un paramètre qui prend la valeur 1 lorsque l'évènement $Z = z_k$ permet de déterminer la dernière version de la donnée et la valeur 0 dans le cas contraire. Lorsque l'évènement $Z = z_k$ se présente, soit $\rho_{z_k,l}$ un paramètre qui prend la valeur 1 si le protocole peut récupérer la version de la donnée en utilisant les nœuds se trouvant au niveau l et sinon la valeur 0.

$$\rho_{z_k,l} = \begin{cases} 1 & \text{si } z_k(2, l+1) \geq D^l - w_l + 1 \\ 0 & \text{sinon} \end{cases} \tag{3.49}$$

Alors le paramètre v_{z_k} peut être exprimé fonction de $\rho_{z_k,l}$ pour l allant de 0 à h. En effet, le paramètre v_{z_k} prend la valeur 1 si il existe au moins un niveau $l \in \{0, 1, \ldots, h\}$ de l'arbre tel que $\rho_{z_k,l} = 1$ et sinon il prend la valeur 0. Par conséquent :

$$v_{z_k} = 1 - \prod_{l=0}^{l=h}(1 - \rho_{z_k,l}) \tag{3.50}$$

Nous allons définir par la suite le paramètre δ_{z_k} qui va permettre au protocole de décider si la donnée peut être reconstruite lorsque l'évènement $Z = z_k$ se présente. Ce paramètre δ_{z_k} prend la valeur 1 lorsque la donnée peut être reconstruite (c'est-à-dire lorsque le protocole peut déterminer la version de la donnée et si il existe au moins r nœuds disponibles contenant des répliques à jour de la donnée) et la valeur 0 dans le cas contraire. Ainsi,

$$\delta_{z_k} = \begin{cases} 1 & \text{si } v_{z_k} = 1 \text{ et } \sum_{l=0}^{l=h} z_k(1, l+1) \geq r \\ 0 & \text{sinon} \end{cases} \tag{3.51}$$

Par conséquent, la disponibilité en lecture en utilisant le protocole *quorum en arbre général* peut être exprimée comme suit :

$$P_{read} = \sum_{k=1}^{k=\prod_{l=0}^{l=h}(1+D^l)^3} \delta_{z_k}.\mathbb{P}(Z = z_k) \tag{3.52}$$

En remplaçant $\mathbb{P}(Z = z_k)$ par son expression définie dans la formule (3.48), page 88, nous obtenons :

$$P_{read} = \sum_{k=1}^{k=\prod_{l=0}^{l=h}(1+D^l)^3} \left[\delta_{z_k}. \prod_{l=0}^{l=h} \mathbb{P}\Big[|U| = z_k(1, l+1), |R| = z_k(2, l+1), |W| = z_k(3, l+1)\Big]\right] \tag{3.53}$$

Nous allons exprimer P_{read} en fonction des paramètres $n_{u,l}$, $n_{r,l}$ et $n_{w,l}$ pour tout $l \in \{0,\ldots,h\}$. En utilisant la formule (3.9) dans la page 69, nous avons :

$$\mathbb{P}\left(|U| = n_{u,l}, |R| = n_{r,l}, |W| = n_{w,l}\right) = \Psi_{D^l}(t, n_{r,l}).\Psi_{n_{r,l}}(t, n_{u,l}).\Psi_{D^l - n_{r,l}}(t, n_{w,l} - n_{u,l}).\xi_{n_{u,l},n_{r,l},n_{w,l}} \quad (3.54)$$

avec (en appliquant la formule (3.4) dans la page 68 dans un niveau l de l'arbre) :

$$\xi_{n_{u,l},n_{r,l},n_{w,l}} = \begin{cases} 1 & \text{si } n_{w,l} \in [max(w_l, n_{u,l}), D^l] \text{ et } n_{r,l} \in [n_{u,l}, D^l + n_{u,l} - n_{w,l}] \\ 0 & \text{sinon} \end{cases} \quad (3.55)$$

Or, par définition $U = R \cap W$. Nous avons donc $0 \leq n_{u,l} \leq n_{rw} = min(n_{r,l}, n_{w,l})$. Par ailleurs, lors de la dernière opération d'écriture valide, la condition suivante devait être remplie : $w_l \leq n_{w,l} \leq D^l$. En définissant les trois ensembles suivants :

- $E_u = \{(n_{u,0},\ldots,n_{u,h}) \mid \forall l \in \{0,\ldots,h\}\ 0 \leq n_{u,l} \leq min(n_{r,l}, n_{w,l})\}$;
- $E_r = \{(n_{r,0},\ldots,n_{r,h}) \mid \forall l \in \{0,\ldots,h\}\ 0 \leq n_{r,l} \leq D^l\}$;
- $E_w = \{(n_{w,0},\ldots,n_{w,h}) \mid \forall l \in \{0,\ldots,h\}\ w_l \leq n_{w,l} \leq D^l\}$.

l'expression de P_{read} dans la formule (3.53) est équivalente à :

$$P_{read} = \sum_{\substack{(n_{u,0},\ldots,n_{u,h}) \in E_u \\ (n_{r,0},\ldots,n_{r,h}) \in E_r \\ (n_{w,0},\ldots,n_{w,h}) \in E_w}} \left[\delta_x . \prod_{l=0}^{l=h} \mathbb{P}\Big[|U| = n_{u,l}, |R| = n_{r,l}, |W| = n_{w,l}\Big]\right] \quad (3.56)$$

avec $x = \begin{pmatrix} n_{u,0} & n_{u,1} & \ldots & n_{u,h} \\ n_{r,0} & n_{r,1} & \ldots & n_{r,h} \\ n_{w,0} & n_{w,1} & \ldots & n_{w,h} \end{pmatrix}$. Ainsi, en remplaçant $\mathbb{P}\Big[|U| = n_{u,l}, |R| = n_{r,l}, |W| = n_{w,l}\Big]$ par son expression dans la formule (3.54), la disponibilité en lecture est égale à :

$$P_{read} = \sum_{\substack{(n_{u,0},\ldots,n_{u,h}) \in E_u \\ (n_{r,0},\ldots,n_{r,h}) \in E_r \\ (n_{w,0},\ldots,n_{w,h}) \in E_w}} \left[\delta_x . \prod_{l=0}^{l=h} \left[\Psi_{D^l}(t, n_{r,l}).\Psi_{n_{r,l}}(t, n_{u,l}).\Psi_{D^l - n_{r,l}}(t, n_{w,l} - n_{u,l}).\xi_{n_{u,l},n_{r,l},n_{w,l}}\right]\right] \quad (3.57)$$

3.4.4 Nombre moyen de messages échangés

Pour le calcul du nombre moyen de messages échangés pour l'opération d'écriture et de lecture, nous allons définir la variable aléatoire X qui à chaque évènement élémentaire, associe le h-uplet formé par le nombre de nœuds disponibles dans chaque niveau de l'arbre. Ainsi, la variable aléatoire X correspond respectivement à $(n_{w,0}, n_{w,1}, \ldots, n_{w,h})$ lors d'une opération d'écriture et à $(n_{r,0}, n_{r,1}, \ldots, n_{r,h})$ lors d'une opération de lecture.

Nous allons commencer par le cas où l'évènement élémentaire correspond à une opération d'écriture. Pour tout $l \in \{0, 1, \ldots, h\}$, nous avons $0 \leq n_{w,l} \leq D^l$ c'est-à-dire qu'il existe $1 + D^l$ valeurs possibles pour $n_{w,l}$. Alors, le cardinal de l'ensemble des valeurs que peut prendre la variable aléatoire X est égal à $\prod_{l=0}^{l=h}(1 + D^l)$. Soient $x_1, x_2, \ldots, x_{\prod_{l=0}^{l=h}(1+D^l)}$ toutes les éventualités possibles de la variable aléatoire X. Pour tout $l \in \{0, 1, \ldots, h\}$, $x_k(l)$ correspond à $n_{w,l}$. Alors, pour tout $k \in \{1, 2, \ldots, \prod_{l=0}^{l=h}(1 + D^l)\}$, l'éventualité $\{X = x_k\}$ se présente avec une probabilité égale à :

$$\begin{aligned}\mathbb{P}(X = x_k) &= \prod_{l=0}^{l=h}\left[\binom{D^l}{x_k(l)}.t^{x_k(l)}.(1-t)^{D^l - x_k(l)}\right] \\ &= t^{\left[\sum_{l=0}^{l=h} x_k(l)\right]}.(1-t)^{\left[\sum_{l=0}^{l=h}\left[D^l - x_k(l)\right]\right]}.\prod_{l=0}^{l=h}\binom{D^l}{x_k(l)} \end{aligned} \tag{3.58}$$

Soit $\sigma_{x_k,l}$ un paramètre qui représente l'état de l'opération d'écriture dans le niveau l (il prend la valeur 1 si la condition requise pour la validation d'une tentative d'opération d'écriture est satisfaite, la valeur 0 sinon). on a :

$$\sigma_{x_k,l} = \begin{cases} 1 & \text{si } x_k(l) \geq w_l \\ 0 & \text{sinon} \end{cases} \tag{3.59}$$

Nous allons définir ensuite le paramètre $\alpha_{x_k,l}$ qui représente l'état de l'opération d'écriture après avoir visité les niveaux $\{0, 1, \ldots, l\}$ et le paramètre μ_{x_k} qui représente le nombre de messages échangés lors d'une opération d'écriture lorsque l'éventualité $\{X = x_k\}$ se présente (avec $1 \leq k \leq \prod_{l=0}^{l=h}(1+D^l)$ et $0 \leq l \leq h$). Le paramètre $\alpha_{x_k,l}$ prend la valeur 1 si les niveaux $\{0, 1, \ldots, l\}$ vérifient chacun la condition requise pour la validation d'une tentative d'opération d'écriture (autrement dit $\sigma_{x_k,i} = 1\ \forall i \in \{0, 1, \ldots, l\}$). Donc,

$$\alpha_{x_k,l} = \prod_{i=0}^{i=l}\sigma_{x_k,i} \tag{3.60}$$

Sans perte de généralité, nous allons supposer que les l premiers niveaux visités lors d'une opération d'écriture sont les niveaux $\{0, 1, \ldots, l-1\}$, le niveau suivant étant le niveau l. Lors du passage au niveau l, les différents messages échangés à ce niveau de l'arbre sont :

- ⇨ $\alpha_{x_k,l}.D^l$ messages de requête d'écriture, pour récupérer les anciennes répliques résidant au niveau l de l'arbre (le protocole envoie les D^l messages de requête d'écriture si et seulement si cette opération a été validée dans les l premiers niveaux visités, ce qui correspond à $\alpha_{x_k,l} = 1$) ;
- ⇨ $\alpha_{x_k,l}.x_k(l)$ messages de réponse venant des nœuds disponibles (si les D^l messages de requête d'écriture ont été envoyés) ;

- ⇨ $\alpha_{x_k,h}.x_k(l)$ messages d'envoi des nouvelles répliques de la donnée vers les nœuds disponibles, c'est-à-dire que la donnée ne sera écrite que si la condition suivante est remplie : $\forall i \in \{0,1,\ldots,h\}$, au moins w_l nœuds sont disponibles au niveau i de l'arbre lors de cette opération.

Ainsi, le nombre de messages échangés lors d'une opération d'écriture lorsque l'éventualité $\{X = x_k\}$ se présente est égal à :

$$\begin{aligned}\mu_{x_k} &= \sum_{l=0}^{l=h}\Big[\alpha_{x_k,l}.D^l + \alpha_{x_k,l}.x_k(l) + \alpha_{x_k,h}.x_k(l)\Big] \\ &= \sum_{l=0}^{l=h}\Big[\alpha_{x_k,l}.D^l + \alpha_{x_k,l}.x_k(l) + \underbrace{\alpha_{x_k,l}.\alpha_{x_k,h}}_{=\alpha_{x_k,h}}.x_k(l)\Big] \\ &= \sum_{l=0}^{l=h}\left[\alpha_{x_k,l}.\Big[D^l + (1+\alpha_{x_k,h}).x_k(l)\Big]\right] \end{aligned} \tag{3.61}$$

Le nombre moyen de messages échangés est égal à :

$$N_{write} = \sum_{k=1}^{k=\prod_{l=0}^{l=h}(1+D^l)} \mu_{x_k}.\mathbb{P}(X = x_k)$$

En remplaçant $\mathbb{P}(X = x_k)$ par son expression dans la formule (3.58), nous obtenons :

$$N_{write} = \sum_{k=1}^{k=\prod_{l=0}^{l=h}(1+D^l)} \left[\mu_{x_k}.t^{\left[\sum_{l=0}^{l=h} x_k(l)\right]}.(1-t)^{\left[\sum_{l=0}^{l=h}\left[D^l - x_k(l)\right]\right]}.\prod_{l=0}^{l=h}\binom{D^l}{x_k(l)}\right] \tag{3.62}$$

Sachant que :

$$\{x_k \mid k \in \{1,2,\ldots,\prod_{l=0}^{l=h}(1+D^l)\} = \{(n_{w,0},\ldots,n_{w,h}) \mid \forall l \in \{0,1,\ldots,h\}, 0 \leq n_{w,l} \leq D^l\}$$

l'expression de N_{write} peut être exprimée en fonction des $n_{w,l}$ (pour l allant de 0 à h) :

$$N_{write} = \sum_{\substack{0\leq n_{w,0}\leq D^0 \\ \vdots \\ 0\leq n_{w,h}\leq D^h}} \left[\mu_x.t^{\left[\sum_{l=0}^{l=h} n_{w,l}\right]}.(1-t)^{\left[\sum_{l=0}^{l=h}\left[D^l - n_{w,l}\right]\right]}.\prod_{l=0}^{l=h}\binom{D^l}{n_{w,l}}\right] \tag{3.63}$$

avec $x = (n_{w,0},\ldots,n_{w,h})$.

Nous allons maintenant déterminer l'expression du nombre moyen de messages échangés lors d'une opération de lecture. Pour cela nous avons besoin d'utiliser la matrice Z définie page 88. Lorsque l'évènement $\{Z = z_k\}$ se présente, nous allons définir les cinq paramètres suivants : $\gamma_{z_k,l}$, $\upsilon_{z_k,l}$, $\theta_{z_k,l}$, $\lambda_{z_k,l}$ et $\delta_{z_k,l}$.

- ⇨ $\gamma_{z_k,l}$: permet de déterminer si la version de la donnée peut être obtenue ou non en utilisant uniquement les nœuds résidant au niveau l de l'arbre :

$$\gamma_{z_k,l} = \begin{cases} 1 & \text{si } z_k(2,l+1) \geq D^l - w_l + 1 \\ 0 & \text{sinon} \end{cases} \tag{3.64}$$

  Ainsi, il retourne 1 si le niveau l de l'arbre permet de déterminer la version de la donnée, la valeur 0 sinon ;
- ⇨ $\upsilon_{x_k,l}$: permet de déterminer si la version de la donnée peut être obtenue ou non en utilisant l'un des niveaux $\{0,1,\ldots,l\}$ de l'arbre. Ce paramètre peut être exprimé en fonction de $\gamma_{z_k,l}$:

$$\upsilon_{z_k,l} = 1 - \prod_{i=0}^{i=l}(1-\gamma_{z_k,l}) \tag{3.65}$$

  Il retourne 1 si le protocole peut récupérer la version de la donnée, la valeur 0 dans le cas contraire ;
- ⇨ $\theta_{x_k,l}$: permet de déterminer si le nombre de nœuds disponibles contenant des répliques à jour de la donnée aux niveaux $\{0,1,\ldots,l\}$ de l'arbre est supérieur ou égal à r (nombre de répliques à jour de la donnée requises par la technique de distribution des données pour reconstruire la donnée originale) :

$$\theta_{z_k,l} = \begin{cases} 1 & \text{si } \sum\limits_{i=0}^{i=l} z_k(1,l+1) \geq r \\ 0 & \text{sinon} \end{cases} \tag{3.66}$$

  Il retourne 1 si ce nombre est supérieur à r et sinon 0 ;
- ⇨ $\lambda_{z_k,l}$: permet de savoir si les nœuds se trouvant aux niveaux $\{0,1,\ldots,l\}$ de l'arbre permettent de reconstituer la donnée originale, c'est-à-dire d'obtenir la version de la donnée et d'avoir au moins r répliques à jour :

$$\lambda_{z_k,l} = \begin{cases} 1 & \text{si } \upsilon_{z_k,l} = 1 \text{ et } \theta_{z_k,l} = 1 \\ 0 & \text{sinon} \end{cases} \tag{3.67}$$

  Ce paramètre retourne 1 si la donnée originale peut être reconstituée, 0 sinon ;
- ⇨ $\delta_{z_k,l}$: permet de déterminer le premier niveau de l'arbre qui vérifie $\lambda_{z_k,l} = 1$ pour l variant de 0 à h, c'est-à-dire le plus petit l tel que les niveaux $\{0,1,\ldots,l\}$ de l'arbre sont suffisants pour lire la donnée :

$$\delta_{z_k,l} = \begin{cases} \lambda_{z_k,0} & \text{si } l = 0 \\ (1-\lambda_{z_k,l-1}).\lambda_{z_k,l} & \text{si } 1 \leq l \leq h \end{cases} \tag{3.68}$$

  Ce paramètre retourne 1 uniquement pour le premier niveau de l'arbre, si il existe, qui vérifie $\lambda_{z_k,l} = 1$ et 0 pour le reste.

Lorsque l'évènement $\{Z = z_k\}$ se présente, soit β_{z_k} le nombre de messages échangés lors d'une opération de lecture (avec $1 \leq k \leq \prod\limits_{l=0}^{l=h}(1+D^l)$). Notons $l_{z_k,min}$ le seul niveau de l'arbre qui vérifie $\delta_{z_k,l_{z_k,min}} = 1$. Nous avons :

$$1 - \lambda_{z_k,l} + \delta_{z_k,l} = \begin{cases} 1 & \text{si } 0 \leq l \leq l_{z_k,min} \\ 0 & \text{si } l_{z_k,min} \leq l \leq h \end{cases}$$

Alors, les différents messages échangés peuvent être exprimés comme suit :

- ☞ $(1-\lambda_{z_k,l}+\delta_{z_k,l}).D^l$ messages de requête de lecture. Le terme $(1-\lambda_{z_k,l}+\delta_{z_k,l})$ impose que la requête de lecture s'arrête au niveau $l_{z_k,min}$;
- ☞ $(1-\lambda_{z_k,l}+\delta_{z_k,l}).z_k(2,l+1)$ messages de réponse venant des nœuds disponibles du niveau l.

Ainsi, l'expression de β_{z_k} est égale à :

$$\begin{aligned}\beta_{z_k} &= \sum_{l=0}^{l=h}\Big[(1-\lambda_{z_k,l}+\delta_{z_k,l}).D^l + (1-\lambda_{z_k,l}+\delta_{z_k,l}).z_k(2,l+1)\Big] \\ &= \sum_{l=0}^{l=h}\Big[(1-\lambda_{z_k,l}+\delta_{z_k,l})(D^l + z_k(2,l+1))\Big] \end{aligned} \tag{3.69}$$

Le nombre moyen de messages échangés lors d'une opération de lecture est donc défini comme suit :

$$N_{read} = \sum_{k=1}^{k=\prod_{l=0}^{l=h}(1+D^l)^3} \beta_{z_k}.\mathbb{P}(Z=z_k)$$

En se référant aux formules (3.52) et (3.57), l'expression de N_{read} est égale à :

$$N_{read} = \sum_{\substack{(n_{u,0},\ldots,n_{u,h})\in E_u \\ (n_{r,0},\ldots,n_{r,h})\in E_r \\ (n_{w,0},\ldots,n_{w,h})\in E_w}} \left[\beta_x.\prod_{l=0}^{l=h}\Big[\Psi_{D^l}(t,n_{r,l}).\Psi_{n_{r,l}}(t,n_{u,l}).\Psi_{D^l-n_{r,l}}(t,n_{w,l}-n_{u,l}).\xi_{n_{u,l},n_{r,l},n_{w,l}}\Big]\right] \tag{3.70}$$

avec $x = \begin{pmatrix} n_{u,0} & n_{u,1} & \ldots & n_{u,h} \\ n_{r,0} & n_{r,1} & \ldots & n_{r,h} \\ n_{w,0} & n_{w,1} & \ldots & n_{w,h} \end{pmatrix}$.

3.5 Quorum en trapèze général

3.5.1 Définition

Le *quorum en trapèze général*, que nous avons présenté dans [87], est une généralisation du quorum en trapèze classique afin de supporter plusieurs types de techniques de distribution des données. Comme dans le quorum en trapèze classique, les répliques de la donnée sont organisées, de façon logique, de manière à former un trapèze de hauteur $(h+1)$. Le trapèze est ainsi constitué de $h+1$ niveaux, numérotés de 0 à h (de haut vers bas). Pour tout $l \in \{0,1,\ldots,h\}$, le niveau l du trapèze contient $s_l = a.l+b$ nœuds où a et b sont deux entiers naturels (avec $a \geq 0$ et $b \geq 1$). Ainsi, le trapèze contient au total $n = \sum_{l=1}^{l=h} s_l = (h+1)\left[\frac{ah+2b}{2}\right]$ répliques. Pour tout $l \in \{0,1,\ldots,h\}$, soit w_l le nombre de nœuds disponibles requis au niveau l du trapèze pour que l'opération d'écriture soit validée à ce niveau (avec $1 \leq w_l \leq s_l$). La taille du quorum d'écriture est alors égale à :

$$w = w_0 + w_1 + \ldots + w_h \tag{3.71}$$

Soit r le nombre de répliques à jour requises pour pouvoir reconstruire la donnée originale en utilisant le quorum en arbre général. Pour qu'une opération de lecture puisse être faite juste après la validation d'une opération d'écriture, la taille du quorum d'écriture doit être supérieure ou égale au nombre de répliques à jour requises pour la reconstruction de la donnée originale, c'est-à-dire $w_0 + w_1 + \ldots + w_h \geq r$. La dernière version de la donnée peut être obtenue avec n'importe quelles $s_l - w_l + 1$ répliques résidant au niveau l (avec $0 \leq l \leq h$). Après avoir déterminé la dernière version de la donnée, le protocole procédera ensuite à la sélection de r répliques à jour pour reconstituer la donnée originale.

La figure 3.3 représente un exemple d'un trapèze de paramètres : $a = 2$, $b = 3$ et $h = 2$. Si nous utilisons une technique de distribution des données avec $r = 8$ alors deux

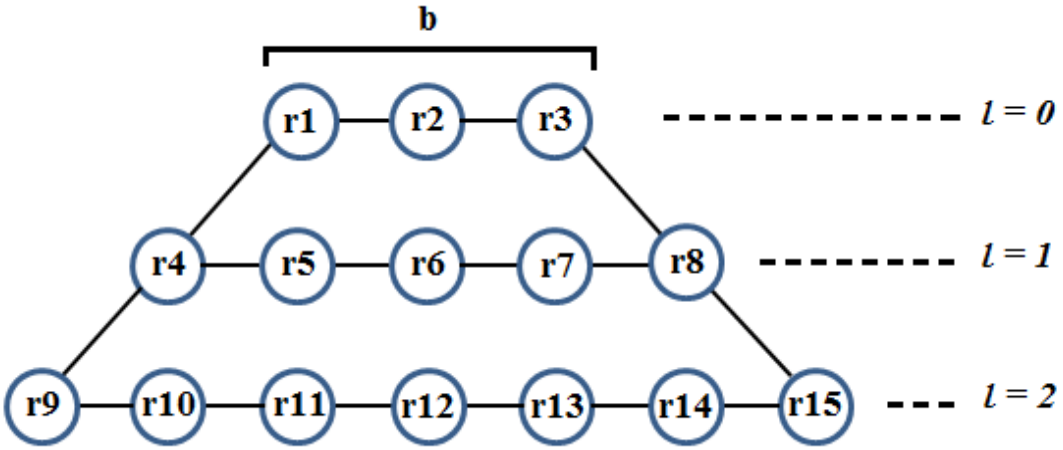


FIGURE 3.3 – Exemple de quorums en trapèze : $a = 2$, $b = 3$ et $h = 2$

répliques au niveau $l = 0$, trois répliques à chacun des niveaux $l = 1$ et $l = 2$ (c'est-à-dire $w_0 = 2$, $w_1 = 3$ et $w_2 = 3$) peuvent former un quorum d'écriture. Ainsi, l'ensemble des répliques **{r1, r3, r5, r6, r8, r10, r12, r15}** forme un quorum d'écriture avec cet exemple. Avec ce quorum d'écriture, deux $(s_0 - w_0 + 1)$ répliques résidant au niveau $l = 0$ ou trois $(s_1 - w_1 + 1)$ répliques résidant au niveau $l = 1$ ou cinq $(s_2 - w_2 + 1)$ répliques résidant au niveau $l = 2$ permettent de déterminer la dernière version de la donnée. A titre d'exemple, chacun des ensembles de répliques suivants permettront au protocole de retrouver la dernière version de la donnée : **{r1, r2}**, **{r4, r7, r8}** et **{r9, r11, r12, r13, r14}**. Quand la dernière version de la donnée est obtenue, le protocole n'a plus qu'à sélectionner r répliques à jour pour reconstituer la donnée originale.

3.5.2 Disponibilité en écriture

En utilisant le *quorum en trapèze général*, la validation d'une opération d'écriture nécessite la sélection d'au moins $w = w_0 + w_1 + \ldots + w_h$ répliques, w_l représentant, pour tout $l \in \{0, 1, \ldots, h\}$, le nombre de répliques choisies au niveau l. Pour $0 \leq l \leq h$, la

probabilité de trouver au moins w_l nœuds disponibles au niveau l est égale à $\Phi_{s_l}(t, w_l, s_l)$ (où Φ est définie selon la formule (3.2) page 67). Par conséquent, la disponibilité en écriture est égale à :

$$\begin{aligned} P_{write} &= \prod_{l=0}^{l=h} \Phi_{s_l}(t, w_l, s_l) \\ &= \prod_{l=0}^{l=h} \left[\sum_{k=w_l}^{k=s_l} \left[\binom{s_l}{k} . t^k . (1-t)^{(s_l-k)} \right] \right] \end{aligned} \tag{3.72}$$

Nous proposons dans l'algorithme 7 suivant une procédure pour former un quorum d'écriture avec le protocole quorum en trapèze général. La procédure d'écriture d'une donnée est définie dans l'algorithme 9 page 105.

Algorithm 7 Procédure de recherche d'un quorum d'écriture

Données
x ▷ Identité de la donnée à écrire
w_l ▷ Nombre de nœuds requis au niveau l lors d'une opération d'écriture
P_l ▷ Liste des nœuds se trouvant au niveau l
$P \leftarrow \{P_0, P_1, \ldots, P_h\}$ ▷ L'ensemble des nœuds utilisés pour le stockage de x

procedure GETWRITEQUORUM(x, P) ▷ Recherche d'un quorum d'écriture
    $Q \leftarrow \emptyset$
    $v \leftarrow 0$
    $test\_version \leftarrow 0$
    for $l \leftarrow 0, h$ **do**
        $compteur \leftarrow 0$
        for all $q \in P_l$ **do**
            $v_i \leftarrow$ requête d'accès (x, q) ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
            if estValide(v_i) **then** ▷ estValide(v_i) retourne 1 si v_i est un numéro de version valide et 0 sinon
                if $compteur < w_l$ **then**
                    $Q \leftarrow Q \cup \{q\}$
                end if
                $compteur \leftarrow compteur + 1$
                if $test\_version = 0$ **then**
                    if $v_i > v$ **then**
                        $v \leftarrow v_i$
                    end if
                    if $compteur = s_l - w_l + 1$ **then** ▷ La version de la donnée est obtenue
                        $test\_version \leftarrow 1$
                    end if
                end if
            end if
            if $test\_version \neq 0$ **and** $|Q| = \sum_{j=0}^{j=l} w_j$ **then** ▷ On passe au niveau suivant
                break
            end if
        end for
        if $compteur < w_l$ **then**
            return $NULL$
        end if
    end for
    return $[v + 1, Q]$
end procedure

3.5.3 Disponibilité en lecture

En utilisant le *quorum en trapèze général*, une opération de lecture est validée si et seulement si les deux conditions suivantes sont remplies :

- ⇨ il existe un niveau l du trapèze (avec $0 \leq l \leq h$) dans lequel au moins w_l nœuds sur un total de s_l nœuds sont disponibles (cette condition est nécessaire pour déterminer la dernière version de la donnée) ;
- ⇨ au moins r nœuds disponibles contiennent chacun une réplique à jour de la donnée (cette condition est requise par le protocole de distribution des données pour la reconstruction de la donnée originale).

Nous détaillons dans l'algorithme 8 suivant une procédure pour former un quorum de lecture avec le protocole quorum en trapèze général. La procédure de lecture d'une donnée est définie dans l'algorithme 10 page 106.

Algorithm 8 Procédure de recherche d'un quorum de lecture

```
1: Données
2: x                                              ▷ Identité de la donnée à lire
3: h                                              ▷ Hauteur du trapèze
4: s_l                                            ▷ Nombre de nœuds résidant au niveau l du trapèze
5: w_l                                            ▷ Nombre de nœuds requis au niveau l lors d'une opération d'écriture
6: P_l                                            ▷ Liste des nœuds se trouvant au niveau l
7: P ← {P_0, P_1, ..., P_h}                       ▷ L'ensemble des nœuds utilisés pour le stockage de x

8: procedure getReadQuorum(x, P)                  ▷ Recherche d'un quorum de lecture
9:     Q ← ∅
10:    v ← 0
11:    test_version ← 0
12:    for l ← 0, h do
13:        compteur ← 0
14:        for all q ← P_l do
15:            v_i ← requête d'accès (x, q)       ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
16:            if estValide(v_i) then             ▷ estValide(v_i) retourne 1 si v_i est un numéro de version valide et 0 sinon
17:                compteur ← compteur + 1
18:                if test_version = 0 then
19:                    if v_i > v then            ▷ Une version plus récente
20:                        Q ← ∅
21:                    end if
22:                    if compteur = s_l − w_l + 1 then   ▷ La version de la donnée est obtenue
23:                        test_version ← 1
24:                    end if
25:                end if
26:                if v_i >= v and |Q| < r then
27:                    Q ← Q ∪ {q}
28:                end if
29:            end if
30:            if |Q| = r and test_version = 1 then   ▷ Un quorum de lecture est obtenu
31:                return [v, Q]
32:            end if
33:        end for
34:    end for
35:    return NULL
36: end procedure
```

Nous allons adopter les notations suivantes :

- ⇨ $n_{w,l}$: désigne le nombre de nœuds disponibles au niveau l du trapèze durant la dernière opération d'écriture valide ($0 \leq l \leq h$) ;
- ⇨ $n_{r,l}$: dénote le nombre de nœuds disponibles au niveau l du trapèze durant l'opération de lecture en cours ($0 \leq l \leq h$) ;
- ⇨ $n_{u,l}$: représente le nombre de nœuds disponibles au niveau l du trapèze durant à la fois la dernière opération d'écriture valide et l'opération de lecture en cours ($0 \leq l \leq h$). Il désigne également le nombre de répliques à jour de la donnée.

Soit la variable aléatoire Z qui à chaque évènement élémentaire associe les valeurs des paramètres $n_{w,l}$, $n_{r,l}$ et $n_{u,l}$ de tous les niveaux du trapèze. Cette variable aléatoire peut

être représentée sous la forme d'une matrice $3 \times (h+1)$ comme suit :

$$Z = \begin{pmatrix} n_{u,0} & n_{u,1} & \dots & n_{u,h} \\ n_{r,0} & n_{r,1} & \dots & n_{r,h} \\ n_{w,0} & n_{w,1} & \dots & n_{w,h} \end{pmatrix} \tag{3.73}$$

Pour tout $l \in \{0, 1, \dots, h\}$, ces trois paramètres sont régis par les trois conditions suivantes : $w_l \leq n_{w,l} \leq s_l$, $0 \leq n_{r,l} \leq s_l$ et $0 \leq n_{u,l} \leq min(n_{r,l}, n_{w,l})$. Nous pouvons, par ailleurs, définir les trois ensembles suivants :

- $E_u = \{(n_{u,0}, \dots, n_{u,h}) \mid \forall l \in \{0, \dots, h\}\ 0 \leq n_{u,l} \leq min(n_{r,l}, n_{w,l})\}$;
- $E_r = \{(n_{r,0}, \dots, n_{r,h}) \mid \forall l \in \{0, \dots, h\}\ 0 \leq n_{r,l} \leq s_l\}$;
- $E_w = \{(n_{w,0}, \dots, n_{w,h}) \mid \forall l \in \{0, \dots, h\}\ w_l \leq n_{w,l} \leq s_l\}$.

Soient $e_u \in E_u$, $e_r \in E_r$ et $e_w \in E_w$. Nous allons calculer la probabilité pour que l'évènement $\{Z = z\}$ se présente lors de cette opération de lecture avec $z = \begin{pmatrix} e_u \\ e_r \\ e_w \end{pmatrix}$.

$$\begin{aligned} \mathbb{P}(Z = z) &= \prod_{l=0}^{l=h} \mathbb{P}\Big[|U| = e_u(l), |R| = e_r(l), |W| = e_w(l)\Big] \\ &= \prod_{l=0}^{l=h} \mathbb{P}\Big[|U| = n_{u,l}, |R| = n_{r,l}, |W| = n_{w,l}\Big] \end{aligned} \tag{3.74}$$

où U, R et W sont les trois variables aléatoires définies section 3.1 page 67. En utilisant la formule (3.9) de la page 69, nous avons :

$$\begin{aligned} &\mathbb{P}\left(|U| = n_{u,l}, |R| = n_{r,l}, |W| = n_{w,l}\right) = \\ &\qquad \Psi_{s_l}(t, n_{r,l}).\Psi_{n_{r,l}}(t, n_{u,l}).\Psi_{s_l - n_{r,l}}(t, n_{w,l} - n_{u,l}).\xi_{n_{u,l}, n_{r,l}, n_{w,l}} \end{aligned} \tag{3.75}$$

avec (en appliquant la formule (3.4) de la page 68 à un niveau l du trapèze)

$$\xi_{n_{u,l}, n_{r,l}, n_{w,l}} = \begin{cases} 1 & \text{si } n_{w,l} \in [max(w_l, n_{u,l}), s_l] \text{ et } n_{r,l} \in [n_{u,l}, s_l + n_{u,l} - n_{w,l}] \\ 0 & \text{sinon} \end{cases} \tag{3.76}$$

En remplaçant $\mathbb{P}\Big[|U| = n_{u,l}, |R| = n_{r,l}, |W| = n_{w,l}\Big]$ par son expression, la formule (3.74) devient :

$$\mathbb{P}(Z = z) = \prod_{l=0}^{l=h} \Big[\Psi_{s_l}(t, n_{r,l}).\Psi_{n_{r,l}}(t, n_{u,l}).\Psi_{s_l - n_{r,l}}(t, n_{w,l} - n_{u,l}).\xi_{n_{u,l}, n_{r,l}, n_{w,l}}\Big] \tag{3.77}$$

Nous allons définir les trois paramètres suivants qui vont permettre au protocole de décider si l'évènement $\{Z = z\}$ (avec $z = \begin{pmatrix} e_u \\ e_r \\ e_w \end{pmatrix}$) permet ou non de retrouver la donnée originale :

- ⇨ ρ_{l,e_r} : indique si le niveau l du trapèze permet de déterminer la version de la donnée :

$$\rho_{l,e_r} = \begin{cases} 1 & \text{si } e_r(l) \geq s_l - w_l + 1 \\ 0 & \text{sinon} \end{cases} \tag{3.78}$$

  Ce paramètre prend la valeur 1 si le protocole peut déterminer la version de la donnée en utilisant uniquement les nœuds se trouvant au niveau l, la valeur 0 sinon ;
- ⇨ v_{e_r} : indique si la version de la donnée peut être obtenue au moins à un niveau du trapèze. Ce paramètre peut être exprimé en fonction de ρ_{l,e_r}pour l allant de 0 à h :

$$v_{e_r} = 1 - \prod_{l=0}^{l=h} (1 - \rho_{l,e_r}) \tag{3.79}$$

  Il prend la valeur 1 si la version de la donnée peut être obtenue, 0 sinon ;
- ⇨ δ_{e_u,e_r} : indique si la donnée originale peut être reconstruite :

$$\delta_{e_u,e_r} = \begin{cases} 1 & \text{si } v_{e_r} = 1 \text{ et } \sum_{l=0}^{l=h} e_u(l) \geq r \\ 0 & \text{sinon} \end{cases} \tag{3.80}$$

  Ainsi, δ_{e_u,e_r} prend la valeur 1 si la donnée originale peut être reconstruite, la valeur 0 sinon.

La disponibilité en lecture peut donc être exprimée comme suit :

$$P_{read} = \sum_{\substack{e_u \in E_u \\ e_r \in E_r \\ e_w \in E_w}} \Big[\delta_{e_u,e_r} . \mathbb{P}(Z = z) \Big] \tag{3.81}$$

Avec $z = \begin{pmatrix} e_u \\ e_r \\ e_w \end{pmatrix}$. En remplaçant $\mathbb{P}(Z = z)$ par son expression dans la formule (3.77), nous obtenons :

$$P_{read} = \sum_{\substack{e_u \in E_u \\ e_r \in E_r \\ e_w \in E_w}} \left[\delta_{e_u,e_r} . \prod_{l=0}^{l=h} \Big[\Psi_{s_l}(t, n_{r,l}) . \Psi_{n_{r,l}}(t, n_{u,l}) . \Psi_{s_l - n_{r,l}}(t, n_{w,l} - n_{u,l}) . \xi_{n_{u,l}, n_{r,l}, n_{w,l}} \Big] \right] \tag{3.82}$$

Avec $e_u = (n_{u,0}, \ldots, n_{u,h})$, $e_r = (n_{r,0}, \ldots, n_{r,h})$ et $e_w = (n_{w,0}, \ldots, n_{w,h})$.

3.5.4 Nombre moyen de messages échangés

Dans cette sous-partie, nous allons calculer le nombre moyen de messages pour les opérations d'écriture et de lecture. Soit $X = (n_{w,0}, \ldots, n_{w,h})$, une variable aléatoire

qui, à chaque opération d'écriture, associe le h-uplet formé par le nombre de nœuds disponibles à chaque niveau du trapèze. Concrètement, lors de cette tentative d'opération d'écriture, pour tout $l \in \{0, 1, \ldots, h\}$, $n_{w,l}$ nœuds sont disponibles au niveau l du trapèze, avec $0 \leq n_{w,l} \leq s_l$. Le cardinal de l'ensemble des valeurs que peut prendre la variable aléatoire X égal à $\prod_{l=0}^{l=h}(1+s_l)$. Soient $x_1, x_2, \ldots, x_{\prod_{l=0}^{l=h}(1+s_l)}$ toutes les éventualités possibles de la variable aléatoire X. Alors, pour tout $k \in \{1, 2, \ldots, \prod_{l=0}^{l=h}(1+s_l)\}$, l'éventualité $\{X = x_k\}$ se présente avec une probabilité égale à :

$$\begin{aligned}\mathbb{P}(X = x_k) &= \prod_{l=0}^{l=h}\left[\binom{s_l}{x_k(l)}.t^{x_k(l)}.(1-t)^{s_l - x_k(l)}\right] \\ &= t^{\left[\sum_{l=0}^{l=h} x_k(l)\right]}.(1-t)^{\left[\sum_{l=0}^{l=h}[s_l - x_k(l)]\right]}.\prod_{l=0}^{l=h}\binom{s_l}{x_k(l)} \end{aligned} \tag{3.83}$$

Nous allons chercher à déterminer le nombre de messages échangés lors de chaque éventualité pour, *in fine*, calculer le nombre de messages moyen. Nous allons noter μ_{x_k} le nombre de messages échangés lors d'une opération d'écriture lorsque l'éventualité $\{X = x_k\}$ se présente. Soit $\sigma_{x_k,l}$ un paramètre qui représente l'état de l'opération d'écriture au niveau l (il prend la valeur 1 si la condition requise pour la validation d'une opération d'écriture est remplie, la valeur 0 sinon) :

$$\sigma_{x_k,l} = \begin{cases} 1 & \text{si } x_k(l) \geq w_l \\ 0 & \text{sinon} \end{cases} \tag{3.84}$$

Nous allons définir ensuite le paramètre $\alpha_{x_k,l}$ qui représente l'état de l'opération d'écriture après avoir visité les niveaux $\{0, 1, \ldots, l\}$ et le paramètre μ_{x_k} qui dénote le nombre de messages échangés lors d'une opération d'écriture lorsque l'éventualité $\{X = x_k\}$ se présente (avec $1 \leq k \leq \prod_{l=0}^{l=h}(1+s_l)$ et $0 \leq l \leq h$). Le paramètre $\alpha_{x_k,l}$ prend la valeur 1 si les niveaux $\{0, 1, \ldots, l\}$ vérifient chacun la condition requise pour la validation d'une opération d'écriture (autrement dit $\sigma_{x_k,i} = 1\ \forall i \in \{0, 1, \ldots, l\}$). Donc,

$$\alpha_{x_k,l} = \prod_{i=0}^{i=l}\sigma_{x_k,i} \tag{3.85}$$

Sans perte de généralité, nous allons supposer que les l premiers niveaux visités lors d'une opération d'écriture sont les niveaux $\{0, 1, \ldots, l-1\}$, le niveau suivant étant le niveau l. Ainsi, lors du passage au niveau l, les différents messages échangés à ce niveau du trapèze sont :

- ⇨ $\alpha_{x_k,l}.s_l$ messages de requête d'écriture, pour récupérer les anciennes répliques résidant au niveau l du trapèze. Le protocole envoie donc les s_l messages de requête d'écriture si et seulement si cette opération a été validée aux l premiers niveaux visités (ce qui correspond à $\alpha_{x_k,l} = 1$) ;

- ⇨ $\alpha_{x_k,l}.x_k(l)$ messages de réponse venant des nœuds disponibles si les s_l messages de requête d'écriture ont été envoyés ;
- ⇨ $\alpha_{x_k,h}.x_k(l)$ messages d'envoi des nouvelles répliques de la donnée vers les nœuds disponibles, c'est-à-dire que la donnée ne sera écrite que si la condition suivante est remplie : $\forall i \in \{0,1,\ldots,h\}$, au moins w_l nœuds sont disponibles au niveau i du trapèze lors de cette opération.

Par conséquent, le nombre de messages échangés lors d'une opération d'écriture lorsque l'éventualité $\{X = x_k\}$ se présente est égal à :

$$\begin{aligned}\mu_{x_k} &= \sum_{l=0}^{l=h}\Big[\alpha_{x_k,l}.s_l + \alpha_{x_k,l}.x_k(l) + \alpha_{x_k,h}.x_k(l)\Big] \\ &= \sum_{l=0}^{l=h}\Big[\alpha_{x_k,l}.s_l + \alpha_{x_k,l}.x_k(l) + \underbrace{\alpha_{x_k,l}.\alpha_{x_k,h}}_{=\alpha_{x_k,h}}.x_k(l)\Big] \\ &= \sum_{l=0}^{l=h}\left[\alpha_{x_k,l}.\Big[s_l + (1+\alpha_{x_k,h}).x_k(l)\Big]\right] \end{aligned} \tag{3.86}$$

Le nombre moyen de messages échangés est égal à :

$$N_{write} = \sum_{k=1}^{k=\prod_{l=0}^{l=h}(1+s_l)} \mu_{x_k}.\mathbb{P}(X = x_k)$$

En remplaçant $\mathbb{P}(X = x_k)$ par son expression dans la formule (3.83), nous obtenons :

$$N_{write} = \sum_{k=1}^{k=\prod_{l=0}^{l=h}(1+s_l)} \left[\mu_{x_k}.t^{\left[\sum_{l=0}^{l=h} x_k(l)\right]}.(1-t)^{\left[\sum_{l=0}^{l=h}[s_l - x_k(l)]\right]}.\prod_{l=0}^{l=h}\binom{s_l}{x_k(l)}\right] \tag{3.87}$$

Comme nous avons cette égalité :

$$\{x_k \mid k \in \{1,2,\ldots,\prod_{l=0}^{l=h}(1+s_l)\} = \{(n_{w,0},\ldots,n_{w,h}) \mid \forall l \in \{0,1,\ldots,h\}, 0 \le n_{w,l} \le s_l\}$$

alors l'expression de N_{write} peut être exprimée en fonction des $n_{w,l}$ (pour l allant de 0 à h) :

$$N_{write} = \sum_{\substack{0\le n_{w,0}\le D^0 \\ \vdots \\ 0\le n_{w,h}\le D^h}} \left[\mu_x.t^{\left[\sum_{l=0}^{l=h} n_{w,l}\right]}.(1-t)^{\left[\sum_{l=0}^{l=h}\left[s_l - n_{w,l}\right]\right]}.\prod_{l=0}^{l=h}\binom{s_l}{n_{w,l}}\right] \tag{3.88}$$

Avec $x = (n_{w,0},\ldots,n_{w,h})$.

Quant au calcul du nombre moyen de messages échangés pour l'opération de lecture N_{read}, nous allons utiliser la variable aléatoire Z déjà définie dans la sous-section 3.5.3, page 98. Soit β_z le nombre de messages échangés lors d'une opération de lecture lorsque l'éventualité $\{Z = z\}$ se présente. Alors, l'expression de N_{read} est égale à :

$$N_{read} = \sum_{\substack{e_u \in E_u \\ e_r \in E_r \\ e_w \in E_w}} \Big[\beta_{z_k} . \mathbb{P}(Z = z) \Big] \tag{3.89}$$

où E_u, E_r et E_w sont les trois ensembles définis dans la sous-section 3.5.3 page 98, et $z = \begin{pmatrix} e_u \\ e_r \\ e_w \end{pmatrix}$. Nous allons garder cette notation dans le reste de cette sous-section. L'expression de $\mathbb{P}(Z = z)$ est déjà définie dans la formule (3.77). Ainsi, il nous reste à déterminer l'expression de β_z. Pour cela, supposons que l'évènement $\{Z = z\}$ se présente. Nous allons définir les cinq paramètres $\gamma_{e_r,l}$, $\upsilon_{e_r,l}$, $\theta_{e_u,l}$, $\lambda_{z,l}$ et $\delta_{z,l}$ de la manière suivante :

- ⇨ $\gamma_{e_r,l}$: permet de déterminer si la version de la donnée peut être obtenue ou non en utilisant uniquement les nœuds résidant au niveau l du trapèze :

$$\gamma_{e_r,l} = \begin{cases} 1 & \text{si } e_r(l) \geq s_l - w_l + 1 \\ 0 & \text{sinon} \end{cases} \tag{3.90}$$

  Ainsi, il retourne 1 si le niveau l du trapèze permet de déterminer la version de la donnée, 0 sinon ;
- ⇨ $\upsilon_{e_r,l}$: permet de déterminer si la version de la donnée peut être obtenue ou non en utilisant l'un des niveaux $\{0, 1, \ldots, l\}$ du trapèze. Ce paramètre peut être exprimé en fonction de $\gamma_{e_r,l}$:

$$\upsilon_{e_r,l} = 1 - \prod_{i=0}^{i=l} (1 - \gamma_{e_r,l}) \tag{3.91}$$

  Il retourne 1 si le protocole peut récupérer la version de la donnée, 0 sinon ;
- ⇨ $\theta_{e_u,l}$: permet de déterminer si le nombre de nœuds disponibles contenant des répliques à jour de la donnée aux niveaux $\{0, 1, \ldots, l\}$ du trapèze est supérieur ou égal à r (nombre de répliques à jour de la donnée requises par la technique de distribution des données pour reconstruire la donnée originale) :

$$\theta_{e_u,l} = \begin{cases} 1 & \text{si } \sum\limits_{i=0}^{i=l} e_u(l) \geq r \\ 0 & \text{sinon} \end{cases} \tag{3.92}$$

  Il retourne 1 si ce nombre est supérieur à r et 0 sinon ;
- ⇨ $\lambda_{z,l}$: permet de savoir si les nœuds se trouvant aux niveaux $\{0, 1, \ldots, l\}$ du trapèze permettent de reconstituer la donnée originale, c'est-à-dire d'obtenir la version de la donnée et d'avoir au moins r répliques à jour :

$$\lambda_{z,l} = \begin{cases} 1 & \text{si } \upsilon_{e_r,l} = 1 \text{ et } \theta_{e_u,l} = 1 \\ 0 & \text{sinon} \end{cases} \tag{3.93}$$

Ce paramètre retourne 1 si la donnée originale peut être reconstituée et sinon 0 (avec $z = \begin{pmatrix} e_u \\ e_r \\ e_w \end{pmatrix}$) ;

➪ $\delta_{z,l}$: permet de déterminer le premier niveau du trapèze qui vérifie $\lambda_{z,l} = 1$ avec l variant de 0 à h, c'est-à-dire le plus petit l tel que les niveaux $\{0, 1, \ldots, l\}$ du trapèze sont suffisants pour lire la donnée :

$$\delta_{z,l} = \begin{cases} \lambda_{z,0} & \text{si } l = 0 \\ (1 - \lambda_{z,l-1}).\lambda_{z,l} & \text{si } 1 \leq l \leq h \end{cases} \tag{3.94}$$

Ce paramètre retourne 1 uniquement pour le premier niveau du trapèze, si il existe, qui vérifie $\lambda_{z,l} = 1$ et 0 pour les autres niveaux.

Notons $l_{z,min}$ le seul niveau du trapèze qui vérifie $\delta_{z,l_{z,min}} = 1$. Nous avons :

$$1 - \lambda_{z,l} + \delta_{z,l} = \begin{cases} 1 & \text{si } 0 \leq l \leq l_{z,min} \\ 0 & \text{si } l_{z,min} \leq l \leq h \end{cases}$$

Le nombre de messages échangés peut donc être exprimé en fonction de $1 - \lambda_{z,l} + \delta_{z,l}$. Les différents messages échangés sont ainsi :

☞ $(1 - \lambda_{z,l} + \delta_{z,l}).s_l$ messages de requête de lecture, le terme $(1 - \lambda_{z,l} + \delta_{z,l})$ imposant donc que la requête de lecture s'arrête au niveau $l_{z,min}$;
☞ $(1 - \lambda_{z,l} + \delta_{z,l}).e_r(l)$ messages de réponse venant des nœuds disponibles au niveau l.

Ainsi, l'expression de β_z est égale à :

$$\begin{aligned} \beta_z &= \sum_{l=0}^{l=h} \Big[(1 - \lambda_{z,l} + \delta_{z,l}).s_l + (1 - \lambda_{z,l} + \delta_{z,l}).e_r(l) \Big] \\ &= \sum_{l=0}^{l=h} \Big[(1 - \lambda_{z,l} + \delta_{z,l})(s_l + e_r(l)) \Big] \end{aligned} \tag{3.95}$$

En utilisant les formules (3.89) et (3.95), nous obtenons :

$$N_{read} = \sum_{\substack{e_u \in E_u \\ e_r \in E_r \\ e_w \in E_w}} \left[\sum_{l=0}^{l=h} \Big[(1 - \lambda_{z,l} + \delta_{z,l})(s_l + e_r(l)) \Big] . \mathbb{P}(Z = z) \right]$$

En remplaçant $\mathbb{P}(Z = z)$ par son expression dans la formule (3.77), l'expression de N_{read} devient :

$$N_{read} = \sum_{\substack{(n_{u,0},\ldots,n_{u,h})\in E_u \\ (n_{r,0},\ldots,n_{r,h})\in E_r \\ (n_{w,0},\ldots,n_{w,h})\in E_w}} \left[\sum_{l=0}^{l=h} \Big[(1-\lambda_{z,l}+\delta_{z,l})(s_l+n_{r,l}) \Big]. \right.$$
$$\left. \prod_{l=0}^{l=h} \Big[\Psi_{s_l}(t,n_{r,l}).\Psi_{n_{r,l}}(t,n_{u,l}).\Psi_{s_l-n_{r,l}}(t,n_{w,l}-n_{u,l}).\xi_{n_{u,l},n_{r,l},n_{w,l}} \Big] \right] \quad (3.96)$$

avec avec $z = \begin{pmatrix} n_{u,0} & n_{u,1} & \ldots & n_{u,h} \\ n_{r,0} & n_{r,1} & \ldots & n_{r,h} \\ n_{w,0} & n_{w,1} & \ldots & n_{w,h} \end{pmatrix}$.

3.6 Algorithmes d'écriture et de lecture

L'algorithme 9 décrit une procédure d'écriture d'une donnée avec un protocole de cohérence basé sur les quorums. Cette procédure d'écriture fait appel à une autre procédure "getWriteQuorum" (voir algorithme 9, ligne 9), qui se charge de trouver un quorum d'écriture. Cette procédure récupère la version de la donnée à écrire et un quorum d'écriture si elle réussit, et sinon la valeur NULL. Chaque protocole de cohérence que nous avons proposé a sa propre procédure "getWriteQuorum". Les procédures "getWriteQuorum" sont définies par :

- l'algorithme 1 page 71 pour le quorum majoritaire général ;
- l'algorithme 3 page 77 pour le quorum en grille général ;
- l'algorithme 5 page 87 pour le quorum en arbre général ;
- l'algorithme 7 page 96 pour le quorum en trapèze général.

L'algorithme 10 décrit une procédure de lecture d'une donnée avec un protocole de cohérence basé sur les quorums. Cette procédure de lecture utilise la procédure "getReadQuorum" (voir algorithme 10, ligne 7), pour former un quorum de lecture. Cette procédure retourne la dernière version de la donnée et un quorum de lecture si elle réussit, et sinon la valeur NULL. Les procédures "getReadQuorum" sont définies par :

- l'algorithme 2 page 72 pour le quorum majoritaire général ;
- l'algorithme 4 page 78 pour le quorum en grille général ;
- l'algorithme 6 page 88 pour le quorum en arbre général ;
- l'algorithme 8 page 97 pour le quorum en trapèze général.

Les algorithmes que nous avons décrits précédemment sont basés sur l'hypothèse selon laquelle la défaillance d'un nœud entre le moment où on détermine le quorum et le moment où on valide l'opération est négligeable. Nous travaillons actuellement sur l'amélioration de ces algorithmes. Le processus normal d'écriture serait le suivant :

Algorithm 9 Procédure d'écriture de la donnée x

```
1: Données
2: x                                                              ▷ Identité de la donnée à écrire
3: d_x                                                            ▷ Nouvelle valeur de la donnée x
4: n                                                              ▷ Nombre de morceaux de la donnée
5: r                                  ▷ Nombre de morceaux requis pour reconstituer la donnée originale
6: error_write ← 0                                                ▷ Nombre d'erreur d'écriture
7: E ← ∅
8: S ← {S_1, S_2, ..., S_n}                                       ▷ n nœuds (sites) utilisés pour le stockage de x

9: procedure ECRIRE(x, d_x, r, n, S)                              ▷ Écriture de la donnée x
10:     w ← max(r, ⌊n/2⌋ + 1)
11:     [v, Q] ← GETWRITEQUORUM(x, S)                             ▷ v : version de la donnée, Q : quorum d'écriture
12:     if Q ≠ NULL then                                          ▷ On a trouvé un quorum d'écriture
13:         {d_x1, d_x2, ..., d_xn} ← Générer les n morceaux de la donnée x   ▷ En utilisant un code correcteur (n, r)
14:         for all q ∈ Q do
15:             ∃j ∈ {1, 2, ..., n}/q = S_j
16:             écrire (d_xj, v) dans S_j
17:             if erreur d'écriture then
18:                 error_write + +
19:                 E ← E ∪ {q}
20:             end if
21:         end for
22:         if error_write > 0 then
23:             while (Q ∪ E) ≠ S do
24:                 Q1 ← GETOTHERSNODES(E, Q, S)   ▷ Retourne error_write nœuds pour que (Q \ E) ∪ Q1 forme un
    nouveau quorum d'écriture
25:                 for all q ∈ Q1 do
26:                     ∃j ∈ {1, 2, ..., n}/q = S_j
27:                     écrire (d_xj, v) dans S_j
28:                     if erreur d'écriture then
29:                         error_write + +
30:                         E ← E ∪ {q}
31:                     else
32:                         error_write − −
33:                     end if
34:                     if error_write == 0 then
35:                         Confirmer la réussite de l'opération d'écriture
36:                         Le reste des morceaux peut être écrit de manière synchrone ou asynchrone
37:                         return
38:                     end if
39:                 end for
40:             end while
41:         else
42:             Confirmer la réussite de l'opération d'écriture
43:             Le reste des morceaux peut être écrit de manière synchrone ou asynchrone
44:         end if
45:     end if
46:     Confirmer l'échec de l'opération d'écriture
47:     return
48: end procedure
```

1. Choix du quorum ;
2. Récupération de la version ;
3. Envoi de la mise à jour aux membres du quorum ;
4. Validation des nouvelles copies des membres du quorum.

Lors de l'envoi de la mise à jour aux membres du quorum, l'idée consiste :

- à conserver toujours deux versions de la donnée lors d'une mise à jour : la donnée valide, et la donnée en cours de validation. La donnée en cours de validation ne devient valide que lors de la réception d'un message de validation ;
- à écraser les versions non validées lors de la réception d'une requête d'écriture de version supérieure ;

Algorithm 10 Procédure de lecture de la donnée x

```
1: Données
2: x                                                          ▷ Identité de la donnée à lire
3: n                                                          ▷ Nombre de morceaux de la donnée
4: r                                          ▷ Nombre de morceaux requis pour reconstituer la donnée originale
5: error_read ← 0                                             ▷ Nombre d'erreur de lecture
6: S ← {S_1, S_2, ..., S_n}                                   ▷ n nœuds (sites) stockant les morceaux de x

7: procedure LIRE(x, r, n, S)                                 ▷ Lecture de la donnée x
8:     [v, Q] ← GETREADQUORUM(x, S)                           ▷ Q : quorum de lecture
9:     if Q ≠ NULL then                                       ▷ On a trouvé un quorum de lecture
10:        V_x ← ∅
11:        for all q ∈ Q do
12:            ∃j ∈ {1, 2, ..., n}/q = S_j
13:            v_{x_j} ← lire x_j dans S_j
14:            if v_{x_j} ≠ NULL then
15:                V_x ← V_x ∪ {v_{x_j}}
16:            else
17:                error_read + +
18:            end if
19:        end for
20:        if error_read > 0 then
21:            for all q ∈ S \ Q do
22:                version ← requête d'accès (x, q)   ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
23:                if estValide(version) then   ▷ estValide(version) retourne 1 si version est un numéro de version valide et 0 sinon
24:                    if version = v then
25:                        ∃j ∈ {1, 2, ..., n}/q = S_j
26:                        v_{x_j} ← lire x_j dans S_j
27:                        if v_{x_j} ≠ NULL then
28:                            error_read − −
29:                            V_x ← V_x ∪ {v_{x_j}}
30:                        end if
31:                    end if
32:                end if
33:                if error_read = 0 then
34:                    break
35:                end if
36:            end for
37:        end if
38:        if error_read = 0 then
39:            v_x ← décoder (r, n, V_x)   ▷ Reconstruction de x en utilisant un code correcteur (n, r)
40:            return v_x                                      ▷ L'opération de lecture réussit
41:        end if
42:    end if
43:    return NULL                                            ▷ L'opération de lecture échoue
44: end procedure
```

- à ne pas tenir compte des nœuds non validés pour la détermination de la valeur de la nouvelle version.

Avec ces hypothèses, même si tous les nœuds ne reçoivent pas le rollback, cela ne pose pas de problème car la version non validée sera automatiquement écrasée lors de l'écriture suivante. Les algorithmes 11 et 12 décrivent respectivement la sélection d'un quorum d'écriture et de lecture pour le quorum majoritaire général.

Algorithm 11 Procédure de recherche d'un quorum d'écriture

```
1: Données
2: x                                                  ▷ Identité de la donnée à écrire
3: (n, r)                                             ▷ Paramètres du protocole de distribution utilisé
4: P ← {P_1, P_2, ..., P_n}                           ▷ n nœuds utilisés pour le stockage de x

5: procedure getWriteQuorum(x, P)                     ▷ Recherche d'un quorum d'écriture
6:     Q ← ∅                                          ▷ Quorum de lecture
7:     v ← 0                                          ▷ Numéro de version validée de la donnée
8:     w ← max(r, ⌊n/2⌋ + 1)                          ▷ Taille d'un quorum d'écriture
9:     for all q ∈ P do
10:        [v_i, c_i, rb_i] ← requête d'accès (x, q)  ▷ Requête d'accès à q pour x : (v_i : version validée de la donnée,
       c_i : version non validée de la donnée, rb_i : indicateur binaire de rollback avec 1 indique la présence d'un rollback et
       0 sinon)
11:        if estValide(v_i, c_i, rb_i) then
12:            if c_i = 0 then
13:                if v < v_i + rb_i then
14:                    v ← v_i + rb_i
15:                end if
16:            end if
17:            Q ← Q ∪ {q}
18:            if |Q| = w then                        ▷ Un quorum d'écriture est obtenu
19:                return [v + 1, Q]                  ▷ Version de la donnée à écrire et un quorum d'écriture
20:            end if
21:        end if
22:    end for
23:    return NULL                                    ▷ La recherche d'un quorum d'écriture échoue
24: end procedure
```

3.7 Évaluations numériques

3.7.1 Analyse théorique

Dans cette section, nous allons effectuer quelques évaluations numériques des nouveaux protocoles proposés précédemment. Ces évaluations numériques concernent la disponibilité en écriture, la disponibilité en lecture et le nombre moyen de messages échangés pour ces deux opérations classiques. Voici les différents paramètres de chaque protocole que nous allons utiliser tout au long de ces évaluations numériques :

- ☞ Quorum majoritaire général (QMG) : $n = 15$;
- ☞ Quorum en grille général (QGG) : $N = 3$ et $M = 5$;
- ☞ Quorum en arbre général (QAG) : $D = 3$ et $h = 2$;
- ☞ Quorum en trapèze général (QTG) : $a = 2$, $b = 3$ et $h = 2$.

Dans le reste de cette section, nous allons adopter les abréviations *QMG*, *QGG*, *QAG* et *QTG* pour désigner respectivement le quorum majoritaire général, le quorum en grille général, le quorum en arbre général et le quorum en trapèze général. Nous allons considérer trois valeurs du paramètre r qui représente le nombre de répliques à jour requises par le protocole de distribution des données lors d'une opération de lecture ($r = 4$, $r = 8$ et $r = 11$). Comme les paramètres w_i des protocoles *quorum en arbre général* et *quorum en trapèze général* dépendent du paramètre r, le choix de leurs valeurs doit prendre compte la valeur de ce dernier. En effet, la taille du quorum d'écriture dans chaque protocole doit être supérieure ou égale au paramètre r . Ainsi, nous définissons ces différents paramètres selon la valeur de r que nous allons utiliser pour ces évaluations numériques :

Algorithm 12 Procédure de recherche d'un quorum de lecture

```
1: Données
2: x                                                          ▷ Identité de la donnée à lire
3: (n, r)                                                     ▷ Paramètres du protocole de distribution utilisé
4: P ← {P_1, P_2, ..., P_n}                                   ▷ n nœuds stockant les morceaux de x

5: procedure getReadQuorum(x, P)                              ▷ Recherche d'un quorum de lecture
6:     Q ← ∅                                                  ▷ Quorum de lecture
7:     v ← 0                                                  ▷ Numéro de version de la donnée
8:     compteur ← 0                                           ▷ Nombre de nœuds accédés
9:     w ← max(r, ⌊n/2⌋ + 1)
10:    for all q ∈ P do
11:        [v_i, c_i, rb_i] ← requête d'accès (x, q)   ▷ Requête d'accès à q pour x : retourne la version de la donnée si OK
12:        if estValide(v_i, c_i, rb_i) then
13:            compteur ← compteur + 1
14:            if v_i > v then
15:                Q ← {q}
16:                v ← v_i
17:            else if v_i = v or c_i = v then
18:                if |Q| < r then
19:                    Q ← Q ∪ {q}
20:                end if
21:            end if
22:            if compteur >= n − w + 1 then                  ▷ Version de donnée trouvée
23:                if |Q| = r then                            ▷ Tous les morceaux nécessaire pour la reconstruction
24:                    return [v, Q]
25:                end if
26:            end if
27:        end if
28:    end for
29:    if compteur >= n − w + 1 then                          ▷ Version de donnée trouvée mais il manque des morceaux
30:        for all q ∈ P do
31:            if q ∉ Q then
32:                [v_i, c_i, rb_i] ← requête d'accès (x, q)
33:                if estValide(v_i, c_i, rb_i) then
34:                    if c_i = v then
35:                        Q ← Q ∪ {q}
36:                    end if
37:                    if |Q| = r then
38:                        return [v, Q]
39:                    end if
40:                end if
41:            end if
42:        end for
43:    end if
44:    return NULL                                            ▷ La recherche d'un quorum de lecture échoue
45: end procedure
```

⇨ Pour $r = 4$:

- ☞ Quorum en arbre général : $w_0 = 1$, $w_1 = 2$ et $w_2 = 4$;
- ☞ Quorum en trapèze général : $w_0 = 2$, $w_1 = 3$ et $w_2 = 4$.

⇨ Pour $r = 8$:

- ☞ Quorum en arbre général : $w_0 = 1$, $w_1 = 2$ et $w_2 = 5$;
- ☞ Quorum en trapèze général : $w_0 = 2$, $w_1 = 3$ et $w_2 = 4$.

⇨ Pour $r = 11$:

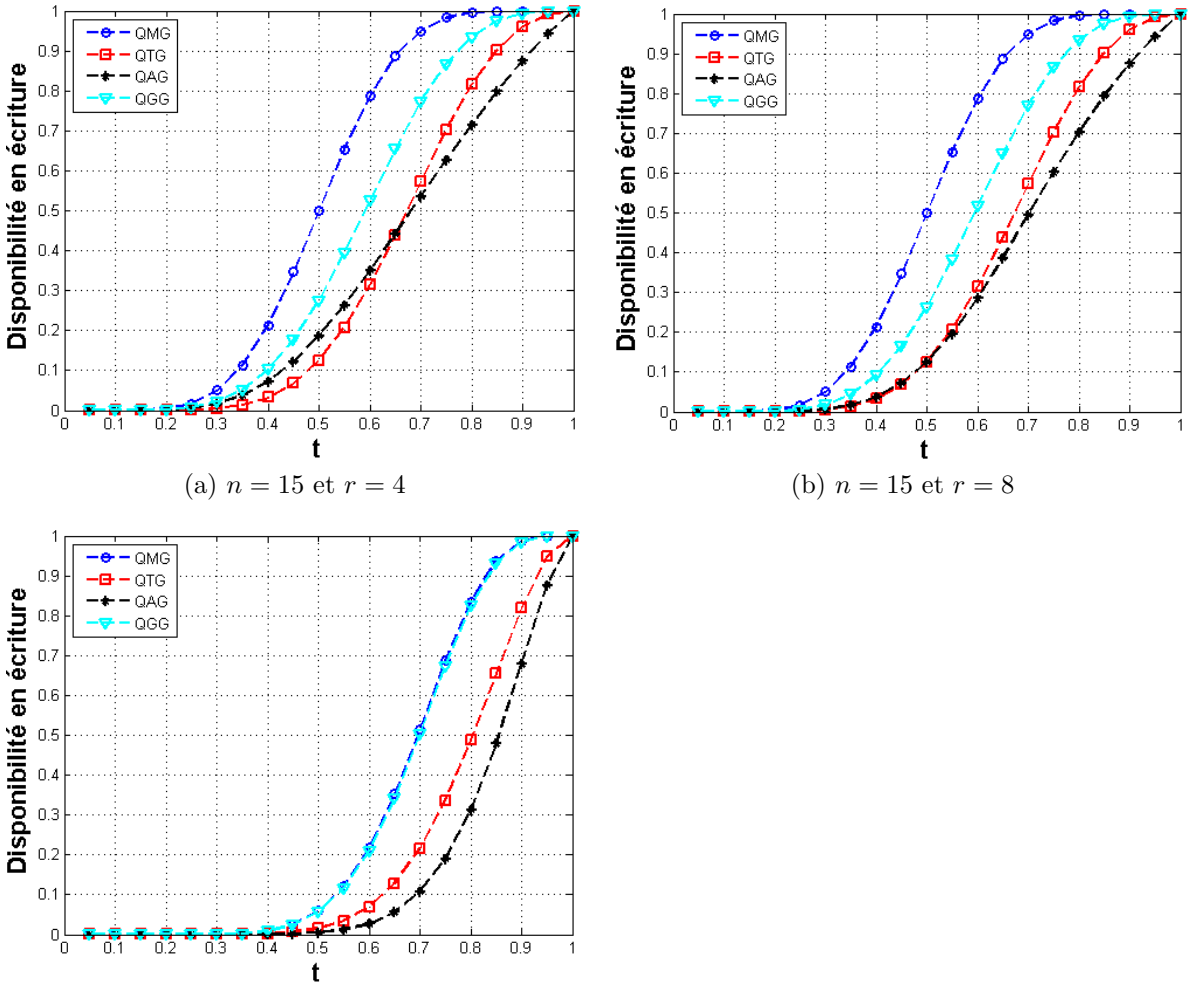
- ☞ Quorum en arbre général : $w_0 = 1$, $w_1 = 2$ et $w_2 = 8$;
- ☞ Quorum en trapèze général : $w_0 = 2$, $w_1 = 3$ et $w_2 = 6$.

Il convient de noter que pour une valeur donnée du paramètre r, il existe plusieurs $(h+1)$-uplets $(w_0, \ldots, w_h)$ possibles. Le choix d'un $(h+1)$-uplet peut se baser sur la priorité de l'utilisateur (disponibilité en écriture, disponibilité en lecture ou encore le nombre moyen de messages échangés, etc.) voire un compromis entre ces différents

critères.

Disponibilité en écriture

La figure 3.4 montre la disponibilité en écriture des différents protocoles pour les trois valeurs de r que nous avons choisies. Nous avons présenté trois cas de figure correspondant respectivement à $r = 4$, $r = 8$ et $r = 11$ afin de déterminer l'évolution de la disponibilité en écriture des différents protocoles en fonction du paramètre r.



(a) $n = 15$ et $r = 4$

(b) $n = 15$ et $r = 8$

(c) $n = 15$ et $r = 11$

FIGURE 3.4 – Disponibilité en écriture

La disponibilité en écriture correspondant au cas $r = 4$ est présentée sur la figure 3.4(a). D'après les paramètres que nous avons définis pour le cas $r = 4$, la taille du quorum d'écriture est égale à :

- ☞ Quorum majoritaire général : $w = \max(r, \lfloor \frac{n}{2} \rfloor + 1) = 8$;
- ☞ Quorum en grille général : $w = \max(r, N + M - 1) = 7$;

- ☞ Quorum en arbre général : $w = w_0 + \ldots + w_h = 7$;
- ☞ Quorum en trapèze général : $w = w_0 + \ldots + w_h = 9$.

Ainsi, en gardant les valeurs des différents paramètres des protocoles, les disponibilités en écriture des quatre protocoles (*quorum majoritaire général*, *quorum en grille général*, *quorum en arbre général*, *quorum en trapèze général*) restent identiques à celles affichées sur la figure 3.4(a) pour tout r compris entre 1 et 7. Dans cette plage de valeurs de r, la meilleure disponibilité en écriture correspond au protocole QMG. Cela est dû au fait que tous les protocoles ont presque la même taille du quorum d'écriture mais contrairement aux trois autres protocoles, le QMG n'impose aucune structure logique pour l'organisation des nœuds. Donc, dans cet exemple, n'importe quels huit nœuds disponibles permettent de valider une opération d'écriture. Or cela n'est toujours pas le cas pour les autres protocoles où la validation d'une opération d'écriture dépend non seulement du nombre de nœuds disponibles mais aussi de leur organisation logique. En effet, supposons qu'il existe exactement 9 nœuds disponibles lors d'une opération d'écriture. Avec le protocole QGG, même si le nombre de nœuds disponibles est supérieur à la taille du quorum d'écriture ($9 \geq w = 7$), la validation ou non de l'opération d'écriture dépend de la répartition dans la grille de ces 9 nœuds disponibles. Elle sera uniquement validée si chaque colonne de la grille contient au moins un nœud disponible. Avec le protocole QTG, le nombre de nœuds disponibles est égal à la taille du quorum d'écriture. Pour que l'opération d'écriture soit validée, il faut que ces neuf nœuds disponibles soient organisés comme suit : 2 nœuds au niveau $l = 0$, 3 nœuds au niveau $l = 1$ et 4 nœuds au niveau $l = 2$. Voici un exemple d'organisation de ces neuf nœuds où l'opération d'écriture échoue : 2 nœuds au niveau $l = 0$, 2 nœuds au niveau $l = 1$ et 5 nœuds au niveau $l = 2$. L'opération d'écriture n'est pas validée au niveau $l = 1$ car le nombre de nœuds disponibles résidant à ce niveau est inférieur à $w_1 = 3$. Le même raisonnement s'applique également dans le cas où le protocole QAG est utilisé. Ce dernier présente la plus faible disponibilité en écriture par rapport aux autres protocoles. La raison en est que le seul nœud résidant au niveau $l = 0$ constitue un goulet d'étranglement pour ce protocole. La défaillance de ce nœud implique forcément la non validation d'une opération d'écriture.

La figure 3.4(b) représente les disponibilités en écriture des différents protocoles correspondant à $r = 8$. Quand le paramètre r passe de 4 à 8 (figures 3.4(a) et 3.4(b)), seules les courbes de disponibilité en écriture des protocoles QGG et QAG changent. En effet, d'après la définition des différents paramètres des protocoles pour $r = 8$, nous avons :

- ☞ Quorum majoritaire général : $w = \max(r, \lfloor \frac{n}{2} \rfloor + 1) = 8$;
- ☞ Quorum en grille général : $w = \max(r, N + M - 1) = 8$;
- ☞ Quorum en arbre général : $w = w_0 + \ldots + w_h = 8$;
- ☞ Quorum en trapèze général : $w = w_0 + \ldots + w_h = 9$.

Donc, seules les tailles des quorums d'écriture des protocoles QGG et QAG changent et elles passent toutes les deux de 7 à 8. Or, la disponibilité en écriture est inversement proportionnelle à la taille du quorum d'écriture, ce qui explique la légère diminution des courbes de disponibilité de ces deux protocoles quand le paramètre r passe de 4 à 8.

Par ailleurs, nous pouvons remarquer que le protocole QAG est beaucoup plus sensible à une modification de la taille du quorum d'écriture que le protocole QGG. Ceci est dû à la structure logique utilisée pour organiser les répliques des données. En effet, le protocole QGG est tel qu'un quorum d'écriture valide dans le cas $r = 4$ peut être aisément transformé en quorum valide pour le cas $r = 8$ en ajoutant n'importe quel un nœud de la grille (exceptés bien sûr ceux qui sont déjà dans le quorum, c'est-à-dire choisi dans les $N.M - \max(r, N + M - 1) = 8$ nœuds restants). En revanche, avec le protocole QAG, seul l'ajout d'un nœud venant du niveau $l = 2$ permet de constituer un quorum d'écriture dans le cas $r = 8$ lorsqu'on se base sur un quorum valide pour le cas $r = 4$ (car c'est w_2 qui a changé lorsque r passe de 4 à 8). Ainsi, le protocole QGG a une probabilité de trouver un nœud disponible égale à $1-(1-t)^8$ tandis qu'elle est seulement de $1-(1-t)^5$ pour le protocole QAG. Par ailleurs, la taille du quorum d'écriture du protocole QMG reste toujours égale à 8 tandis que celle du protocole QTG reste inchangée et est égale à 9. Les courbes de disponibilité d'écriture des protocoles QMG et QTG restent donc inchangées dans les deux cas de figure.

Enfin, les disponibilités en écriture des différents protocoles correspondant au cas $r = 11$ sont affichées dans la figure 3.4(c). Lorsque le paramètre r passe de 8 à 11 (figures 3.4(b) et 3.4(c)), toutes les courbes se déplacent vers le bas. En effet, pour $r = 11$, la taille du quorum d'écriture est égale à :

- ☞ Quorum majoritaire général : $w = \max(r, \lfloor \frac{n}{2} \rfloor + 1) = 11$;
- ☞ Quorum en grille général : $w = \max(r, N + M - 1) = 11$;
- ☞ Quorum en arbre général : $w = w_0 + \ldots + w_h = 11$;
- ☞ Quorum en trapèze général : $w = w_0 + \ldots + w_h = 11$.

La taille du quorum d'écriture a augmenté pour chaque protocole, ce qui explique la diminution de la disponibilité en écriture des quatre protocoles lorsque le paramètre r passe de 8 à 11. Nous pouvons remarquer dans la figure 3.4(c) que même si la taille du quorum d'écriture est identique pour les quatre protocoles, ils n'ont forcément pas la même disponibilité en écriture pour la même disponibilité des nœuds. Les écarts s'expliquent principalement par la structure logique imposée par chaque protocole. La plus grande disponibilité en écriture est obtenue par le protocole QMG car ce dernier n'impose aucune structure logique aux répliques des données. Par conséquent, n'importe 11 nœuds disponibles (parmi les $n = 15$ nœuds utilisés pour stocker les répliques de la donnée) permettent de valider une opération d'écriture. En revanche, même si il existe 11 nœuds disponibles, la validation d'une opération d'écriture dépend encore de la répartition de ces 11 nœuds lorsque l'un des trois autres protocoles (QGG, QAG et QTG) est utilisé. Voici un exemple pour chaque protocole où le nombre de nœuds diponibles est égal à 11 mais où l'opération d'écriture échoue :

- ⇨ Quorum en grille général : la première colonne ne contient aucun nœud disponible, la deuxième colonne contient 2 nœuds disponibles et les trois colonnes restantes contiennent chacune 3 nœuds disponibles. L'opération d'écriture ne sera pas validée car la première colonne ne contient aucun nœud disponible ;
- ⇨ Quorum en arbre général : 1 nœud disponible au niveau $l = 0$, 3 nœuds disponibles au niveau $l = 1$ et 7 nœuds disponibles au niveau $l = 2$. L'opération d'écriture

échoue car le nombre de nœuds disponibles au niveau $l = 2$ est inférieur à w_2 ($7 < w_2 = 8$) ;

- ⇨ Quorum en trapèze général : 2 nœuds disponibles au niveau $l = 0$, 2 nœuds disponibles au niveau $l = 1$ et 7 nœuds disponibles au niveau $l = 2$. L'opération d'écriture ne sera validée car le nombre de nœuds disponibles au niveau $l = 1$ est inférieur à w_1 ($2 < w_1 = 3$).

Ceci explique donc la plus grande disponibilité en écriture du protocole *QMG* par rapport aux trois autres protocoles qui eux, imposent une structure logique pour organiser les nœuds utilisés pour stocker les données. Par ailleurs, pour $r = 11$, la disponibilité en écriture du protocole *QGG* est très proche de celle du protocole *QMG* comme montre la figure 3.4(c). Cela est dû au fait qu'il existe seulement deux scénarios où le nombre de nœuds disponibles est supérieur ou égal à 11 mais où l'opération d'écriture échoue en utilisant le protocole *QGG*, à savoir :

- ⇨ **scénario 1** (nombre de nœuds égal à 11) : une colonne parmi les cinq existantes ne contient aucun nœud disponible, une colonne parmi les quatre restantes en contient 2 et les trois colonnes restantes contiennent chacune 3 nœuds disponibles ;
- ⇨ **scénario 2** (nombre de nœuds égal à 12) : une colonne parmi les cinq existantes ne contient aucun nœud disponible et les quatre colonnes restantes contiennent chacune 3 nœuds disponibles.

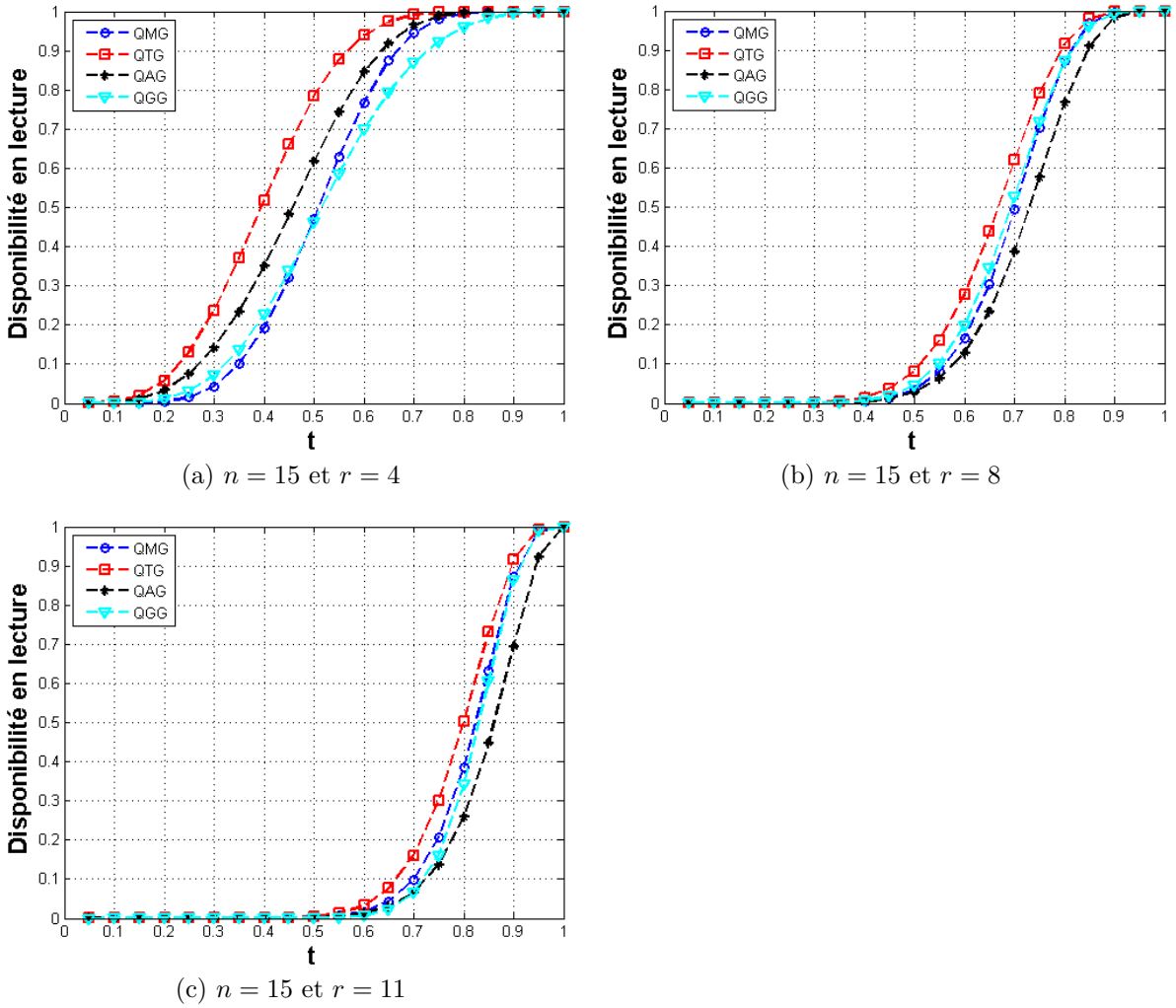
Ainsi, la probabilité d'apparition de ces deux scénarios reste très faible par rapport à celle de toutes les éventualités possibles. Par ailleurs, lorsque le nombre de nœuds disponibles est égal à 13, les cinq colonnes de la grille doivent forcément contenir chacun au moins un nœud disponible. En effet, quatre colonnes de la grille ne peuvent contenir au total que 12 nœuds à raison de 3 nœuds par colonne. Nous pouvons donc en déduire que pour une taille du quorum d'écriture supérieur ou égale à 13, les protocoles *QGG* et le *QMG* présentent la même disponibilité en écriture. En revanche, il existe plusieurs scénarios où l'opération d'écriture n'est pas validée bien que le nombre de nœuds disponibles soit supérieur ou égal à 11 en utilisant les protocoles *QAG* ou *QTG*. Ceci explique leur faible disponibilité en écriture par rapport aux deux autres protocoles.

Disponibilité en lecture

La figure 3.5 montre la disponibilité en lecture des différents protocoles pour les trois valeurs de r que nous avons choisies ($r = 4$, $r = 8$ et $r = 11$).

Pour $r = 4$ (figure 3.5(a)), la plus grande disponibilité en lecture correspond à celle du protocole *QTG*. Ceci s'explique principalement par l'organisation logique des nœuds et la taille du quorum d'écriture, plus élevée que dans le cas des autres protocoles. Selon une répartition spécifique, 4 nœuds disponibles peuvent être suffisants pour lire la donnée en utilisant le protocole *QTG* tandis qu'au moins 5 nœuds disponibles sont requis avec le protocole *QGG* et il en faut au minimum 8 avec le protocole *QMG*. En effet, à titre d'exemple, en utilisant le protocole *QTG*, 3 nœuds disponibles stockant des répliques à jour au niveau $l = 0$ et 1 nœud disponible avec une réplique à jour au

niveau $l = 1$ ou $l = 2$ sont suffisants pour la lecture des données. En revanche, avec le protocole QGG, 5 nœuds disponibles dont au moins 4 contiennent chacun une réplique à jour de la donnée (à raison d'un nœud par colonne de la grille) sont nécessaires pour reconstituer la donnée originale. De plus, avec le protocole QGG, il existe plusieurs scénarios où l'opération de lecture échoue bien que le nombre de nœuds disponibles soit supérieur ou égal à 5. Jusqu'à 12 nœuds disponibles peuvent être insuffisants si ces nœuds résident dans 4 colonnes de la grille, la colonne restante ne contenant aucun nœud disponible. La différence de disponibilité de lecture entre les protocoles QTG



(a) $n = 15$ et $r = 4$

(b) $n = 15$ et $r = 8$

(c) $n = 15$ et $r = 11$

FIGURE 3.5 – Disponibilité en lecture

et QMG s'explique principalement par le fait qu'avec cet exemple, tout quorum de lecture avec le protocole QMG l'est aussi avec le protocole QTG alors que l'inverse n'est pas forcément vrai. En effet, le protocole QMG requiert 8 nœuds disponibles pour déterminer la version de la donnée. De même, avec le protocole QTG, n'importe quels 8 nœuds disponibles (indépendamment de leur localisation) permettent de déterminer la version de la donnée. En revanche, dans ce dernier cas, il ne s'agit pas d'une exigence

au sens strict car seulement deux nœuds disponibles sont suffisants pour retrouver la version de la donnée avec ce protocole (à condition que ces nœuds soient situés au niveau $l = 0$).

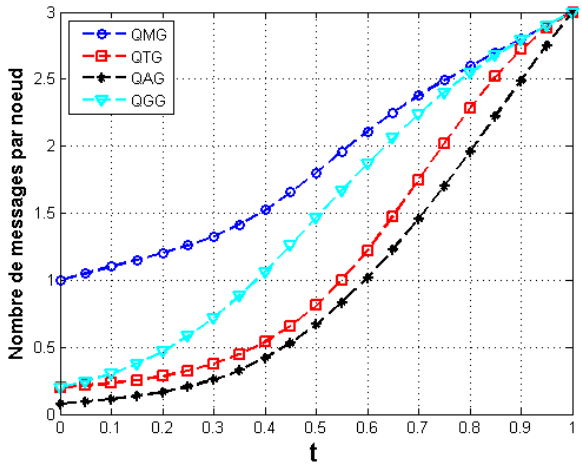
La disponibilité en lecture pour $r = 8$ et $r = 11$ est présentée respectivement sur les figures 3.5(b) et 3.5(c). Nous remarquons que dans ces deux figures la courbe de la disponibilité en lecture du protocole QAG passe en dessous de celle des protocoles QMG et QGG et devient la plus faible disponibilité en lecture. Mais cela doit être relativisé car ce changement s'explique principalement par le fait que le protocole QAG utilise seulement au total 13 nœuds contrairement aux trois autres protocoles qui en utilisent 15. Par exemple, avec un nombre total de nœuds égal à 13, la courbe de disponibilité en lecture du protocole QAG passe devant de celle du protocole QMG. Le choix d'utiliser un nombre de nœuds différent des autres protocoles pour le protocole QAG a été motivé par le fait qu'il n'y a pas un nombre raisonnable (i.e. pas trop grand pour l'exemple) qui fédère les quatre protocoles.

La figure 3.5 montre que la disponibilité en lecture est inversement proportionnelle à la valeur du paramètre r. Ceci s'explique par le fait que plus la valeur du paramètre r est grande, plus le nombre de répliques à jour requises est élevé. Ainsi, en augmentant la valeur du paramètre r, on perd en disponibilité en lecture mais on gagne en sécurité et en espace de stockage car il faut plusieurs répliques de la donnée pour reconstituer cette dernière et l'espace de stockage diminue car il est proportionnel à $\frac{1}{r}$. Par voie de conséquence, l'utilisation d'une grande valeur du paramètre r exige une disponibilité de nœuds très élevée pour ne pas pénaliser l'opération de lecture.

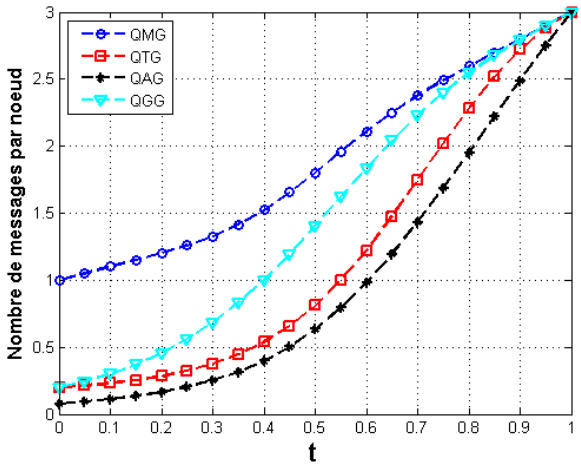
Nombre moyen de messages échangés pour l'opération d'écriture

La figure 3.6 présente le nombre moyen de messages échangés (envoyés et reçus) par nœud pour l'opération d'écriture en fonction de la disponibilité des nœuds t. L'objectif principal est d'avoir une estimation du nombre de messages générés par chaque protocole lors d'une opération d'écriture car ce nombre peut avoir un impact sur le temps d'accès aux données et l'utilisation des ressources système telles que la bande passante. Dans cette sous-section, nous allons utiliser l'abréviation $NMOE$ pour désigner le " *nombre moyen de messages échangés par nœud pour l'opération d'écriture* ".

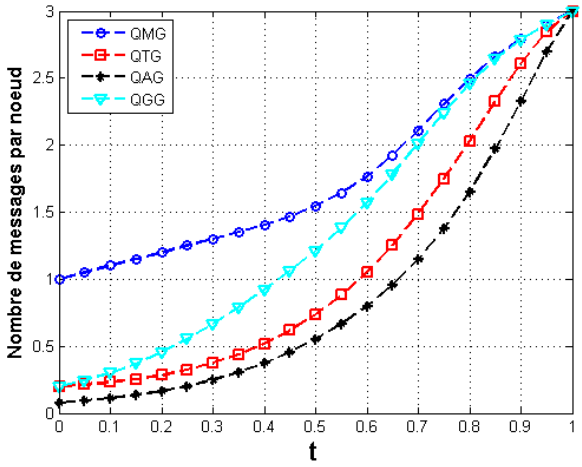
La première constatation (figure 3.6) est que pour tous les protocoles, le $NMOE$ croît avec la disponibilité des nœuds t. Cela est dû au fait que, par définition, plus la disponibilité des nœuds t est élevée, plus il y a des chances que plusieurs nœud soient disponibles lors d'une opération d'écriture. Ce qui implique donc que plusieurs nœuds vont participer à l'opération d'écriture et cela génère naturellement plus de messages. Prenons par exemple le cas $t = 1$. Indépendamment du protocole utilisé, il y aura au total $3n$ messages échangés car les n nœuds utilisés pour stocker la donnée seront disponibles et un nœud disponible génère 3 messages lors d'une opération d'écriture valide (voir section 3.1, page 66). Donc $NMOE = \frac{3n}{n} = 3$ pour tous les protocoles, comme le montrent les courbes de la figure 3.6.



(a) $n = 15$ et $r = 4$



(b) $n = 15$ et $r = 8$



(c) $n = 15$ et $r = 11$

FIGURE 3.6 – Nombre moyen de messages échangés par nœud pour l'opération d'écriture

La figure 3.6 montre également que lorsque la valeur de la disponibilité des nœuds t est faible, le protocole *QMG* a un *NMOE* significativement plus élevé que les trois autres protocoles. En effet, le protocole *QMG* n'impose aucune structure logique pour les nœuds stockant les répliques de la donnée. Or, les structures logiques des nœuds permettent de détecter rapidement une défaillance de l'opération d'écriture. Lorsqu'une défaillance est détectée, le protocole stoppe le processus d'écriture en évitant ainsi de générer des messages inutiles. Par exemple, la défaillance d'un seul nœud au niveau $l = 0$ avec le protocole *QAG* (un message échangé : *une requête d'écriture*), la défaillance de deux nœuds au niveau $l = 0$ avec le protocole *QTG* (quatre messages échangés : *trois requêtes d'écriture* et *une réponse venant du seul nœud disponible à ce niveau*) et la défaillance de trois nœuds dans une colonne de la grille avec le protocole *QGG* (trois messages échangés : *trois requêtes d'écriture* dans le meilleur des cas) permettent chacune au protocole de décider que l'opération va surement échouer. En revanche il faut 8 défaillances pour pouvoir détecter un échec d'écriture avec le protocole *QMG*

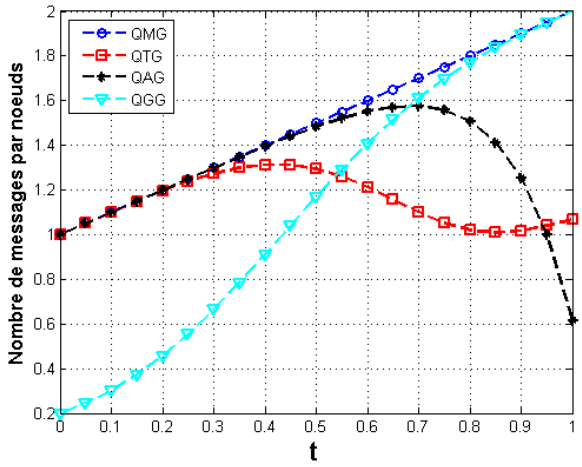
(soit, dans le meilleur des cas, huit messages échangés : *huit requêtes d'écriture*). Cette détection rapide d'une défaillance de l'opération d'écriture permet au protocole *QAG* d'obtenir un faible *NMOE* par rapport aux autres protocoles car dans le cas le plus favorable, un seul message échangé lui suffit pour détecter une défaillance de l'opération d'écriture.

Les figures 3.6(a) et 3.6(b) montrent que, comme dans le cas de la courbe de disponibilité en écriture, même si on augmente la valeur du paramètre r, la courbe de *NMOE* reste inchangée tant qu'on ne modifie pas la taille du quorum d'écriture. Ainsi, seules les courbes correspondant aux protocoles *QGG* et *QAG* changent lorsque la valeur du paramètre r passe de 4 à 8 car la taille du quorum d'écriture pour ces deux protocoles passe de 7 à 8 alors que la taille des quorums pour les protocoles *QMG* et *QTG* restent inchangées et sont égales, respectivement, à 8 et à 9. En revanche, lors du passage de la valeur de r de 8 à 11 (figures 3.6(b) et 3.6(c)), toutes les courbes se déplacent vers le bas car la taille du quorum d'écriture a augmenté pour tous les protocoles. On constate ainsi que le *NMOE* est inversement proportionnel à la taille du quorum d'écriture. Cela s'explique par le fait que lorsque la taille du quorum d'écriture augmente, la probabilité d'obtenir un quorum d'écriture diminue, ce qui implique naturellement moins de messages échangés.

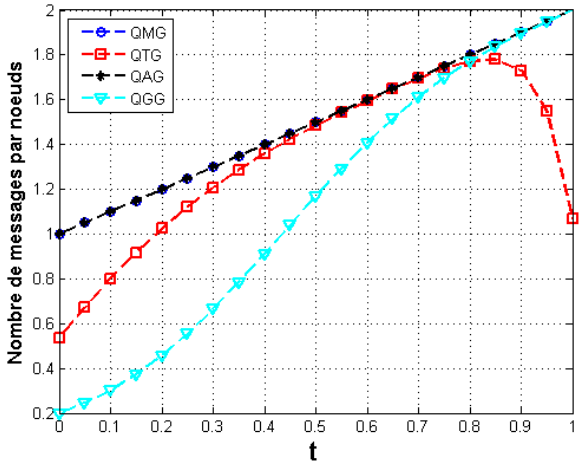
Nombre moyen de messages échangés pour l'opération de lecture

La figure 3.7 présente le nombre moyen de messages échangés (envoyés et reçus) par nœud pour l'opération de lecture en fonction de la disponibilité des nœuds t pour trois valeurs de r (4, 8 et 11). Le calcul de ce nombre prend en compte tous les messages lors d'une opération de lecture que cette dernière réussisse ou non. Dans cette sous-section, nous allons utiliser l'abréviation *NMOL* pour désigner le " *nombre moyen de messages échangés par nœud pour l'opération de lecture* ".

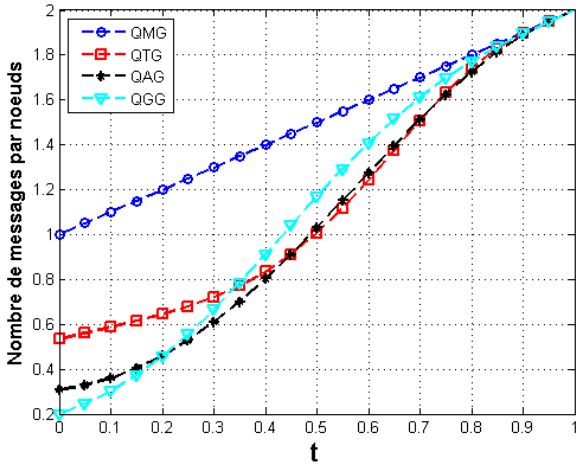
La figure 3.7(a) affiche les courbes de *NMOL* pour les quatre protocoles lorsque $r = 4$. Le plus grand *NMOL* correspond à celui du protocole *QMG* car il détecte tardivement la défaillance d'une opération de lecture par rapport aux trois autres protocoles. Pour une faible disponibilité en lecture ($t \leq 0.5$), le protocole *QGG* présente le plus faible *NMOL* comme montre la figure 3.7(a). En effet, lorsque la disponibilité des nœuds t est faible, il y a plus de chance de trouver une colonne de la grille ne contenant aucun nœud disponible. Or, lorsque le protocole rencontre cette colonne lors d'une opération de lecture, cette dernière ne sera pas validée car il n'est plus possible de retrouver avec certitude la dernière version de la donnée. Le protocole stoppe donc automatiquement le processus de lecture pour éviter de générer des messages inutiles. Dans le meilleur des cas, si c'est la première colonne consultée qui ne contient aucun nœud disponible alors il n'y aura que trois messages générés (3 *requêtes de lecture*) avant de stopper l'opération. A contrario, avec les protocoles *QTG* ou *QAG*, même si tous les nœuds aux niveaux $l = 0$ et $l = 1$ sont défaillants, il est encore possible d'obtenir la version de la donnée et de reconstituer la donnée originale en utilisant uniquement les nœuds résidant au niveau $l = 2$. En effet, le protocole *QAG* ne nécessite qu'un minimum de $D^2 - w_2 + 1 = 5$



(a) $n = 15$ et $r = 4$



(b) $n = 15$ et $r = 8$



(c) $n = 15$ et $r = 11$

FIGURE 3.7 – Nombre moyen de messages échangés par nœud pour l'opération de lecture

nœuds disponibles dont 4 au moins stockant une réplique à jour de la donnée. De même, le protocole *QTG* requiert seulement un minimum de $s_2 - w_2 + 1 = 4$ nœuds disponibles et contenant chacun une réplique à jour de la donnée. Il en résulte que ces protocoles nécessitent un nombre de messages échangés supérieur avant de se rendre compte que l'opération de lecture ne peut être validée. Lorsque la disponibilité des nœuds t est grande, les structures logiques imposées aux nœuds avec les protocoles *QTG* et *QAG* permettent à ces deux protocoles d'obtenir un faible *NMOL* (voir figure 3.7(a)). En effet, avec ces deux protocoles, lorsque la disponibilité des nœuds est élevée, il y a plus de chance que les niveaux $l = 0$ et $l = 1$ du trapèze ou de l'arbre soient suffisants pour retrouver la donnée originale. A l'inverse, le protocole *QGG* nécessite la visite des cinq colonnes pour obtenir la version de la donnée.

Lorsque la valeur du paramètre r passe à 8, le changement le plus perceptible concerne le *NMOL* du protocole *QAG* : sa courbe croît de manière constante avec la

disponibilité des nœuds t (voir figure 3.7(b)). Ce changement s'explique principalement par le fait que désormais les nœuds résidant aux niveaux $l = 0$ et $l = 1$ de l'arbre ne suffisent plus pour reconstituer la donnée originale. En effet, ces deux niveaux contiennent uniquement 4 nœuds alors qu'il en faut au moins 8. Par contre, avec le protocole *QTG*, les niveaux $l = 0$ et $l = 1$ du trapèze contiennent au total 8 nœuds. Ainsi, pour une grande valeur de la disponibilité des nœuds, la probabilité que ces 8 nœuds soient disponibles et contiennent chacun une réplique à jour de la donnée est plus importante. Ceci explique la décroissance de la courbe du *NMOL* du protocole *QTG* à partir de $t = 0.85$ environ. On remarque également une amélioration de la courbe du *NMOL* du protocole *QTG* pour les faibles valeurs de t. Ceci est dû au fait que contrairement au cas $r = 4$, les nœuds résidant au niveau $l = 2$ du trapèze ne suffisent plus pour reconstituer la donnée originale. En effet, lorsque tous les nœuds des niveaux $l = 0$ et $l = 1$ sont défaillants, il ne reste plus que 7 nœuds dans le niveau $l = 2$ alors qu'il en faut au moins 8 pour reconstituer la donnée originale. Par conséquent, dès que les nœuds aux niveaux $l = 0$ et $l = 1$ sont tous défaillants, le protocole stoppe automatiquement le processus d'opération de lecture.

Pour $r = 11$, toutes les courbes croissent avec la disponibilité des nœuds, y compris celle de *QTG*. Ceci s'explique par le fait que lorsque la valeur du paramètre r est élevée, il y a plus de chance que le protocole *QTG* utilise souvent les trois niveaux du trapèze lors d'une opération de lecture. En effet, les nœuds résidant aux niveaux $l = 0$ et $l = 1$ ne sont plus suffisants pour la reconstruction de la donnée originale. Nous pouvons remarquer également que pour des faibles valeurs de la disponibilité des nœuds t, il y a une diminution du *NMOL* des protocole *QTG* et *QAG* par rapport au cas $r = 8$. Par exemple, lorsque $t = 0.2$, le *NMOL* du protocole *QAG* est égal à 1.2 pour $r = 8$ (figure 3.7(b)) alors qu'il vaut 0.43 environ pour $r = 11$ (figure 3.7(c)). C'est surtout la structure logique imposée aux nœuds par les protocoles qui contribue à cette optimisation de *NMOL* pour des faibles valeurs de r. Comme le montre la figure 3.7(c) seuls les protocoles qui imposent une structure logique à l'organisation des nœuds présentent un faible *NMOL*.

Synthèse

Nous avons évalué plusieurs critères de performance de chacun des quatre protocoles que nous avons élaborés. L'objectif principal de ces évaluations est de dégager les avantages et les inconvénients de chaque protocole. Cela va permettre à CloViS de choisir le protocole de cohérence à utiliser selon les besoins prioritaires de l'utilisateur. Les critères de performances que nous avons retenus pour ces évaluations numériques sont la disponibilité en écriture et en lecture et le nombre de messages échangés par nœud pour ces opérations.

Le *quorum majoritaire général* présente une grande disponibilité en écriture mais son opération de lecture n'est pas adaptée lorsque la valeur du paramètre r est petite du fait que le nombre de nœuds requis pour obtenir la dernière version de la donnée reste toujours égale à $\lfloor \frac{n}{2} \rfloor + 1$. Cette contrainte conditionne la disponibilité en lecture

avec ce protocole. Par ailleurs, ce protocole présente également un nombre significatif de messages échangés pour les opérations d'écriture et de lecture. Ceci peut constituer un obstacle majeur pour les applications où la bande passante est un paramètre critique. Il n'est donc pas adapté, par exemple, aux flux multimédia (applications de *streaming multimédia*).

Le *quorum en grille général* présente également une disponibilité en écriture assez élevée et une disponibilité en lecture est faible par rapport à celle des autres protocoles de cohérence étudiés pour les faibles valeurs du paramètre r. Concernant le nombre de messages échangés, du fait de sa capacité à détecter rapidement la défaillance d'une opération d'écriture et de lecture, ce protocole est particulièrement adapté à une faible disponibilité des nœuds. En revanche, pour une grande valeur de disponibilité des nœuds, ce protocole échange beaucoup de messages lors des opérations de lecture et d'écriture. Ainsi, comme le *quorum majoritaire général*, ce protocole n'est pas adapté aux applications gourmandes en termes de bande passante.

Le *quorum en arbre général* est efficace surtout lorsque les données stockées sont en lecture seule ou qu'elles ne subissent que peu de modifications (lectures majoritaires ou exclusives). L'inconvénient majeur de ce protocole est sa faible disponibilité en écriture, due principalement à la structure de l'arbre où le seul nœud résidant au niveau $l = 0$ constitue un goulot d'étranglement lors d'une opération d'écriture. La défaillance de ce seul nœud lors d'une opération d'écriture entraîne la défaillance de cette dernière. Parmi les quatre protocoles considérés, c'est le *quorum en arbre général* qui présente le plus faible nombre de messages échangés par nœud pour les opérations d'écriture et de lecture. Cette solution pourrait-être utilisée dans les applications de base de données où la disponibilité et le taux de transactions sont des critères déterminants.

Le *quorum en trapèze général* présente la meilleure disponibilité en lecture par rapport aux trois autres protocoles. Il propose également une disponibilité en écriture assez élevée spécialement lorsque la disponibilité des nœuds est élevée. Le nombre moyen de messages échangés pour l'opération d'écriture est faible pour ce protocole, ce qui contribue à l'optimisation de la bande passante. Pour une faible disponibilité des nœuds, ce protocole échange peu de messages durant l'opération de lecture lorsque la valeur du paramètre r est strictement supérieur à s_h. Pour une forte disponibilité des nœuds, ce protocole échange également peu de messages lorsque la valeur du paramètre r est strictement inférieur à $n - s_h$. Le *quorum en trapèze général* présente non seulement un bon compromis entre disponibilité en écriture et en lecture mais aussi entre le nombre moyen de messages échangés lors des opérations d'écriture et de lecture.

3.7.2 Analyse de performances dans un environnement réel

Le système de fichiers de CloViS : CloViSFS

Avant d'effectuer une évaluation de performances dans un environnement réel, nous avons d'abord décidé d'améliorer le système de fichiers de CloViS, appelé CloViSFS. CloViSFS est implémenté en espace utilisateur et s'appuie sur FUSE(***F****ilesystem in*

U**ser**S**pac**E). Le choix de la librairie FUSE a été fait afin de permettre une intégration rapide au noyau linux tout en développant ClovisFS en espace utilisateur. Du fait de l'utilisation d'un système de gestion de fichiers compatible POSIX, l'ensemble des commandes shell relatives à la gestion des répertoires et des fichiers peuvent être utilisées. Le choix de cette sémantique permet d'assurer que la majorité des applications écrites ne nécessitent pas de réécriture/recompilation pour pouvoir être exécutées sur notre plateforme. Grâce à cette interface, un utilisateur peut accéder au système de fichiers à partir d'un point de montage sur sa machine.

Nous avons procédé à une ré-écriture totale du système de fichiers CloViS, en s'appuyant sur l'API bas niveau de FUSE plutôt que celle haut niveau utilisée dans l'ancienne version. Cette interface bas niveau offre une meilleure performance et plus de contrôle sur les opérations du système de fichiers au prix d'une complexité d'implémentation plus importante par rapport à l'interface haut niveau. Par exemple, l'interface manipule directement les i-nœuds alors celle haut niveau travaille sur les chemins de fichier. Dans ce dernier cas, FUSE garde à sa charge la translation d'un chemin de fichier en i-nœud de façon transparente pour le développeur, mais au prix d'une sollicitation accrue des ressources mémoire et cpu. En outre, nous avons implémenté l'algorithme CRUSH (***C**ontrolled **R**eplication **U**nder **S**calable **H**ashing*) pour gérer le placement des répliques dans l'ensemble des ressources de stockage de CloViS. L'algorithme CRUSH est détaillé dans [110]. Il permet de sélectionner, selon des critères définis par l'utilisateur, les nœuds à utiliser pour stocker une donnée lors d'une opération d'écriture. De façon symétrique, il est aussi responsable de la recherche des différents nœuds qui stockent les répliques d'une donnée lors d'une opération de lecture. Il supporte également la gestion de manière hiérarchique des dispositifs de stockages tels que l'armoire (rack), la salle, le bâtiment, la ville, le pays voire le continent pour affiner les règles de distribution.

Analyse réelle sur des charges simulées avec IOzone

La plate-forme de test est composée de nœuds de stockage de type *DELL Poweredge C2100*, équipés d'un système d'exploitation *Scientific Linux SL6.3* et interconnectés par un réseau infiniband 40 G. Tous les nœuds sont des bi-processeurs Intel Xeon 5500 (4 cœurs) à 2.2 Ghz, dotés de 24 Go de mémoire et équipés chacun de 7 disques :

- un disque SSD SATA 3GB/s de 200 Go réservé pour le système (*Seagate Pulsar.2 ST200FM0012*),
- 6 disques SAS 3 Gb/s de 146 Go pour les données (*Seagate ST3146855SS*).

L'évaluation des performances dans un environnement réel a été effectuée grâce au *benchmark* IOzone (version 3.430). Le *benchmark* IOzone génère et mesure la performance d'une grande variété d'opérations sur des fichiers [4]. Dans nos tests, pour chaque protocole de cohérence, nous nous intéressons à trois paramètres particuliers, à savoir, la durée d'exécution du test sur IOzone, la performance en écriture et celle en lecture. Pour chaque protocole de cohérence, nous avons effectué quatre séries de tests puis nous avons calculé ensuite la moyenne des résultats obtenus.

Dans ce test, nous avons utilisé les paramètres suivants :

- ☞ Quorum majoritaire général (QMG) : $n = 15$;
- ☞ Quorum en grille général (QGG) : $N = 3$ et $M = 5$;
- ☞ Quorum en arbre général (QAG) : $D = 3$, $h = 2$, $w_0 = 1$, $w_1 = 2$ et $w_2 = 4$;
- ☞ Quorum en trapèze général (QTG) : $a = 2$, $b = 3$, $h = 2$, $w_0 = 2$, $w_1 = 3$ et $w_2 = 4$.

Le nombre de répliques à jour requises par le protocole de distribution des données lors d'une opération de lecture est égal à $r = 4$.

- ⇨ **Durée d'exécution sur IOzone** : Ce paramètre correspond à la durée moyenne globale d'un test en utilisant IOzone. La figure 3.8 récapitule pour chaque protocole la durée moyenne globale d'un test. Les résultats obtenus confirment ceux présentés dans la sous-section 3.7.1 précédente. En effet, nous pouvons remarquer que la durée du test est proportionnelle au nombre de messages échangés lors de l'opération d'écriture et de lecture. Le *QAG* qui échangeait moins de messages par rapport aux autres protocoles, représente le meilleur temps de test dans nos évaluations (figure 3.8). Et inversement, c'est le *QMG* qui est le moins performant, confirmant ainsi, le nombre élevé de messages échangés lors d'une opération d'écriture et de lecture dans l'analyse théorique faite précédemment (sous-section 3.7.1). Le *QGG* et le *QTG* ont à peu près les mêmes temps d'exécution. Cela s'explique principalement par le fait que le *QGG* présente une taille de quorum en écriture inférieure à celle du *QTG* (donc un meilleur temps en écriture par rapport au *QTG*) et inversement, dans le meilleur des cas, le *QTG* peut accéder uniquement à quatre nœuds lors d'une opération de lecture alors qu'avec le *QGG*, lors d'une opération de lecture, il doit accéder au moins $N + M - 1 = 7$ nœuds (un meilleur temps en lecture par rapport au *QGG*). Ainsi, il est tout à fait possible donc que la somme de ces deux temps se rapprochent pour ces deux protocoles.

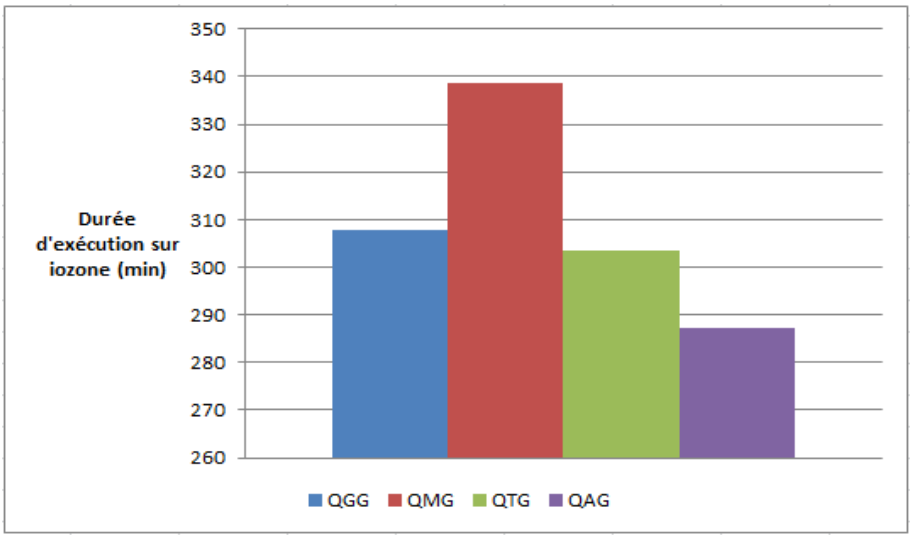


FIGURE 3.8 – Temps d'exécution sur IOzone

⇨ **Performance de l'opération d'écriture** : Ce paramètre mesure la performance de l'opération d'écriture pour chaque protocole de cohérence. Il s'agit d'une moyenne de performance de plusieurs types d'écriture de fichier (écriture d'un nouveau fichier, écriture d'un fichier qui existe déjà, écriture à des emplacements aléatoire dans le fichier, etc.). La figure 3.9 présente, pour chaque protocole, la performance en écriture mesurée lors de nos tests dans un environnement réel. Les résultats obtenus montrent que la meilleure performance en écriture correspond à celle du *QAG* et la moins bonne performance correspond à celle du *QMG*. La performance en écriture pour le *QGG* et celle de *QTG* se situent entre ces deux seuils. Ces résultats concordent en général avec ceux obtenus dans la sous-section 3.7.1 concernant le nombre moyen de messages échanges pour l'opération d'écriture. Néanmoins, il existe une petite discordance entre le *QTG* et le *QGG*. En effet, ce dernier présente une meilleur performance en écriture alors qu'en moyenne, il échange plus de messages lors de l'opération d'écriture que le *QTG*. Mais ce résultat était prévisible car cela provient du fait que la taille du quorum d'écriture du *QGG* est inférieure à celle du *QTG*, respectivement égales à sept et neuf. Ainsi, même si le nombre moyen de messages échangés lors de l'opération d'écriture avec le *QGG* est supérieur à celui de *QTG*, il y a des cas où le nombre de nœuds accédés lors d'une opération d'écriture avec le *QGG* est inférieur à celui de *QTG*. Par exemple, ce scénario se présente lorsque les sept premiers nœuds accédés sont tous disponibles, qui constituent donc un quorum d'écriture avec le *QGG* alors qu'il en faut neuf avec le *QTG*.

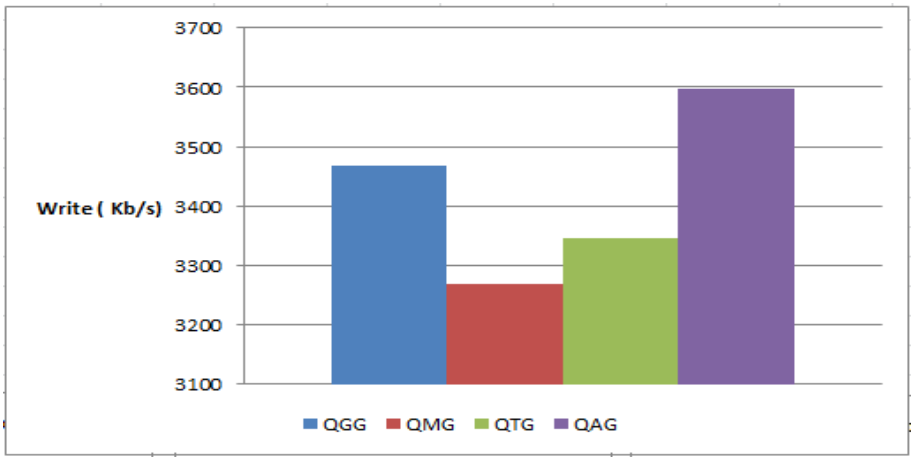


FIGURE 3.9 – Performance de l'opération d'écriture

⇨ **Performance de l'opération de lecture** : Ce paramètre mesure la performance de l'opération de lecture pour chaque protocole de cohérence. Il s'agit d'une moyenne de performance de plusieurs types de lecture (lecture d'un fichier, lecture d'un fichier qui a été lu récemment, lecture à des endroits aléatoire dans le fichier, etc.). La figure 3.10 présente, pour chaque protocole, la performance en lecture mesurée lors de nos tests dans un environnement réel. Les résultats montrent que le *QTG* et le *QAG* présentent les meilleurs performances en lecture, vient ensuite le *QGG* et enfin, le *QMG* représente la moins bonne performance en lecture. Ce

résultat était prévisible car il confirme le fait que la performance en lecture est inversement proportionnelle au nombre de nœuds accédés lors d'une opération de lecture.

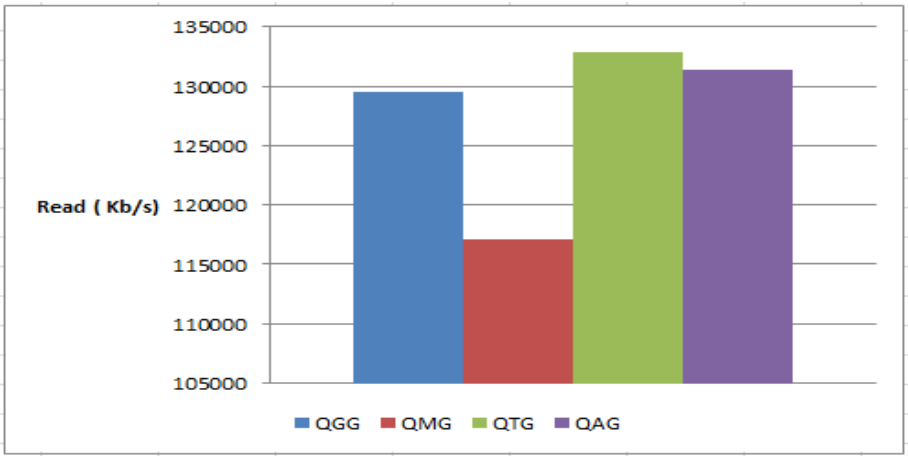


FIGURE 3.10 – Performance de l'opération de lecture

Synthèse

Nous avons évalué les différents protocoles de cohérence élaborés sur une infrastructure réelle. Les résultats obtenus confirment ceux présentés lors de l'évaluation numériques dans la sous-section 3.7.1. Le QAG offre une meilleure performance en écriture et en lecture mais son principal inconvénient est sa faible disponibilité en écriture dûe au fait que l'unique nœud se trouvant au niveau $l = 0$ constitue un goulot d'étranglement. Le QMG présente la plus faible performance en écriture et en lecture parmi les quatre protocoles de cohérence élaborés et donc un temps d'exécution plus long. Le QGG a une performance comparable à celle de QTG en terme de temps d'exécution. Mais en moyenne le QGG échange plus de messages que le QTG, ceci est dû au coût élevé de la formation d'un quorum d'écriture avec le QGG en cas de défaillance d'un ou plusieurs nœuds.

Conclusion et perspectives

Le stockage des données est devenu l'une des composantes fondamentales du cloud computing. Il doit répondre à un certain nombre d'objectifs parmi lesquels figurent la performance (temps d'accès aux données et débit), le passage à l'échelle, la fiabilité des opérations qui y sont effectuées, la disponibilité des données, etc. Dans cette thèse, nous nous sommes focalisés sur les deux critères que sont la disponibilité des données et la gestion de la cohérence des répliques. Pour répondre au critère de disponibilité des données au sein de CloViS, nous avons utilisé des techniques de distribution des données issues de la littérature. Elles sont présentées sous forme de qualité de services implémentées par le biais de librairies à liaison dynamique enfichables à chaud. La gestion de la cohérence des données est implémentée, dans CloViS, par le composant CCCC. Ce composant doit mettre en œuvre un protocole de cohérence efficace et performant pour ne pas pénaliser le temps d'accès aux données de CloViS et éviter la perte des données. En effet, la défaillance de ce composant est susceptible de causer une perte irréparable partielle voire totale de données.

Ainsi, dans un premier temps, nous avons proposé d'utiliser au sein de CloViS des techniques de distribution des données de types codes correcteurs ou schémas à seuil qui permettent de trouver un meilleur compromis entre la disponibilité des données et le surcoût généré comparé au mécanisme habituel de réplication totale. Ces techniques offrent également à CloViS la possibilité de définir plusieurs niveaux de qualité de services en termes de disponibilité des données ou d'espace de stockage, permettant ainsi d'adapter les paramètres du protocole de distribution des données selon les critères prioritaires de l'utilisateur (qu'il s'agisse de la disponibilité des données ou de l'espace de stockage, voire d'un compromis entre les deux). En effet, ces protocoles de distribution des données sont beaucoup plus souples que la réplication totale. Même si cette dernière offre la meilleure disponibilité des données possible, elle implique également une utilisation excessive des ressources du système telles que le réseau, l'espace de stockage, etc.

Ensuite, nous avons élaboré quatre protocoles de gestion de répliques motivés par CloViS : le *quorum majoritaire général*, le *quorum en grille général*, le *quorum en arbre général* et le *quorum en trapèze général*. Pour cela, nous avons repris différents protocoles de cohérence à quorums issues de la littérature et nous y avons apporté les modifications nécessaires afin qu'ils puissent garantir toutes les conditions requises par la technique de distribution des données utilisée. Ces conditions sont, entres autres, de pouvoir garantir

le nombre de répliques requises pour la reconstitution de la donnée originale lors d'une opération de lecture, et d'ajuster la taille minimale du quorum d'écriture lors d'une opération d'écriture. Pour cela, nous avons redéfini les quorums d'écriture et de lecture afin de maintenir la cohérence des données à tout moment et de supporter plusieurs types de techniques de distribution des données dont, entre autres, les codes correcteurs et les schémas à seuil. Les adaptations apportées à ces protocoles permettent à CloViS de disposer d'un large choix de techniques de distribution des données à utiliser comme les codes correcteurs, les schémas à seuil, le RAID, etc. Les évaluations numériques que nous avons effectuées nous ont permis de mettre en exergue les avantages et les inconvénients respectifs de chaque protocole. Les résultats ont montré que parmi les quatre protocoles de gestion de répliques proposés c'est le protocole de *quorum en trapèze général* qui présente un meilleur compromis entre les quatre critères de performance que nous avons considérés, en l'occurrence, la disponibilité en écriture et en lecture, et le nombre de messages échangés par nœud lors des opérations d'écriture et de lecture.

Enfin, un dernier point sur lequel nous avons travaillé pendant cette thèse est l'amélioration de la performance du systèmes de fichiers de CloViS : CloViSFS. Pour cela, nous avons effectué une ré-écriture totale du système de fichiers de CloViS, en utilisant l'interface bas niveau de l'API de FUSE plutôt que celle de haut niveau utilisée précédemment. Cette interface bas niveau offre une meilleure performance et plus de contrôle sur les opérations du système de fichiers au prix d'une complexité d'implémentation plus importante par rapport à l'interface haut niveau. Nous avons également mis en place l'algorithme CRUSH (***C**ontrolled **R**eplication **U**nder **S**calable **H**ashing*) pour optimiser le placement des données dans l'ensemble des ressources de stockage physiques mises à disposition [110]. Lors d'une opération d'écriture, cet algorithme sélectionne les nœuds à utiliser pour le stockage selon les critères indiqués. De même, lors d'une opération de lecture, il se charge de trouver les nœuds qui stockent les différentes répliques de la donnée. On peut y définir des hiérarchisations telles que l'armoire (rack), la salle, le bâtiment, la ville, le pays voire le continent pour affiner les règles de distribution.

Perspectives

Adapter d'autres protocoles de cohérence issus de la littérature

Il est souhaitable d'explorer d'autres types de protocoles de cohérence dans le contexte de CloViS. En effet, la performance d'un protocole de cohérence dans le contexte de la réplication totale n'est pas liée à la performance dans notre contexte, plus général. Pour déterminer les avantages et inconvénients de ces autres protocoles issus de la littérature, il faut donc procéder à une nécessaire adaptation, puis à des évaluations numériques. Cela devrait permettre à CloViS de proposer une liste significative de protocoles de cohérence, avec leurs avantages et inconvénients respectifs, afin de pouvoir sélectionner le meilleur protocole de cohérence des données selon les besoins de l'utilisateur.

Envisager d'autres techniques de distribution des données

Nous avons, jusqu'à présent, considéré que chaque bloc de données était scindé en plusieurs sous-blocs, ces sous-blocs étant utilisés avec une technique de type code correcteur ou schéma à seuil avant d'être distribués sur l'ensemble des nœuds de stockage. Cette technique présente l'inconvénient d'un morcellement important des données. Une autre approche est possible : chaque bloc de données peut être combiné avec d'autres blocs existants dans le système par le biais des techniques précédemment évoquées avant d'être stocké. Cela entraine un morcellement moins important, mais soulève d'autres problèmes tels que la mise à jour des blocs "appairés". De surcroit le nombre et la taille des messages varie significativement. Il convient donc d'étudier plus en détail ces deux solutions pour évaluer leurs avantages et inconvénients respectifs et déterminer les conditions optimales pour leur mise en œuvre.

Évaluer les différents protocoles élaborés sur une infrastructure représentative

Après avoir effectué des évaluations numériques sur les quatre protocoles de cohérence définis, nous avons effectué des évaluations de ces protocoles en environnement réel, sur une grappe de stockage composée de quatre nœuds. Ces premiers résultats, présentés dans la section 3.7.2, ont montré la pertinence de notre évaluation numérique. Néanmoins ils doivent être confirmés dans des conditions plus proches d'une utilisation réelle de CloViS : nombre de nœuds supérieur, répartis sur plusieurs sites. Ainsi, ces évaluations permettront, nous l'espérons, de valider le modèle présenté. Ces expérimentations utiliserons les ressources disponibles dans GRID 5000 pour se rapprocher d'un environnement de production représentatif. Ces tests seront complétés par des mesures de performance comparatives entre CloViS et d'autres outils dédiés au stockage Cloud tels que Ceph et GlusterFS.

Bibliographie

[1] AWS Elastic Beanstalk. `https://aws.amazon.com/fr/elasticbeanstalk/`.

[2] ConPaaS : User Guide. `https://conpaas-team.readthedocs.org/en/1.3.1/userguide.html`.

[3] Filesystem in Userspace(FUSE). `http://fuse.sourceforge.net/`.

[4] IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[5] Terrastore. https ://code.google.com/p/terrastore/.

[6] Sun Microsystems, Inc., Santa Clara, CA, USA : Lustre file system - High performance storage architecture and scalable cluster file system, December 2007.

[7] Best Practices for Architecting a Lustre - Based Storage Environment. Whitepaper : DataDirect Networks, Inc., 2008.

[8] Gluster : An introduction to gluster architecture, 2011.

[9] Le cloud computing une nouvelle filière fortement structurante, september 2012. Direccte Île-de-France : Etudes sectorielles.

[10] Ortiz A., Thiébolt F., Jorda J., and Mzoughi A. How to use multicast in distributed mutual exclusion algorithms for grid file systems. *Conference on High Performance Computing and Simulation (HPCS)*, pages 122–130, 2009.

[11] Ortiz A., Jorda J., and Mzoughi A. Toward a new direction on data management in grids. *15th IEEE International Conference on High Performance Distributed Computing*, pages 377–378.

[12] D. Agrawal and A. El Abbadi. The tree quorum protocol : An efficient approach for managing replicated data. *Proceedings of the 16th VLDB Conference Brisbane, Australia*, 1990.

[13] Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1) :1–20, February 1991.

[14] Marcos K. Aguilera, Ramaprabhu Janakiraman, and Lixao Xu. Using erasure codes efficiently for storage in a distributed system. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN'05, pages 336–345. IEEE, 2005.

[15] M. Ahamad, M. Ammar, and S. Cheung. Replicated data management in distributed systems. *T.L. casavant and M. Singhal, editors, Readings in distributed computing systems*, pages 572–591, January 1994.

[16] B. Allcock, J. Bester, J. Bresnahan, J. Bester, Ann L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. *IEEE Mass Storage Conference*, 2001.

[17] Pistirica Sorin Andrei, Asavei Victor, Geanta Horia, Moldoveanu Florica, Moldoveanu Alin, Negru Catalin, and Mocanu Mariana. Evolution towards distributed storage in a nutshell. *IEEE International Conference on High Performance Computing and Communications (HPCC),IEEE 6th International Symposium on Cyberspace Safety and Security (CSS) and IEEE 11th International Conference on Embedded Software and Systems (ICESS)*, pages 1267–1274, 2014.

[18] Rowstron Antony and Druschel Peter. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[19] H. Peter Anvin. The mathematics of raid-6. `http://ftp.tux.org/pub/kernel/people/hpa/raid6.pdf`, January 2004.

[20] Masayuki Arai, Tabito Suzuki, and Mamoru Ohara. Analysis of read and write availability for generalized hybrid data replication protocol. *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*, pages 143–150, March 2004.

[21] Jerzy B. and Sobaniec C. Kalewski, M. Safety of a session guarantees protocol using plausible clocks. In *Proceedings of the Seventh international Conference on Parallel Processing and Applied Mathematics (PPAM ?2007), LNCS 4967*, Gdansk, Poland, 2008.

[22] V. Balegas, N. Preguiça, R. Rodrigues, S. Duarte, C. Ferreira, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. *The European Conference on Computer Systems (EuroSys)*, April 2015.

[23] E. R. Berlekamp. *Algebraic Coding Theory.* McGraw-Hill, New York, 1968.

[24] Eric Brewer. Towards robust distributed system. *Symposium on Principles of Distributed Computing (PODC)*, 2000.

[25] R. Buyya, T. Cortes, and H. Jin. A case for redundant arrays of inexpensive disks (raid). *High Performance Mass Storage and Parallel I/O :Technologies and Applications*, pages 2–14, 2002.

[26] Brad Calder et al. Windows azure storage : A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[27] M. J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Transactions on Database System*, 16(4) :703–746, December 1991.

[28] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4) :12–27, 2011.

[29] Eugenio Cesario, Toni Cortes, Erich Focht, Matthias Hess, Felix Hupfeld, and et al. Xtreemfs - an object-based file system for federated it infrastructures, 2007.

[30] Ye-In Chang and Yao-Jen Chang. A fault-tolerant trangular mesh protocol for dirstibuted mutual exclusion. *Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 694–701, 1995.

[31] Peter M Chen, Garth A. Gibson, Randy H. Katz, and David A. Patterson. An evaluation of redundant arrays of disks using an amdahl 5890. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990.

[32] Peter M Chen, Edward K. Lee, Garth A. Gibson, David A. Patterson, and Randy H. Katz. Raid : High-performance, reliable secondary storage. *ACM Computing Surveys*, October 1993.

[33] Shun Yan Cheung, Mostafa H. Ammar, and Mustaque Ahamad. The grid protocol : A high performance scheme for maintaining replicated data. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 438–445, Washington, DC, USA, 1990. IEEE Computer Society.

[34] Houssem-Eddine Chihoub, Shadi Ibrahim, Yue Li, Gabriel Antoniu, and et al. Exploring energy-consistency trade-offs in cassandra cloud storage system. *27th International Symposium on Computer Architecture and High Performance Computing*, 2015.

[35] Cédric Coulon. *Réplicaation Préventive dans une grappe de bases de données.* PhD thesis, Université de Nantes, Septembre 2006.

[36] N. das Chagas Mendonça and R. de Oliveira Anido. Using extended hierarchical quorum consensus to control replicated data : from traditional voting to logical structures. *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, 2 :303–312, jan 1994.

[37] Benjamin Depardon, Gaël Le Mahec, and Cyril Séguin. Analysis of six distributed file systems. *[Research Report]*, 2013.

[38] M.Mat Deris, J.H. Abawajy, and H.M. Suzuri. An efficient replicated data access approach for large-scale distributed systems. *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2004.

[39] Alexandros G. Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3) :476–489, 2011.

[40] Yann Dupont. Stockage distribué : retour d'expérience avec ceph. *Journées réseaux (JRES)*, 2013.

[41] Jon G. Elerath and Michael Pecht. A highly accurate method for accessing reliability of redundant arrays of inexpensive disks (raid). *IEEE Computer Society*, Sep 2008.

[42] Ivan Frain. *Systèmes à quorums dynamiques adaptés aux grilles informatiques*. PhD thesis, Université Toulouse III - Paul Sabatier, Décembre 2006.

[43] Ada Wai-Chee Fu, Yat Sheung Wong, and Man Hon Wong. Diamond quorum consensus for high capacity and efficiency in a replicated database system. *Distrib. Parallel Databases*, 8(4) :471–492, October 2000.

[44] Ada Waichee Fu, Fu Wai, Kwong Lau, Fuk Keung, Ng Man, and Hon Wong. Hypercube quorum consensus for mutual exclusion and replicated data management. *Computers and Mathematics with Applications, An International Journal*, 36(5) :45–59, 1998.

[45] Gregory R. Ganger, Pradeep K. Khosla, and Mehmet Bakkaloglu. Survivable storage systems. *DARPA Information Survivability Conference & Exposition II*, 2 :184–195, 2001.

[46] Devarshi Ghoshal and Lavanya Ramakrishnan. Frieda : Flexible robust intelligent elastic data management in cloud environments. *SC Companion : High Performance Computing, Networking Storage and Analysis*, pages 1096–1105, 2012.

[47] G.A. Gibson. Redundant disks arrays : Reliable, parallel secondary storage. *PhD dissertation, Dept. Computer Science, UC Berkeley*, Apr 1991.

[48] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–159, 1979.

[49] Jim Gray, Bob Horst, and Mark Walker. Parity striping of disc arrays : Low-cost reliable storage with acceptable throughput. In *Proceedings of the 16th Very Large Database Conference*, pages 148–160, May 1990.

[50] Diana Gudu, Marcus Hardt, and Streit Achim. Evaluating the performance and scalability of the ceph distributed storage system. *IEEE International Conference on Big Data*, 2014.

[51] Tonglin Hawk, Ioan Raicu, and Lavanya Ramakrishnan. Scalable state management for scientific applications in the cloud. *IEEE International Congress on Big Data*, 2014.

[52] Maurice Herlihy. General quorum consensus : a replication method for abstract data types. *Tech. Rep. CMU-CS-84-164, Carnegie-Mellon University*, December 1984.

[53] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1) :32–53, February 1986.

[54] R. J. Honicky and Ethan L. Miller. A fast algorithm for online placement and reorganization of replicated data. *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, Apr 2003.

[55] R. J. Honicky and Ethan L. Miller. Replication under scalable hashing : A family of algorithms for decentralized data distribution. *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Apr 2004.

[56] Cheng Huang, Minghua Chen, and Jin Li. Pyramid codes : Flexible schemes to trade space for access efficiency in reliable data storage systems. *Trans. Storage*, 9(1) :3 :1–3 :28, March 2013.

[57] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

[58] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. Xtreemfs - a case for object-based storage in grid data management. In *Proceedings of 33th International Conference on Very Large Data Bases (VLDB) Workshops*, 2007.

[59] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. Xtreemfs - a case for object-based file systems in grids. In *Concurrency and Computation : Practice and Experience*, volume 20, 2008.

[60] Douceur John R. and Wattenhofer Roger P. Optimizing file availability in a secure serverless distributed file system. In *Proceedings of the 20th Symposium on Reliable Distributed Systems*, 2001.

[61] Ehud D. Karnin, Jonathan W. Greene, and Martin E. Hellman. On secret sharing systems. *IEEE Transaction on Informations Theory*, 29(1) :35–41, January 1983.

[62] Sohyun Koo, Se Jin Kwon, Sungsoo Kim, and Tae-Sun Chung. Dual raid technique for ensuring high reliability and performance in ssd. *IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS)*, pages 339–404, June 2015.

[63] Akhil Kumar. Performance analysis of a hierarchical quorum consensus algorithm for replicated objects. *Proceedings., 10th International Conference on Distributed Computing Systems*, pages 378–385, 1990.

[64] Akhil Kumar. Hierarchical quorum consensus : A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9) :996–1004, sep 1991.

[65] Bing Li, Yutao He, and Ke Xu. The nosql principles and basic application of cassandra model. *International Conference on Computer Science and Service System*, pages 1332–1335, 2012.

[66] Hsiao-Ying Lin, Li-Ping Tung, and Bao-Shuh P. Lin. An empirical study on data retrievability in decentralized erasure code based distributed storage systems. In *Proceedings of IEEE 7th International Conference on Software Security and Reliability - SERE'13*, pages 30–39, 2013.

[67] Hsiao-Ying Lin and Wen-Guey Tzeng. A secure erasure code-based cloud storage system with secure data forwarding. *IEEE Transactions on Parallel and Distributed Systems*, 23(6) :995–1003, 2012.

[68] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The Theory of Error Correcting Codes, Part 1*, volume 16 of *North-Holland mathematical library*. North-Holland Publishing Company, 1977.

[69] Mamoru Maekawa. A sqrt(n) algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2) :145–159, May 1985.

[70] Dahlia Malkhi, Michael Reiter, and Rebecca Wright. Probabilistics quorum systems. *Information and Computation*, 170 :184–206, 2001.

[71] Gunnar Mathiason, Sten F. Andler, and Sang H. Son. Virtual full replication by adaptative segmentation. *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 327–336, 2007.

[72] Peter Mell and Timothy Grance. NIST : The NIST Definition of Cloud Computing. `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`.

[73] M. Naor and A. Wool. The load, capacity and availability of quorum systems. *SIAM Journal on Computing*, 27(2) :423–447, April 1998.

[74] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Transaction on Computer Systems*, 6(1) :134–154, February 1988.

[75] Aurélien Ortiz. *Contrôle de la concurrence dans les grilles informatiques. Application au projet ViSaGe*. PhD thesis, Université Toulouse III - Paul Sabatier, Décembre 2009.

[76] Mayur Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids : a viable solution ? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64, June 2008.

[77] David A. Patterson, Peter Chen, G. Gibson, and Randy H. Katz. Introduction to redundant arrays of inexpensive disks (raid). *COMPCON Spring '89. Thirty-Fouth IEEE Computer Society International Conference : Intellectual Leverage, Digest of Papers*, pages 112–117, 1989.

[78] David A. Patterson, Garth A. Gibson, and Randy H. Katz. Redundant arrays of inexpensive disks (raid). *University of California,Technical Report UCB/CSD 87/391, Berkeley CA*, December 1987.

[79] David A. Patterson, Garth A. Gibson, and Randy H. Katz. Redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, June 1988.

[80] David Peleg and Avishai Wool. Crumbling walls : a class of practical and efficient quorum systems. *Distributed Computing*, 10 :87–97, 1997.

[81] Ying Peng, Yan Zhu, and Jinling Luo. A network storage framework based on san. *The 7th International Conference on Computer Science & Education (ICCSE)*, pages 818–821, July 2012.

[82] W. W. Peterson and E. J. Weldon. *Error-Correcting Codes, Second Edition*. The MIT Press, Cambridge, Massachusetts, 1972.

[83] Guillaume Pierre and Corina Stratan. Conpaas : A platform for hosting elastic cloud applications. *IEEE Internet Computing*, pages 88–92, 2012.

[84] Michael Rabinovich and Edward D. Lazowska. An efficient and highly available read-one write-all protocol for replicated data management. *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 56–65, 1993.

[85] K.V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 331–342, New York, NY, USA, 2014. ACM.

[86] Théodore Jean Richard Relaza, Jacques Jorda, and Abdelaziz M'zoughi. Majority quorum protocol dedicated to general threshold schemes. *15th IEEE/ACM international Symposium on Cluster, Cloud and the Grid Computing (CCGrid)*, pages 785–788, 2015.

[87] Théodore Jean Richard Relaza, Jacques Jorda, and Abdelaziz M'zoughi. Trapezoid quorum protocol dedicated to erasure resilient coding based schemes. *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 1082–1088, 2015.

[88] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants : novel erasure codes for big data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 325–336. VLDB Endowment, 2013.

[89] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11) :612–613, nov 1979.

[90] Marc Shapiro. A principled approach to eventual consistency. *20th IEEE International Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises (WETICE)*, June 2011.

[91] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. *[Research Report] RR-7506*, June 2011.

[92] Andrew M. Shooman and Martin L. Shooman. A comparison of raid storage schemes : Reliability and efficiency. *Reliability and Maintainability Symposium (RAMS), 2012 Proceedings - Annual*, pages 1–6, 2012.

[93] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[94] N. S. Sudharsan and Dr. K. Latha. Improvising seeker satisfaction in cloud community portal : Dropbox. *International conference on Communication and Signal Processing*, pages 321–325, 2013.

[95] Tabito Suzuki and Mamoru Ohara. Analysis of probabilistic trapezoid protocol for data replication. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 782–791, 2005.

[96] A. Tanenbaum and M. Van Steen. Distributed systems : Principles and paradigms, 2002.

[97] Yusuke Tanimura, Seiya Yanagita, and Takahiro Hamanishi. A high performance, qos-enabled, s3-based object store. *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 784–791, 2014.

[98] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2) :180–209, June 1979.

[99] Sterling Thomas, Becker Donald J., Savarese Daniel, Dorband John E., Ranawake Udaya A., and Packer Charles V. Beowulf : A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[100] Martin Tompa and Heather Woll. How to share a secret with cheaters. *Journal of Cryptology*, pages 133–138, 1988.

[101] Alejandro Z. Tomsic, Tyler Crain, and Marc Shapiro. An empirical perspective on causal consistency. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, April 2015.

[102] B.G. Tudorica and C. Bucur. A comparison between several nosql databases with comments and notes. *10th Romanian Educational Network International Conference (RoEduNet)*, pages 1–5, June 2011.

[103] L. Richard Turner. Inverse of the vandermonde matrix with applications. 1966.

[104] Steven Verkuil. A comparison of fault-tolerant cloud storage file systems. *19th Twente Student Conference on IT*, June 2013.

[105] Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. Understanding lustre filesystem internals. *Technical Report ORNL/TM-2009/117, Oak Ridge National Laboratpry, National Center for Computational Sciences*, 2009.

[106] Haiyang Wang, Ryan Shea, Feng Wang, and Jiangchuan Liu. On the impact of virtualization on dropbox-like cloud file storage/synchronization services. *IEEE 20th International Workshop on Quality of Service (IWQoS)*, pages 1–9, June 2012.

[107] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication : A quantitative comparison. In *Revised Papers from the First International Work-*

shop on Peer-to-Peer Systems, IPTPS '01, pages 328–338, London, UK, UK, 2002. Springer-Verlag.

[108] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Cloudtps : Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing, Special Issue on Cloud Computing*, 2011.

[109] Sage Weil. *CEPH : Reliable, scalable, and high-performance distributed storage.* PhD thesis, University of California - SANTA CRUZ, 2007.

[110] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush : Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE SC—06 Conference (SC'06)*, 2006.

[111] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush : Controlled, scalable, decentralized placement of replicated data. *Proceedings of the 2006 ACM/IEEE SC—06 Conference (SC'06)*, 2006.

[112] Bolosky William J., Douceur John R., Ely David, and Theimer Marvin. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems*, 2000.

[113] Jay J. Wylie, Michael W. Brigrigg, and John D. Strunk. Survivable information storage systems. *IEE Computer*, 2000.

[114] Feng Yan, Alma Riska, and Evgenia Smirni. Fast eventual consistency with performance guarantees for distributed storage. *32nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 23–28, June 2012.

[115] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past : Causal consistency for client-side applications. In *Annual ACM/IFIP/Usenix Middleware coference*, Vancouver, BC, Canada, December 2015.

[116] Guangyan Zhang, Weimin Zheng, and Keqin Li. Rethinking raid-5 data layout for better scalability. *IEEE Transaction on Computers*, 63(11) :2816–2828, 2014.