Towards Malleable Distributed Storage Systems: From Models to Practice

Nathanaël Cheriere

THÈSE DE DOCTORAT DE

L'ÉCOLE NORMALE SUPÉRIEURE DE RENNES
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Nathanaël CHERIERE

Towards Malleable Distributed Storage Systems: from Models to Practice

Thèse présentée et soutenue à RENNES, le 5 novembre 2019
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Toni Cortés, Associate Researcher, Universitat Politècnica de Catalunya
Frédéric Desprez, Directeur de recherche, Inria - Grenoble

Composition du jury :

Présiden :	**François Taiani**, Professeur des Universités, Université Rennes 1
Examinateurs :	**Toni Cortés**, Associate Researcher, Universitat Politècnica de Catalunya
	Frédéric Desprez, Directeur de recherche, Inria - Grenoble
	Kate Keahey, Senior Research Scientist, Argonne National Laboratory
Dir. de thèse :	**Gabriel Antoniu**, Directeur de recherche, Inria - Rennes
Co-dir. de thèse :	**Matthieu Dorier**, Software Development Specialist, Argonne National Laboratory

All models are wrong but some are useful
- George Box

ACKNOWLEDGMENTS

I would like to start by thanking my reviewers, Toni Cortés and Frédéric Desprez, and the other members of the jury, Kate Keahey and François Taiani, for taking the time to evaluate this work.

Many persons have made this work possible, starting from my PhD advisors: Matthieu Dorier, who provided among many things very helpful technical help and had an impressive response time, and Gabriel Antoniu who pushed me to publish and present my work in top conferences and gave me the freedom to follow my ideas. This work would also not have been possible without Shadi Ibrahim inviting me to the Kerdata team.

I am also very grateful to Rob Ross for hosting me at Argonne National Laboratory and for the fruitful high level discussions. Many thanks to Stefan Wild and Sven Leyffer for the discussions about formalizing multi-objective problems and techniques to find exact solutions.

I need to thank my good friends Luis and Lucy for tolerating my complaints and for all the awesome food and drinks we shared.

I also want to thank my office-mates, Lokmam, Luis, Yacine, and Laurent for the time spent together and the long discussions about pretty much anything. I extend my thanks to all the members of Kerdata I have met, Orçun, Tien-Dat, Alex, Luc, Ovidiu, Hadi, Paul, Amelie and Pedro, who made Kerdata a comfortable place to be in.

This PhD could not have been possible without the help of the team assisants Gaëlle, and Aurélie, as well as Elodie who helped with a lot of paperwork and with the missions.

I cannot forget to thank my interns, Tom and Juliette, and my students Titouan, Santiago, Alexandre, and Nicolas who were a pleasure to advise and with whom I learned a lot.

There are a lot more persons to thank, my parents, siblings, the friends from Argonne, from the JLESC, the Mochi team, random encounters in conferences... to all of you, thank you!

RÉSUMÉ EN FRANÇAIS

Contexte

En juin 2016, Sunway Taihulight, le premier supercalculateur dépassant les 10 millions de cœurs, prit la première place du classement mondial des plus puissants supercalculateurs, le Top500 [1].[1] Cette machine, installée au National Supercomputing Center à Wuxi (Chine), a pu atteindre 93 014,6 Tflops (93×10^{15} opérations en virgule flottante par seconde) en utilisant un total de 10 649 600 cœurs. À la même période, le nombre de serveurs utilisés par Amazon Web Services [2] était estimé à 1,3 millions [3]. Une telle puissance de calcul est utile dans de nombreux domaines. En recherche, elle permet la simulation de phénomènes complexes comme la physique des particules, et aide à l'analyse de grandes quantités de données obtenues par observation et expérimentation. De nombreuses entreprises utilisent aussi les calculs intensifs pour fournir des services utiles dans la vie de tous les jours. Par exemple, les prévisions météorologiques ont été considérablement améliorées depuis 1950 grâce à l'utilisation de supercalculateurs. En 1950, les scientifiques pouvaient prévoir la météo avec succès 24 heures à l'avance. Aujourd'hui, il est possible de prévoir la météo plusieurs semaines à l'avance grâce au calcul de modèles météorologiques très précis [4].

Les systèmes à large échelle utilisés pour exécuter ces applications nécessitant une grande puissance de calcul sont rarement dédiés à une seule application; la plupart du temps les ressources de la plateforme sont partagées entre plusieurs applications exécutées simultanément. Le partage de ressources de calcul est de fait un problème apparu en même temps que les ordinateurs eux-mêmes. Les plateformes de calcul sont souvent mutualisées entre plusieurs applications et utilisateurs principalement à cause de leur coût d'acquisition et d'entretien. Les ordinateurs centraux achetés par les universités dans les années 1960 et 1970 étaient partagées en allouant des plages horaires aux utilisateurs ou simplement en exécutant les applications dans l'ordre de leur soumission. John McCarthy présenta le concept d'*utility computing* en 1961 [5]. Il anticipait la possibilité d'accéder à de la puissance de calcul comme d'autres services comme l'eau et l'électricité. Une première mise en place de cette idée fut la création de *grilles de calcul* dans les années 1990 : les ressources de calcul de plusieurs institutions furent mises en commun pour donner accès aux utilisateurs à un plus grand ensemble de ressources. Cette approche permet aussi la réduction du coût d'entretien de ces ressources en rassemblant et mutualisant des services essentiels tels que le support utilisateur. Le Cloud, développé dans les années 2000, est plus proche de la vision de McCarty : les utilisateurs peuvent louer des ressources à des entreprises et elles leur sont facturées selon l'utilisation.

La gestion fine des ressources de plateformes partagées permet aux utilisateurs de réserver de nouvelles ressources quand elles sont nécessaires, et de les rendre à la plateforme au moment où elles ne sont plus utiles. Cette gestion élastique des ressources a de nombreux avantages pour les applications : celles-ci peuvent ajuster les ressources qu'elles utilisent à

[1] Il est classé troisième en Juin 2019.

leur charge de travail même si cette charge est imprévisible, à un coût optimal puisque seules les ressources nécessaires sont réservées.

Cependant, cette gestion fine des ressources peut parfois être limitée par le stockage et la gestion de grandes quantités de données manipulées par certaines applications. L'avènement récent des Big Data a conduit à la création de nombreuses applications manipulant d'immenses quantités de données. Par exemple, certains ensembles de données utilisés pour entrainer des réseaux de neurones profonds dépassent 100 Tio [6]. La manipulation de telles quantités de données rend l'efficacité de leur gestion critique pour assurer une bonne performance de ces applications. En particulier, la co-location entre tâches et données est essentielle : en exécutant les tâches de calcul sur les nœuds (serveurs) qui stockent les données qui leur sont nécessaires, l'utilisation du réseau et la latence sont réduites. Cette technique a été popularisée par les systèmes utilisés pour le traitement des Big Data comme MapReduce [7], Hadoop [8], ou Spark [9]. Utiliser des systèmes de stockage de données co-localisés avec les applications empêche la gestion fine des ressources de calcul puisque la *commission* (ajout) et la *décommission* (retrait) de nœuds d'un système de stockage de données nécessite le transfert de grandes quantités de données pour garantir la disponibilité de ces données. Ces transferts sont habituellement supposés trop lents pour être utilisés dans le cadre d'une gestion fine de ressources. Cela force les applications basées sur la co-location des tâches et des données à seulement utiliser les ressources qui leur ont été allouées lors de leur lancement. Cette limitation n'est pas confinée au Cloud; les systèmes de stockage de données co-localisés avec les applications sont aussi mis en avant et développés pour les infrastructures de calcul hautes performances (HPC) [10, 11].

Inclure la malléabilité aux systèmes de stockages permettrait l'utilisation de systèmes de stockage co-localisés avec une gestion fine des ressources. La malléabilités est la possibilité pour un système distribué d'avoir des ressources commissionnées et décommissionnées pendant son exécution en suivant les ordres d'un gestionnaire de ressources. Un système de stockage distribué pourrait voir sa taille réajustée selon les besoins de l'application qui y accède. Les systèmes de stockage existants n'ont pas été développés avec la malléabilité comme principe de conception puisque les transferts de données nécessaires pour les *opérations de redimensionnement* (la commission et la décommission) sont supposés lents. Cependant, les technologies réseaux et de stockage de données ont grandement été améliorées récemment(adoption des SSDs et NVRAM, réduction de leurs coût, amélioration de leur vitesse, de leur capacité, etc.). Il faut donc réévaluer ces anciennes hypothèses.

Dans cette thèse, nous cherchons à développer une meilleure compréhension de la malléabilité des systèmes de stockage distribués et à résoudre certains des défis qui lui sont liés.

Contributions

Face au défi de l'étude de la malléabilité des systèmes de stockage, il est tentant d'immédiatement commencer par l'implémentation d'un tel système. Nous avons choisi une approche différente (Figure 1).

Puisqu'il est inutile de commencer l'implémentation de mécanismes de commission et décommission rapide si un modèle mathématique réfute en avance son utilité, notre première étape est la modélisation de la durée minimale des opérations de redimensionnement. Ces

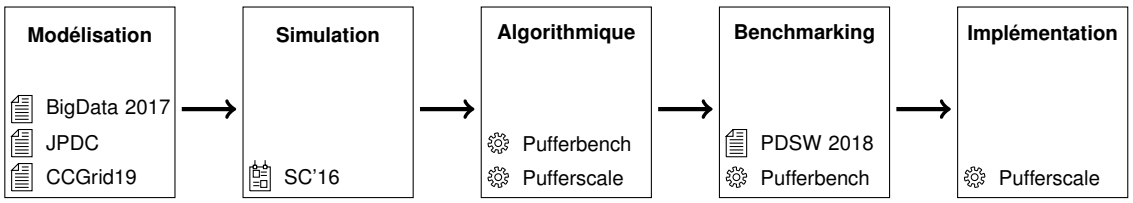


Figure 1: Contributions de cette thèse et publications et logiciels associés.

modèles nous fournissent une référence pour évaluer l'efficacité des opérations de redimensionnement. Ils permettent aussi d'identifier les goulots d'étranglement inhérents à ces opérations.

L'efficacité des transferts de données est primordiale pour avoir des opérations de redimensionnement rapides. La topologie du réseau, notamment, peut avoir un impact important sur les transferts de données. L'apparition de topologies réseau de faible diamètre a donc motivé une nouvelle évaluation des opérations de communication pour les plateformes utilisant ce type de topologie. C'est pourquoi notre seconde étape est l'étude et l'optimisation des opérations collectives de communication pour le supercalculateur Theta en utilisant des outils de simulation.

Les modèles de la durée minimale des opérations de redimensionnement donnent des informations précises sur les goulots d'étranglement qui doivent être considérés pendant ces opérations. Cependant, les modèles ne fournissent ni les algorithmes nécessaires pour déterminer le placement des données après le redimensionnement, ni comment ordonnancer les transferts pendant l'opération. Ces algorithmes sont nécessaires pour l'implémentation de mécanismes rapides de redimensionnement de systèmes de stockage distribués. Notre troisième étape est donc la conception de ces algorithmes.

Les modèles donnent une estimation de la durée minimale des opérations de redimensionnement qui peut être difficile à atteindre puisque les hypothèses (notamment sur le matériel) utilisées pour concevoir ces modèles sont rarement respectées en pratique. Se baser seulement sur les modèles pour décider de l'implémentation de mécanismes de redimensionnement signifie donc prendre le risque que la plateforme elle-même ne soit pas en mesure de supporter ces opérations. Notre quatrième étape est l'implémentation d'un benchmark nommé **Pufferbench** pour mesurer en pratique la durée minimale des opérations de redimensionnement sur une plateforme donnée.

La dernière étape est l'implémentation de **Pufferscale** un gestionnaire de redimensionnement pour des services de gestion de données utilisés en HPC. Pufferscale contient tous les algorithmes nécessaires pour avoir des opérations de redimensionnement rapides.

En fait, nous suivons la méthode scientifique de Karl Popper : nous établissons une hypothèse falsifiable - les opérations de redimensionnement ne sont pas assez rapides pour être avantageuses en pratique - que nous infirmons dans des contextes de complexité croissante, nous conduisant à supposer que le contraire est probablement vrai : la malléabilité est viable, et mérite d'être étudiée plus en détails. Les contributions peuvent être résumées comme suit.

Modélisation de la durée minimale des opérations de redimensionnement

Pour réévaluer sans biais l'hypothèse selon laquelle les opérations de redimensionnement durent trop longtemps pour avoir le moindre intérêt en pratique, l'étude de ces opérations doit être faite indépendamment des systèmes de stockage existants. En effet, ces systèmes n'ont pas été développés en considérant la malléabilité. Dans cette thèse, nous évaluons la viabilité de la malléabilité *en tant que principe de conception* pour un système de stockage distribué. Plus précisement, nous modélisons la durée minimale des commissions et des décommissions. Nous étudions ensuite HDFS, et nous montrons que notre modèle peut être utilisé pour évaluer les performances des algorithmes de redimensionnement. Nos résultats montrent que le mécanisme de décommission de HDFS est efficace (il présente des performances proches du modèle) quand le réseau est le facteur limitant, mais pourrait être trois fois plus rapide quand les composants de stockage sont le facteur limitant. De plus, la commission implémentée dans HDFS peut être grandement accélérée. Grâce à une meilleure compréhension des opérations de redimensionnement, nous proposons des changements pour accélérer ces opérations dans HDFS. Cette contribution à été publiée à la conférence BigData'17 (voir [12]), et un article à été soumis au Journal of Parallel and Distributed Computing (voir [13]).

Étude du compromis entre tolérance aux fautes et vitesse des opérations

Décommissionner le plus vite possible les nœuds inutilisés permet au gestionnaire de ressources de rapidement ré-allouer ces nœuds à d'autres applications. La décommission de nœuds dans un système de stockage distribué nécessite le transfert de larges quantités de données avant de rendre les nœuds à la plateforme pour assurer leur disponibilité et garantir la tolérance aux fautes. Dans cette thèse, nous modélisons et évaluons la performance de la décommission lorsque la tolérance aux fautes est relâchée pendant l'opération. Intuitivement, la quantité de données à transférer avant de rendre les nœuds est réduite, donc les nœuds sont rendus plus rapidement. Nous quantifions théoriquement combien de temps et de ressources sont économisés par cette *décommission rapide* comparée à la décommission standard durant laquelle la tolérance aux fautes n'est pas diminuée. Nous modélisons la durée minimale des différentes phases d'une décommission rapide, et utilisons ces modèles pour estimer quand une décommission rapide est utile pour réduire la consommation de ressources. Avec un prototype de la décommission rapide, nous validons expérimentalement le modèle et confirmons en pratique les résultats obtenus théoriquement. Ce travail a été publié à CCGrid'19 (voir [14]).

Optimisation des communications collectives de MPI pour une topologie Dragonfly

Les contributions précédemment présentées sont basées sur l'hypothèse d'un réseau parfait, connectant tous les nœuds selon un graphe complet, pour les transferts de données. En pratique, les réseaux de plateformes de calcul à large échelle n'ont pas cette topologie idéale. Pour autant, l'efficacité des transferts de données et des communications entre les nœuds de calcul est essentielle pour les performances de nombreuses applications et opérations, y compris les opérations de redimensionnement.

De nouvelles topologies réseau à faible diamètre pour supercalculateurs ont initié de nouvelles études des communications collectives pour ces plateformes. Nous étudions les opérations collectives Scatter et AllGather de MPI pour la topologie Dragonfly du supercalculateur

Theta. Nous proposons un ensemble d'algorithmes pour ces opérations qui exploitent différents aspects tels que la topologie et l'utilisation de communications point-à-point non bloquantes. Nous réalisons une campagne de simulations en utilisant le simulateur de réseaux CODES. Nos résultats montrent que, contrairement à nos attentes, exploiter la topologie du réseau n'améliore pas significativement la vitesse de ces opérations. A la place, de simples algorithmes basés sur des communications non bloquantes ont de meilleures performances grâce au faible diamètre de la topologie et à l'algorithme de routage utilisé. En utilisant ce principe, Scatter peut être 6 fois plus rapide que l'état de l'art, tandis que AllGather est 4 fois plus rapide.

Cette contribution n'est pas détaillée dans ce manuscrit. Un poster présentant ces travaux a été présenté pendant la conférence SC'16 et a été récompensé par le 3$^{\text{ième}}$ prix de la compétition ACM Student Research Competition (voir [15]).

Évaluation du potentiel de malléabilité d'une plateforme avec un benchmark

Évaluer la performance des opérations de redimensionnement sur une plateforme spécifique est un défi en soit : il n'existe actuellement pas d'outils pour cela. Nous introduisons **Pufferbench**, un benchmark conçu pour évaluer la vitesse maximale des opérations de redimensionnement d'un système de stockage distribué sur une plateforme donnée, et, par la même occasion, l'intérêt d'implémenter la malléabilité dans des systèmes de stockage déployés sur ladite plateforme. Pufferbench peut aussi servir à rapidement prototyper et évaluer les mécanismes de redimensionnement de systèmes de stockage distribués existants. Nous validons Pufferbench en le comparant aux durées minimales des opérations de redimensionnement obtenues grâce aux modèles. Nos résultats montrent que les temps mesurés avec notre benchmark sont au maximum 16% plus lents que la durée théorique minimale. Nous utilisons Pufferbench pour évaluer en pratique les opérations de redimensionnement de HDFS : la commission pourrait être 14 fois plus rapide dans certains cas! Nos résultats montrent que (1) les modèles de la durée minimale des opérations de redimensionnement sont cohérents et peuvent être approchés en pratique, (2) HDFS pourrait réaliser ces opérations plus rapidement, et plus important (3) *la malléabilité des systèmes de stockage distribués est viable et devrait être exploitée par les applications Big Data*. Ce travail a conduit à l'implémentation de Pufferbench, et à un article publié dans le workshop PDSW-DISCS'18, tenu conjointement avec la conférence SC'18 (voir [16]).

Ajout de la malléabilité dans des services de données pour le HPC

De nombreuses critiques ont été faites envers l'approche traditionnelle du stockage de données en HPC, basée sur des fichiers : les performances des systèmes de fichiers parallèles sont de plus en plus contraignantes. Une des solutions proposées est l'utilisation de *services de données* lancés par les utilisateurs comme alternative aux systèmes de fichiers traditionnels. Les services de données sont des services habituellement proposés par le système de stockage mais qui sont déployés sur les nœuds utilisés par les applications. En particulier, ces services peuvent être ajustés aux besoins des applications tout en éliminant des surcouts dus à l'organisation en fichiers imposée par les systèmes de fichiers parallèles.

De tels services peuvent avoir besoin d'être redimensionnés pour s'adapter à des charges de travail changeantes afin d'optimiser l'utilisation des ressources. Dans cette thèse, nous formalisons le redimensionnement d'un service de données comme un problème d'optimisation multi-critères : l'équilibrage de charge, l'équilibrage des données, et la durée de l'opération de redimensionnement. Une heuristique pour obtenir rapidement une solution à ce problème est proposée, et permet aux utilisateurs d'ajuster l'importance de chaque critère. Cette heuristique est évaluée grâce à **Pufferscale**, un gestionnaire de redimensionnement développé pour les services de données et systèmes de stockage basés sur des microservices. Pour valider notre approche dans un écosystème réel, nous démontrons l'utilisation de Pufferscale comme un moyen d'ajouter la malléabilité dans HEPnOS, un système de stockage pour des applications en physique des hautes énergies. Ce travail à conduit a l'implémentation de Pufferscale.

Collaborations

Cette thèse a été principalement faite dans le contexte du JLESC (Joint Laboratory for Extreme Scale Computing), un laboratoire commun regroupant l'Inria (France), l'University of Illinois at Urbana-Champaign (UIUC, États-Unis), l'Argonne National Laboratory (ANL, État-Unis), le Barcelona Supercomputing Center (BSC, Espagne), Jülich Supercomputing Centre (JSC, Allemagne), le RIKEN Center for Computational Science (R-CCS, Japon), et l'University of Tennessee Knoxville (UTK, États-Unis).

Publications

Conférences internationales

- **Nathanaël Cheriere**, Gabriel Antoniu. *How Fast can One Scale Down a Distributed File System?*, Proceeding of 2017 IEEE International Conference on Big Data **(BigData 2017)**, *Boston, décembre 2017.* (taux d'acceptation 17.8%).

- **Nathanaël Cheriere**, Matthieu Dorier, Gabriel Antoniu. *Is it Worth Relaxing Fault Tolerance to Speed Up Decommission in Distributed Storage Systems?*, Proceedings of the 19th IEEE/ACM International Symposium on Cluster Computing and the Grid **(CCGrid 19)**. *Larnaca, mai 2019.* Core RANK A (taux d'acceptation 23%).

Articles de journaux soumis

- **Nathanaël Cheriere**, Matthieu Dorier Gabriel Antoniu. *How Fast can One Resize a Distributed File System?*, Journal of Parallel and Distributed Computing **(JPDC)**.

Workshops dans des conférences internationales

- **Nathanaël Cheriere**, Matthieu Dorier, Gabriel Antoniu. *Pufferbench: Evaluating and Optimizing Malleability of Distributed Storage*, Proceedings of 2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems **(PDSW-DISCS 2018)**, tenu conjointement avec International Conference for High Performance Computing, Networking, Storage and Analysis (SC 18), *Dallas, novembre 2018*.

Posters dans des conférences internationales

- **Nathanaël Cheriere**, Matthieu Dorier. *Design and Evaluation of Topology-aware Scatter and AllGather Algorithms for Dragonfly Networks*, International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16), *Salt Lake City, novembre 2016.* **3ième prix de l'ACM Student Research Competition**.

Logiciels

Pufferbench est un benchmark modulaire développé afin de pouvoir rapidement mesurer la durée d'une opération de redimensionnement d'un système de stockage distribué sur une plateforme donnée. Pour accomplir son objectif, Pufferbench émule un système de stockage distribué, et exécute uniquement les entrées et sorties requises par l'opération de redimensionnement sur le réseau et les composants de stockage. Pufferbench a été testé sur Grid'5000 [17] et sur Bebop [18] un supercalculateur d'Argonne National Laboratory. Pufferbench est au centre de la partie III de ce manuscrit.
Lien : `http://gitlab.inria.fr/Puffertools/Pufferbench`
Taille et langage : 6500 lignes, C++
License : MIT

Pufferscale est un gestionnaire de redimensionnement pour systèmes de stockage distribués. Pufferscale inclut une heuristique conçue pour équilibrer le plus rapidement possible la quantité de données et la charge sur chaque nœud. Les rôles de Pufferscale sont de suivre les données hébergées sur chaque nœud, d'ordonnancer les transferts de ces données pendant les opérations de redimensionnement, et de démarrer et d'éteindre les microservices sur les nœuds qui sont respectivement commissionés et décommissionnés. Pufferscale est la contribution majeure du chapitre 13.
Lien : `http://gitlab.inria.fr/Puffertools/Pufferscale`
Taille et langage : 3500 lignes, C++
License : MIT

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Context

In June 2016, Sunway TaihuLight, the first supercomputer to use more than 10 millions cores, took the first place in the Top500 [1], the global ranking of the most powerful supercomputers.[1] Sunway TaihuLight, installed at the National Supercomputing Center in Wuxi (China), reaches 93,014.6 TFlops (93×10^{15} floating point operations per second) using a total of 10,649,600 cores. At the same time, the number of servers used by Amazon Web Services [2] was estimated to 1.3 millions [3]. Such an immense amount of computational power is useful for many fields. In research, it helps with the simulation of complex phenomena, like particle physics, and with the analysis of large amounts of data obtained from experiments and from observations. Intensive computations are also used by companies to provide services useful for everyday life. For example, weather forecasting has been drastically improved since 1950 thanks to the use of supercomputers. In 1950, scientists were successfully predicting the weather 24 hours in advance. Today, accurate weather forecast can be made several weeks in advance thanks to the possibility to compute weather models with greater accuracy [4].

Large-scale computing systems used to run these computationally intensive applications are rarely dedicated to a single application; most of the time, the resources of the platform are shared between multiple applications running simultaneously. In fact, the problem of sharing computing resources is almost as old as computers themselves. Due to the cost of acquiring and operating computing platforms, they have often been shared between applications and between users. Mainframes bought by universities in the 60s and 70s were often shared by allocating time slots to users, or by running applications in the order they were submitted. John McCarthy introduced the concept of *utility computing* in 1961 [5]. He envisioned the possibility to access computing power when needed like other utilities such as water and electricity. A first implementation of this vision was introduced in the 1990s with the creation of *computing grids*: computing resources from institutions were pooled together in order to grant their users access to a larger set of resources. This approach also reduces the cost of maintaining the resources since essential services such as user support are also merged and shared. The Cloud, developed in the 2000s, is one step closer to the vision of McCarthy. Users can rent resources that are managed by companies and are charged for their usage.

The fine-grained resource management available in shared platforms gives the opportunity for users to get new resources as they need them, and to release them as soon as they are not used anymore. This elasticity in resource management offers many advantages to the applications: they can adjust their own resources to be able to efficiently process their workload,

[1] As of June 2019, Sunway TaihuLight is ranked third in the Top500.

even if the latter is unpredictable, while maintaining a low operational cost since only the needed resources are used.

However, this type of fine-grained resource management may sometimes be limited by the need for applications to store and manage massive amounts of data. The recent advent of Big Data led to many applications manipulating immense amounts of data, for example, some datasets used to train a single deep neural network exceed 100 TiB [6]. The manipulation of such a large amount of data makes efficient data management critical for the performance of these applications. In particular, maximizing the co-location of tasks and data is essential: network utilization and latency is reduced by launching tasks on the node (server) that hosts the data they need. This technique has been popularized by the frameworks used to process Big Data, such as MapReduce [7], Hadoop [8], or Spark [9]. Using co-located storage systems hinders fine-grained resource management due the fact that *commissioning* (adding) nodes to and *decommissioning* (removing) nodes from a storage system are operations that require large data transfers to guarantee that no data is lost. These data transfers have been traditionally assumed to be too slow for a practical use. This forces applications relying on the co-location of tasks and data to only use the amount of resources allocated to them when they were launched. The limitation of fine-grained resource management due to data storage is not confined to the Cloud; storage systems co-located with the applications are advocated and developed for high performance computing (HPC) infrastructures as well [10, 11].

Malleable storage systems would enable the use of co-located storage systems with a fine-grained resource management. Malleability is the ability for a distributed system to have resources commissioned and decommissioned during its execution upon request from a resource manager. A malleable distributed storage system could be rescaled following the needs of the application co-located with it. State-of-the-art distributed storage systems have not been designed with malleability in mind due to the assumption of long data transfers needed by *rescaling operations* (commissions and decommissions). However, both the network and storage technologies and the approaches to data storage have greatly changed recently. Old assumptions need to be revisited.

In this thesis, we aim to develop a better understanding of the malleability of distributed storage systems and to address some challenges associated with it.

1.2 Contributions

Faced with the challenge of studying storage system malleability, one would be tempted to jump right into implementing such a system. We took a different approach (Figure 1.1).

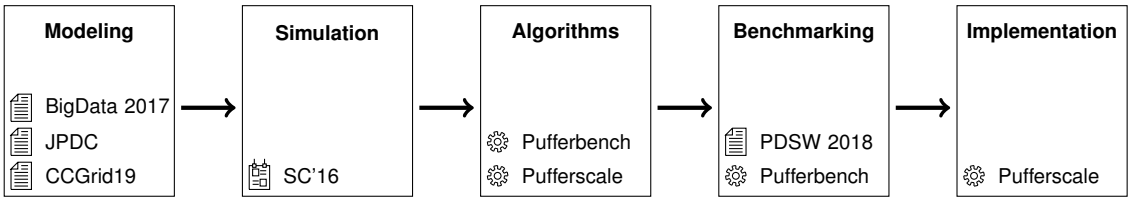


Figure 1.1: Contributions of this thesis and their associated publications and software.

Since it is pointless to start the implementation of fast rescaling mechanisms if a mathematical model disproves in advance its usefulness, our first step was to establish a model of the minimal duration of rescaling operations. This gave us a reference to evaluate the efficiency of rescaling operations. It also highlighted the inherent bottlenecks of rescaling operations.

Efficient data transfers are required for fast rescaling operations. The topology of the network can have an important impact on the performance of the communications. The emergence of new low-diameter topologies motivated the reevaluation of communication algorithms for the platforms using this type of topology. Thus, our second step was the study and optimization of collective communications on the Theta supercomputer using discrete-event simulation network simulations.

The models of the minimal duration of rescaling operations gave precise information about the bottlenecks that need to be mitigated during the operations. However, modeling the operations did not provide the required algorithms determining the placement of data objects after the rescaling, and how to schedule the data transfers during the operation. These algorithms are needed for the implementation of fast rescaling mechanisms in distributed storage systems. Thus, the third step was the design of these algorithms.

Models give an estimation of the minimal duration of rescaling operations that may be hard to reach since the assumptions about the hardware used to produce them are rarely met in practice. Thus, relying only on the models to decide whether to implement rescaling operations in a distributed storage system is taking the risk that the platform itself may not be able to support efficient rescaling operations. This is why our fourth step was the implementation of a benchmark named **Pufferbench** to measure in practice the minimal duration of rescaling operations on a given platform.

The last step was the implementation of **Pufferscale**, a rescaling manager for HPC data services implementing all algorithms required for fast rescaling operations.

In fact, we follow Karl Popper's scientific method: we establish a falsifiable hypothesis — namely, that commissions and decommissions are not fast enough for malleability to present any advantage in practice —, that we disprove in contexts of increasing complexity, leading us to assume that its opposite must ultimately be true: malleability is viable, and worthy of further investigation. These contributions can be summarized as follows.

Modeling the minimal duration of a rescaling operation

To fairly revisit the assumption that rescaling operations last too long to present any real-time benefit, their study has to be done as independently as possible from existing storage systems. Indeed, these systems were not designed with malleability as a base principle. In this thesis, we evaluate the viability of malleability *as a design principle* for a distributed storage system. We specifically model the minimal duration of the commissions and decommissions. We then consider HDFS as a use case, and we show that our mathematical model can be used to evaluate the performance of the commission and decommission algorithms. We show that the

existing decommission mechanism of HDFS is good (in the sense that it is close to our model) when the network is the bottleneck, but it can be accelerated by up to a factor 3 when storage is the limiting factor. We also show that the commission in HDFS can be greatly accelerated. With the highlights provided by our model, we suggest improvements to speed up both operations in HDFS. This work led to a publication at the BigData'17 conference (see [12]), and to a paper submitted to the Journal of Parallel and Distributed Computing (see [13]).

Studying the trade-off between fault tolerance and rescaling duration

Decommissioning the idle nodes as soon as possible allows the resource manager to quickly reallocate those nodes to other jobs. The decommission of nodes in a distributed storage system requires transferring large amounts of data before releasing the nodes in order to ensure data availability and a certain level of fault tolerance. In this thesis, we model and evaluate the performance of the decommission when relaxing the level of fault tolerance (i.e., the number of replicas) during this operation. Intuitively, this is expected to reduce the amount of data transfers needed before nodes are released, thus allowing nodes to be returned to the resource manager faster. We quantify theoretically how much time and resources are saved by such a *fast decommission* strategy compared with a standard decommission that does not temporarily reduce the fault tolerance level. We establish mathematical models of the minimal duration of the different phases of a fast decommission. These models are used to estimate when fast decommission would be useful to reduce the usage of node-hours. We implement a prototype for fast decommission and experimentally validate the model and confirm in practice our theoretical findings. This work was published at CCGrid'19 (see [14]).

Optimizing MPI collective operations for a Dragonfly topology

The contributions previously introduced have assumed a perfect, all-to-all network topology to handle data transfers. In practice, networks for large computing platforms do not have such an ideal topology. Nonetheless, the efficiency of data transfers and communications between compute nodes is essential for the performance of many applications and operations, including rescaling operations.

Recent low diameter network topologies for supercomputers have motivated a redesign of collective communication algorithms. We study the Scatter and AllGather collectives of MPI for the Theta supercomputer's dragonfly topology. We propose a set of algorithms for these operations that leverage different factors such as the topology or the use of non-blocking point-to-point communications. We conduct an extensive simulation campaign using the CODES network simulator. Our results show that, contrary to our expectations, topology awareness does not significantly improve the speed of these operations. Instead, we note that simple algorithms based on non-blocking communications perform well thanks to the low diameter topology coupled with the routing algorithm. Using this principle, Scatter can be up to 6x faster than state-of-the-art algorithms, while there is a 4x improvement for AllGather.

This contribution is not detailed in this manuscript. A poster about this work was presented during SC'16 and was awarded the 3rd prize of the ACM Student Research Competition (see [15]).

Benchmarking the viability of malleable storage on a platform: Pufferbench

Evaluating the performance of rescaling operations on a given platform is a challenge in itself: no tool currently exists for this purpose. We introduce **Pufferbench**, a benchmark for evaluating how fast one can scale up and down a distributed storage system on a given infrastructure and, thereby, how viable it is to implement storage malleability on it. Pufferbench can also serve to quickly prototype and evaluate mechanisms for malleability in existing distributed storage systems. We validate Pufferbench against the theoretical minimal duration of commissions and decommissions. We show that it can achieve performance within 16% of these theoretical minimal duration. Pufferbench is used to evaluate in practice these operations in HDFS: commission in HDFS could be accelerated by as much as 14 times! Our results show that: (1) the models of the minimal duration of the commissions and decommissions we previously established are sound and can be approached in practice; (2) HDFS could handle these operations much more efficiently; and most importantly, (3) *malleability in distributed storage systems is viable and should be further leveraged for Big Data applications*. The outcomes of this work include the implementation of Pufferbench itself and a paper published at the PDSW-DISCS'18 workshop, held in conjunction with the SC'18 conference (see [16]).

Adding malleability to HPC data services: Pufferscale

Numerous criticisms have been raised about the standard approach to storage in HPC infrastructures: the performance of parallel file systems have been increasingly problematic over the past decade. One of the proposed solutions is the use of *user-space HPC data services* as an alternative to traditional parallel file systems. Such services traditionally managed by the storage system are deployed on the nodes used by the application instead. In particular, they can be tailored to applications needs while eliminating unnecessary overheads incurred by the file-based data organization imposed by parallel file systems.

Such services may need to be rescaled up and down to adapt to changing workloads, in order to optimize resource usage. In this thesis, we formalize the problem of rescaling a distributed storage system as a multi objective optimization problem considering three criteria: load balance, data balance, and duration. We propose a heuristic for rapidly finding a good approximate solution, while allowing users to weight the criteria as needed. The heuristic is evaluated with **Pufferscale**, a new, generic rescaling manager for microservice-based distributed storage systems. To validate our approach in a real-world ecosystem, we showcase the use of Pufferscale as a means to enable storage malleability in the HEPnOS storage system for high energy physics applications. This work led to the implementation of Pufferscale.

Collaborations

This work was mainly carried out in the context of the associate team Data@Exascale and of the JLESC (Joint Laboratory for Extreme Scale Computing), a joint laboratory between Inria (France), University of Illinois at Urbana-Champaign (UIUC, USA), Argonne National Laboratory (ANL, USA), Barcelona Supercomputing Center (BSC, Spain), Jülich Supercomputing Centre (JSC, Germany), RIKEN Center for Computational Science (R-CCS, Japan), and University of Tennessee Knoxville (UTK, USA).

1.3 Publications

International conferences

- **Nathanaël Cheriere**, Gabriel Antoniu. *How Fast can One Scale Down a Distributed File System?*, in Proceeding of 2017 IEEE International Conference on Big Data **(BigData 2017)**, *Boston, December 2017.* (Acceptance rate 17.8%).

- **Nathanaël Cheriere**, Matthieu Dorier, Gabriel Antoniu. *Is it Worth Relaxing Fault Tolerance to Speed Up Decommission in Distributed Storage Systems?*, in Proceedings of the 19th IEEE/ACM International Symposium on Cluster Computing and the Grid **(CCGrid 19)**. *Larnaca, May 2019.* Core RANK A (acceptance rate 23%).

Submitted journal papers

- **Nathanaël Cheriere**, Matthieu Dorier Gabriel Antoniu. *How Fast can One Resize a Distributed File System?*, in Journal of Parallel and Distributed Computing **(JPDC)**.

Workshops at international conferences

- **Nathanaël Cheriere**, Matthieu Dorier, Gabriel Antoniu. *Pufferbench: Evaluating and Optimizing Malleability of Distributed Storage*, in Proceedings of 2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems **(PDSW-DISCS 2018)**, held in conjunction with the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 18), *Dallas, November 2018*.

Posters at international conferences

- **Nathanaël Cheriere**, Matthieu Dorier. *Design and Evaluation of Topology-aware Scatter and AllGather Algorithms for Dragonfly Networks*, International Conference for High Performance Computing, Networking, Storage and Analysis (SC 16), *Salt Lake City, November 2016.* **3rd prize at the ACM Student Research Competition**.

1.4 Software

Pufferbench is a modular benchmark developed to measure the duration of the rescaling operations of a distributed storage system on a given platform. To this end, Pufferbench emulates a distributed storage system, executing only the inputs and outputs on the storage devices and the network that are needed for a rescaling operation. Pufferbench was tested on Grid'5000 [17] and on Bebop [18], a cluster from Argonne National Laboratory. Pufferbench is at the core of Part III of this manuscript.
Link: `http://gitlab.inria.fr/Puffertools/Pufferbench`
Size and language: 6500 lines, C++
License: MIT

Pufferscale is a generic rescaling manager for distributed storage systems. Pufferscale implements a heuristic designed to balance the amount of data and the load on each node as fast as possible. The roles of Pufferscale are to track the data hosted on each node, schedule the data migrations using the previous heuristic, and start and stop microservices on compute nodes that are being respectively commissioned and decommissioned. Pufferscale is at the core of Chapter 13.
Link: `http://gitlab.inria.fr/Puffertools/Pufferscale`
Size and language: 3500 lines, C++
License: MIT

1.5 Organization of the manuscript

The rest of this manuscript is organized in four parts.

In the first part, we present the context of our research. In Chapter 2, we introduce the concept of malleability and its benefits for applications. We then focus on the problem of storing data for malleable applications in Chapter 3, where we motivate the integration of malleability in distributed storage systems, and detail the main challenges to address.

The second part is centered on modeling the minimal duration of rescaling operations. After detailing the assumptions used to build the models in Chapter 4, a model for the shortest duration of the commission is established in Chapter 5. The model of the duration of the fastest possible decommission is presented in Chapter 6. In Chapter 7 we use the models to evaluate the rescaling operations implemented in HDFS, a popular distributed storage system. In Chapter 8, we study in detail the trade-off between relaxing the fault tolerance during decommissions and quickly releasing the decommissioned nodes.

The third part focuses on measuring the potential rescaling duration on a given platform. To this end, we present Pufferbench in Chapter 9. We validate Pufferbench against the previously established models in Chapter 10, and use it in Chapter 11 to estimate the potential speed of the rescaling operations of HDFS.

The fourth part focuses on adding malleability to user-space HPC data services. In Chapter 12, we detail the problem of rescaling a data service while ensuring load balance and formalize it as a multiobjective optimization problem. In Chapter 13, we propose a heuristic to quickly provide an approximate solution, implement it in Pufferscale, and evaluate it.

Chapter 14 concludes this manuscript by summarizing our contributions and presenting open problems.

PART I

Distributed Storage and Malleability

CHAPTER 2

JOB MALLEABILITY

Large-scale computing platforms are often shared by multiple users due to the overall high cost of running and managing such platforms. However, sharing resources is a challenge in itself. Platform managers want user satisfaction and high resource utilization. Users want their applications to start as soon as possible, and to have access to the amount of resources they requested. These objectives are orthogonal and may not all be satisfied at the same time.

One approach studied to improve resource management on shared platforms is job malleability. Malleable jobs can have some of their resources commissioned (added) or decommissioned (removed) during their execution. Job malleability can be leveraged by resource managers to improve response times and increase the resource utilization of the platform. The same mechanisms can also be used by the job itself to adapt its resources to its needs. Moreover, the commission and decommission[1] of resources to malleable jobs is done with minimal interruption in order to efficiently use available resources.

In this chapter, we define the malleability of a system and detail the benefits for users and platform owners of using malleable applications. We then introduce the commission and the decommission, the simplest operations required for a job management system to provide malleability. We conclude by presenting existing works on the malleability of distributed systems and applications.

2.1 The concept of malleability

A job is said to be *malleable* if the resource manager can give the job more resources to use during its execution, and if it can order it to decommission any set of resources the job is using. In this context, the notion of job is wide; it can be a single distributed application, or a workflow of multiple malleable applications or applications that can run in parallel. The scheduling of these types of jobs has been extensively studied by the scheduling community [19, 20, 21], which also coined the term *malleability*.

Classification

Jobs can be classified according to their resource requirements. Feitelson and Rudolph [22] proposed the classification presented in table 2.1.[2]

[1] We denote as commission the operation during which resources are added to a system. Similarly, the decommission is the operation during which resources are removed from a system.

[2] This classification is not always followed; moldable jobs are sometimes called malleable jobs in works on theoretical scheduling.

	When is it decided?	
Who decides?	At Job Submission	During Job Execution
User or job	Rigid	Evolving
Resource manger	Moldable	Malleable

Table 2.1: Classification of jobs according to the dynamicity of their resource requirements.

Rigid jobs require a fixed number of resources to run. The size of the resource allocation (set of resources used by the job to run) is specified by the user during the job submission. This job cannot be executed on fewer resources, and having additional resources does not improve its performances.

Moldable jobs are flexible in the number of resources they need. The resource manager chooses before launching the job how many resources will be allocated to run the job. The moldable job behaves like a rigid job once launched.

Evolving jobs have the ability to change their resource requirements during their execution. It is important to note that the job *itself* initiates the change. The resource manager must then satisfy the resource requirements for the job to continue its execution.

Malleable jobs are the most flexible jobs, they can adapt to changes in resources *initiated by the resource manager.* The resource manager can request to have any resource released and a malleable job can adapt itself without terminating. Moreover, it should be able to take advantage of any extra resources commissioned during its execution.

Note that a job can be at the same time evolving and malleable; it can adapt itself to external changes in resources as well as request changes by itself.

Resources

Feitelson and Rudolph [22] only considered one type of resources in their work: processors. However, the notion of resources can easily be extended to other resources, such as nodes, cores, storage, and network bandwidth.

Elasticity, scalability, and malleability

Scalability and elasticity of distributed systems are properties similar to the malleability, but differ on some aspects.

According to Bondi [23],

> "The concept [of scalability] connotes the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement."

The study of scalability is about how a system performs at large-scale. However, scalability does not imply malleability: a rigid job can be scalable. In fact, most of the scalability stud-

ies are done by starting the studied system with increasingly large resource allocations. The studies on malleability aim to improve rescaling operations themselves, and require jobs to be scalable.

According to Herbst *et al.* [24],

> "Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible."

This means malleability constitutes a prerequisite for elasticity: malleability gives the possibility for a job to be rescaled, but elasticity also requires a decision system to chose *when* to rescale in order to maximize performance under the current workload.

Focus on the notion of malleability

Scalability has been the focus of decades of research and is now well understood. Malleability however, is partially understood to date, and research mainly focuses on the management of a single type of resources: processors. Malleability needs to be mastered in order to efficiently implement and use truly elastic systems.

2.2 Benefits of malleable and evolving jobs

Malleable and evolving jobs have been the focus of many works in recent years due to the popularization of shared platforms for distributed computing. These types of jobs have majors advantages for the platforms and the users.

2.2.1 Advantages of malleable jobs

Managing malleable jobs has two clear benefits for the platforms: lowering the response times, and increasing the resource utilization.

Response time

Multiple works [25, 26, 27] have shown that job malleability can be leveraged to decrease response time (i.e. the time elapsed between the submission of a job and its completion). Malleable jobs can start on few resources if needed, and can also be downsized to free some resources required for other jobs.

For example, Hungershofer [26] used simulation and traces collected on various large scale platforms to show that if only 25% of the jobs running on the platform are moldable or malleable, the response time could be halved.

Another example is the work of Prabhakaran *et al.* [27]. They proposed an extension of the Torque/Maui batch system to leverage malleability. When all jobs running on the platform are malleable, they observed a throughput increased by 32% compared to rigid job allocations.

Moreover, they showed a 20% speed up of job execution when at least 40% of the jobs running are malleable. Resource utilization on the platform was also increased by 12.5%.

Resource utilization

Malleable jobs can also be used to improve the resource utilization on a given platform. Indeed, job malleability gives the possibility for the resource manager to use otherwise idle resources to improve the performances of already running jobs, or to start jobs with few resources and later increase the allocated resources as other resources become available.

For example, Mercier *et al.* [28] exploited the inherent malleability of MapReduce jobs to run them on unused resources of supercomputers. The approach consists of scheduling tasks from MapReduce jobs on resources from the HPC cluster left idle due to scheduling gaps. This approach maximizes the resource utilization of the HPC platform: resource utilization reached 100%. However, it leads to a mean waiting time for HPC jobs increased by 17%, and MapReduce jobs have an average efficiency (the proportion of time spent in tasks that complete over the total time spent running tasks) of 68% due to tasks being killed for their resources as priority is given to HPC jobs.

2.2.2 Advantages of evolving jobs

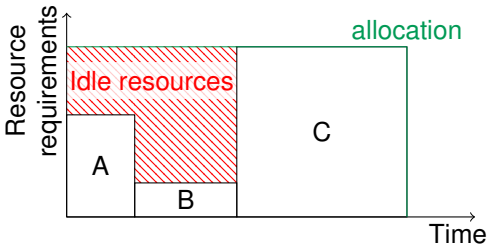


Figure 2.1: Execution of the workflow as a rigid job.

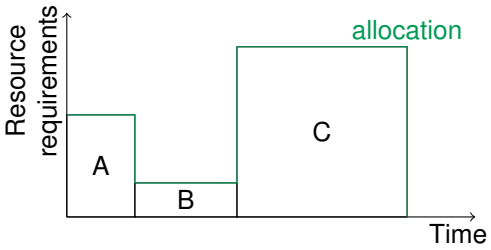


Figure 2.2: Execution of a workflow as an evolving job.

There are also clear benefits for evolving that rely on rescaling mechanisms similar to those of malleable jobs.

Adaptation to the workload

Evolving jobs can adapt their own resource usage to their workload, ensuring a constant right provisioning, even in the case of a varying workload. If the workload increases, an evolving job can request more resources from the resource manager and similarly return unneeded resources when the workload decreases.

Auto-scaling techniques are developed in order to match workload and resources [29, 30].

Roy *et al.* [29] proposed an algorithm able to change the number of nodes allocated to an application based on the prediction of future resource requirements. They aim to satisfy the application's quality of service and to minimize the cost of the resources used.

Niu *et al.* [30] used auto-scaling for video-on-demand applications to reduce bandwidth usage while ensuring quality of service. For their use case, the relevant resource that is adjusted is the bandwidth of data centers.

Cost minimization

Moreover, evolving jobs are able to minimize their cost (financial and energetic) by reducing the amount of resources allocated to them. Indeed, many jobs, especially workflows, have multiple successive phases with different resource requirements. If such a workflow is considered as a rigid job (Figure 2.1) by the resource manager or the user, enough resources must be allocated to satisfy the requirements of the phase (C) requiring the most resources, leaving many resources idle during the first two phases (A and B). However, if it is built as an evolving job, the amount of resources allocated to the workflow can closely follow its needs.

Mao and Humphrey [31] proposed an auto-scaling mechanism designed to minimize the execution cost of workflow in the cloud. The mechanism rely on the heterogeneity of the nodes users can rent in the cloud (with various capabilities and cost), and on the fact that workflows are evolving jobs. By carefully choosing which nodes to use for each task in the workflow, they exhibit cost savings ranging from 9.8% to 40.4% compared with other state-of-the-art methods.

Dougherty *et al.* [32] proposed a model-driven approach to resource provisioning to precisely match the resource allocation of an application to its workload. This removes resources that would otherwise be underutilized and waste power. They showed a reduction of the energy consumption and cost by 50% compared with a rigid resource allocation.

2.3 Basic rescaling operations: commission and decommission

The commission (adding resources) and decommission (removing resources) are the two basic operations a job needs to be malleable. We denote these operations as rescaling operations. In this work, we do not consider reconfiguration operations during which some resources are commissioned while other are decommissioned. Most results can easily be expended to this specific case.

Commission During a commission, new resources are added to the resource allocation of a job. New resources may have to be prepared in order to be used by the job. For example, they may have to start some services, and contact a master node to inform it that new resources are ready to run some tasks.

Decommission: During a decommission, part of the allocated resources are returned to the resource manager, reducing the ones used by the job. The main challenge in designing an efficient decommission mechanism is the minimization of work lost due to resources leaving. For instance, the results computed by tasks on a decommissioned resource may have to be transferred to resources that will remain after the operation.

Benefits of fast rescaling operations: Ensuring a fast decommission presents benefits for both the user and the platform. Because physical resources are returned to the platform during a decommission, finishing the operation quickly reduces the core hour usage of the jobs, which in turn reduces the cost of running jobs in pay-as-you-go pricing models, and reduces the energy consumption. Moreover, if the decommissioned resources are awaited by another job, the sooner they are available, the sooner the other job can benefit from these resources, reducing the platform's response time.

Similarly, a fast commission enables newly added resources to be used quickly after being provisioned. It helps to react faster to changes in the workload since new resources are available soon after the workload increases. In the case of a reconfiguration, finishing it quickly also reduces the core-hour usage and the energy consumed by the job.

2.4 Previous work on job malleability

Multiple works have been done in order to add malleability to shared platforms. They can be organized in two categories: resource managers, and malleable applications.

2.4.1 Leveraging job malleability

Job malleability can be leveraged by the resource managers of the platforms to improve their own objectives.

KOALA-F [33] is designed to adjust the resources allocated between various computing frameworks such as Hadoop [8], Fluent [34], and Spark [9] depending on their workload. In this case, the malleability of the frameworks is leveraged. Each framework running on the platform managed by KOALA-F reports regularly how it performs: nominally with extra resources, nominally without extra resources, overloaded. Based on this feedback, KOALA-F adjusts the node allocation of each framework running on the platform in order for all frameworks to perform nominally without extra resources. This guarantees that tasks running on the frameworks can progress without hindrances while keeping a pool of available resources to quickly allocate when needed.

Morpheus [35] is a resource manager that focuses on automatically improving service level objectives such as completion time and deadlines, using various means, including job malleability. It monitors the progress of malleable jobs and increases or decreases the resource allocation to compensate execution variability. Experiment have shown a reduction of deadline violation by 5 to 13 times while maintaining similar level of resource utilization.

2.4.2 Enabling job malleability

Implementing a malleable job is not a straightforward task since the application needs to be able to cope with unexpected rescaling operations, while minimizing their impact on its performance.

Applications built for some frameworks such as MapReduce [7] or Hadoop [8] are inherently malleable. The frameworks themselves are malleable (commissions and decommissions have been implemented for maintenance purposes). MapReduce jobs are also malleable. The map and reduce phases are composed of well-defined tasks that each run on one node, so when

Processors	Shrink time (ms)	Expand time (ms)
128 to 64	614	502
64 to 32	660	538
32 to 16	696	506
16 to 8	594	461
8 to 4	564	489

Table 2.2: Duration of the rescaling operations.[40]

new nodes are commissioned, tasks can be started on them. Similarly, when resources are decommissioned, one can either wait for tasks running on these resources to finish, or kill and restart them on other nodes. This is leveraged by KOALA-F [33] and Bebida [28] for example.

Likewise, many workflows are malleable by nature. If a workflow has to run multiple tasks in parallel, it can cope with multiple allocation sizes. In case of decommissions, one can wait for the tasks running on the decommissioned nodes to finish, or terminate and restart them on other resources, without interrupting the execution of the overall workflow. The evolving character of workflows has been taken into account by most workflow engines [36, 37, 38]: they do not rely on rigid resource allocation to run the whole workflow as one job.

Other works, such as the ones of Buisson *et al.* [39], Kale *et al.* [40], and Vadhiyar *et al.* [41] have proposed frameworks and tools to easily implement malleable applications.

The SRS Checkpointing library [41] proposes a method to implement malleable applications based on checkpointing. To resize an application, a checkpoint of it is realized, the application is stopped and then restarted on the resized allocation. The library handles the required data redistribution automatically.

Kale *et al.* [40] introduce a framework conceived to help programmers design malleable applications. The framework manages the migration of threads (including message forwarding) as well as the load balancing on the nodes allocated to the job. They evaluated the performance of rescaling operations on the NCSA Platinum Cluster (a Linux cluster with 512 dual-processor 1 Ghz Pentium III nodes connected by Myrinet) by rescaling an application using 10 MB of data per processor. The duration of the migration (Table 2.2) is around 600 milliseconds for the decommissions and 500 milliseconds for the commission, highlighting the possible fast rescaling of computing applications.

Conclusion

In this chapter, we presented the concept of malleability, and its advantages for both platforms and applications. We detailed the two basic rescaling operations for malleability: the commission, and the decommission. We advocated for the importance of completing both rescaling operations quickly. Lastly, we introduced and discussed previous efforts that enable or leverage malleability for computing resources.

CHAPTER 3

TOWARDS MALLEABLE DISTRIBUTED STORAGE

Most applications rely on persistent data storage for their execution. They may rely directly on data storage to read input data or to save their output, or to ensure other properties such as fault tolerance. Overall, the efficiency of storage systems is critical for the performance of many applications.

Distributed applications have storage requirements that are amplified by their size compared to single node applications. Specialized storage systems have been designed to accommodate distributed applications. However, existing distributed storage systems are not able to fully support malleable and evolving applications, and limit their malleability.

In this chapter, we present the challenge of storing data for large-scale applications and highlight the importance of an adapted storage system to ensure good application performance. We motivate the use of malleable co-located storage systems (distributed storage systems deployed on the nodes used to run the application that needs the stored data) in the context of cloud computing. We then show the relevance of malleability for specialized data services in the field of high performance computing (HPC). We conclude by presenting the challenges of adding malleability to distributed storage systems, and detail the ones studied in the remainder of this work.

3.1 Storing data for large-scale applications

Storing data for large scale applications is a complex task that requires to address a number of challenges and to make a number of decisions. The task is critical since the performance of the storage system directly impacts the performance of applications relying on it.

3.1.1 A necessarily distributed storage

The two primary challenges when storing data for distributed applications are the usually high amount of data to store and the bandwidth requirements to satisfy. Distributed applications usually read and write data. This data may consist of the input data, the results of the computation, or temporary data. Moreover, some mechanisms also rely on a persistent data storage: when ensuring fault tolerance with checkpointing, the state of the application is periodically written to persistent storage in case the application needs to be restarted. Distributed applications can access very large amounts of data during their run time. For example, the CyberShake workflow [42] that runs on clusters of machines has been studied by Juve *et al.* [43]. Traces obtained during the execution of the workflow on 36 nodes have shown that 198 TiB of data

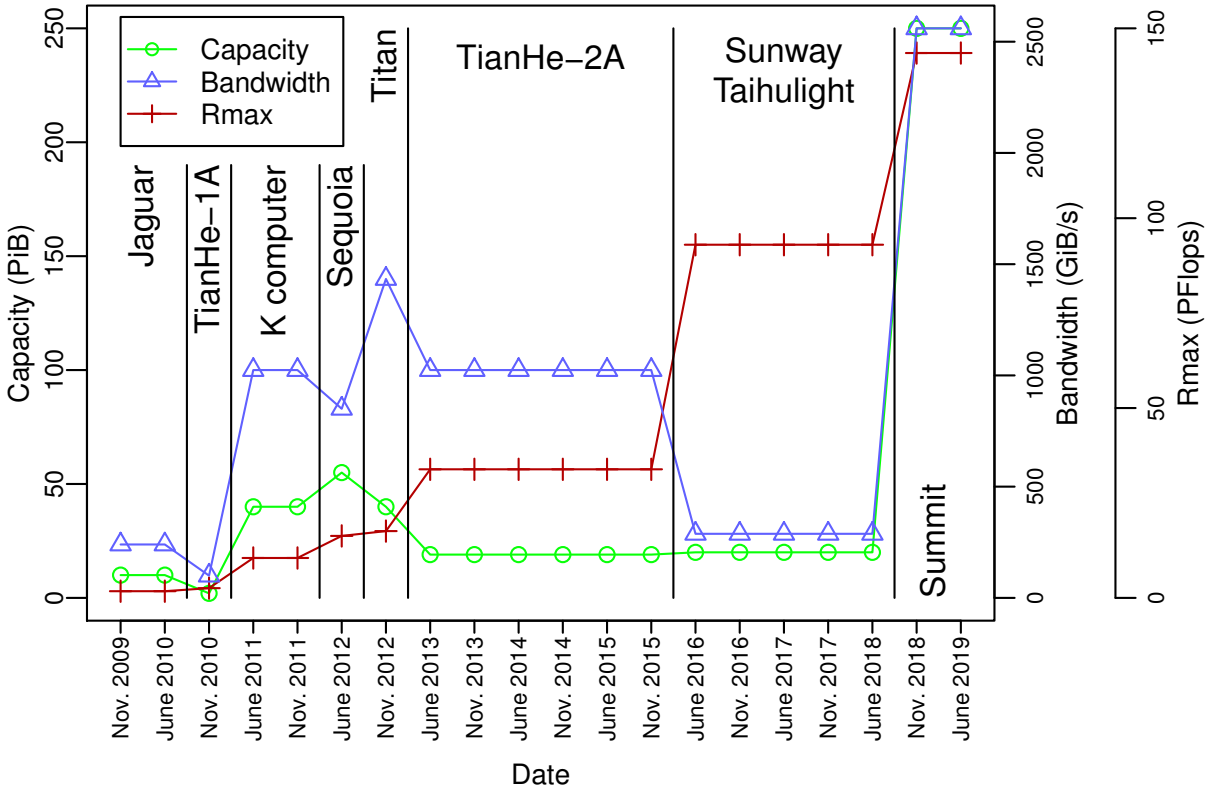


Figure 3.1: Capacity and bandwidth provided by the storage used by the supercomputer ranked first in the Top500 from November 2009 to June 2019. The number of floating point operations per second achieved by the machine on the Linpack benchmark used to rank the supercomputers. Sources [1, 47, 48, 49, 50, 51, 52, 53, 54]

was read as input, and 858 GiB of output data was written over 12 hours. Besides an obvious need for a storage with large capacity, the storage system needs to provide an efficient access to its data. Some applications do not require much storage space, but will access their data frequently. Snyder *et al.* [44] studied the HMMER application [45]. This single node application reads 4 TiB of data out of a single 253.4 MiB file in 710 seconds, which corresponds to an average read throughput of 5.8 GiB/s. From this observation, they proposed to put the file in cache on the compute node running HMMER and reduced the I/O time to 390.8 seconds.

These requirements are taken into consideration when dimensioning the storage systems of computing platforms. Summit [46] is the most powerful supercomputer in the world as of November 2018 according to the TOP500 ranking [1]. It is composed of 4,608 nodes each with 2 CPUs (2×22 cores) and 6 GPUs. To satisfy the storage demand of its applications, Summit provides a storage capacity of 250 PiB and a storage bandwidth of 2.5 TiB/s [47].

Such requirements cannot be achieved by a single machine. The storage system of Summit is deployed on 77 nodes each with 1 TiB of memory, 104 disks and 2 SSDs for 4 PiB of raw storage [55]. Over the past decade, the capacity of the storage provided to the best supercomputer in the world has been multiplied by 25, while the bandwidth provided has been increased by a factor 10 (Figure 3.1).

Huge storage requirements also exist outside of the domain of high performance computing.

Large companies, such as Twitter and Facebook, store and process large amounts of data. Twitter revealed in 2017 that its HDFS [56] cluster hosted more than 500 PiB of data and only represented 40.8% of all the data stored [57]. Moreover, Twitter provides an aggregated bandwidth of 120 GiB/s to its users [57].

Last but not least, the advent of deep learning in all fields of computer science is in part due to the ability to process very large datasets used to train deep neural networks. Chilimbi *et al.* [58] and Hazelwood *et al.* [6] report dataset sizes of 100s TiB and only limited by the capacity of the clusters.

> *Applications and systems from the Cloud and high performance computing fields require an efficient storage that provides high capacity and high bandwidth. Such storage systems are distributed and use many nodes to satisfy these requirements.*

3.1.2 Types of deployments

Many distributed storage systems have been proposed to efficiently support distributed applications. They can be categorized in two large families according to their deployment with respect to the computing cluster that executes applications. The storage can either be *separated* or *co-located* with respect to the computing resources.

Separated storage

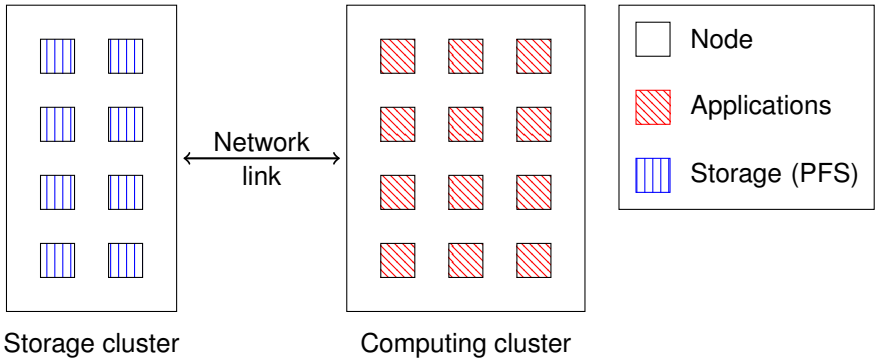


Figure 3.2: Separated storage system deployed alongside a computing cluster.

A separated storage is deployed on a cluster distinct from the computing cluster (Figure 3.2). Data can only be accessed through the network since it is remote from the applications, and the storage is shared between all applications running on the computing nodes.

Such a deployment has clear advantages. The storage system can run on nodes designed for data storage with specialized hardware since applications run on another cluster. This also gives the opportunity for the computing cluster to be specialized for computation: until recently, compute nodes of HPC platforms did not have any form of persistent storage.

Separated storage is the usual storage approach for high performance computing. Many systems have been proposed such as Ceph [59] based on RADOS [60], Lustre [61], PVFS [62],

GPFS [63], BeeGFS [64], zFS [65], OrangeFS [66], and MooseFS [67]. They are also used in cloud environments with systems such as Amazon S3 [68] and Microsoft Azure [69, 70, 71]. All these systems have one property in common, they are *parallel file systems (PFS)*[1]: each data object stored is cut in stripes of fixed size and distributed among multiple storage nodes to maximize access throughput.

Co-located storage

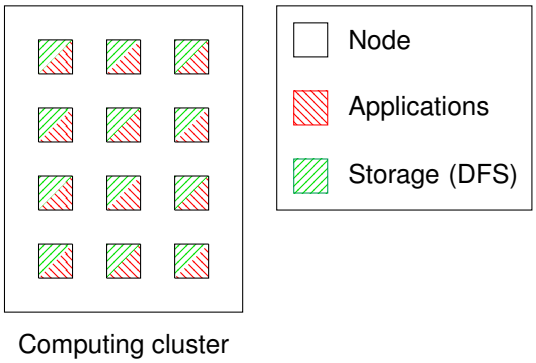


Figure 3.3: Co-located storage and computation deployed on a cluster.

The other type of deployment is the co-located storage. The storage system runs on the nodes used by the application (Figure 3.3). This deployment has been popularized with the advent of Big Data and its processing techniques. Cloud has emerged from Grid computing, which used commodity hardware to form a pool of computing resources. In such a setting, storage was traditionally an aggregation of each machine's disks, which motivated the need for a distributed storage system aggregating such storage capacity and, therefore, collocated with applications. Eventually, Clouds offered the same model because of the benefits it provides in terms of access locality.

Co-locating storage and computation has one major advantage: it enables data locality, that is, running tasks on the nodes that already host the data they require. Many Big Data frameworks such as Hadoop [8], Spark [9], and Flink [74] leverage this property. Enforcing data locality when scheduling tasks has two major benefits. First, it reduces network usage since the data does not have to be transferred from its storage node to the task. Second, it enables local accesses to the data: local data accesses have lower latency and possibly better throughput than remote data accesses. Besides data locality, another strong characteristic of co-located storage systems is their scalability: adding nodes to a cluster with co-located storage and computation increases not only the computation capabilities but also the storage capacity.

Multiple distributed storage systems have been designed to be co-located with computation, such as HDFS [56], GFS [72], Tachyon [75], and RAMCloud [73].

[1]This is why separated storage systems are commonly referred to as PFS, even if co-located storage systems such as HDFS [56], GFS [72], and RAMCloud [73] are also parallel file systems.

3.1.3 Efficient storage for application performance

The choice of the storage system to use can have a significant impact on applications. Tantisiriroj *et al.* [76] showed a difference in the run time of the distributed Hadoop grep application depending on the used storage system: it was 10 times faster when Hadoop relied on HDFS for storage than when PVFS was used. They also highlighted numerous modifications to bring to PVFS in order for it to match the performances of HDFS. Similarly, Zaharia *et al.* [77] demonstrated a 20 times speedup on Map Reduce iterative jobs with modifications of the storage system, in particular, data objects can be kept in memory when needed.

Such improvements are possible since many applications rely heavily on the storage system, and data accesses take a major part of their run time. Luu *et al.* [78] analyzed I/O logs from three supercomputers: Intrepid [79], Mira [80], and Edison [81], collected with Darshan [82]. On Edison, they observed jobs that spent up to 87% of their run time doing I/O operations.

On the contrary, storage systems can also be the source of performance degradation. I/O interference between independent applications have been shown to be one of the root causes for the performance variability of HPC applications [83, 84, 85, 86].

3.2 Bringing malleability to cloud storage

In this section we discuss the use of co-located storage for malleable applications in the cloud. Although separated storage systems are also used in the cloud, we discuss their usefulness for malleable applications in the next section in conjunction with storage for HPC.

3.2.1 Rigid storage limits cloud elasticity

Elasticity is one of the main selling points for cloud infrastructures. Users can virtually get as many resources as they need, when they need them, and release them when they are not needed anymore. This is encouraged by the business model since users are charged only for the resources reserved.

However, the elasticity of the cloud is limited by the rigidity of storage systems. Databases are expected to become the bottleneck of storage processing due to their lack of malleability [87, 88, 89].

Determining the number of nodes needed to run a storage system is not a trivial problem, even for rigid applications. If the storage system is under-provisioned (uses too few nodes), it is not able to ensure efficient data accesses, which in turn deteriorates application performance. If the storage system is over-provisioned, nodes allocated to it are under-utilized and may increase the cost (financial and energetic) of running the system.

The complexity of the problem of provisioning nodes for the storage system is exacerbated when malleable or evolving applications are using it. Malleable and evolving applications do not have a fixed size. However, the rigidity of the storage system limits their maximal size because the storage would become under-provisioned. It also limits their minimal size since co-located storage runs on the same nodes as the application so the application cannot release nodes that host the storage system even if they are not needed.

A solution to fully enable cloud elasticity is to enable malleable distributed storage systems. Malleable applications can be rescaled with their own co-located malleable storage, and evolving applications can rescale their co-located malleable storage to fit their need.

3.2.2 Previous works related to malleable storage

With the emergence of cloud infrastructures, many works close to storage malleability have been conducted. In this section, we propose a classification of these works and detail them.

Malleability for maintenance purposes

In practice most distributed storage systems can accommodate the addition and removal of nodes for maintenance purposes. Storage systems usually stay deployed on a platform for long periods of time during which they must ensure data is available. Thus, when a node needs maintenance, it can be decommissioned from the system. Similarly, the commission of nodes is possible to increase the size of the storage system.

Among the co-located storage systems, HDFS [56], Tachyon [75] and RAMCloud [73] implement such operations. However, these operations are optimized to limit their impact on the performance of the system as seen by the running applications. They are usually not fast enough to fit the needs of malleable and evolving applications. Moreover, they are not always fully automatized. The commission operation often simply adds an empty node to the storage system; it is the case for HDFS, Tachyon, and RAMCloud. Thus, a rebalancing operation has to be executed to have fully operational new nodes, this mechanism may be provided (HDFS, RAMCloud) or not (Tachyon). The decommission mechanism may not be implemented and users may have to rely on the fault tolerance mechanism to remove nodes from the cluster (Tachyon).

Malleable databases

Dynamo [90] (from Amazon) and PNUTS [91] (from Yahoo!) are two distributed database systems that mention malleability as one of their design principles. Providing malleability enables these systems to quickly grow during periods of high activity. The malleability of these database systems is however not evaluated in the litterature.

Shutting down nodes to save energy

Sierra [92], Rabbit [93], and SpringFS [94] are three works based on the same idea: unneeded nodes are shut down to save energy. They enable this with a clever placement of the replicas of the data stored (each data object stored on the system is duplicated multiple times forming replicas in order to ensure fault tolerance). At all time, even with most of the nodes shut down, at least a replica of each data object stored is available.

These systems are not malleable for two reasons. First, not any node can be shut down, the data placement on the nodes ensures that at least a replica is place on a specific subset of nodes. The nodes in this subset cannot be shut down. Second, the shut-down nodes are

not released: the resource manager cannot allocate them to other jobs. They still host some replicas, and need to be available to rejoin the storage (with their data) when needed.

Elasticity managers

Lim *et al.* [95], Trushkowsky *et al.* [96], Al-Shishtawy *et al.* [97], and Jyothi *et al.* [35] propose elasticity managers. Their works aim to decide when to add and remove resources to malleable applications. They can also rescale co-located storage systems if needed by leveraging the existing malleability of co-located storage systems. These works aim to bring storage elasticity to the cloud without improving the malleability of the storage systems, and they suffer from the poor performance of existing rescaling operations.

For example, the work of Lim *et al.* [95] rescales HDFS when needed. They report a duration of 1750 seconds to remove a single node out of a 10 node cluster, and 7200 seconds to add 9 nodes to a cluster of 3 even though the storage only hosts 36 GiB of data.

Rebalancing algorithms and systems

Rebalancing the data objects across the nodes of the storage system is part of the rescaling operations. It reduces the risk of having overloaded nodes degrading the performances of the storage system.

The rebalancing operation in the context of peer-to-peer storage systems have been extensively studied by Rao *et al.* [98], Ganesan *et al.* [99], and Zhu and Hu [100]. Similar techniques have been applied to distributed storage systems [101] and RAID systems [102]. These works aim to balance a metric (such as network usage or processor utilization) across multiple nodes or devices as quickly as possible. However, these works ignore the strong constraints of the decommission operation: leaving nodes must be emptied before leaving to avoid any data loss.

> *To the best of our knowledge, no work focuses on improving the malleability of distributed storage systems.*

3.3 Bringing malleability to HPC storage

In this section, we discuss the relevance of malleable storage systems in the context of high performance computing. The discussion can be extended to clouds relying on separated storage since they have similar storage deployments.

3.3.1 Separated storage systems

Adding an efficient support of malleability to separated storage systems, which are the main form of storage systems used in HPC, does not present any major benefit for malleable and evolving applications running on HPC infrastructures. The specialized storage clusters that host the separated storage systems are usually exclusively and entirely used to run them, leaving no room to grow without physically adding nodes.

However, some efforts have been made in order to run HPC applications with separated storage systems in the cloud [103, 104]. These works involve the deployment of traditional

separated storage systems such as Lustre [59] on some nodes, and the execution of HPC applications on other nodes. Due to the pay-as-you-go model of the cloud, a support of distributed storage malleability for this usage could bring the benefits of malleability to this use-case.

3.3.2 Data services

Criticisms have been voiced about separated storage systems. Their performance have been increasingly problematic over the past decade, and a lot of research is being done to improve their scalability [59, 105, 106]. Moreover, over the same time period, the computing power of the most powerful supercomputers increased by more than 80 times, while the bandwidth to the storage was only multiplied by 10, and the capacity by 25 (Figure 3.1). Separated storage systems are expected to be a major bottleneck for exascale supercomputers (machines able to compute 10^{18} floating-point operations per second) [10].

Moreover, since separated storage systems are shared among multiple users, they need to accommodate a variety of applications with different requirements. Thus, they have to provide a generic interface, and follow standards such as POSIX that cover the needs of the broadest range of applications. Enforcing such a generality limits the performances of the storage [107].

Furthermore, developers need multiple libraries [108, 109, 110] and middleware [111] to transform the content of files to in-memory data representations ready to be used by applications [112].

A first approach to solve these problems has been the addition of burst buffers [113, 114] to supercomputers. Burst buffers are nodes placed in the periphery of the computing cluster near the storage nodes and that absorb bursts of I/O operations from applications in fast memory (RAM and SSD). While improving the apparent storage bandwidth for applications, burst buffers suffer from limitations due to their need to provide a generic interface like separated storage systems.

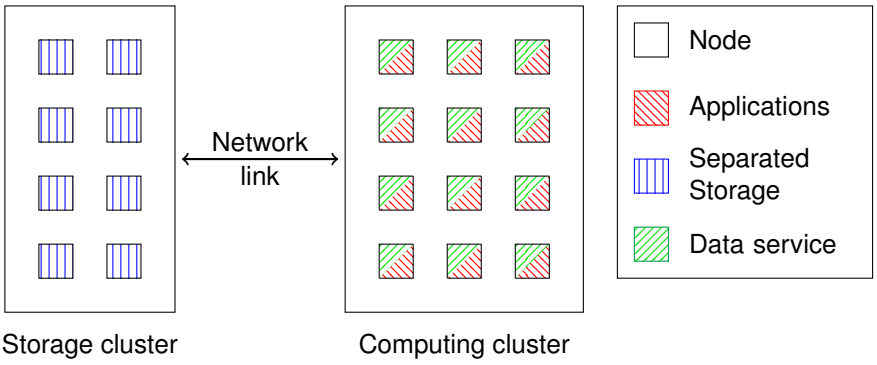


Figure 3.4: Data service deployed on a computing cluster alongside a separated storage system.

Another solution to the problems of data storage for high performance computing is *data services* [11, 115]. Data services use compute nodes to run services (Figure 3.4) that had been traditionally managed by the separated storage system, like managing metadata, or even storing all the temporary data generated by an application. This reduces the workload of the

separated storage system and offers some benefits of co-located storage systems: local data accesses, low latency, and good scalability.

Data services are deployed on the computing cluster alongside the application. They may be deployed on the same nodes, or have the exclusive usage of a set of nodes to run. It is important to note that data services are only deployed for a limited duration: it may be only for the execution of an application, or for the duration of a scientific campaign (few weeks to months).

Since data services are used by a single application or at most a limited set of applications, they can be tailored to these applications. Not only data can be stored in a format and with an organization that is convenient for the applications, the basic storage properties such as data consistency and fault tolerance can also be adjusted.

Transient storage systems are a category of data services. They are co-located storage systems deployed with the application only for the duration of the application. This type of storage has been made possible by the recent addition of persistent storage such as solid state drives and NVRAM to computing clusters in HPC infrastructure.

3.3.3 Malleability for data services

Adding malleability to data services presents the same advantages as adding malleability to co-located storage systems because of their similarities (Section 3.2.1). It would improve the support of malleable and evolving jobs, such as workflows that are a commonly used way to organize scientific applications. Moreover, when the data service is kept deployed for the duration of a scientific campaign, its malleability would enable a reduction of its footprint in between job submissions: when the data service is not used, it can be rescaled to fit on fewer nodes, which reduces the cost (financial and energetic) of the campaign.

3.3.4 Existing solutions

Some data services have been developed to date. Many have been designed to improve the performances of the separated storage system existing on the platform [116, 117, 118, 119, 120, 115]. For example, DeltaFS [115] manages most of the metadata operations on compute nodes thus speeding up metadata operations by orders of magnitude compared to relying on the parallel file system for metdata management.

The Mochi project [121, 11] aims to provide building blocks that can be used to create application-tailored data services and transient storage systems with minimal development work. The building blocks proposed by Mochi are well-defined, optimized, simple components such as libraries for threading, tasking, networking, group membership, local storage and databases among others. These components can be combined with a minimal amount of code to create application-specific data services.

> *While data services are designed for the specific needs of particular applications, to the best of our knowledge, none is malleable.*

3.3.5 An example of transient storage system: HEPnOS

An example of transient storage system is HEPnOS [11] developed in the context of the Mochi project [121] and the "HEP on HPC" project [122]. HEPnOS has been designed for High Energy Physics applications.

The goal of HEPnOS is to replace the storage system used for these applications. Thus, HEPnOS has been adapted to the needs of these applications. The data stored are serialized C++ objects instead of the ROOT files [123] currently used by the targeted High Energy Physics applications. A simple data hierarchy consistent with the applications is implemented. The system is optimized for small data objects (few bytes), and most of the data is stored in memory to be readily available and minimize latency. Initially data replication was envisioned for fault tolerance, but due to the nature of the produced data, the choice has been made to simply copy only the most relevant data to the separated storage system of the supercomputer, which is enough to recover from failure.

The addition of malleability to HEPnOS would be welcome due to the nature of its envisioned deployment. HEPnOS would stay deployed for the duration of a campaign (few weeks to months) but during this time applications would not be continuously using it. There would be periods without applications requiring HEPnOS. However, HEPnOS is deployed on compute nodes and keeping these nodes reserved has a financial and energetic cost. Thus, with a malleable HEPnOS, the storage could be shrunk to a minimum number of nodes when it is not used, and expanded again when needed. Besides, the number of nodes used by HEPnOS could be chosen on a per-application basis, instead of making a choice for the duration of the campaign. Thus, HEPnOS would need to be malleable in order to minimize the cost of the campaign by reserving fewer nodes when no applications are running, and to improve application performance by reconfiguring the storage before each application runs.

3.4 Challenges of storage malleability

For distributed storage systems to efficiently support malleability, a number of challenges must be addressed. In this section, we detail the main challenges related to distributed storage malleability and discuss the ones studied in this work. We denote as *distributed storage system* both co-located storage systems and separated storage systems as they would face the same challenges even if the usefulness for separated storage systems to implement malleability is unclear.

3.4.1 Challenges

The challenges related to the malleability of distributed storage systems can be organized in two categories: challenges arising from the guarantees storage systems have to ensure at all time even during rescaling operations, and challenges specific to storage malleability.

Ensuring guarantees of distributed storage systems

Fault tolerance: Distributed storage systems are designed to tolerate some types of failures under specific assumptions. For instance, most of the existing fault tolerance mechanisms at

least ensure that no data is rendered permanently unavailable if a single storage node suddenly becomes unavailable, as long as the system is not already recovering from another failure. This property must be ensured during rescaling operations.

Data consistency: Similarly, distributed storage systems follow a consistency model that gives guarantees to the users about the consistency of stored data. The same guarantees should be ensured during rescaling operations.

Addressing challenges specific to rescaling operations

Readiness of commissioned nodes: One of the first challenges to address is the behavior of newly added nodes. The simplest commission operation consists of simply adding empty nodes. However, it limits their usefulness since they can only start servicing requests once data has been written to them.

Duration: The speed at which rescaling operations can be completed must be considered, since faster operations will make new resources available earlier, and conversely unneeded resources can be released quickly to reduce costs. The benefits of fast rescaling operations are the same as for malleable applications presented in Chapter 2 (Section 2.3).

Impact on applications: Doing rescaling operations relies on the same resources (network bandwidth and storage device bandwidth) as the ones required for the normal utilization of the storage system. Thus, there exists a trade-off between the speed of rescaling operations and the efficiency of data accesses for the applications. On one hand, prioritizing a fast rescaling operation may quickly degrade the performances of applications relying on the storage (see Section 3.1.3). On the other hand, guaranteeing data accesses at all time is likely to slow down rescaling operations, reducing the potential benefits of distributed storage malleability.

Organization of data transfers: Rescaling distributed storage systems involves data transfers between nodes. The strategies and algorithms used to schedule these data transfers are the aspects that, in practice, implement the solutions chosen for the other challenges.

Challenges related to the elasticity of distributed storage systems

The previously mentioned challenges are only related to the malleability of storage systems, but leveraging this malleability to create elastic storage systems also brings many challenges.

The main one is the decision system required to decide when to rescale, which size should the storage be after rescaling and which nodes should be released or kept. The work can be done at the application level for evolving applications, but can also be done at the level of the platform resource manager to exploit the malleability of jobs to the benefit of the platform.

3.4.2 Challenges addressed in this work

This work focuses on *minimizing the duration of rescaling operations* since fast rescaling operations is a prerequisite for making malleability usable in practice with acceptable costs. From this work, we deduce strategies to schedule data transfers efficiently in order to enable fast rescaling operations.

However, this study could not have be done without considering other challenges, such as the fault tolerance, and the readiness of commissioned nodes. For the readiness of commissioned nodes, we assume newly commissioned nodes should behave as if they had always been part of the cluster, which means they must host as much data as the old nodes, among other properties. Similarly, we assume the fault tolerance to be enforced through most of this work. We study its relaxation in Chapter 8. In Part III, the fault tolerance is ignored due to the nature of the use case.

The impact of rescaling distributed storage systems on applications relying on them and the data consistency during these operations are also topics worth investigating in the future.

Conclusion

In this chapter, we discussed how storage systems impact the performance of distributed applications. We motivated the investigation of malleability for co-located storage systems in the clouds as well as for data services used in HPC. Lastly, we presented the main challenges of distributed storage malleability, and precised our focus on the duration of rescaling operations.

PART II

Modeling the Commission and Decommission Operations

CHAPTER 4

SCOPE AND ASSUMPTIONS

One of the main reasons for the reluctance to the adoption of distributed storage system malleability is the preconception that rescaling operations would be too slow due to the large number of data transfers required. However, the technologies involved in data transfers (storage and network) have greatly evolved during the past decades. The assumption of long rescaling operations needs to be re-evaluated.

To estimate the duration of such operations, we build performance models of commissions and decommissions in this second part of the manuscript. Having a performance model of rescaling operations enables the understanding of their bottlenecks, their cost, and how fast they can be executed. Based on these models, efficient rescaling operations can be implemented and evaluated, thereby allowing malleability to become an efficient means to optimize resource usage on large-scale infrastructures.

In this chapter, we first argue for a focus on the duration of the rescaling operations. We then lay down a few assumptions about the platform and the initial data distribution on the nodes. These assumptions are made to reach a trade-off between the accuracy and the complexity of the model. We later determine and explain the constraints and objectives of the rescaling operations. All the assumptions detailed in this chapter are used as a basis to build the performance models of commissions and decommissions in the following chapters.

4.1 Modeling the duration of storage rescaling operations

We have seen in Chapter 2 the benefits of having fast commissions and decommissions: cost and energy consumption reduction, as well as high reactivity to varying workloads. In this section, we discuss the focus of the model on the *minimal* duration of the rescaling operations, and the potential uses of such a model.

4.1.1 Revisiting the preconception of slow storage rescaling operations

Rescaling distributed storage systems is assumed to be slow since they require large data transfers that are assumed to be slow. However, the bandwidth of storage devices has been greatly improved over the last decades: data that used to be stored on hard drives can now be stored in solid state drives, or even in memory. Thus, the storage bandwidth increased by about four orders of magnitude over the past two decades (from tens of megabytes per second for hard drives, to a hundred of gigabytes per second for in-memory storage). Moreover, the emergence of new storage technologies such as non-volatile memory needs to be considered.

During the same time period, network technologies have also been improved. For instance, the ASCI Red Supercomputer [124] (ranked first in the Top500 [1] in June 1999) had a node

link bandwidth of 800 MiB/s, while Summit [46] (ranked first in the Top500 in June 2019) has a 100 Gib/s interconnect. This is a 16 times speed up.

We re-evaluate the supposed length of rescaling operations by modeling the *minimal* duration of rescaling operations, regardless of their implementation.

4.1.2 On the usefulness of a performance model

Quickly estimating the relevance of distributed storage system malleability: Estimating the duration of the rescaling operations without a need for experimental measurement allows to quickly determine if using a malleable distributed storage system is relevant for a given workload on a given platform. For instance, if the rescaling takes too long for marginal performance improvements, using a malleable distributed storage system is likely detrimental.

Improving scheduling: Having an accurate estimation of the duration of rescaling operations is critical for the scheduling process. With a prediction of the duration of a decommission, a resource manager can anticipate the retrieval of nodes needed for another application. With an accurate estimation of the duration, nodes can be decommissioned from a job, and then given to another one without idle time between the two operations, and without delaying the latter job.

Having a precise estimation of the duration of an operation limits job slowdown, especially at high system utilization [125]. Besides, a model for the duration of both rescaling operations enables informed decisions on whether to commission nodes.

Evaluating the rescaling mechanism of distributed storage systems: Knowing the theoretical performance limits of an operation also gives a reference for the evaluation of actual implementation of rescaling mechanisms, helping developers of such systems to estimate their margin of improvement.

Highlighting bottlenecks inherent to the rescaling operations: Last, a precise modeling work highlights the inherent bottlenecks of the operations that cannot be overcome by implementations, helping developers to identify and optimize critical aspects of their implementation.

4.2 Scope of the models

The decommission mechanism is similar to the one often used for fault tolerance. When a node crashes, its data needs to be recreated on the remaining nodes of the cluster. Similarly, when a node is decommissioned, its data needs to be moved onto the remaining nodes of the cluster.

With this in mind, we reduce the scope of our study to storage systems using *data replication* as their fault tolerance mechanism. In this case, the crash recovery mechanism is highly parallel and is fast: most of the nodes share some replicas of the data with the crashed nodes and thus can send their data to restore the replication level to its original level.

We do not consider *full node replication*, used in systems where sets of nodes host exactly the same data, since the recovery mechanism is fundamentally different from the one used with data replication. We also exclude from our scope systems using erasure coding for fault

tolerance, such as Pelican [126], since such mechanisms require CPU power to regenerate missing data, and a model of the minimal duration of rescaling operations would therefore have to take into account the usage of the CPU (unlike the case of data replication that is not CPU intensive) to be as realistic as possible.

Another major fault tolerance mechanism for storage systems is lineage, used in Tachyon [75] for Spark [9]. We do not consider lineage in this work because the base principles differ greatly from the ones needed for efficient decommission. With lineage, the sequence of operations used to generate the data is saved safely; and, in case of a crash, the missing data is regenerated. Consequently, a file system using lineage must be tightly coupled with a data processing framework, and the CPU power needed to recover data depends on the application generating the data.

4.3 Assumptions and notations

In order to obtain a comprehensible and useful model, a set of assumptions about the infrastructure and the initial data placement must be made. These assumptions are voluntarily simple to enable general conclusion, not specific to a particular platform or implementation. We show in Chapter 7 that these assumptions allow to closely model the behavior of HDFS.

4.3.1 Assumptions about the cluster infrastructure

We make three assumptions about the cluster in order to build a comprehensible model.

Assumption 1: Homogeneous cluster

All nodes have the same characteristics, in particular they have the same network throughput (S_{Net}) and the same throughput for reading from / writing to storage devices (S_{Read}, S_{Write}).

Moreover, we consider that either the network or the storage is the bottleneck of commissions and decommissions. Both operations rely heavily on data transfers, so these components are likely to be the bottlenecks.

Having a storage bottleneck is common with most storage systems such as HDFS [56], because they store data mostly on hard drives that have lower bandwidths than the network. However, the case of a network bottleneck is common for systems that store their data in memory such as RAMCloud [73].

Estimating whether the storage devices or the network are the bottleneck can be delicate. A rough estimation would be the following. The network is the bottleneck if it limits at any point the reading or writing of data from storage: $S_{Net} < S_{Read}$. Conversely, the bottleneck is located at the storage level if the read/write speeds cannot keep up with the speed at which data is sent and received through the network. All proofs are provided in Appendix A.

The storage is a bottleneck if $\frac{S_{Read}S_{Write}}{S_{Read}+S_{Write}} < S_{Net}$. **(Prop. 4.1)**

Note that this estimation does not include the possibility of buffering data in memory. There is also a possibility of having bottlenecks both at the storage and the network level at the

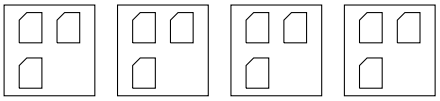


Figure 4.1: Data-balanced storage system (assuming each data object has the same size). Each node hosts the same amount of data.

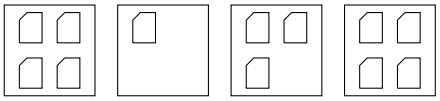


Figure 4.2: Data imbalanced storage system (assuming each data object has the same size). Nodes do not host the same amount of data.

same time. While this less-intuitive situation is outside the scope of this study, a model can be obtained by extending the present work.

Assumption 2: Ideal network

The network is full duplex, data can be sent and received with a throughput of S_{Net} at any time, and there is no interference.

This assumption determines the maximum throughput each node can individually reach. Thus, building upon this assumption ensures we model the fastest possible decommission. In order to build generic models, we do not consider the topology of the network and focus only on the I/O of each individual node.

The latency of the network is ignored in this assumption because of the large amount of data transferred during rescaling operations. Thus, transfer times should be dominated by the bandwidth and the latency should be a negligible part of the transfer times.

However, since no network is ideal, the bisection bandwidth per node is likely to be a better metric to use for the network, and the interference problem can be taken into account by using a bandwidth measured in the presence of interference (or provided by some model that is interference-aware).

Assumption 3: Ideal storage devices

Storage devices can either read or write, but cannot do both simultaneously. Also, the writing speed is not higher than the reading speed ($S_{Write} \leq S_{Read}$)

Assumption 3 holds for most modern storage devices.

Moreover, we assume that all resources are available for the commissions and decommissions without restrictions.

4.3.2 Assumptions on initial data placement

We denote as a *data object* the unit of information stored by users on the storage system (which can be files, objects, blobs, or even chunks of larger files). We distinguish data objects from the space they occupy on the storage devices. We denote the occupied storage space as simply *data*. The size of the data objects is not the same as the size of the data because of the replication. With a replication factor of r, the data is r times larger than the size of the data objects. Finally, we denote as a *replica* each copy of a data object and do not consider any hierarchy between replicas.

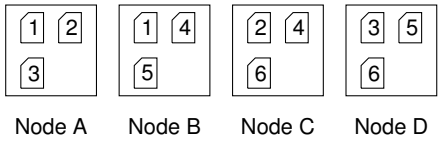


Figure 4.3: Uniform data placement with $r = 2$. To simplify, we assume all objects have the same size. Each pair of nodes has exactly one object in common.

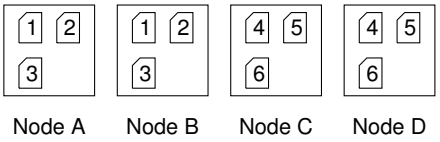


Figure 4.4: Data placement not uniform with $r = 2$. Nodes A and B have all their data in common, like nodes C and D.

The initial *data placement* (the placement of all replicas of all data objects on the nodes of the storage cluster) is important for the performance of commissions and decommissions. Thus, we make assumptions in this respect.

Assumption 4: Data balance

Each node initially hosts the same amount of data D.

Assumption 4 matches the data-balancing target of policies implemented in existing file systems, such as HDFS [56] or RAMCloud [73]. Each node should host the same amount (or volume) of data, which is not necessarily the same number of data objects. For example, in Figure 4.1 the data is balanced across the 4 nodes, while it is not the case in Figure 4.2.

Assumption 5: Uniform data replication

Each object stored in the storage system is replicated on $r \geq 1$ distinct nodes.

Assumption 5 simply states that data replication is the fault tolerance strategy used by the considered distributed storage systems. In Figures 4.3 and 4.4 each data object is replicated twice.

We denote as *exclusive data* of a subset of nodes the data objects that have all their replicas on nodes included in the specified subset.

Assumption 6: Uniform data placement

All sets of r distinct nodes host the same amount of exclusive data, independently of the choice of the r nodes.

This assumption reflects the way data is placed on storage systems using data replication. In case of failures of up to $r - 1$ nodes, each of the remaining nodes can participate equally to the system's recovery since they all host exactly the same amount of data that needs to be replicated. This situation is approached by some existing storage systems such as HDFS by randomly choosing the hosts of each replica.

Figure 4.3 shows an example of uniform data placement with 4 nodes and a replication factor of 2. Node replication, the fault tolerance strategy that consists of having two nodes hosting the replicas of the same data set (Figure 4.4) does not have a uniform data placement.

4.3.3 Formalizing the problem

At the end of both rescaling operations (commissions and decommissions), the data placement should satisfy the following objectives.

Objective 1: No data loss
No data can be lost during either operation.

Objective 2: Maintenance of the replication factor
Each object stored on the storage system is replicated on r distinct nodes. Moreover, the replication factor of the objects should not drop below r during the operations.

Objective 3: Maintenance of data balance
All nodes host the same amount of data D' at the end of the operation.

Objective 4: Maintenance of a uniform data placement
All sets of r distinct nodes host the same amount of exclusive data, independently of the choice of the r nodes.

Objective 1 is obvious for a storage system. Objectives 2, 3, and 4 are the counterparts of Assumptions 5, 4, and 6 and are here to ensure a data placement that is the same as if the cluster always had its new size. Moreover, reaching the objectives also prepares the data placement for a future rescaling operation.

4.4 Discussion

These strong assumptions are used to build a generic model of the duration of commissions (Chapter 5), a generic model of the duration of decommissions (Chapter 6), and to study the possibility to relax the fault tolerance during decommissions (Chapter 8). However, such assumptions and objectives are rarely attained in practice: they reflect the goal of the data-balancing policies implemented in many current state-of-the-art distributed file systems such as HDFS [56] or RAMCloud [73]. We show in Chapter 7 that the models obtained from these assumptions match the performance of HDFS when the rescaling mechanism is designed to minimize the duration of the operation.

Conclusion

In this chapter, we argued for a focus on the minimal duration of rescaling operations. We laid down assumptions about the platform (cluster homogeneity, ideal network, and ideal storage devices) and about the initial data placement on the distributed storage system (data balance, data replication, uniformity of the data placement). We defined the objectives of the rescaling operations that are to ensure the absence of data loss, the maintenance of the replication factor, an even data placement, and a uniform data placement. These assumptions and objectives are required not only to define the scope of the performance models but also to reach a trade-off between the accuracy and the complexity of the model.

CHAPTER 5

A MODEL FOR THE COMMISSION

The commission of new resources to a running system is one of the two basic operations required for malleability. Having a fast commission has one main benefit: the newly provisioned resources are ready to be used quickly, improving the performance of the system faster. This also reduces the cost (financial and energetic) of the preparation of these resources.

In this chapter, we use the assumptions presented in Chapter 4 to build a model for the commission. With it, we show that the duration of the commission decreases when the number of nodes decommissioned at once increases. We also highlight the bottlenecks of the operation, and suggest strategies to minimize the duration of the operation.

5.1 Problem definition

Commissioning nodes to a storage system involves two steps. First, the storage cluster receives a notification about new nodes ready to be used. Then, the data stored in the cluster is balanced among all nodes to homogenize the load on the servers.

Ideally, at the end of the operation, the system should not have any traces of the commission; it should appear as if it always had the larger size. This aspect is important to ensure a normal operating state, as well as to prepare for any operation of commission or decommission that could happen afterwards. It is enforced by Objectives 2, 3, and 4.

In this chapter, we look for a model of the duration of the fastest commission operation, we denote as t_{com} the time needed to commission a set of x empty nodes (new nodes) to a cluster of N nodes (the old nodes). The commission terminates when all objectives defined in the previous chapter (Objectives 1, 2, 3, and 4) are satisfied.

5.2 Identifying required data movements

The commission is mainly a matter of transferring data from old nodes to new nodes. In the following parts, the amount of data to transfer from sets to sets is quantified.

5.2.1 Data needed by the new nodes

With the objectives of not losing data, of maintaining data replication, and of load-balancing (Objectives 1, 2, and 3), and the fact that each of the N nodes initially host D data (Assumption 4), we deduce the following.

Each node must host $D' = \dfrac{ND}{N+x}$ *of data at the end of the commission.* **(Prop. 5.1)**

With the amount of data needed per node, we obtain the amount of data that must be written onto the new nodes.

$$D_{\rightarrow new} = xD' = \frac{xND}{N+x} \qquad \textbf{(Prop. 5.2)}$$

5.2.2 Required data movements from old to new nodes

If data replication is not considered, all the data that new nodes have to host at the end of the commission should be sent from the old nodes.

Because of data replication, however, some objects must have multiple replicas to be written on the new nodes. This requirement is particularly important because those objects could be sent once to new nodes and then forwarded from new nodes to new nodes to reduce the amount of data to send from old nodes to new ones.

Let us denote as p_i the probability that an object has exactly i replica(s) on the new nodes. Since we want a specific final distribution of data, these probabilities are known.

$$p_i = \begin{cases} \frac{\binom{x}{i}\binom{N}{r-i}}{\binom{N+x}{r}} & \forall 0 \leq i \leq r \\ 0 & \forall i > r \end{cases}. \qquad \textbf{(Definition 5.1)}$$

The problem is modeled as an urn problem: x new nodes, N old ones, we extract r of them (Assumption of uniformity 6) and compute the probability that exactly i new nodes are selected.

Minimum amount of data to read and send from old nodes Of all unique data, only the part that has at least a replica to place on the new nodes must be moved. This amount is expressed as $D_{old \rightarrow new}$.

$$D_{old \rightarrow new} = \frac{ND}{r}(1 - p_0). \qquad \textbf{(Prop. 5.3)}$$

Data that can be moved from either new or old nodes to new nodes Of course, reading and sending the minimum amount of data is not enough to complete the commission. The remaining data can be read either from old nodes or from new nodes after they receive the first replicas (from the old nodes). The total amount of this data is $D_{old/new \rightarrow new}$.

$$D_{old/new->new} = \frac{ND}{rx}(\frac{rx}{N+x} + p_0 - 1). \qquad \textbf{(Prop. 5.4)}$$

5.2.3 Avoiding data transfers between old nodes

The preceding sections focused on the data transfers to new nodes; however, data transfers between old nodes could compete for resources against data transfers to new nodes that are mandatory to complete the operation.

To avoid data transfers between old nodes, we need to design a data redistribution scheme (Algorithm 5.1) for the old and new nodes that has the following property: the data that was present initially on an old node is either staying on it or being transferred to new nodes but never transferred to another old node.

1 Group objects according to the placement of their replica; *i.e.*, two objects whose replicas are on the same set of servers will be considered in the same group.
2 Divide {groups} according to the proportions in the new placement; *i.e.*, from a given group C of objects, select a proportion p_i (for all i in $[0, r]$) of objects that will be replicated i times in the new servers.
3 For each subdivision, assign the corresponding number of replicas to the new nodes uniformly and remove the same number of replicas from the old nodes uniformly.

Algorithm 5.1: Algorithm designed to rebalance data without transferring data between old nodes.

Assuming that objects can always be divided into multiple objects of any smaller size, Algorithm 5.1 avoids all data transfers between old nodes and satisfies all the objectives. **(Prop. 5.5)**

With this result, since no data transfers are required between old nodes, they will not have any impact on the model of the minimal duration of the commission of storage nodes.

Of course, in practice objects cannot be indefinitely divided. So when relaxing the goals of load-balancing and uniform data distribution, as done in practice, data transfers between old nodes can be avoided.

5.3 A model when the network is the bottleneck

We can determine the time needed to transfer data from the amount of data. However, two cases must be considered, depending on the relative speed of the network with respect to that of the storage devices. In the first case, a slow network is the bottleneck, and the nodes do not receive data fast enough to saturate their storage devices' bandwidth. In the second case, the storage device is slow and becomes a bottleneck (i.e., the storage device cannot write at the speed at which the data is received from the network).

In this section, we consider the case where the network is the bottleneck of the system.

5.3.1 Many possible bottlenecks

The operation of commission is composed of multiple concurrent actions such as sending and receiving data. Moreover, two strategies are possible when sending data: sending the minimum

amount of data from old nodes and forwarding it between new nodes, or balancing the amount of data sent by the old and the new nodes.

Thus, we design a model of the minimal duration needed by each action; and then, because all actions must finish to complete the commission, we extract the maximum of the times required for each action to obtain the model of the minimal duration needed to commission nodes.

5.3.2 Receiving data

Each new node must receive D' data, with a network throughput of S_{Net}. Thus, the time needed to do so is at least T_{recv}.

$$T_{recv} = \frac{ND}{(N+x)S_{Net}} \quad \textbf{(Prop. 5.6)}$$

5.3.3 Sending the minimum amount of data from old nodes

If one chooses the strategy of sending as little data as possible from old nodes, the minimal time needed is $T_{old \to new}$.

$$T_{old \to new} = \frac{D}{rS_{Net}}(1 - p_0) \quad \textbf{(Prop. 5.7)}$$

Of course, sending the minimum amount of data from old nodes means that the new nodes will spend some time $T_{new \to new}$ to forward the data.

$$T_{new \to new} = \frac{ND}{rxS_{Net}} \sum_{i=2}^{r} (i-1)p_i \quad \textbf{(Prop. 5.8)}$$

5.3.4 Balancing the sending operations between old and new nodes

The previous strategy can be easily improved when the new nodes spend more time forwarding data than the old nodes spend sending it (*i.e.* $T_{new \to new} > T_{old \to new}$). In this situation, the old nodes should send more data, thus reducing the amount that must later be forwarded by new nodes.

In that case, the minimum time required to send all the needed data to their destination is $T_{\to new}^{balanced}$.

$$T_{\to new}^{balanced} = \frac{xND}{(N+x)^2 S_{Net}} \quad \textbf{(Prop. 5.9)}$$

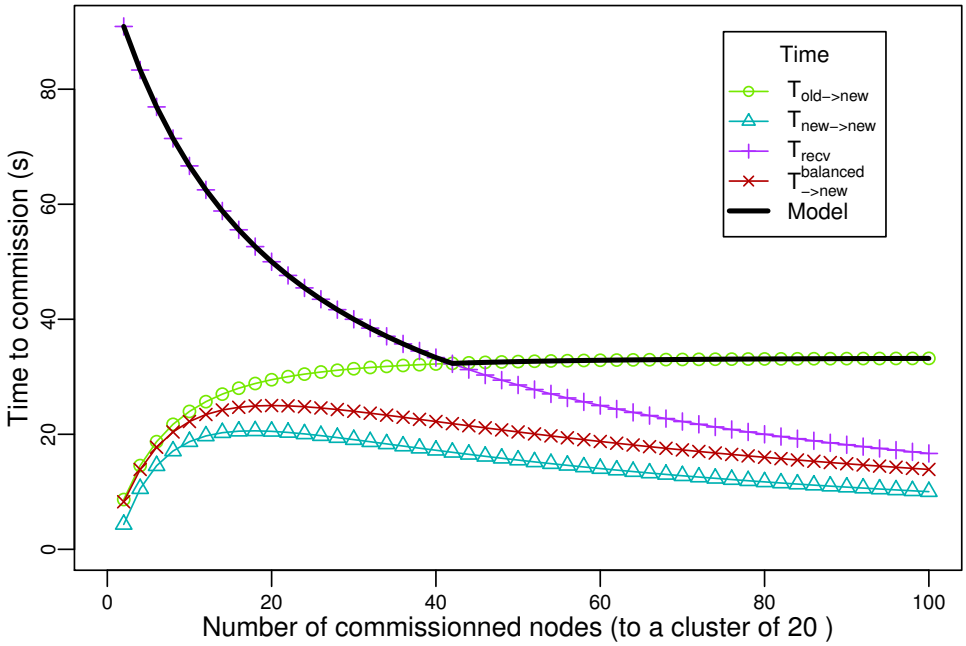


Figure 5.1: Important times when adding nodes to a cluster of 20 nodes each hosting 100 GiB of data, $S_{Net} = 1\ GiB/s$, $r = 3$.

5.3.5 Duration of the data transfers

A few useful properties can be deduced.

If $T_{old \to new} \leq T_{new \to new}$, *then* $T_{old \to new} \leq T_{\to new}^{balanced} \leq T_{new \to new}$. **(Prop. 5.10)**

If $T_{old \to new} \geq T_{new \to new}$, *then* $T_{old \to new} \geq T_{\to new}^{balanced} \geq T_{new \to new}$. **(Prop. 5.11)**

$T_{recv} \geq T_{\to new}^{balanced}$ **(Prop. 5.12)**

In particular, it follows that $T_{\to new}^{balanced}$ is always smaller than the time needed for the new nodes to receive data (Property 5.12). Thus, equally distributing the task of sending data between new and old node does not have any impact on the duration of the operation.

5.3.6 Commission time with a network bottleneck

The commission, in the case of a network bottleneck, cannot be faster than t_{com}.

$t_{com} = max(T_{old \to new}, T_{recv})$ **(Prop. 5.13)**

Indeed, the minimum commission time is at least as long as the time needed to receive the data and at least as long as the time needed to send it (balancing the sending operations between old and new nodes if needed).

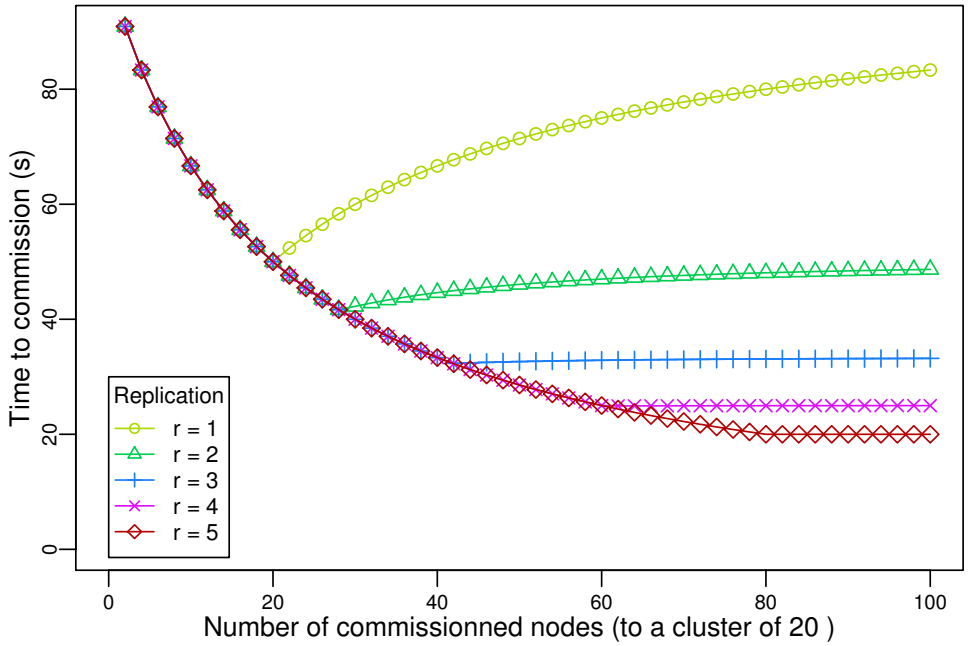


Figure 5.2: Minimal duration of commissions with different replication factors. Nodes are added to a cluster of 20 nodes each hosting 100 GiB of data (including replicas, thus the size of the data objects hosted decreases with the replication factor). $S_{Net} = 1\ GiB/s$.

We note that the time to commission a set of nodes to a storage cluster is proportional to the amount of data initially present on each node D. This is discussed in more details in Section 5.5.

In Figure 5.1, we can observe the different minimum times that have been used in constructing the model in the context of a 20-node cluster initially hosting 100 GiB of data per node. When less than 40 nodes are added at once, the bottleneck is the reception of the data by the new nodes. When more than 40 nodes are added, however, the old nodes do not manage to send the data they have to send as fast as the new nodes can receive it, and thus the emission is the bottleneck.

The impact of the replication factor can be observed in Figure 5.2. The minimal duration of the commission of storage nodes decreases when the replication factor increases. In fact, the minimal amount of data that must be sent from the old nodes decreases when the replication factor increases, which decreases in turn the time needed to send it to the new nodes.

> *To have the fastest commission with a network bottleneck, the old nodes must send only one replica of each transferred object. The other replicas can be transferred between newly commissioned nodes.*

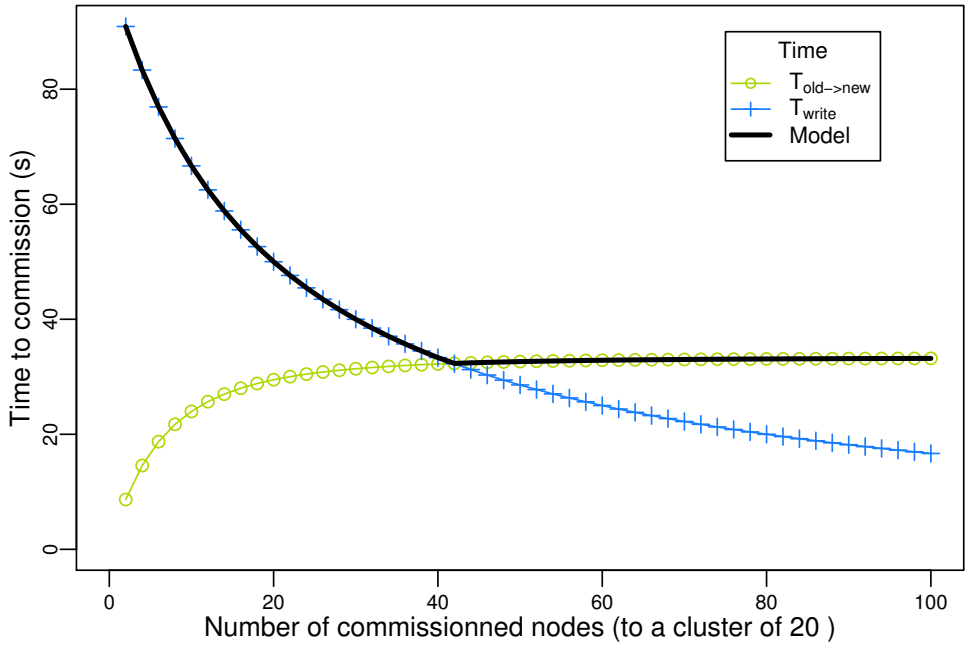


Figure 5.3: Important times when adding nodes to a cluster of 20 nodes each hosting 100 GiB of data. $S_{Read} = S_{Write} = 1\ GiB/s$, $r = 3$.

5.4 A model when the storage devices are the bottleneck

In the case of a storage bottleneck, similar actions to the network bottleneck case are required (reading and writing data), but the minimal amount of time needed to finish each action depends on the characteristics of the storage devices.

In the following, the minimal duration of each action is evaluated in the context of a storage bottleneck.

5.4.1 Writing data

Each new node must write D' data on its storage devices, it takes at least T_{write}.

$$T_{write} = \frac{ND}{(N+x)S_{Write}} \qquad \textbf{(Prop. 5.14)}$$

5.4.2 Reading the minimum amount of data from old nodes

The part of the data that must be read from old nodes can be read in at least $T_{old \to new}$.

$$T_{old \to new} = \frac{D(1-p_0)}{rS_{Read}} \qquad \textbf{(Prop. 5.15)}$$

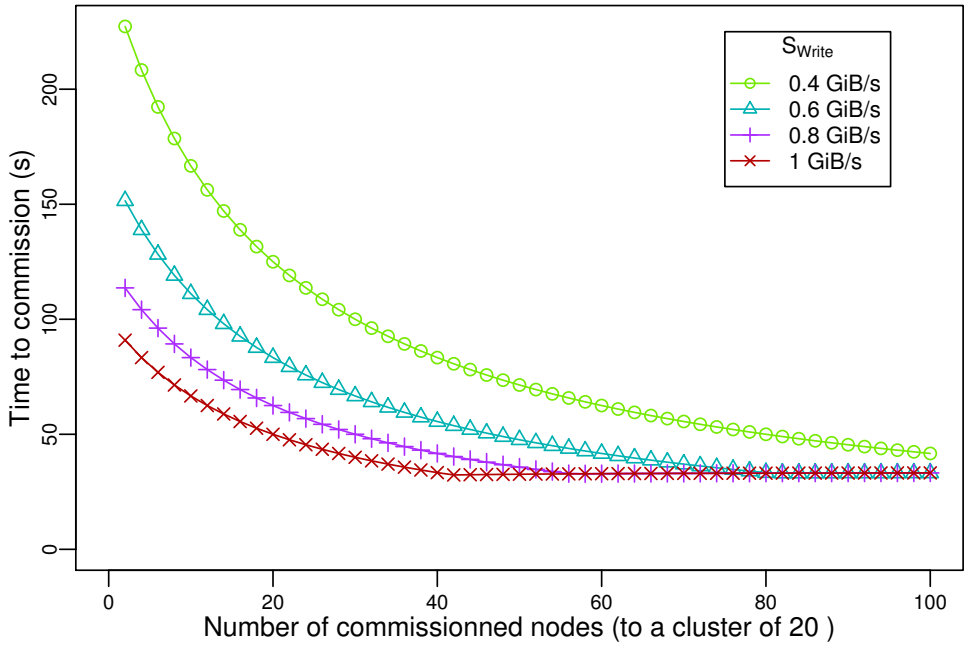


Figure 5.4: Minimal duration of the commission of nodes to a cluster of 20 nodes each hosting 100 GiB of data with different ratios S_{Write}/S_{read}. $S_{Read} = 1\ GiB/s$, $r = 3$.

5.4.3 When buffering is possible

If the data can be put in a buffer that is faster than the storage device (typically from drive to memory), reading from memory is orders of magnitude faster than from disk and thus can be ignored.

In this case, the relevant objects are read once from the storage devices of the old nodes, sent to new nodes, stored in the storage device and onto a buffer, and then forwarded from the buffer to other new nodes if needed.

In that case, the minimal time needed for the commission is defined as follows.

$$t_{com} = max(T_{write}, T_{old \to new}). \qquad \textbf{(Prop. 5.16)}$$

In Figure 5.3, we can observe the different times that are important in constructing the model in the context of a 20-node cluster initially hosting 100 GB of data per node. As in the case of a network bottleneck, when less than 40 nodes are added at once, writing is the bottleneck. In contrast, when more than 40 nodes are added simultaneously, the old nodes are not numerous enough to read the unique data they must read as fast as the new nodes can write it.

When the replication factor changes, the model behaves similarly to the case of a network bottleneck, if fact, if $S_{Write} = S_{Read}$, it is exactly the same formula.

In Figure 5.4, we show the minimal duration of the commission with different writing speed of the storage device. A slow writing speed compared to the reading speed of the storage device lengthens the time needed to write data on the new nodes. When $S_{Write} = S_{Read}$, reading data from the old nodes is the bottleneck when 42 or more nodes are commissioned, however, when

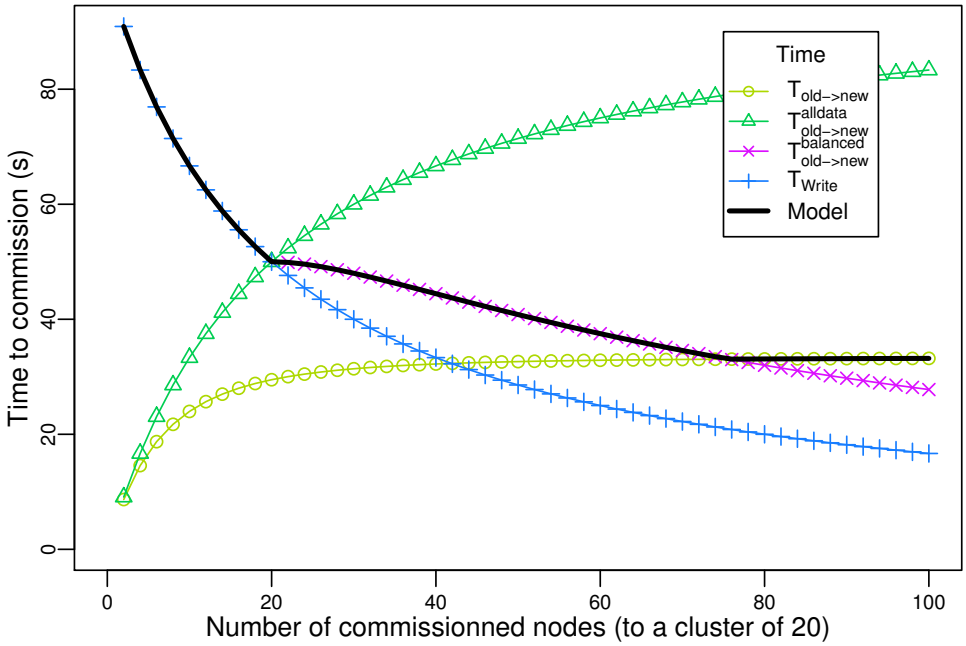


Figure 5.5: Important times when adding without buffering nodes to a cluster of 20 nodes each hosting 100 GiB of data. $S_{Read} = S_{Write} = 1\ GiB/s$, $r = 3$.

$S_{Write} = 0.6\ GiB/s$ and $S_{Read} = 1\ GiB/s$, at least 80 nodes must be added to have the same bottleneck.

5.4.4 When buffering is not possible

Buffering may not be usable, in particular in the case of in-memory storage (in this case, the buffer would have the same throughput as the main storage).

Because the storage devices cannot read and write at the same time (Assumption 3), the new nodes should prioritize writing. When the number of commissioned nodes increases, however, the old nodes will spend more time reading all the data ($T^{alldata}_{old\to new}$) than the new nodes will spend writing it (T_{Write}).

$$T^{alldata}_{old\to new} = \frac{xD}{(N+x)S_{Read}} \qquad \textbf{(Prop. 5.17)}$$

In this situation, the new nodes can spend some time to forward data to other new nodes and reduce the time ($T^{balanced}_{old\to new}$) needed to read data from the old nodes.

If $x \geq \frac{NS_{Read}}{S_{Write}}$*, new nodes can forward data, and* $T^{balanced}_{old\to new} = \frac{xND}{(N+x)^2}\frac{S_{Read}+S_{Write}}{S_{Read}S_{Write}}$. **(Prop. 5.18)**

Thus, when no buffering is possible, the time needed to commission nodes is as follows.

$$t_{com} = \begin{cases} T_{write} & \text{if } x \leq \dfrac{NS_{Read}}{S_{Write}}, \\ max(T_{old \to new}, T^{balanced}_{old \to new}) & \text{otherwise} \end{cases} \quad \textbf{(Prop. 5.19)}$$

Similarly to the case of a network bottleneck, the time to commission a set of nodes to a storage cluster is proportional to the amount of data initially present on each node D (see Section 5.5).

In Figure 5.5, we show the different times that are important in constructing the model in our usual example of a 20-node cluster. When less than 20 nodes are added, the bottleneck is the new nodes that are not writing fast enough. When between 20 and 80 nodes are added at once, the old nodes cannot read all the data fast enough by themselves; thus the new nodes start to forward part of the data they receive to other new nodes. But when more than 80 nodes are added at once, the new nodes do not manage to read the unique data they must read fast enough: they are the bottlenecks.

In the case of a storage bottleneck without buffering, the minimal duration of the commission behaves similarly to case of a storage bottleneck with buffering when the replication factor or the writing speed are changed and are thus not detailed here.

> *No optimization can be done to mitigate the bottlenecks when the data can be buffered by the nodes. However, when this is not the case, new nodes should write then forward part of the data they receive from the old ones to reduce the load on the old nodes, thus reducing the duration of the commission.*

5.5 Discussion

In this section, we discuss some aspects of the model.

5.5.1 Duration proportional to the initial amount of data per node

The minimal duration of the commission operation is always proportional to the initial amount of data per node. This is due to the objective of final data balance: each node must host the same amount of data at the end of the operation, amount that depends on the initial amount of data per node.

5.5.2 Commissioning many nodes at once

In both cases of a storage bottleneck and of a network bottleneck, the more nodes are commissioned at once, the faster the operation finishes.

Thus, it is faster to add many nodes at once to match the workload than to add few nodes after few nodes until the workload is matched.

5.5.3 Relaxing data balance

If the objective of data balancing (Objective 3) is not enforced, the new nodes can be added and left without any data, making the commission duration null. However, without data on these

new nodes, they cannot serve read requests until data is written to them, which reduces the relevance of adding new nodes.

If data balance is relaxed (for example, by allowing a difference in the amount of data per node between the least and the most loaded node), less data needs to be placed on the new nodes, speeding up the commission.

5.5.4 Relaxing data uniformity

Removing the objective of data uniformity (Objective 4) has for consequence to reduce the amount of data to read from old nodes. Indeed, one can simply put r replicas of each read data onto the new nodes instead of enforcing a uniform data placement (Section 5.2.2). Having less data to read would reduce $T_{old \to new}$ and thus speed up the commission when reading data is the bottleneck.

Conclusion

In this chapter, we created a model for the minimal duration of a commission.
The model shows that in the case of a network bottleneck, the longest operations are the emission of data from the old nodes to the new ones, and the reception of such data by the new nodes. Thus, the old nodes should send as little data as possible, and new nodes should forward it between its destinations.
In the case of a storage bottleneck, the situation may be slightly more complex and requires a careful balancing of the reading operations between old and new nodes when no buffering of the data is possible.
We use this model in Chapter 7 to hint at potential improvements of the commission mechanisms implemented in HDFS.

CHAPTER 6

A MODEL FOR THE DECOMMISSION

The decommission of resources from a running system is the complementary operation to the commission. Resources are removed from a system and returned to the resource manager that can allocate them to other jobs. Having a fast decommission gives the resource manager the ability to react faster to varying workloads. In the case the job itself requests the decommission (and not the resource manager of the platform), a fast operation allows returning unneeded resources, reducing its cost (financial and energetic).

Having a model of the duration of the operation for distributed storage systems allows making informed decisions on whether and when to decommission nodes. Moreover, by identifying the inherent bottlenecks of the operation, we establish efficient and precise strategies to decommission nodes quickly.

We rely on the assumptions and objectives defined in Chapter 4 to establish a model of the minimal duration of a decommission. We show that the data replication and data placement can be leveraged to enable fast decommissions. We highlight the fact that receiving and writing data on the remaining nodes is the bottleneck of the operation and thus should be optimized in any implementation of decommission mechanism.

6.1 Problem definition

The decommission of nodes from a storage cluster is composed of two steps. First, to ensure that no data is lost (Objective 1), the data is transferred out of the leaving nodes and sent to the remaining nodes. Then, the storage providers on the leaving nodes are shut down, and these nodes are returned to the resource manager.

We are looking for the duration of the fastest possible decommission, thus we establish a model of the minimal duration t_{decom} of the decommission of x nodes from a cluster of N nodes. More precisely, t_{decom} is the minimum time needed between the reception of the order to decommission some nodes from the resource manager (thus the choice of the leaving nodes does not depend on the cluster) and the moment when all leaving nodes can be safely removed. At the end of the decommission, the remaining nodes must satisfy the objectives defined in Chapter 4.

6.2 Identifying required data movements

Since the replication factor must be left unchanged after the decommission (Objective 2), all data present on the leaving nodes must be written on remaining nodes.

Thus, the data to write D_{write} is the amount of data hosted by the leaving nodes.

$$D_{write} = xD \quad \textbf{(Prop. 6.1)}$$

6.3 A model when the network is the bottleneck

To establish the minimal duration of the decommission, we note that data reception is the bottleneck of this operation, for three reasons. First, any remaining node shares some data with all the leaving nodes (Assumption 6); thus, any remaining node can read and send data at the same rate as leaving nodes. Second, only the remaining nodes can receive and write some data to their storage devices: since leaving nodes will be removed form the cluster, having them store more data is pointless. Third, storage devices have a lower writing speed than their reading speed (Assumption 3).

Thus, the minimal duration of the decommission is equal to the amount of data to write (D_{write}) divided by the writing speed of the whole cluster $S_{write}^{cluster}$ (Definition 6.1).

$$t_{decom} = \frac{D_{write}}{S_{write}^{cluster}} \quad \textbf{(Definition 6.1)}$$

We assume the network is full duplex, and without interference (Assumption 2). In this case, the remaining nodes can receive data at the network speed S_{Net}, even if they send data at the same time. Each of the $N-x$ remaining nodes can receive and write data at the network speed S_{Net}. Thus, we deduce the writing speed of the cluster $S_{write}^{cluster}$.

$$S_{write}^{cluster} = S_{Net}(N-x) \quad \textbf{(Prop. 6.2)}$$

With this, the overall decommission time t_{decom} is defined as follows.

$$t_{decom} = \frac{xD}{S_{Net}(N-x)} \quad \textbf{(Prop. 6.3)}$$

In Figure 6.1 we observe the duration of the decommission. We note that, unlike the minimal duration of the commission, the minimal duration of the decommission in the case of a network bottleneck does not depend on the value of the replication factor.

> *Reception of data is the bottleneck of the operation. Because of this, data transfers should primarily be scheduled to balance the reception on all remaining nodes. However, the emissions should also be balanced to some level in order to avoid creating a bottleneck at the sending level.*

6.4 A model when the storage devices are the bottleneck

If the storage is the bottleneck ($S_{Read} \leq S_{Net}$), the situation is slightly different: most storage devices (disk, RAM, or NVRAM) cannot read and write at the same time (Assumption 3).

However, by using buffering (reading once from the storage) and keeping a copy in memory, what is read can be written more than once. From this, we denote as $R(N,x)$ the ratio of the total amount of data written divided by the total amount of data read across all nodes:

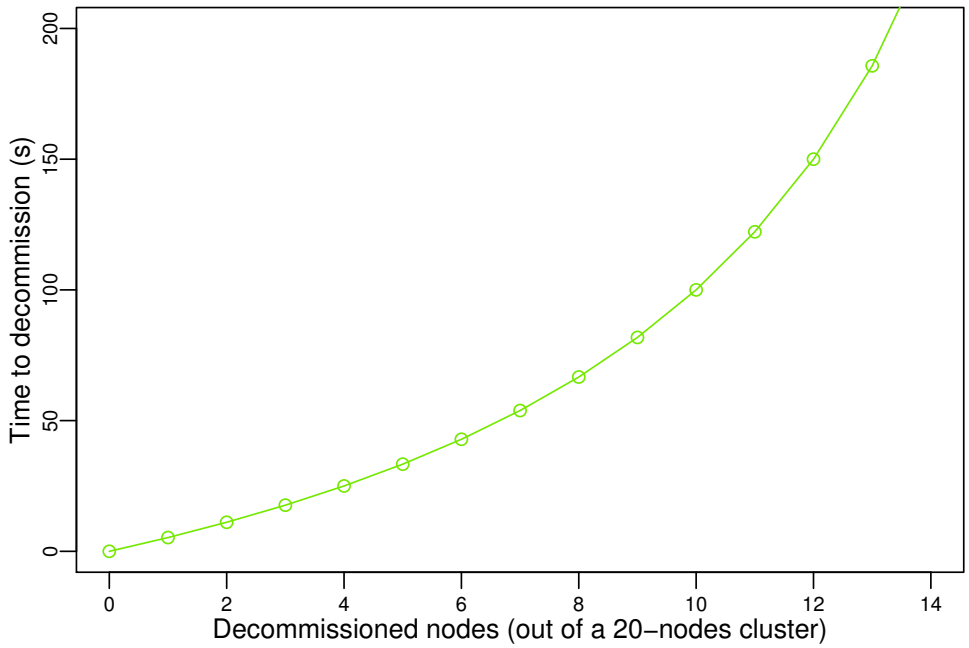


Figure 6.1: Time needed to transfer 100 GiB from each of the leaving nodes when the network is the bottleneck and has a bandwidth of 1 GiB/s. The cluster is composed of 20 nodes.

$$p_i = \begin{cases} 0 & \text{if } i > r, \\ \dfrac{\binom{r}{i}\binom{N-r}{x-i}}{\binom{N}{x}} & \text{for } i \leq r. \end{cases} \qquad \textbf{(Definition 6.2)}$$

$$R(N,x) = \begin{cases} 1 & \text{when buffering is not possible,} \\ \dfrac{\sum_{i=1}^{r} i p_i}{\sum_{i=1}^{r} p_i} & \text{otherwise.} \end{cases} \qquad \textbf{(Prop. 6.4)}$$

The ratio $R(N,x)$ (Property 6.4) is expressed with the probability p_k of an object to have k replicas on the leaving nodes (Definition 6.2), which is a classical urn problem. In this case, the data is read once but is written k times to ensure the replication factor.

There are two possible situations to consider in order to determine the writing speed of the cluster ($S_{write}^{cluster}$): either the leaving nodes can read enough data to saturate the writing on the remaining nodes, or they cannot.

In the first case, the writing speed of the cluster is

$$S_{write}^{cluster} = S_{Write}(N-x). \qquad \textbf{(Prop. 6.5)}$$

In the second case, the remaining nodes are not saturated by the amount of data received from the leaving nodes and thus can also read and write more data to accelerate the decommission. In this case, the writing speed of the cluster is

$$S_{write}^{cluster} = \frac{NR(N,x)S_{Read}S_{Write}}{S_{Write} + R(N,x)S_{Read}}. \qquad \textbf{(Prop. 6.6)}$$

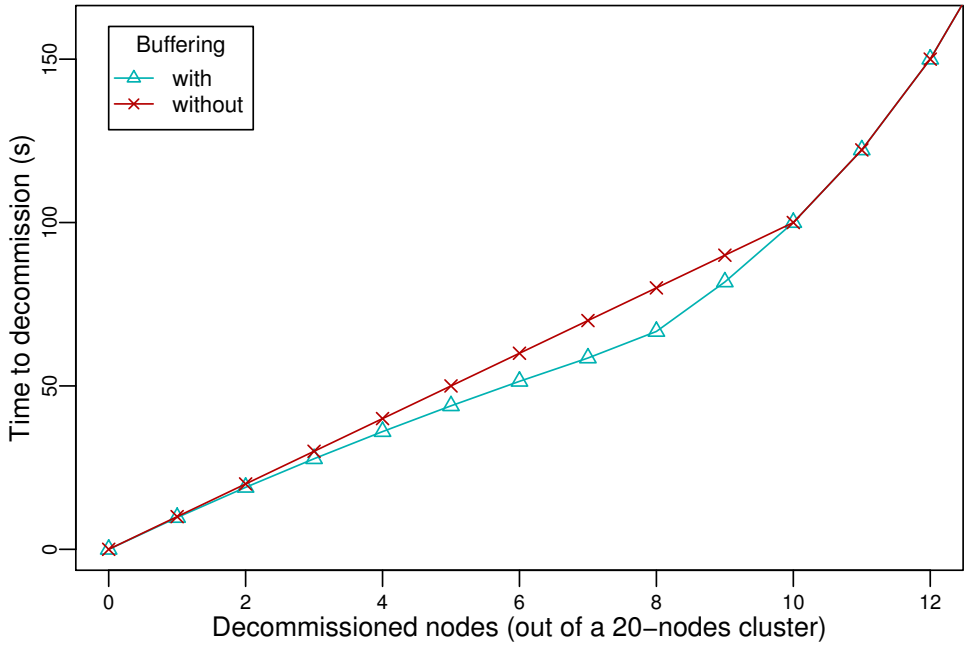


Figure 6.2: Time needed to transfer 100 GiB from each of the leaving nodes when storage devices are the bottleneck, with and without buffering. $S_{Write} = S_{Read} = 1\ GiB/s$, $r = 3$, $N = 20$.

The leaving nodes are able to saturate the remaining nodes when more than $T(N, x)$ nodes are decommissioned at once. The threshold $T(N, x)$ can be expressed as follows.

$$T(N,x) = \frac{NS_{Write}}{R(N,x)S_{Read} + S_{Write}} \qquad \textbf{(Prop. 6.7)}$$

With Properties 6.1, 6.1, 6.5, 6.6, and 6.7, the minimal duration of the decommission in the case of storage as a bottleneck is expressed as follows.

$$t_{decom} = \begin{cases} \dfrac{xD}{S_{Write}(N-x)} & \text{if } x \geq T(N,x), \\ \dfrac{x \cdot D \cdot (S_{Write} + R(N,x)S_{Read})}{N \cdot R(N,x) \cdot S_{Read} \cdot S_{Write}} & \text{in other cases.} \end{cases} \qquad \textbf{(Prop. 6.8)}$$

In Figure 6.2, we present the minimal decommission duration in the case of a storage device bottleneck with and without buffering. We observe that buffering reduces the duration of the operation when few nodes are decommissioned simultaneously. This is due to the fact that storage devices must share their time between read and write operations on the remaining nodes; the possibility to buffer data reduces the overall amount of data to read, thus reducing the storage device usage.

The replication factor impacts the duration of the decommission only when buffering is possible. In Figure 6.3, we show the minimal duration of the operation on clusters configured with different replication factors. The larger the replication factor, the quicker the decommission.

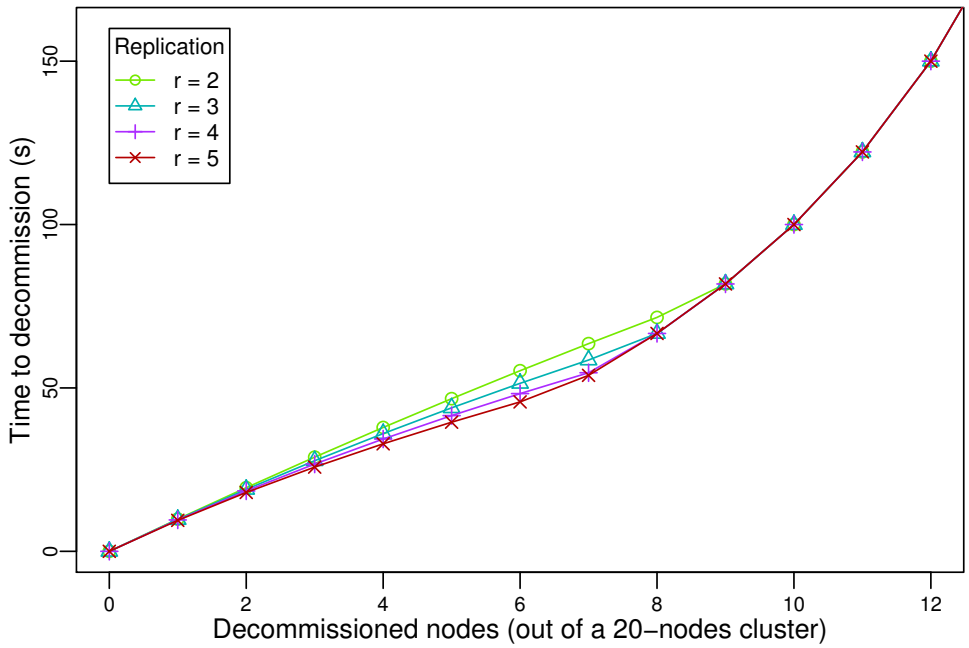


Figure 6.3: Time needed to transfer 100 GiB from each of the leaving nodes when storage devices are the bottleneck with buffering and with different replication factors. $S_{Write} = S_{Read} = 1\ GiB/s$, $N = 20$.

This is also due to the fact that a high replication factor reduces the amount of data to read (with buffering, each data object must only be read once and can be written up to r times) which in turn reduces the duration of the decommission.

> *In the case of a storage bottleneck, writing the data on the remaining nodes is limiting the operation, thus writing data should be prioritized. Because leaving nodes can only read data, they should read as much data as possible. If the storage devices of remaining nodes are not saturated, they can also read some data to speed up the operation.*

6.5 Observations

Five relevant observations can be made regarding the model.

6.5.1 Decommission without replication

The proposed model only works when the data objects stored are replicated. Without replication, the model is simpler: only the nodes leaving can read and send data since they are the only ones hosting the data that needs to be transferred, and thus reading and sending the data can also be a bottleneck.

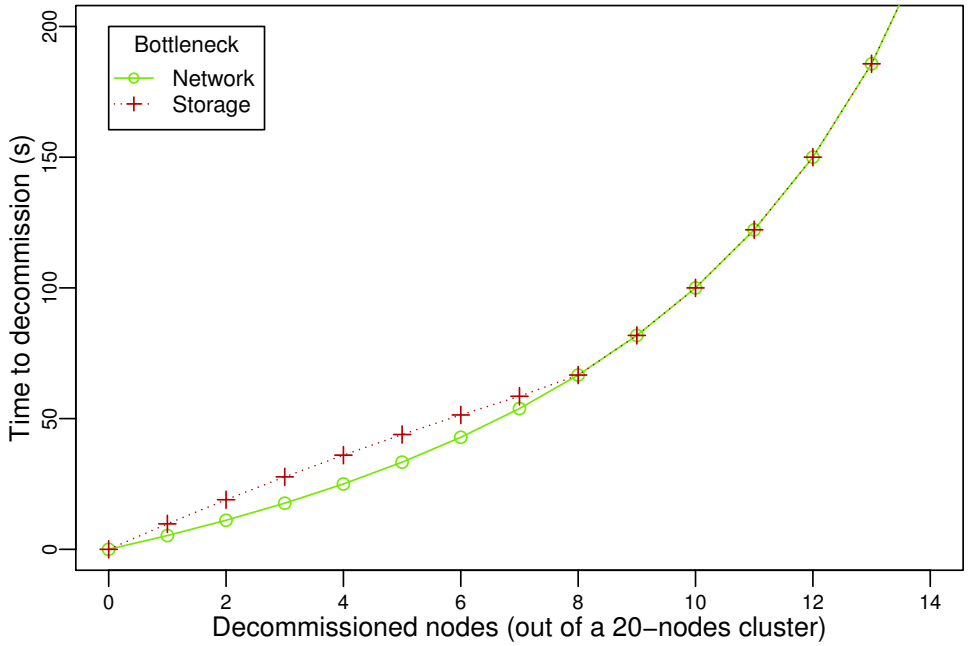


Figure 6.4: Time needed to transfer 100 GiB from each of the leaving nodes on two different settings: either the network is the bottleneck and has a bandwidth of 1 GiB/s, or the storage is the bottleneck and has read and write speeds of 1 GiB/s. The cluster is composed of 20 nodes.

Thus, the model of the minimal duration of the decommission without replication is the maximum of the time needed to read and send the data to move and of the time needed to receive and write it. From this, we deduce the models for a network bottleneck (Property 6.9) and for a storage device bottleneck (Property 6.10).

$$t_{decom} = \max\left(\frac{xD}{(N-x)S_{Net}}, \frac{D}{S_{Net}}\right) \quad \textbf{(Prop. 6.9)}$$

$$t_{decom} = \max\left(\frac{xD}{(N-x)S_{Write}}, \frac{D}{S_{Read}}\right) \quad \textbf{(Prop. 6.10)}$$

6.5.2 Comparison between the two possible bottlenecks

Figure 6.4 summarizes the minimal decommission times for both kinds of bottlenecks, on an artificial platform that exposes the differences in behavior between both bottlenecks. Both bottlenecks have the same bandwidth. When decommissioning 8 or more nodes, the duration of the decommission under both bottlenecks is the same. In this case, all remaining nodes only receive / write data at the same rate (1 GiB/s). However, when fewer nodes are decommissioned, a storage bottleneck leads to a slower operation compared to a network bottleneck. Indeed, the storage devices have share their time reading and writing data (they cannot do both at the same time — Assumption 3) making the decommission slower compared to the case of a network bottleneck in which data can be received and sent simultaneously.

6.5.3 Impact of the data hosted per node on the decommission time

The decommission time is proportional to the amount of data hosted per node. Thus, the decommission scales linearly with the amount of data hosted on a given platform.

6.5.4 Impact of the proportion of decommissioned nodes

In the case of a network bottleneck, the decommission time depends only on the proportion of nodes leaving the cluster. In this situation, decommissioning 20 nodes in a cluster of 100 or 4 in a cluster of 20 will take the same time if each node hosts the same amount of data.

Hence, if each node hosts 100 GiB of data, the decommission can take as little as 20 seconds if the decommission mechanism minimizes the duration of the operation.

6.5.5 Decommissioning nodes one by one or in batch

In a situation with k nodes to decommission, we may wonder whether it is better to decommission all the nodes at once, or one by one. This question can be answered with the model. We have the following property.

In the case of a network bottleneck, decommissioning a set of nodes in k consecutive steps takes as much time as decommissioning the same nodes all at once. **(Prop. 6.11)**

The reason behind this unexpected result is that even if more data is transferred (data is moved to a node that is decommissioned later), the transfer speed is also higher. The bottleneck in this case comes from the number of nodes that can write, which is higher in the first step than in the last steps. Note that this result does not hold in the case where the bottleneck is at the storage level: the storage devices compensate for the nodes that cannot write and has a constant speed.

6.6 Discussion about the models

It is interesting to discuss how the relaxation of some assumptions could impact the models. Another important question is: how to adapt the model if a workload is running on the same nodes.

6.6.1 Can any objective be relaxed during the decommission?

All the assumptions and objectives listed in Chapter 4 are building blocks for the models. They are, however, quite strong and may not match implementations. Some of them can be relaxed.

Relaxing data uniformity: For instance, ensuring a uniform data placement (Objective 4) does not have any impact on the duration of the decommission. However, not enforcing uniform data placement would be detrimental to the duration of the following rescaling operations. Indeed, the model rely on the uniformity of the data placement to balance over all nodes the reading and sending tasks of a decommission. Without data placement uniformity, these tasks

may be so imbalanced that reading / sending data becomes the bottleneck instead of writing it on the remaining nodes, making the decommission last longer.

Relaxing fault tolerance: In Chapter 8 we study the relaxation of fault tolerance during the decommission. The replication factor of the files present on the leaving nodes can be lowered to free the nodes faster and then can be brought back to its initial level once the nodes have left. This approach is faster since only the data that is exclusively on leaving nodes is moved to avoid losses. However, the decommission needs to be followed by a stabilization phase in which the shrunk cluster recreates the missing replicas for fault tolerance and following decommissions. It is a trade-off between decommission speed and fault tolerance.

Relaxing data balance: Since receiving and writing data is the bottleneck of the decommission of storage nodes, the amount of data received and written per node should be balanced to get the fastest operation. Thus, if the initial data placement is balanced (Assumption 4) then the final data placement should also be data-balanced in order to achieve the fastest decommission, regardless of the objective of data balance (Objective 3).

6.6.2 Are the models relevant if a workload is present?

The models presented in Chapters 5 and 6 assume that all resources are available for the rescaling operation, but this is often not the case. Some implementations might limit the bandwidth used for the operations or give a lower priority to the commission or decommission to favor the execution of applications. In both cases, this choice is made when implementing the distributed file system. Thus, one can add trade-offs such as limiting the bandwidth available for the data transfers of the commission or decommission, and hence reduce the S_{Net}, S_{Read} and S_{Write} accordingly. In this case, the models still represent the theoretical minimal duration of the operations.

6.6.3 Can one determine the throughput for network and storage devices?

The established models highlight precise strategies to enable fast rescaling operations. Thus, it possible to obtain precise modeling of the network and storage device throughput for fast rescaling operations. These models show the behavior and the expected bandwidth usage of both potential bottlenecks, and can help to identify inefficient implementations.

Conclusion

In this chapter, we presented a model of the minimal duration of the decommission. The replication of the data and its uniform placement can be leveraged to ensure a fast decommission. In particular, receiving and writing data is an inherent bottleneck of the operation, it should then primarily be optimized in order to obtain good performances. We also discussed the relaxation of the assumptions for both the commission and decommission models.

CHAPTER 7

CASE STUDY: MALLEABILITY OF HDFS

In this chapter we use the previously defined models to evaluate the commission and decommission in a practical setting: we focus on the case of HDFS, a relevant state-of-the-art distributed file system, in which these operations are implemented. We consider two types of deployment: HDFS with its storage in RAM (bottleneck at network level); HDFS with storage on disk drives (bottleneck at storage level).

For each configuration, we compare experimental measurements with the lower bound and propose improvements for the transfer scheduler of HDFS that would decrease the duration of the operation.

We first study the decommission, for which HDFS exhibits good performance. When the bottleneck is at the network level (when data is stored in memory), the decommission follows closely the performance model and uses nearly 90% of the available bandwidth. In the case of a storage-level bottleneck, the duration of the decommission of HDFS has the same behavior as the model, but is about 3 times slower. However, the throughput could easily be improved with an optimization of the disk access patterns. We also study the commission of HDFS, and show that it is not optimized for speed, and could greatly be sped up.

7.1 Decommission in HDFS

This section presents a study on decommission, where HDFS exhibits good practical results for this operation.

7.1.1 Experimental setup

Testbed

The experiments presented in this section have been performed on the Grid'5000 [17] experimental testbed. The *paravance* cluster from Rennes was used for the decommission measurements. Each node has 16 cores, 128 GB of RAM, a 10 Gb/s network interface, and two hard drives. The file system's cache has been reduced to 64 MB in order to limit its effects as much as possible. Unless stated otherwise, 20 nodes from this cluster were used for each experiment.

HDFS

We deployed HDFS and Hadoop 2.7.3. One node acted as both DataNode (slave of HDFS) and NameNode (master of HDFS) while the others were used only as DataNodes. One drive

Table 7.1: Parameters used for the experiments.

Parameter	RAM setup	Disk setup
dfs.namenode.decommission.interval	1	1
dfs.namenode.replication.work.multiplier.per.iteration	25	2
dfs.namenode.replication.max-streams-hard-limit	30	4

was reserved for HDFS to store its data. Most of the configuration was left to its default values, including the replication factor, which was left unchanged at 3.

The data on the nodes was generated using the *RandomWriter* job of Hadoop, which yields a typical data distribution for HDFS.

HDFS in memory

To experiment RAM-based storage with HDFS, we used the same setup as in the paper introducing Tachyon [75]: a tmpfs partition of 96 GiB was mounted, and HDFS used it to store data. A tmpfs partition is a space in RAM that is used exactly (and natively by Linux systems) as a file system. It is seen as a drive by HDFS, but the speeds are that of the underlying RAM (6 GiB/s reading and 3 GiB/s writing) and therefore much higher than disks. It moves the bottleneck from the storage devices to the network.

7.1.2 Experimental protocol

In order to measure the decommission time of HDFS, a random subset of nodes was selected among the DataNodes except the one hosting the NameNode, and the command to decommission these nodes was given to the NameNode. The recorded time is the time elapsed between the moment the NameNode receives the command and the moment the NameNode indicates that the data has been transferred and the decommission process is finished.

For all experiments, measurements were repeated 10 times. Boxplots represent, from top to bottom, the maximum observed value, the third quartile, the median, the first quartile, and the minimum.

Some parameters in the configuration of HDFS were optimized for the experiments: HDFS checked the decommission status every second (instead of 30 seconds by default) with the parameter *dfs.namenode.decommission.interval*. This gives us the decommission times with a precision of 1 second. Moreover, since HDFS schedules data transfers every 3 seconds, we used the parameters presented in Table 7.1 to schedule enough transfers to maximize the bandwidth utilization while avoiding unbalanced work distribution. We confirmed the maximization of the transfer speed of HDFS experimentally.

7.1.3 Decommission in HDFS: when the bottleneck is at the network level

To create a setup with a bottleneck at the network level, we configured HDFS with RAM-based storage (we could measure writing at 3 GiB/s in memory, including an overhead induced by the file system, while transfers on the network are done at 1.1 GiB/s that is, 9.4 Gb/s).

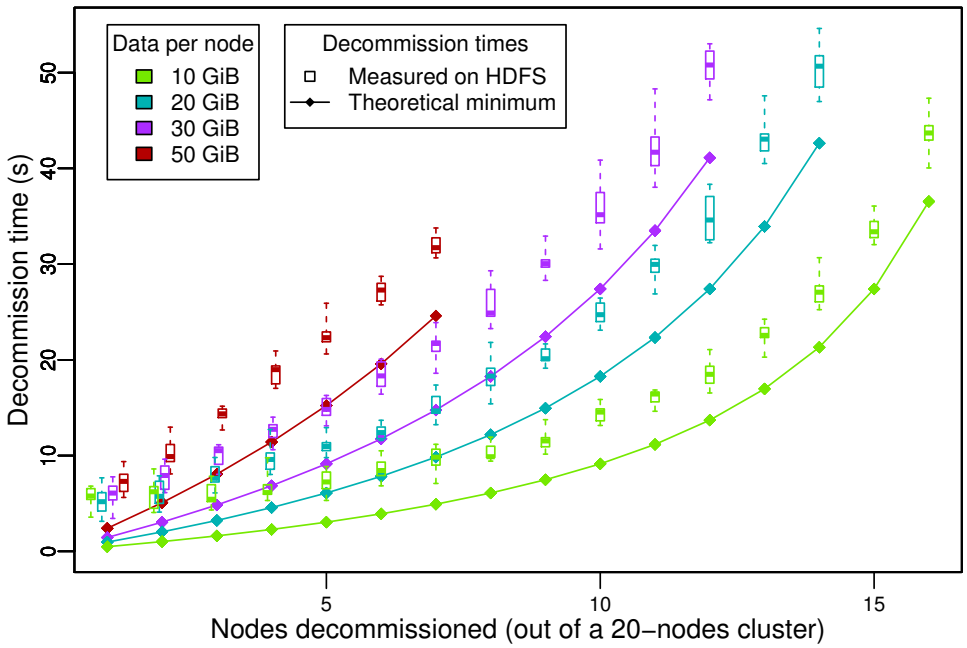


Figure 7.1: Decommission time measured on the platform presented in Section 7.1.3. The minimum theoretical time obtained with the model on this platform is added.

Closeness of HDFS to the theoretical minimum

Figure 7.1 shows the decommission times observed for various amounts of data hosted per node and various numbers of nodes to decommission. In addition, the figure shows the theoretical minimum decommission time for this platform computed with the model presented in Chapter 6. The number of nodes that can be decommissioned is limited by the capacity of the cluster after decommission. Thus, the maximum number of nodes that can be decommissioned is different depending on the amount of data stored per node.

We observe that the decommission times are short, especially for small numbers of decommissioned nodes. In particular, no decommission lasts more than 55 seconds. If we consider the scenario used to validate KOALA-F [33] in which 1 or 2 nodes are added or removed every 5 minutes, the decommission would take less than 13 seconds every 5 minutes, which is a cost of at most 5% of the time to save 5 to 10% of the energy and/or renting cost of the hardware. Moreover, we observe that the measured values are close to the model: the decommission mechanism of HDFS is close to the theoretical minimum duration in this case, but can be improved.

Fitting the model to HDFS

In order to fit the model to the performance of HDFS, we add a constant cost t_0 to the formula: $t_{decom} = \frac{xD}{S_{Net}(N-x)} + t_0$. In this case, the constant cost represent mostly the time needed to start and stop a decommission outside of any data transfers.

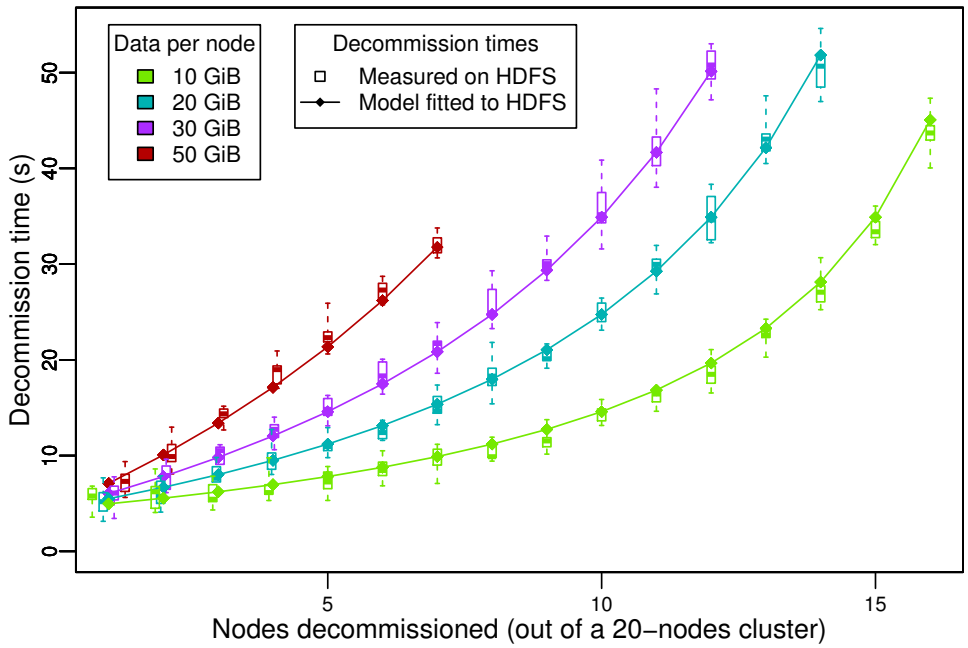


Figure 7.2: Model fitted to HDFS on the platform presented in Section 7.1.3. The model fits the data with a coefficient of determination r^2 of 0.98.

In Figure 7.2, we use linear regression to determine the values of S_{Net} and t_0 that would fit the model and explain the decommission time of HDFS. The values obtained are $t_0 = 4.4$ seconds and $S_{Net} = 0.98$ GiB/s with a coefficient of determination of 0.983, which means that the variance in the measures is explained at 98.3% by the model with these parameters. These values indicate mainly that the decommission process uses 90% of the network bandwidth (which is the main bottleneck) to receive data on the remaining nodes and there is a flat cost of 4.6 seconds.

The network bandwidth determined by the regression matches the observations, as we can see in Figure 7.3, when the transfer durations are long enough to have a steady transfer speed. The value of t_0 includes many delays due to the implementation of HDFS, such as the scheduling of the transfers done only every 3 seconds (on average 1.5 seconds delay) or the verification of the status of the decommission every second (on average 0.5 seconds delay). It also includes the impact of the imperfect data balancing of the initial data placement, and of the straggling data transfers due to the amount of data transferred at the same time.

The model explains the decommission times of HDFS well even though some assumptions needed by the model are not fulfilled by HDFS: the data is not evenly distributed (see Figure 7.4), and the transfer speeds are not constant (see Figure 7.3, especially the reception speed that should not change). These imperfections explain why the value of t_0 is higher than expected: it compensates for the lower transfer speeds for small amounts of data transferred.

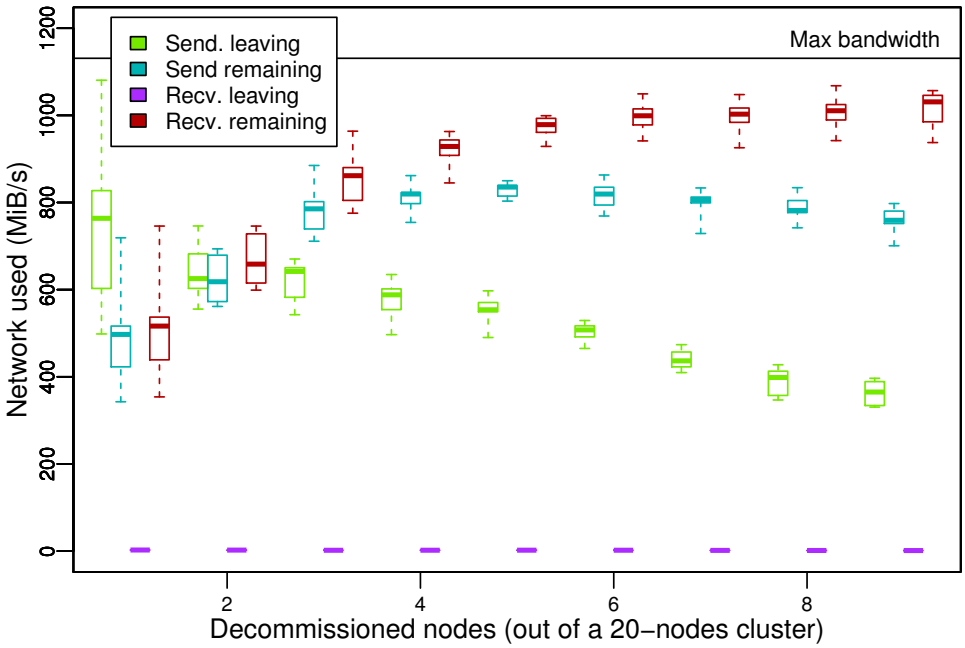


Figure 7.3: Average bandwidth measured for leaving and remaining nodes for both reception and emission on the platform presented in Section 7.1.3. Each node hosts 40 GiB of data, and the maximum bandwidth was measured with a benchmark.

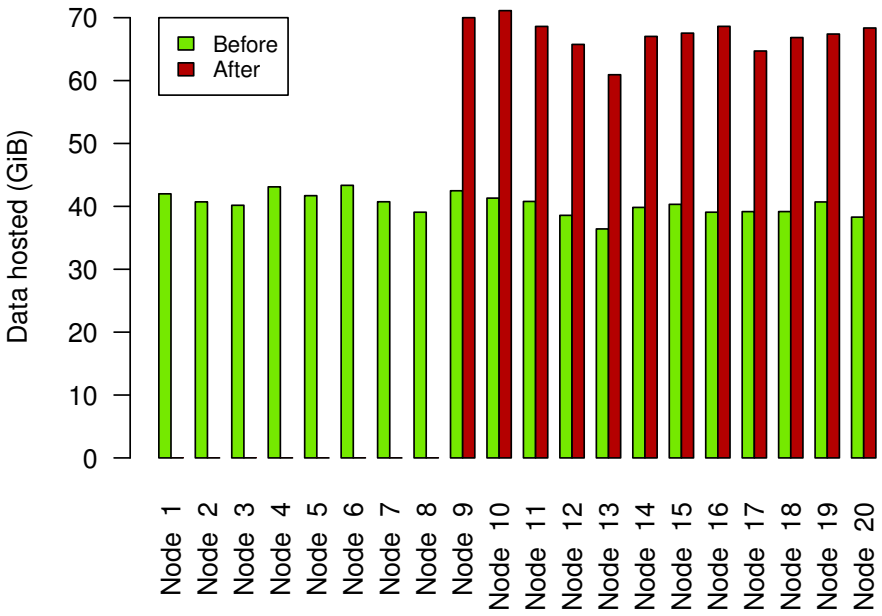


Figure 7.4: Amount of data before and after the decommission on the nodes of the cluster on the platform presented in Section 7.1.3. Data was generated for 40 GiB per node, and 8 nodes were decommissioned.

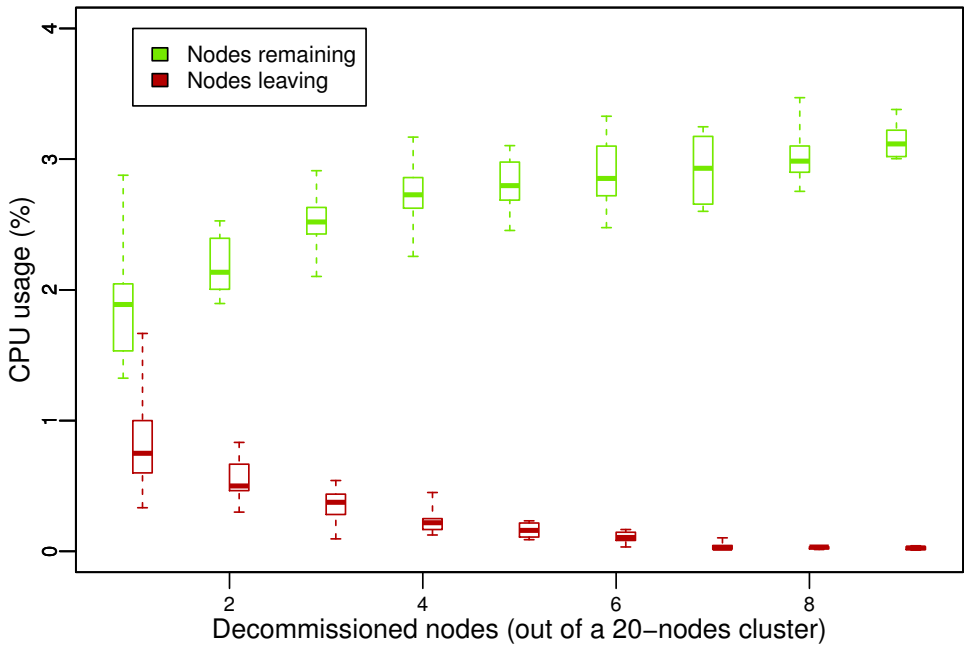


Figure 7.5: Average CPU usage measured for leaving and remaining nodes on the platform presented in Section 7.1.3. Each node hosts 40 GiB of data.

Practical cost of HDFS

Figure 7.3 shows the network usage during the decommission. As expected, the leaving nodes do not receive any data since they are going to leave the cluster after the operation. The nodes send data at a lower bandwidth since the bottleneck is the reception on the remaining nodes. Figure 7.5 shows that the CPU usage is low during the decommission: it is used only for the metadata operations. As shown in Figure 7.4, the storage on remaining nodes does increase.

Potential for improvement of the decommission time in HDFS

Although the performance is already close to the model, it can still be improved. Parameter tuning by reducing the heartbeat rate, increasing the transfer scheduling rate, and checking more often the status of the decommission could decrease the value of t_0. However, the scheduler should be redesigned to improve the bandwidth utilization that becomes important for large amounts of data transferred. Indeed, the current transfer scheduler of HDFS tries to balance the transfers on the sender side, ignoring the receivers; but, as the model shows, the bottleneck is the receiving side. Thus, load-balancing should be done considering primarily the receivers. All the above can serve for the design of future optimized transfer schedulers in HDFS.

Different fits for different platforms

Figure 7.6, presents the parameters obtained by regression for different setups (10, 20, 40, and 60 nodes on the *paravance* cluster) and another platform (10 nodes on the *parapluie* cluster, storage in RAM but only 1 Gb/s network). We observe that the network utilization stays roughly

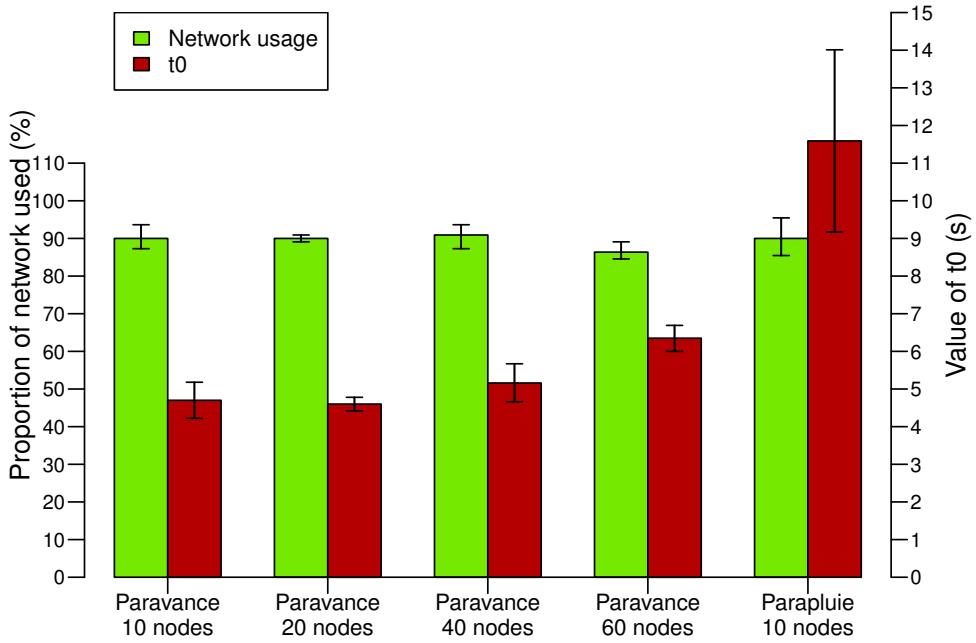


Figure 7.6: Value of t_0 and proportion of network utilization obtained by regression for multiple setups with a network bottleneck.

at 90% of the maximum bandwidth thanks to a good configuration of HDFS. On the other hand, t_0 changes, not because of the larger amount of work to schedule transfers, but mainly because of the impact of scheduling mistakes made by the data transfer scheduler of HDFS.

> *Overall, the decommission mechanism of HDFS is efficient when the network is the bottleneck: its performance is close to the theoretical minimum. The model closely explains the duration of the decommission in HDFS, and can be used to identify possible improvements to the decommission mechanism.*

7.1.4 Decommission in HDFS: when the bottleneck is at storage level

To create a setup where the bottleneck is at storage level, we configured HDFS to store data on the drive (read speed: 180 MiB/s, write speed: 160 MiB/s), significantly slower than the network (1.1 GiB/s).

Closeness of HDFS to the theoretical minimum

Figure 7.7 shows the decommission times observed and the minimal theoretical time. As we can see, even if the measures follow the same trends as the model, HDFS is about 3 times slower than what could theoretically be achieved on the platform.

In Figure 7.7 we present the measurements with the same configuration (number of nodes and data hosts per node) as the ones presented in Section 7.1.3 for better comparison, even if the technical constraint would allow larger experiments. In particular, when comparing with

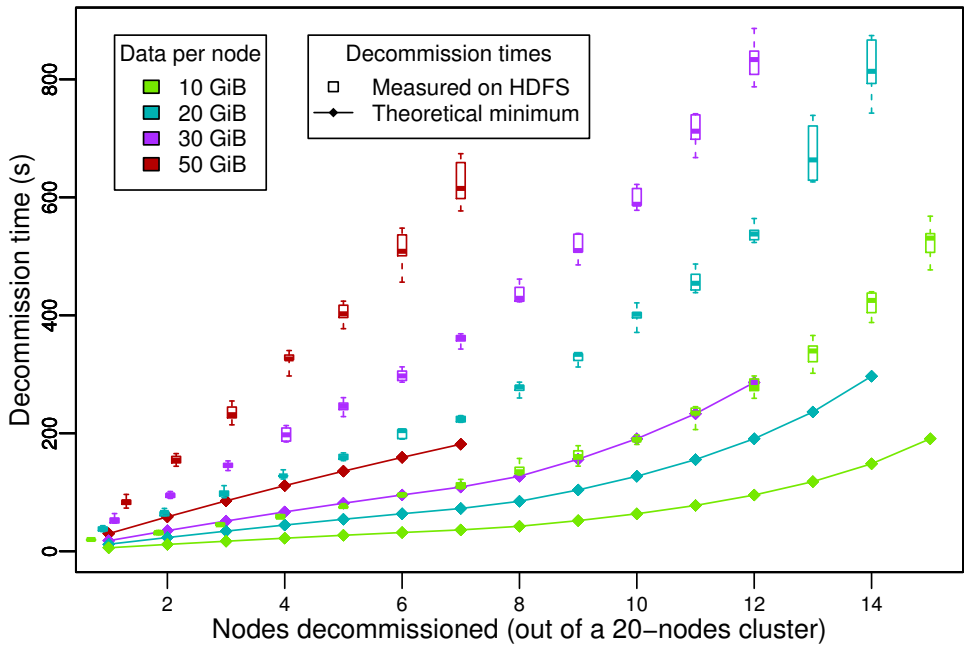


Figure 7.7: Decommission time measured on the platform presented in Section 7.1.4. The minimum theoretical time obtained with the model is added.

Figure 7.1, we observe that the decommission times are up to 20 times slower when using the drive. However, the drive should be only 13 times slower than the network in the worst case (reading and writing at the same time). This confirms that the decommission in this configuration is significantly less efficient than the one presented in Section 7.1.3.

Fitting the model to HDFS

Since the pattern of the measures follows that of the model, we fit the model to HDFS using a regression. The decommission of HDFS matches a model obtained on a platform with a reading speed of 50.7 MiB/s, a writing speed of 55.1 MiB/s, and an initialization time t_0 of -3.55 seconds, with a coefficient of determination of 0.983 as shown in Figure 7.8. The negative initialization time is due to increasing interference in the data transfers when more nodes are decommissioned simultaneously. The low bandwidth of drive accesses is due to the fact that HDFS reads and writes data from the drives in blocks of only 4 KiB.

Potential for improvement of the decommission time in HDFS

HDFS schedules data transfers by balancing the reads and send operations, but the bottlenecks are receive and write operations. Figure 7.9 shows that all nodes read data at approximately the same speed, resulting in high competition for the drive accesses on remaining nodes that must read and write data. In contrast, the disk of leaving nodes are underloaded since they do not write.

A scheduling strategy leveraging the model is simple: the scheduler should balance the

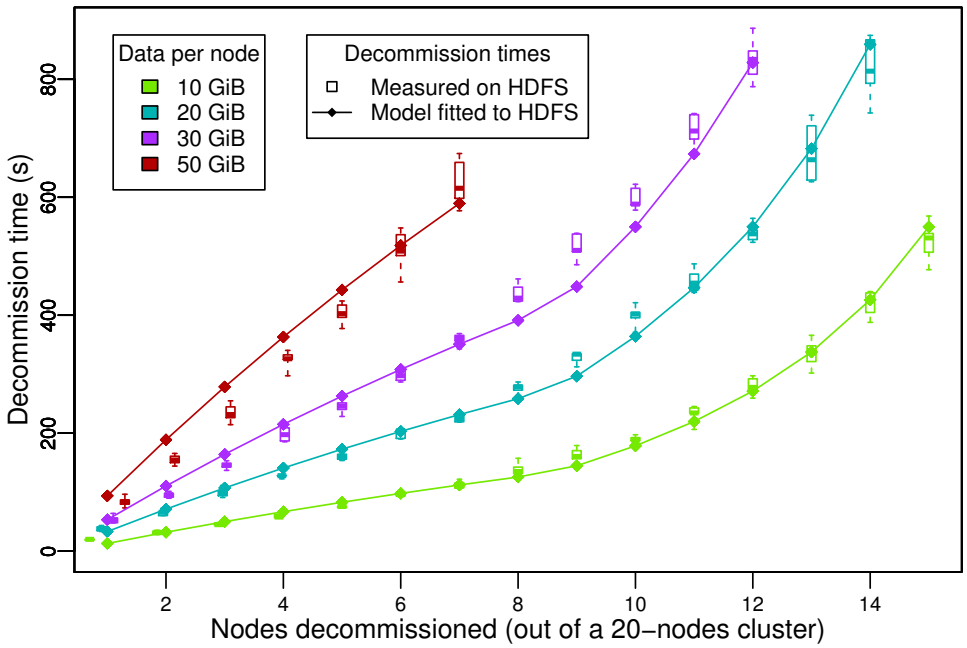


Figure 7.8: Model fitted to HDFS on the platform presented in Section 7.1.4. The model is enough to explain the performance of HDFS and has a coefficient of determination r^2 of 0.983.

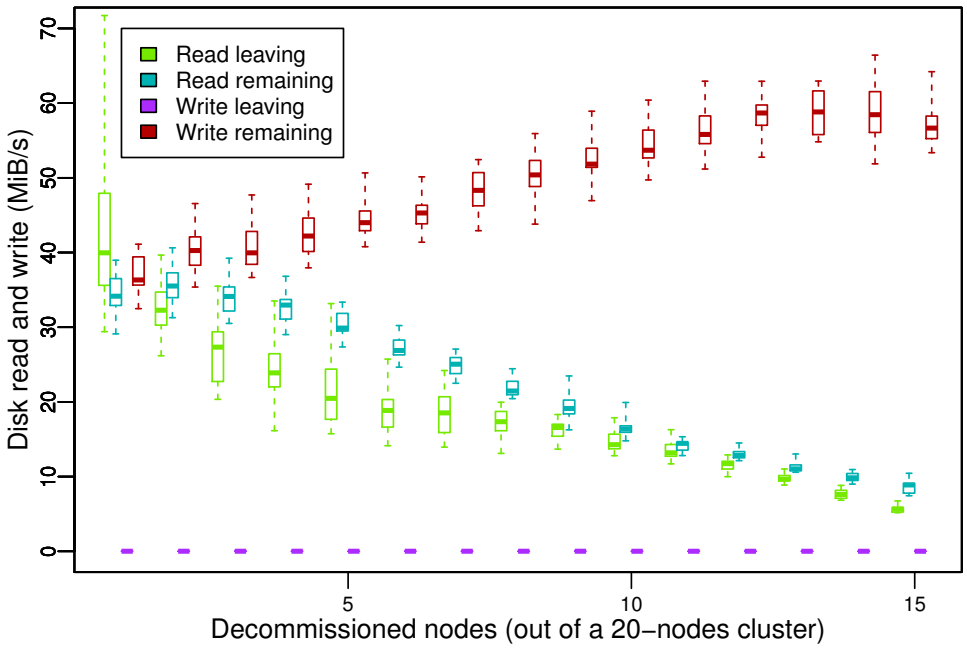


Figure 7.9: Average disk usage measured for leaving and remaining nodes on the platform presented in Section 7.1.4. Each node hosts 40 GiB of data.

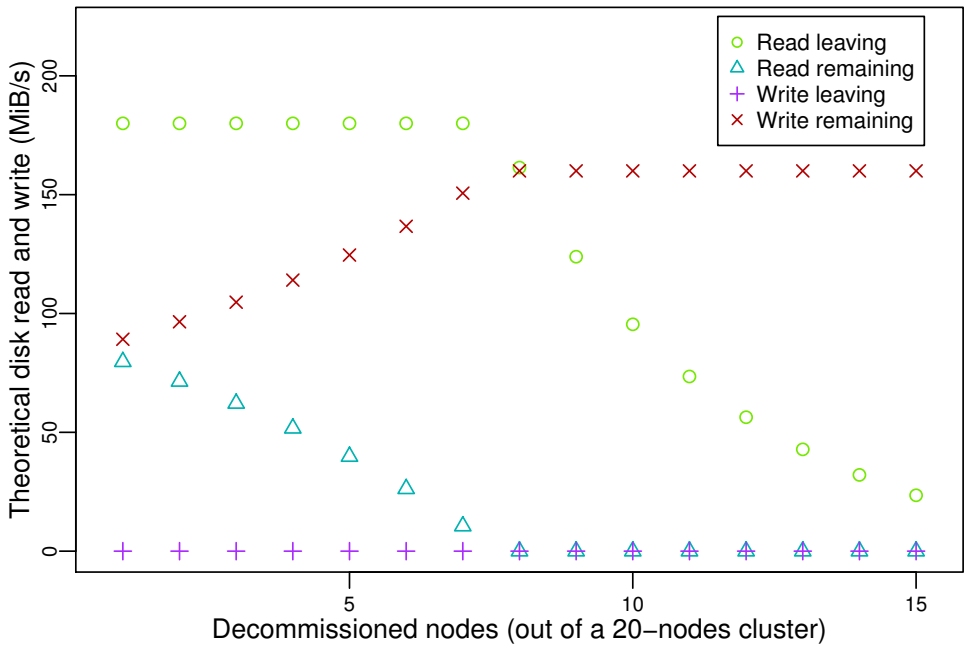


Figure 7.10: Theoretical disk utilization for leaving and remaining nodes obtained with the model on the platform presented in Section 7.1.4.

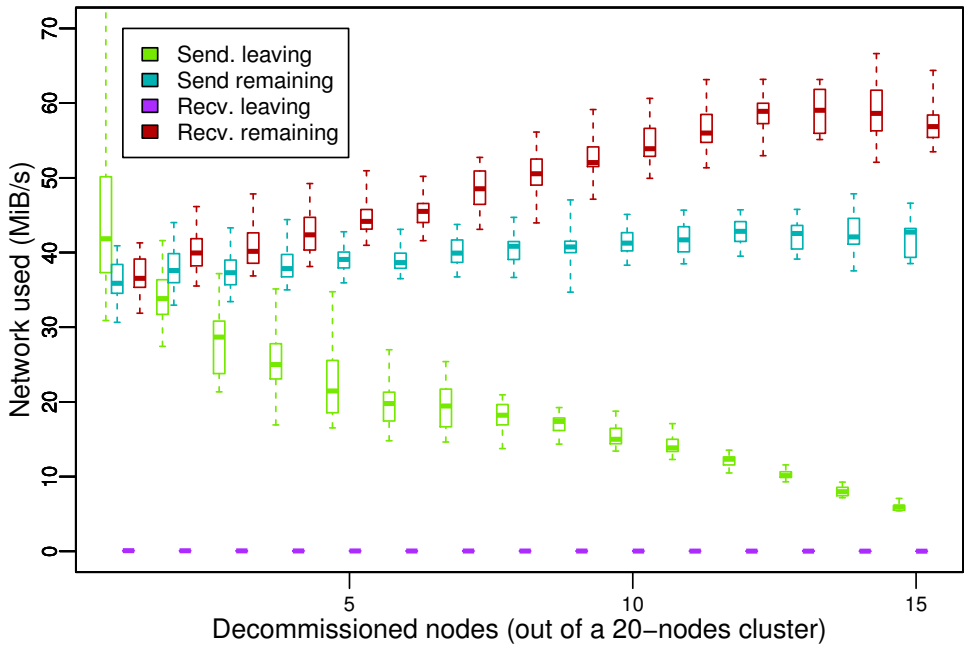


Figure 7.11: Average network usage measured for leaving and remaining nodes on the platform presented in Section 7.1.4. Each node hosts 40 GiB of data.

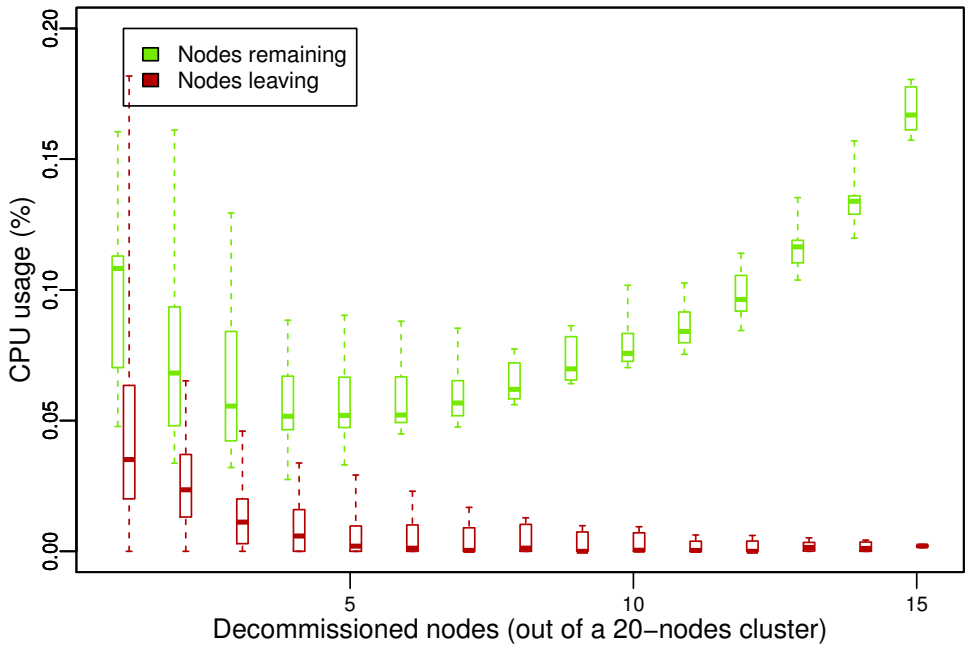


Figure 7.12: Average CPU usage measured for leaving and remaining nodes on the platform presented in Section 7.1.4. Each node hosts 40 GiB of data.

write operations, prioritize them, then maximize reading from leaving nodes. This would lead to read and write patterns like those presented in Figure 7.10. If remaining nodes can write all that is read by leaving nodes, then they also read to accelerate the decommission. If they cannot, the leaving nodes have their reading speed reduced while remaining nodes simply stop reading.

Cost of the decommission

Figure 7.11 shows the network usage during the decommission. We note that the amount of data sent on the network is higher than the amount of data read from the drive: HDFS pipelines the writing of replicas and avoids useless read operations. The average CPU utilization is low (see Figure 7.12): the CPU is used only for metadata operations, which are rare because of the reading and writing speeds of the storage.

> *When the bottleneck is at storage level, the decommission mechanism in HDFS suffers from inappropriate scheduling: the scheduler balances the read load instead of the write load, and disk accesses are inefficient due to the resource contention and access patterns. Solving these problems could make decommission in HDFS up to 3 times faster.*

7.2 Commission in HDFS

In this section we evaluate the performance of the commission in HDFS against the theoretical minimum established in Chapter 5.

7.2.1 Experimental setup

The setup used to evaluate the commission of HDFS is the same as presented in Section 7.1.1 except that we used the *grisou* cluster of Grid'5000 located in Nancy. This cluster has the same hardware as the one described in Section 7.1.1.

7.2.2 Experimental protocol

To measure the commission time of HDFS, we first deployed HDFS on 10 nodes. A subset of the unused nodes in the cluster (with 2 to 30 nodes) was then randomly selected and added to HDFS. HDFS does not rebalance the data by itself, however, thus we used the internal rebalancer to balance the data between new and old nodes. The recorded time is the time taken by the rebalancer to balance the data between old and new nodes, since adding nodes takes hardly any time compared with the time needed to balance the data among the nodes.

For all experiments with in-memory storage, measurements were repeated 10 times (these experiments lasted for 39 hours). Because of the duration of the experiments, however, measurements for disk drive storage were repeated 5 times (the experiments lasted for 84 hours).

7.2.3 Rebalancing algorithm used in HDFS

Algorithm 7.1 is used by HDFS to rebalance the data in a cluster. As done for the decommission, some parameters of this algorithm were adjusted to improve the commission time. The delay between two iterations was reduced from 9 seconds to 1 second. Moreover, HDFS checks whether the wave of transfers is finished only every 30 seconds; this delay has also been reduced to 1 second. The rebalancer limits both the throughput of each node used for rebalancing data and the number of concurrent data transfers. Both limits have been removed. The threshold of the rebalancing done by HDFS is set to 2%, which means that the rebalancing will stop if all nodes are within a 2% margin of their ideal amount of data.

7.2.4 Commission in HDFS

Figures 7.13 and 7.14 show the time needed by HDFS to commission nodes to a cluster of 10 nodes with various amounts of data initially on the nodes.

Figure 7.13 presents the duration of the commission when the network is the bottleneck, while Figure 7.14 shows the duration of the commission when the storage (drives) is the bottleneck. Both figures show the same pattern: the theoretical minimum and the observed results are opposite of each other. The model suggests that the time needed to commission nodes should decrease as the number of added nodes increases, but the commission times of HDFS increases greatly as the number of new nodes grows.

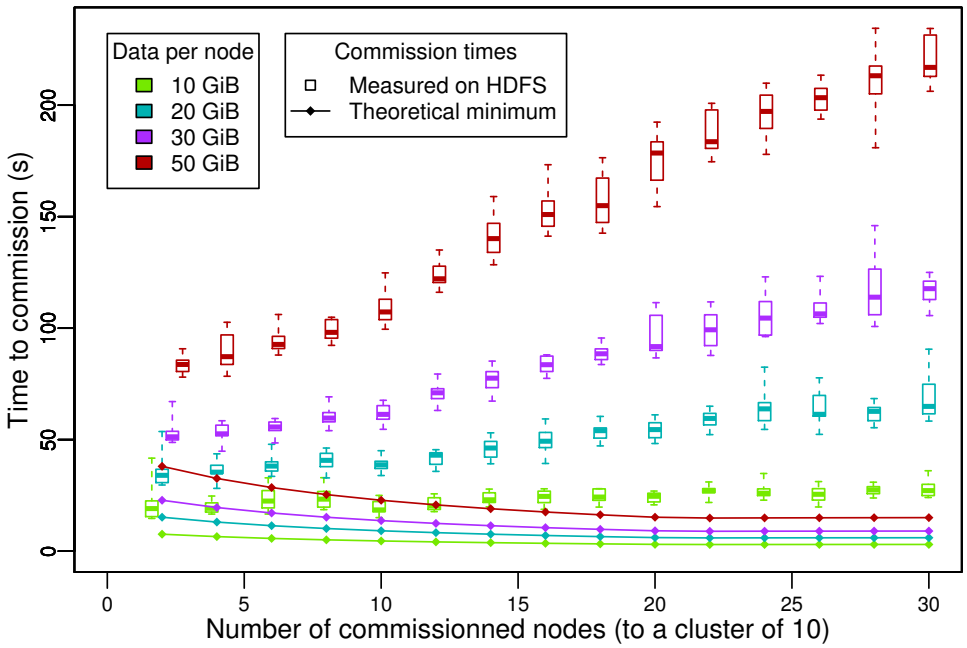


Figure 7.13: Commission time measured when there is a network bottleneck. The minimum theoretical time obtained with the model is added.

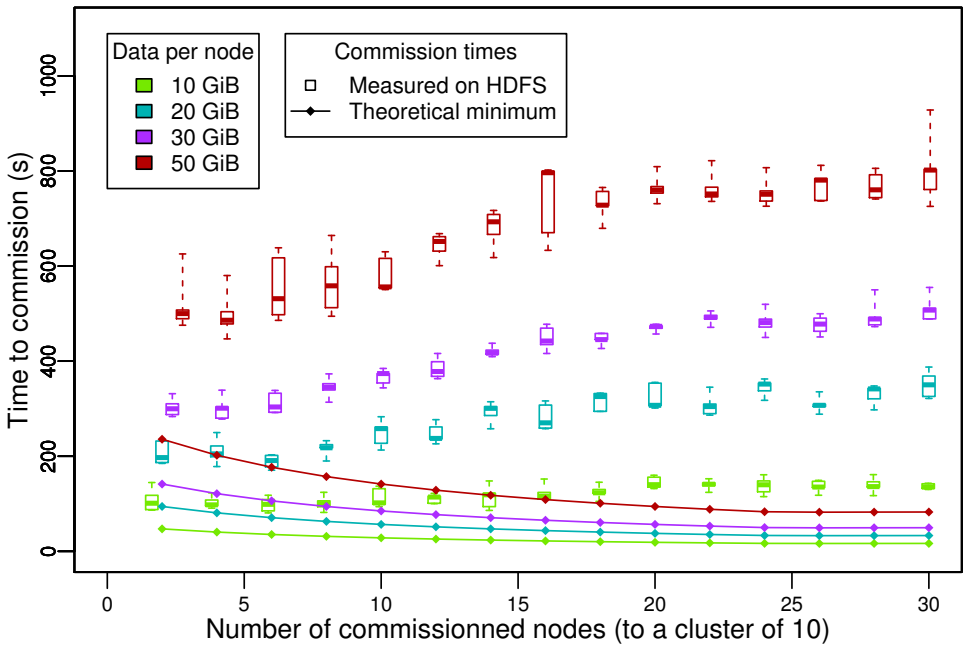


Figure 7.14: Commission time measured when there is a storage bottleneck. The minimum theoretical time obtained with the model is added.

Input: $threshold$: maximum difference between the ideal storage utilization on each node and the final one, provided by the user.

repeat
- Compute the $average$ storage utilization per available node on the cluster.
- Cluster nodes according to their storage utilization (u):
  - **if** $u > average + threshold$ **then** the node is Over-Utilized
  - **else if** $u > average$ **then** the node is Above-Average
  - **else if** $u > average - threshold$ **then** the node is Below-Average
  - **else if** $u \leq average - threshold$ **then** the node is Under-Utilized
- Pair the nodes (source and target) with the following priority:
  - Over-Utilized and Under-Utilized,
  - Over-Utilized and Below-Average,
  - Under-Utilized and Above-Average.
- Select data to move from the source to the target:
  - Target must not already host the same object.
  - Data must not already be scheduled to move.
- Execute the data transfers
  - no more than $threshold * cluster\_capacity$ amount of data is moved during each iteration.
  - Replicas can be sent from the source or from another node hosting the replica.
- Wait for all transfers to finish.

until *All nodes are Above-Average or Below-Average*

Algorithm 7.1: Algorithm used by HDFS to rebalance the data among the nodes, in the case of a single-rack configuration.

> *These are not surprising results: the rebalancing algorithm of HDFS is not optimized to be as fast as possible but to limit the impact on the performance of HDFS. This difference means that the model of the fastest commission cannot be used to represent the commission of HDFS.*

7.2.5 Hints to improve the commission mechanism

The model highlighted two important bottlenecks: the reception (or writing) of the data and the old nodes sending (reading) data. Hence, in order to improve the commission in HDFS (or any distributed storage system using replication), the old nodes should send as little data as possible, while the reception of data on the new nodes should be balanced.

7.3 Discussion

In this section, we discuss some aspects and usages of the models in light of the study of the rescaling operations of HDFS.

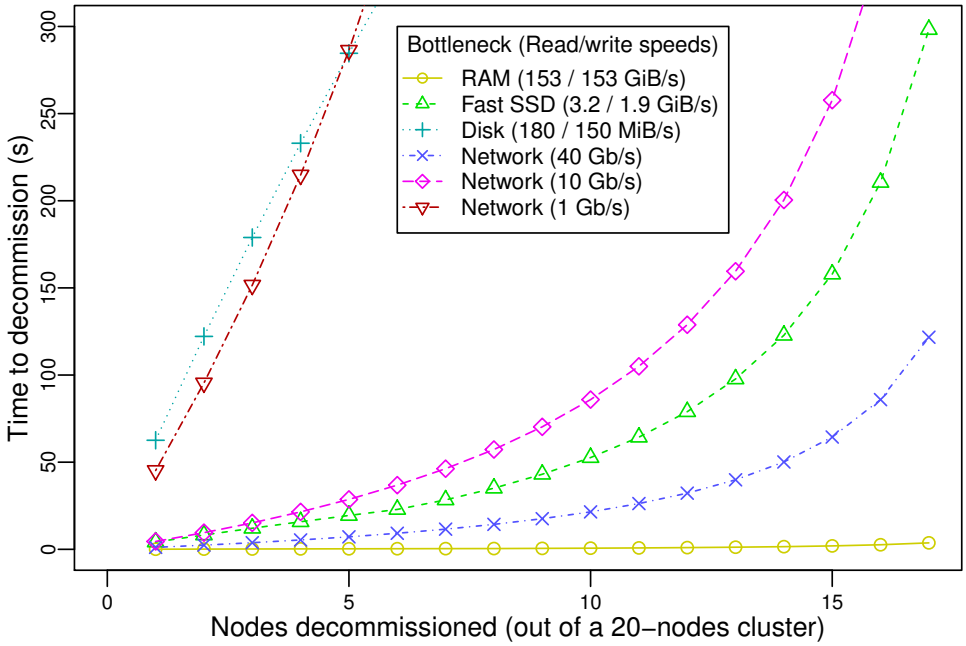


Figure 7.15: Minimal decommission time (for 100 GiB per node) for different existing technologies on a 20-node cluster.

7.3.1 Difference between the commission model and the observations

The observed times to commission nodes in HDFS greatly differ from the theoretical minimum obtained with the model, thus raising a natural question: Isn't this theoretical minimal duration too optimistic? To answer this, we present a benchmark for rescaling operations, Pufferbench, in Part III. With it, we obtain commission duration close to the theoretical minimum duration.

7.3.2 Dependence on HDFS

The models are generic and do not rely on HDFS. Thus, the hints given to improve the transfer scheduler of HDFS can also be used to improve the duration of both operations in any distributed storage system using data replication.

7.3.3 Prediction of the duration of rescaling operations for various technologies

Since the models are generic, one can use them to predict the commission and decommission times that could be reached when other storage technologies, existing or emerging, are used. As an example, Figure 7.15 illustrates expected decommission times for various settings: storage bottleneck with RAM (from the Cray XC series [127]), drive (see Section 7.1.1), and one of the fastest SSDs [128] and network bottlenecks with different bandwidths. In Figure 7.16, the minimum commission time for the same hardware is presented.

From these figures we can see that the commission and decommission times decrease with newer technologies, strengthening the idea that malleable storage systems can currently be useful as the cost of the malleability is decreasing.

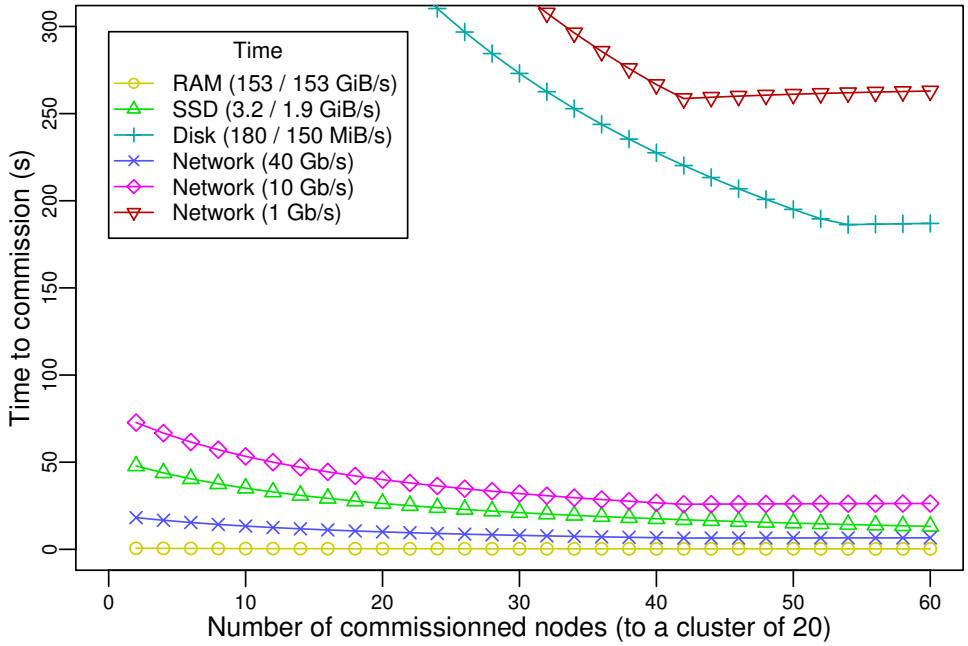


Figure 7.16: Minimal commission time (for 100 GiB per node) for different existing technologies on a 20-node cluster.

7.3.4 Impact of the generality of the assumptions on the model

The mismatch between the model of the commission and the performance of the commission of HDFS highlights an important point. The established model can only represent the performance of rescaling mechanisms that have been optimized with speed in mind. There are, however, rescaling operations of storage systems that can be optimized for other relevant properties, such as guaranteeing a maximal latency for the requests or ensuring a minimal throughput for them.

Conclusion

In this chapter, we used the models established in Chapters 5 and 6 to evaluate the performances of the rescaling operations of HDFS, one well-known distributed storage systems.
The evaluation of the decommission implementation of HDFS showed that, in the case of a network bottleneck, the rescaling was efficient, and followed the model closely. In the case of a storage bottleneck, the operation could be sped up by about 3 times with a better balancing of data reading tasks and improvements to the access patterns on the drive.
On the contrary, the evaluation of the commission showed that HDFS's mechanism is not optimized for fast operations and could even be sped up by up to a factor 14.

CHAPTER 8

RELAXING FAULT TOLERANCE FOR FASTER DECOMMISSIONS

In Chapter 6, we established a model of the minimal duration of the decommission under the self-imposed constraint of maintaining the replication factor of the data stored in the distributed storage system (Objective 2). This constraint is followed to ensure one of the basic guarantees of most distributed storage systems: fault tolerance.

However, fault tolerance can be relaxed during the decommission. The replication factor of the data stored can be lowered temporarily to return the leaving nodes to the resource manager faster, thus trading off safety for faster decommissions. We call this strategy *fast decommission*.

The idea of trading fault tolerance for performance is not new, and is, in fact, rather intuitive. The main contribution of this chapter is to show that *this intuition is incorrect in many situations*. Our goal is to make a step forward in better understanding the actual trade-off that exists between the duration of the decommission and its impact on fault tolerance. To this end, we provide a model of the minimal duration for the main phases of the fast decommission.

8.1 The fault tolerance assumption

In the models presented in Chapter 6, we assumed that the level of fault tolerance of the storage system should not be weakened during the decommission (Objective 2); if the system is configured to keep r replicas of an object at all times, the number of replicas of that object during the decommission should never be strictly less than r. It also means that the decommissioned nodes can be given back to the resource manager only at the end of the decommission, since all objects need to be sufficiently replicated on the remaining nodes.

This is an opportunity for optimization. As long as no data is lost, decommissioned nodes can be returned to the resource manager sooner. We call this strategy *fast decommission*. It is composed of three phases. The first one is the *data-safekeeping* phase, where the system ensures that at least one replica of each object is present on the remaining nodes, transferring objects if needed. The second one is the *node release* phase, the decommissioned nodes are given back to the resource manager. Missing replicas are recreated during the *system stabilization* phase.

With this strategy, the decommissioned nodes are effectively made available for other jobs faster than they are with a standard decommission. However, fast decommission comes at the cost of weakened fault tolerance during the system stabilization phase: not all objects have their required number of replicas until the stabilization finishes.

To build a model of the duration of this operation, we use the same assumptions and objectives as described in Chapter 4, except for the objective of minimum replication (Objective 2).

We replace this objective with Objective 5, hereafter, which allows the replication level of data to temporarily decrease during the decommission as long as it is restored to its initial level at the end of the operation.

Objective 5: Maintenance of the replication factor
At the end of the rescaling operation, each object stored on the storage system is replicated on r distinct nodes.

8.2 Problem definition

We consider a replication-based distributed storage system deployed on a cluster of N nodes. Each node initially hosts an amount of data D. Each of the objects stored in the system is replicated r times. The resource manager requests the decommission of x arbitrarily chosen nodes.

A fast decommission is done in three main steps.

1. **Data-safekeeping:** During the data-safekeeping phase, the objects that are stored only on the leaving nodes have a replica transferred to remaining nodes to ensure that no data is lost during the operation.
2. **Nodes release:** The leaving nodes are given back to the resource manager. They no longer participate in the distributed storage.
3. **System stabilization:** The missing replicas are recreated by the remaining nodes to recreate the target replication level.

We define the time to availability t_{avail} as the minimum duration of the data-safekeeping phase. The stabilization time t_{stab} is the minimum duration of the whole process, assuming that the leaving nodes participated only in the data-safekeeping phase and were all removed from the cluster at time t_{avail}.

8.3 Identifying data movements

Because data should not be lost during a decommission (Objective 1), a minimum amount of data has to be moved from the leaving nodes to the remaining ones. The objects to move are the ones that have all their replicas on the leaving nodes and that would have been lost, had these nodes all been removed at the same time. Thus, we first compute the probability, p_i, for an object to have exactly i replicas on the leaving nodes. From it, we deduce the minimum amount of data to transfer to remaining nodes D_{avail}.

$$p_i = \begin{cases} 0 & \text{if } i > r, \\ \dfrac{\binom{r}{i}\binom{N-r}{x-i}}{\binom{N}{x}} & \text{for } i \leq r. \end{cases} \qquad \textbf{(Definition 8.1)}$$

$$D_{avail} = \begin{cases} NDp_r/r & \text{if } x \geq r \\ 0 & \text{in other cases.} \end{cases} \quad \textbf{(Definition 8.2)}$$

D_{stab} is the amount of data to move in order to recreate all replicas from the leaving nodes onto the remaining nodes. It is the amount of data that was initially present on the leaving nodes and includes D_{avail}.

$$D_{stab} = xD \quad \textbf{(Definition 8.3)}$$

Both D_{avail} and D_{stab} are obtained assuming that objects stored on the nodes can be divided as needed to perfectly balance the data on each node.

8.4 When the network is the bottleneck

In this section we assume that the network is the bottleneck for the data transfers required by the data-safekeeping and stabilization phases. The network is the bottleneck if it limits the storage in any situation ($S_{Net} < S_{Read}$).

8.4.1 Time to availability

During the data-safekeeping phase, only the leaving nodes send data to the remaining ones. As defined by Assumption 2, the network is ideal without interference, and each node can send and receive data with a bandwidth S_{Net} at the same time. Two possible bottlenecks may appear, however: either when sending data from the leaving nodes or when receiving the data on the remaining nodes.

Thus, the time to availability t_{avail} depends on the number of nodes leaving the cluster x, the amount of data to move D_{avail}, and the bandwidth of the network S_{Net}. We express t_{avail} as follows.

$$t_{avail} = \begin{cases} \dfrac{NDp_r}{rxS_{Net}} & \text{if } x \leq N/2 \\ \dfrac{NDp_r}{r(N-x)S_{Net}} & \text{otherwise.} \end{cases} \quad \textbf{(Prop. 8.1)}$$

8.4.2 Stabilization time

The safekeeping phase has the priority over the stabilization phase; decommissioned nodes have to be released as fast as possible. However, depending on whether the bottleneck of the safekeeping phase is receiving or sending data, the stabilization can happen at the same time as the safekeeping without slowing down the latter. Indeed, when the leaving nodes sending data are the bottleneck of the data-safekeeping phase, the remaining nodes do not have their network bandwidth saturated by the reception of the data. Thus, data exchanges needed to stabilize the storage can start before the end of the safekeeping phase without slowing it down.

We denote as t_{over} the time gained on the duration of the stabilization phase by starting it before the end of the safekeeping phase.

$$t_{over} = \begin{cases} \dfrac{(N-2x)NDp_r}{rx(N-x)S_{Net}} & \text{if } x \leq N/2 \\ 0 & \text{otherwise.} \end{cases} \qquad \textbf{(Prop. 8.2)}$$

From this, we obtain the time needed to stabilize the distributed storage system t_{stab}. The stabilization phase can use all available resources only after the end of the safekeeping phase. However, some overlap between the two phases reduces the duration of the stabilization by t_{over}. Thus, t_{stab} is defined by Property 8.3.

$$\begin{aligned} t_{stab} &= t_{avail} - t_{over} + \frac{D_{stab} - D_{avail}}{S_{Recv}} \\ &= \frac{xD}{(N-x)S_{Net}}. \end{aligned} \qquad \textbf{(Prop. 8.3)}$$

8.4.3 Observations

The model of the duration of the whole operation (t_{stab}) is exactly the same as the model of the duration of the standard decommission established in Chapter 6 (in which the replication factor is maintained). Thus, one can relax the fault tolerance to release nodes faster (the fewer the node-hours used, the better the overall platform utilization) without any difference in the length of the operation compared with a standard decommission.

We also infer that keeping the leaving nodes after they have transferred the data needed for the fast decommission does not speed up the duration of the operation: in all cases, receiving data on the remaining nodes is the bottleneck. It would, however, have an impact on the ability of the cluster to service read requests. The network is completely saturated by the stabilization, servicing any request would slow it down.

In Figure 8.1, we observe the differences between a standard decommission and a fast decommission; with the fast decommission, the nodes are released in a fraction of the time needed to decommission nodes while maintaining the replication factor.

> *When the network is the bottleneck of the decommission, using a fast decommission instead of a standard one is a simple trade-off between fault tolerance and the time needed to release the decommissioned nodes. In both cases, the system needs the same time before the data is properly replicated on the remaining nodes. Using the fast decommission also decreases the consumption of node-hours.*

8.5 When storage devices are the bottleneck

When the storage devices are the bottleneck, the situation is different because of the limitations of the storage devices (Assumption 3): data cannot be read and written at the same time.

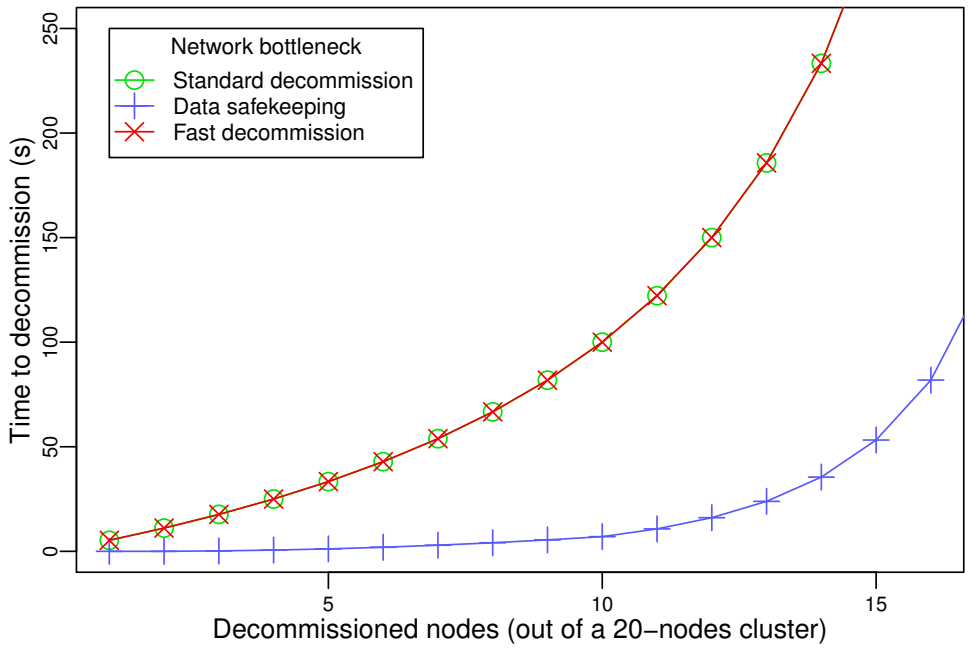


Figure 8.1: Theoretical minimum duration of the data-safekeeping phase and fast decommission compared with the minimum duration of the standard decommission, in case of a network bottleneck. $D = 100$ GiB, $S_{Net} = 1$ GiB/s.

8.5.1 Time to availability

The limitations on the storage, however, do not have any impact on the time to availability since leaving nodes only have to read data, and remaining nodes only have to write it. Thus, the time to availability depends on the data to move during the data-safekeeping phase D_{avail} and the reading and writing speeds of the storage devices S_{Read} and S_{Write}.

$$t_{avail} = \begin{cases} \dfrac{NDp_r}{rxS_{Read}} & \text{if } x \leq \dfrac{NS_{Write}}{S_{Read} + S_{Write}} \\ \dfrac{NDp_r}{r(N-x)S_{Write}} & \text{otherwise.} \end{cases} \qquad \textbf{(Prop. 8.4)}$$

8.5.2 Stabilization time

Similarly to the first case, when the bottleneck of the operation is reading data from the leaving nodes, the storage of the remaining nodes is not saturated: these nodes can read or write more data without slowing down the data-safekeeping process. Thus, the remaining nodes can exchange data to start the stabilization before the data-safekeeping finishes and without impact on the time to availability.

Each remaining node has some time t_{over} to exchange data with other remaining nodes in the data-safekeeping phase.

$$t_{over} = \begin{cases} \dfrac{(N-x)S_{Write} - xS_{Read}}{x(N-x)S_{Read}S_{Write}} & \text{if } x \leq \dfrac{NS_{Write}}{S_{Read}+S_{Write}} \\ 0 & \text{otherwise.} \end{cases} \quad \textbf{(Prop. 8.5)}$$

We determine S_{eff}, the effective writing speed on the cluster when the remaining nodes exchange data among themselves. S_{eff} is not simply the product of the number of remaining nodes by their individual writing speed. Indeed, to exchange data among themselves, remaining nodes must also read data.

To avoid reading multiple times data from storage devices with low read bandwidth, many systems use buffering. The data read is stored in memory (that has a higher bandwidth) and then sent to as many destinations as needed. The buffering relies on the bandwidth of the memory being a few times higher than the bandwidth of the storage device. We denote as R the ratio of data read to data written on the storage device during the stabilization.

$$R = \begin{cases} 1 & \text{in case of in-memory storage,} \\ \dfrac{\sum_{i=1}^{r-1} p_i}{(r-1)p_r + \sum_{i=1}^{r-1} ip_i} & \text{otherwise.} \end{cases} \quad \textbf{(Prop. 8.6)}$$

With the ratio R we deduce S_{eff}. Storage devices have their operation time divided between reads and writes (they cannot read and write at the same time). The cluster must also avoid imbalances between the data read and written. If too much data is read compared with the data written, the amount of memory needed to store it before writing it will increase. On the contrary, if too little data is read, the storage devices will not be used at their maximum capacity and thus the decommission will slow down. Thus, the ratio of data read to data written during any given duration should be equal to R. From this we deduce S_{eff}.

$$S_{eff} = \frac{(N-x)S_{Write}S_{Read}}{S_{Read} + RS_{Write}} \quad \textbf{(Prop. 8.7)}$$

From the speed at which data is effectively exchanged on the cluster during the stabilization (Property 8.7), the amount of data to write (Definition 8.2 and 8.3), the duration of the overlap of data-safekeeping and stabilization (Property 8.5), and the time to availability (Property 8.4), we deduce the stabilization time t_{stab}.

$$t_{stab} = \frac{D}{N-x}\left(\frac{R}{S_{Read}} + \frac{1}{S_{Write}}\right)\left(x - \frac{Np_r}{r}\right) + \frac{NDp_r}{r(N-x)Sw} \quad \textbf{(Prop. 8.8)}$$

8.5.3 Observations

In the case of a storage bottleneck, the data-safekeeping phase and thus the effective decommission of the leaving nodes can be completed significantly faster than with the standard decommission. It is done, however, at the cost of a long stabilization phase: the leaving nodes were reading data in the case of a standard decommission, a task that must be done by remaining nodes in the case of a fast decommission. This situation implies that, contrary to the case of a network bottleneck, the longer the leaving nodes stay in the cluster, the faster the stabilization is. The stabilization cannot be faster than the standard decommission since the

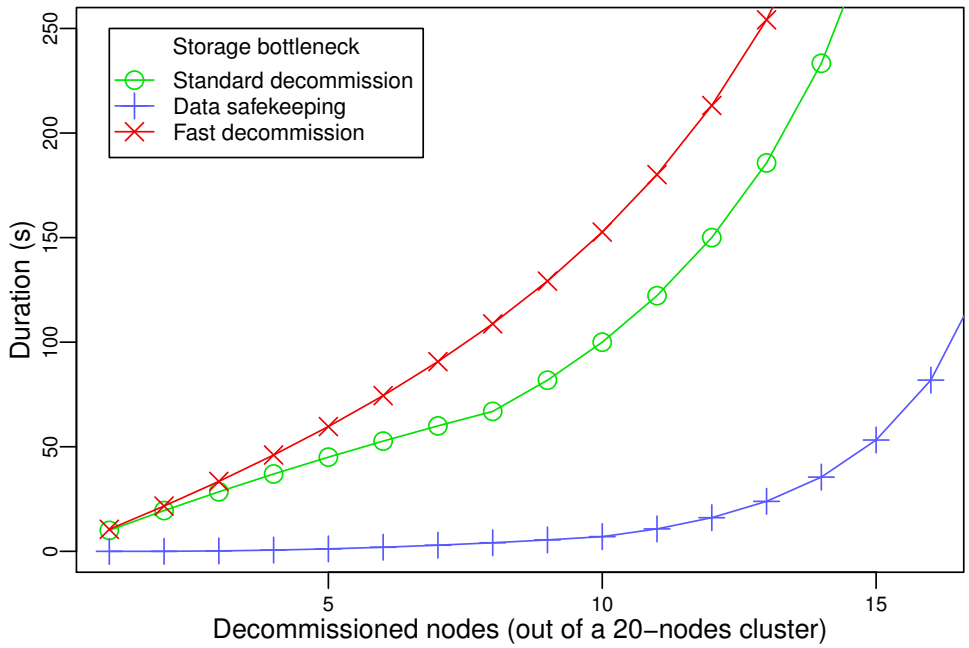


Figure 8.2: Theoretical minimum duration of the data-safekeeping phase and fast decommission compared with the minimum duration of the standard decommission, in case of a storage bottleneck. $D = 100$ GiB, $S_{Read} = S_{Write} = 1$ GiB/s.

standard decommission is the extreme case in which the leaving nodes stay until the end of the stabilization.

During a fast decommission, the storage devices are fully saturated. Thus, servicing any request can only slow down the decommission.

In Figure 8.2, we show the minimal duration of a standard decommission and of the data-safekeeping and stabilization phases of a fast decommission. Decommissioned nodes are available in a fraction of the time needed for a standard decommission. However, it comes at the cost of having the storage devices saturated for a longer time because of the recreation of the missing replicas.

8.6 Node-hour usage

We study the node-hour usage of the strategies since it is linked to the financial and energetic cost of the active nodes during the operation. Comparing the consumption of node-hour equates to comparing the cost of the strategies.

In Figure 8.3 we compare the usage of node-hour for the standard decommission and the fast decommission in the case of a network bottleneck. Since the numbers are based on the minimum duration of the operations, the figure represents the minimal node-hour consumption. We observe that using the fast decommission mechanism always reduces the node-hours consumption when the network is the bottleneck. Moreover, the gain increases greatly with the

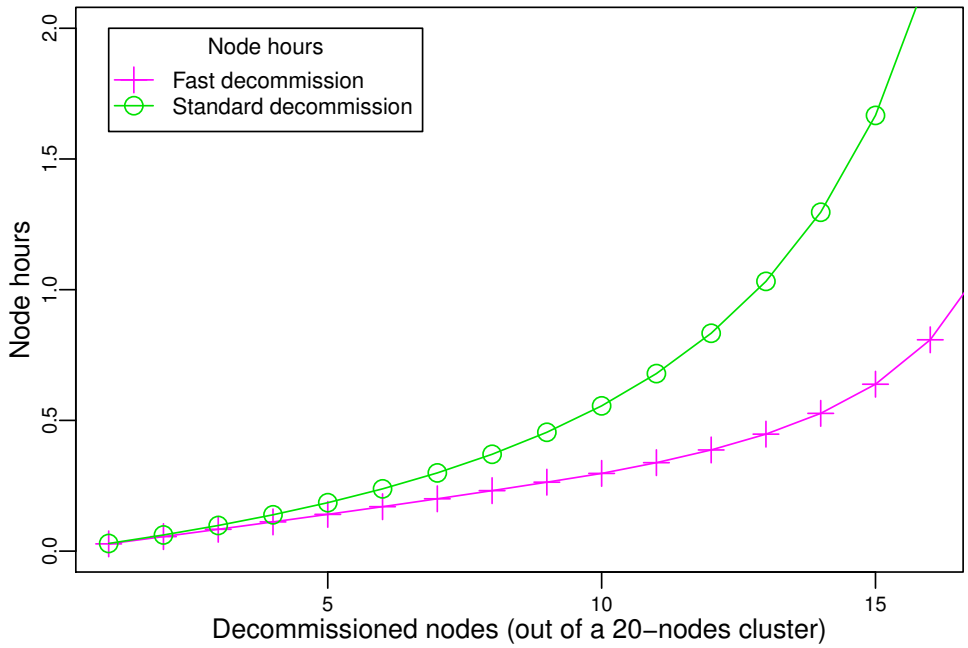


Figure 8.3: Minimum number of node-hours used during a fast decommission compared with the standard decommission in the case of a network bottleneck. $D = 100$ GiB, $S_{Net} = 1$ GiB/s.

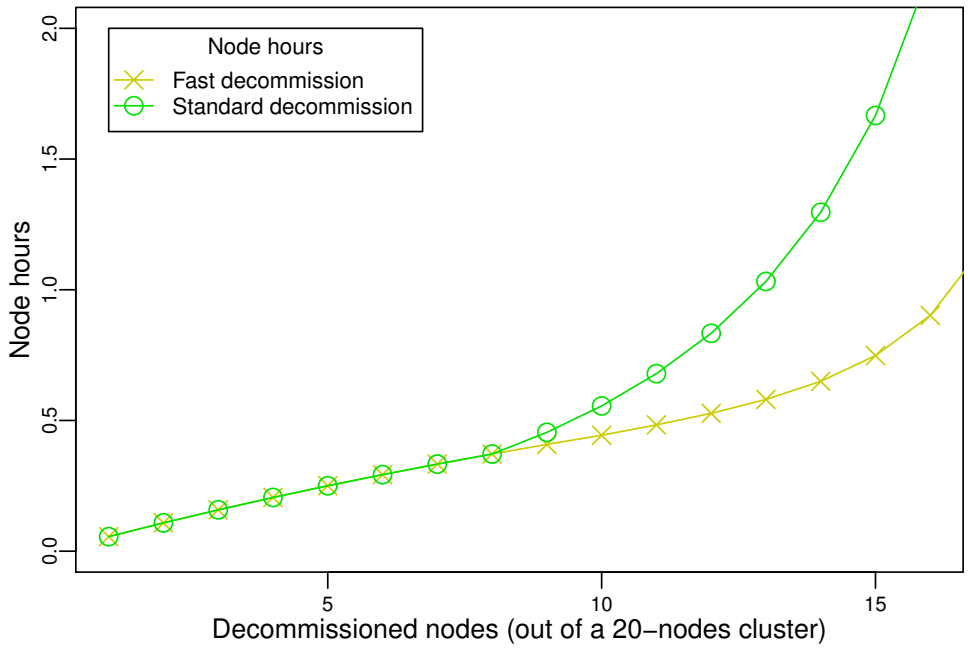


Figure 8.4: Minimum number of node-hours used during a fast decommission compared with the standard decommission in the case of a storage bottleneck. $D = 100$ GiB, $S_{Read} = S_{Write} = 1$ GiB/s.

number of decommissioned nodes, and more than 50% of the node-hour consumption can be saved when many nodes are decommissioned at once.

In Figure 8.4 we compare the node-hours needed for the decommission in the case of a storage bottleneck. When few nodes (less than $NS_{Write}/(S_{Read} + S_{Write})$, that is less than 8 in this case) are decommissioned at once, there are no benefits in using the fast decommission compared with the standard decommission. When many nodes are decommissioned simultaneously, however, the node-hours consumption can be reduced by more than 50%.

> *In the case of a storage bottleneck, using a fast decommission releases the decommissioned nodes faster, however, the stabilization phase of the fast decommission lasts longer than the standard decommission, increasing the impact of the decommission on the storage system. Moreover, there are no gains in node-hours unless many nodes are decommissioned at the same time.*

8.7 Discussion

In this section, we discuss some aspects of the models of the duration of the fast decommission.

8.7.1 The models are lower bounds by design

To obtain a model of the minimal duration of the decommission, the model has been built as a lower bound of the duration of the operation. We show in Chapter 10 that the theoretical minimum duration of the decommission can be approached in practice.

8.7.2 Preserving $k > 1$ replicas

For the models presented in Sections 8.4 and 8.5, the fault tolerance is simply ignored during the decommission: only one replica of each object is required. However, one may want to be able to tolerate $0 < k - 1 < r$ faults during the decommission. In this case, at least $k > 1$ replicas of each object must be preserved on the remaining nodes before the leaving nodes are released.

Models for this situation can be defined. In the case of a network bottleneck (Property 8.9 and Figure 8.5), the time to stabilization is the same as the standard decommission which is also the time to stabilization when maintaining only one replica. For the time to availability, we notice that receiving the data is always the bottleneck, indeed, due to the uniform data distribution (Assumption 6), each and every node hosts some objects that are also stored by leaving nodes, and they can replicate them among themselves.

$$t_{avail} = \sum_{i=1}^{k} ip_{r-k+i} \frac{ND}{r(N-x)S_{Net}} \qquad \textbf{(Prop. 8.9)}$$

$$t_{stab} = \frac{xD}{(N-x)S_{Net}}$$

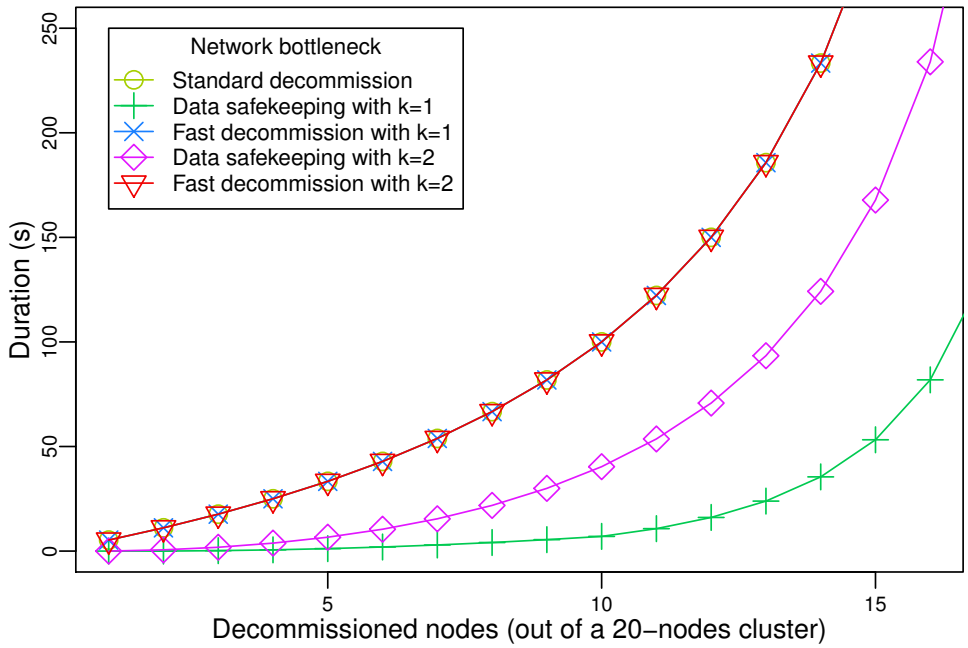


Figure 8.5: Theoretical minimal duration of the data-safekeeping phase and fast decommission compared with the duration of the standard decommission in case of a network bottleneck for $k = 1$ and $k = 2$. $D = 100$ GiB, $S_{Net} = 1$ GiB/s.

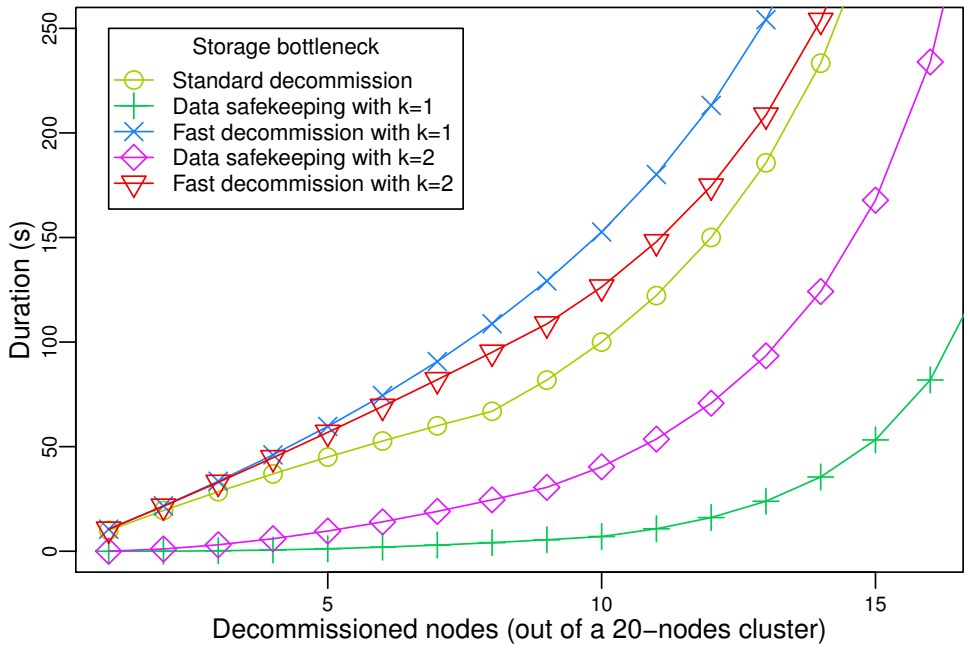


Figure 8.6: Theoretical minimal duration of the data-safekeeping phase and fast decommission compared with the duration of the standard decommission in case of a storage bottleneck for $k = 1$ and $k = 2$. $D = 100$ GiB, $S_{Read} = S_{Write} = 1$ GiB/s.

Let R_{avail} be the ratio of the amount of data to read on the data to write during the data-safekeeping phase.

$$R_{avail} = \begin{cases} 1 & \text{for in-memory storage} \\ \frac{\sum_{i=1}^{k} p_{r-k+i}}{\sum_{i=1}^{k} p_{r-k+i}} & \text{in other cases.} \end{cases}$$

$$t_{avail} = \begin{cases} \sum_{i=1}^{k} ip_{r-k+i} \frac{D}{r} \frac{S_{Read} + R_{avail} S_{Write}}{S_{Write} S_{Read}} & \text{if } x < \frac{R_{avail}(N-x) S_{Write}}{S_{Read}} \\ \sum_{i=1}^{k} ip_{r-k+i} \frac{ND}{r(N-x) S_{Write}} & \text{in other cases.} \end{cases}$$

(Prop. 8.10)

Let R_{stab} be the ratio of the amount of data to read on the amount of data to write during the stabilization phase.

$$R_{stab} = \begin{cases} 0 & \text{in case of storage in-memory} \\ \frac{\sum_{i=1}^{r-k} p_i}{\sum_{i=1}^{r} ip_i - \sum_{i=1}^{k} ip_{r-k+i}} & \text{in other cases.} \end{cases}$$

$$t_{stab} = t_{avail} + \left(\sum_{i=1}^{r} ip_i - \sum_{i=1}^{k} ip_{r-k+i} \right) \frac{ND}{r} \frac{R_{stab} S_{Write} + S_{Read}}{(N-x) S_{Read} S_{Write}}$$

(Prop. 8.11)

In the case of a storage bottleneck (Property 8.10 and 8.11, and Figure 8.6) the time to availability is longer than when keeping only one replica. On the other hand, the time to stabilization is shorter. Indeed, since the leaving nodes stay for a longer time, their storage devices are used to read data during a longer time period, eventually reducing the reading load on the drives of the remaining nodes. Note, however, that reaching the minimal duration of the stabilization phase when $k > 1$ is hardly possible in practice since all the data transferred during the preservation would have to be kept in a buffer to reduce the reading overhead during the stabilization, which induces very large memory buffers.

Conclusion

In this chapter, we established a model of the duration of the decommission operation under a relaxed fault tolerance constraint: the replication factor of the objects can be reduced during the operation.
The obtained models show that the choice of the decommission strategy in the case of a network bottleneck is a simple trade-off of fault tolerance for faster resource availability and lower node-hour usage without drawbacks.
In the case of a storage bottleneck, however, using a fast decommission does release resources faster, but increases the duration of the operation for the storage system, and only reduces the node-hours usage when numerous nodes are decommissioned at the same time. In this situation, the benefits are more incidental.

PART III

Benchmarking Storage Malleability: Pufferbench

CHAPTER 9

PUFFERBENCH: A BENCHMARK FOR RESCALING OPERATIONS

Malleability of distributed storage systems has not been largely adopted mainly because it involves migrating large amounts of data, which is potentially long. With increasingly faster hardware, however, and in light of the theoretical results obtained in earlier chapters, this limitation is worth reevaluating in practice. To know whether distributed storage system malleability would be useful *for a given workload* or *on a given platform*, one should be able to estimate the duration of both commissions and decommissions.

In part II, we modeled the minimal duration of rescaling operations (commission in Chapter 5 and decommission in Chapter 6). These models are useful because they indicate whether rescaling operations would be too slow on a given platform for any practical use. However, if the duration of the rescaling operations given by the model are acceptable, it is still difficult to know *in practice* how fast malleability can be on a specific platform.

In this chapter, we introduce Pufferbench a modular benchmark able to evaluate *in practice* the duration of rescaling operations on a given platform. We detail the customizable components of Pufferbench. Last, we present how to use Pufferbench to evaluate a platform, to evaluate the rescaling of a distributed storage system, and to improve the efficiency of such rescaling mechanisms.

9.1 The need for a benchmark

Assessing the usefulness of malleability: To know whether distributed storage system malleability would be useful *for a given workload* or *on a given platform*, one should be able to estimate the duration of both commissions and decommissions. In Chapters 5, 6, and 8, we have provided a model of the minimal duration of rescaling operations. With them, one can easily estimate whether the operation would be too slow for a specific application on a given platform (i.e., in the case these lower bounds are too high from a practical perspective). However, these models are based on strong assumptions (Chapter 4), including the uniformity of the hardware, perfect load balancing, absence of latency, among others. These assumptions can only be approximated in practice: the latency can be reduced but will not disappear, the hardware may be comprised of the same pieces but wear and tear will induce slight variations in their performance.

Aspects not accounted for by the models: The model has been designed to estimate the minimal duration of a rescaling operation. Thus, it does not cover some aspects of the system that can affect the duration of rescaling operations in practice, such as the network topology,

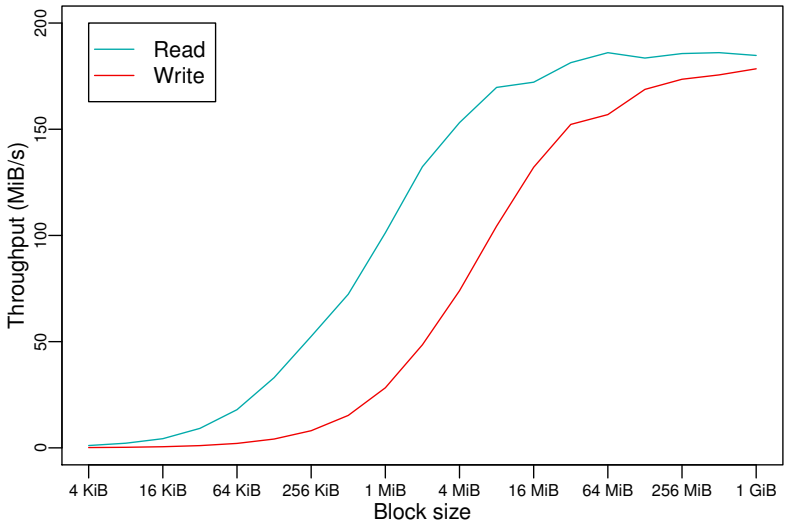


Figure 9.1: Read and write thoughput of random blocks depending on the block size, measured on one of the 600 GB hard drives of a node of the paravance cluster of Grid'5000.

storage caches, and the efficiency of storage devices for various types of data accesses. The network topology, combined with the node allocation, can be the source of network bottlenecks, even though the network interface of the nodes is not itself a bottleneck. For instance, the traffic induced by the rescaling operations may have to be routed through a link with a bandwidth that cannot accommodate it. Operating systems often include a cache for file systems: the cache stores in memory data that was recently read, and can buffer write operations. Because memory is faster than hard drives, the cache can artificially speed up data accesses until it is filled. Some storage devices, hard drives in particular, do not operate at their maximum throughput at all time: the throughput increases with the size of the block read (Figure 9.1).

On the other hand, measuring the duration of rescaling operations on a platform with an actual distributed storage system is impractical. For this, one must deploy an actual distributed storage system, determine an efficient configuration for the rescaling operations, generate data, and record the duration of the operations. This process is time-consuming and not necessarily accurate since rescaling operations implemented in current distributed storage systems are often not optimized for speed but to limit their impact on application execution.

9.2 Pufferbench

We address the problem of evaluating the potential usefulness of malleability by introducing *Pufferbench*,[1] a modular benchmark developed to efficiently measure the duration of commission and decommission operations on a given platform. To this end, Pufferbench emulates a distributed storage system, executing only the inputs and outputs needed for a rescaling operation.

[1] Pufferbench is available at `https://gitlab.inria.fr/Puffertools/Pufferbench`

Pufferbench has been designed with two goals in mind.

1. Evaluate the viability of distributed storage system malleability on a given platform. Pufferbench provides the duration of rescaling operations on a platform, regardless of the distributed storage system that would use these operations.

2. Help optimize rescaling mechanisms in order to improve the malleability of specific distributed storage systems. Pufferbench is independent from any distributed storage system and thus can be used to quickly prototype and test custom rescaling mechanisms (algorithms, network transfers, storage management) on a simpler code before implementing them into a real storage system.

In the second case, Pufferbench also verifies the correctness of the custom migration algorithms by checking that the postconditions (number of replicas, data distribution, etc.) are satisfied.

Pufferbench is implemented as an MPI application that emulates the rescaling operations of a distributed storage system by doing all I/O operations that are needed during such a rescaling operation: data accesses to/from a local storage device (which may be local memory) and across the network. It is a master/workers application with MPI rank 0 acting both as master and worker and all other ranks acting as workers.

Its execution involves four steps.

1. **Migration planning:** Pufferbench's master node applies the migration algorithm chosen in its configuration file to compute the sequence of I/O operations that each node needs to execute to complete the migration (writing/reading to/from local device, sending/receiving to/from other nodes). The trace of I/Os is then sent to worker nodes to be replayed.

2. **Data generation:** All nodes running Pufferbench generate dummy data on their local storage device to have an actual payload to read and transfer.

3. **Execution:** All Pufferbench nodes execute their respective sequence of I/O, recording timing and statistics. In particular, the duration of the rescaling operation is measured.

4. **Statistics aggregation:** Statistics collected by each node are gathered by the master node and output to the user.

While executing the data migration, Pufferbench controls and reports about the following properties:

1. Data conservation: no data is ever lost.

2. Data replication: the replication factor is the same before and after the migration.

3. Data balance: the placement of data across nodes remains balanced.

Hence Pufferbench not only evaluates the performance of rescaling operations, it also assesses the correctness of the migration algorithm.

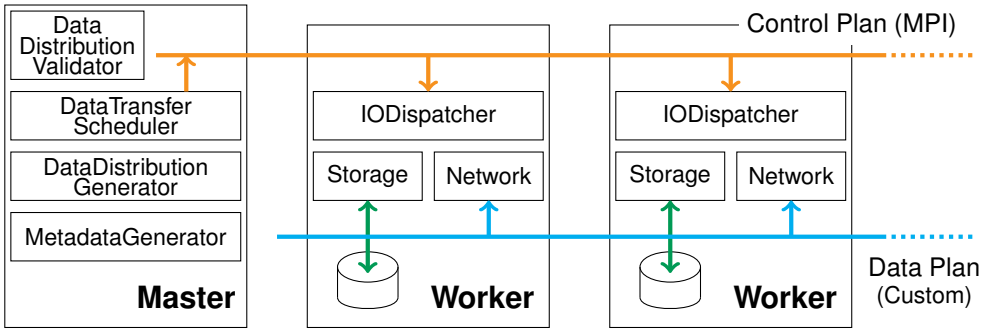


Figure 9.2: Components of Pufferbench and their interactions.

9.3 A highly customizable benchmark

In order to match as many platforms and systems as possible, Pufferbench has been designed with modularity in mind. With a simple change in its configuration file, the main components of the system can be switched for custom ones.

9.3.1 Master node components

Three of these components are used exclusively by the master node (Figure 9.2). In order of action, they are the following.

1. The **MetadataGenerator** generates the basic metadata of the dataset initially present on the emulated storage system. This set of metadata takes the form of a set of pairs *(object id, size)*. Tuning this component enables choosing between various data sizes (e.g., many small objects, few large objects, random size uniformly distributed across a range, gaussian). By changing this component for a custom one, users can plug in data sizes that best match their workload.

2. The **DataDistributionGenerator** takes this metadata set as input and assigns each object to as many virtual storage nodes as necessary to meet the required replication factor. The output of this component is a data distribution map associating each virtual node id with the list of *(object id, size)* that this virtual node manages. Two implementations of this component are provided by default: the first one places data randomly across the nodes, ensuring only the replication factor; the second one balances the load across the nodes. Changing this component enables matching the placement policy of a particular distributed storage or evaluating new ones.

3. The **DataTransferScheduler** is the core of Pufferbench. It takes as input the previously generated placement map as well as the desired migration (e.g. “commissioning 3 nodes”) and produces a sequence of I/O operations (read, write, send, receive) that each virtual node has to replay in order to accomplish this migration. The default DataTransferScheduler redistributes the data randomly but maintains a level of load balancing and carefully chooses the nodes reading and sending the data in order to mitigate the bottlenecks of the operations: receiving and writing the data for the decommission, and reading

the data for the commission. Customizing the DataTransferScheduler allows the user to test new migration algorithms and evaluate their performance before implementing them in a real distributed storage system.

These components are responsible for the simulation of various distributed storage systems. For example, one can simulate HDFS with a MetadataGenerator that generates mostly chunks of 128 MiB and a DataDistributionGenerator that will replicate the chunks three times and place them onto random nodes, as HDFS does.

In addition to these three components, the master node includes a **DataDistributionValidator** component that cannot be customized. This component takes as input a data distribution map that represents the placement of objects on the nodes. From this map, it checks that the data migration is valid, that is, that it respects the requirements listed in Section 9.2. In particular, it checks the replication factor of all objects and evaluates the load balancing of the final distribution by computing the average, minimum, maximum, median, and standard deviation of the amount of data held by each node and the same set of statistics of their number of objects. These statistics enable the user to check whether the final data distribution is more, less, or equally load-balanced than the original one. This validation is done twice in order to control the data distribution maps: before and after the execution of the DataTransferScheduler. Running the DataDistributionValidator twice ensures that the customized DataDistributionGenerator and DataTransferScheduler are both satisfying their requirements.

To evaluate the validity of a particular migration algorithm, Pufferbench can be executed on a single node and stopped at the validation step instead of replaying the I/O operations.

9.3.2 Worker node components

Three components are used by all nodes (including the master) to replay the I/Os.

1. The **Storage** component makes the interface with the local backend storage device by providing the `read` and `write` functions from/to the storage device. By default we provide a Storage component that stores its data in memory, a Storage component that stores its data in a local disk drive, and a Storage component that also stores its data in a local disk drive but ignores the file system cache. Users can plug their own Storage component too, for example using a custom interface of a particular backend device.

2. The **Network** component provides the `send` and `receive` methods used to transfer objects between nodes. Pufferbench's default Network component relies on MPI's nonblocking send and receive functions (`MPI_Isend` and `MPI_Irecv`) so that these operations can complete in parallel with other operations. By default, up to 500 send/receive can proceed concurrently, each of them transferring at most 8 MiB, although these parameters can be configured. Once again, users can plug their own Network component, for example to use other networking libraries, or remote direct memory accesses (RDMA).

3. The **IODisptacher** component takes as input the sequence of I/O operations received from the master's DataTransferScheduler component and dispatches the operations to the Storage and Network components for execution.

9.4 Utilization of Pufferbench

Pufferbench can be used in three different manners. First, it can be used to determine if using storage malleability on a given platform could yield benefits. Second, it can also be leveraged to evaluate the performance and margin of improvement of rescaling mechanisms of existing distributed storage systems. Finally, it can be used as a testbed to design rescaling mechanisms before implementing them into actual distributed storage systems.

9.4.1 To asses the potential benefits of malleability on a platform

Pufferbench can be used with its default components to evaluate the potential for storage malleability on a given platform. This can be done with a simple default configuration, which uses default migration algorithms designed to mitigate the bottlenecks highlighted by the models. Pufferbench replays all I/Os needed to commission or decommission nodes on the target platform. The duration of each operation is recorded along with various other metrics. Since the measurements are done on a real platform, the recorded performance is reachable by any distributed storage system available on that platform, provided the latter is optimized for the rescaling operations.

9.4.2 To evaluate the rescaling mechanisms of a distributed storage system

The duration of rescaling operations obtained by Pufferbench can be used as a reference to evaluate the rescaling mechanisms of existing distributed storage systems. However, to ensure a fair evaluation, the experimenter should ensure similar initial conditions and similar final data placement. This can be achieved thanks to the customizability of Pufferbench: components can be chosen and modified as needed. We use this method to evaluate the rescaling mechanism of HDFS in Chapter 11.

9.4.3 To prototype data migration algorithms

The modularity of Pufferbench allows any user to evaluate and optimize the algorithms used for the rescaling operations (commission and decommission). Thus, Pufferbench can easily be used to optimize and evaluate data migration mechanisms in an existing distributed storage system without modifying it. Users can plug in custom algorithms for data migration, which can replace the default ones provided by Pufferbench. To this end, Pufferbench embeds a DataDistributionValidation component that checks that the plugged-in algorithm yields a valid migration plan. Moreover, writing commission and decommission algorithms can be done in significantly fewer lines of codes than in an actual distributed storage system because of the abstraction. For instance, the commission and decommission algorithms used in Chapter 10 are written in 350 lines of C++ overall.

Conclusion

In this chapter, we motivated the need for a tool that measures in practice the duration of rescaling operations since the previously proposed performance model does not encompass the specificity of each platform. We introduced Pufferbench, a benchmark designed precisely for this task. We detailed the various customizable components of Pufferbench. Customizing these components enables the adaptation of the benchmark to various platforms and use cases. We presented how to use Pufferbench to test a platform, to evaluate rescaling operations of existing distributed storage systems, and to quickly prototype rescaling mechanisms.

CHAPTER 10

VALIDATING PUFFERBENCH AGAINST THE MODELS

In this chapter, we validate Pufferbench against the theoretical minimal duration of the rescaling operations established in Part II. We use clusters from the Grid'5000 testbed [17] to deploy Pufferbench, and use its characteristics (network and storage bandwidths) as parameters for the model. We show that the theoretical minimal duration can be closely approached, since the duration of rescaling operations emulated by Pufferbench are within 16% of it.

Then, we use Pufferbench to study in practice the fast decommission strategy (Chapter 8). We show that the previously established models are realistic: they can be approached by real implementations. Besides, we confirm that using a fast decommission is a simple trade-off between fault tolerance and the time needed to release the leaving nodes without drawbacks when the network is the bottleneck. However, we confirm the strategy can be detrimental in the case of a storage bottleneck, as discussed in Chapter 8.

10.1 Methodology

To evaluate Pufferbench against the theoretical minimal duration of the rescaling operations obtained in Chapters 5 and 6, we used the French Grid'5000 [17] experimental testbed. Experiments on decommission were done on the *paravance* cluster in Rennes, while experiments on commission were done on the *grisou* cluster in Nancy. Both clusters feature the same type of node: Dell PowerEdge R630 with Intel Xeon E5-2630 v3 Haswell 2.40 GHz (2 CPUs/node, 8 cores/CPU), 128 GiB of RAM, and two 558 GiB HDD. They are all connected with a 10 Gb/s Ethernet network to a common Cisco Nexus 6000 switch (for *paravance*) and a Cisco Nexus 9508 (for *grisou*).

Pufferbench emulates a distributed storage system that initially hosts 50 GiB per node. Ten measurements per configuration of Pufferbench were done. The results are represented by box plots showing the minimum, the first quartile, the median, the third quartile, and the maximum duration of the rescaling operation.

In order to create a network bottleneck, the data was stored in memory since it has a bandwidth significantly higher than the network's bandwidth (6 GiB/s reading, 3 GiB/s writing). Similarly, in order to generate a storage bottleneck, the data was stored on the drives of the nodes (207 MiB/s reading, 195 MiB/s writing).

10.1.1 Evaluation of Pufferbench against the Lower Bounds

The models giving theoretical minimal duration of the rescaling operations have been built on strong assumptions (detailed in Chapter 4), such as an all-to-all network topology and the absence of latency in disk accesses. In practice, these hypotheses are not met. The following sections describe to what extent our experimental setup respects the assumptions. In order to safely evaluate any result against the models, the experimental conditions should be such that practical constraints only increase (and never decrease) the duration of the rescaling operations.

Assumption on the hardware

- *Cluster homogeneity (Assumption 1)*: The cluster should be composed of identical nodes. In practice, although the clusters we used are homogeneous, performance variations can be observed across nodes. For instance, the maximum bandwidth of the drives while reading varies between 195 MiB/s and 207 MiB/s on the cluster used for the experiments. The parameters of the models (maximum disk read speed, maximum disk write speed, and maximum network bandwidth) have been set with the maximum measured values prior to the experiments.
- *Ideal network (Assumption 2)*: The models ignore the network latency and the potential interference that can happen. An all-to-all topology is assumed, with identical bandwidth between any two nodes. In practice, all the nodes of our clusters are connected to a common switch through a 10 Gb/s Ethernet link. Hence, this switch may become a bottleneck. This bottleneck may only increase the rescaling time compared with the models.
- *Ideal storage backend (Assumption 3)*: The models ignore the latency (seek times of the drives) and assume that the drives always read and write at their maximum speed. In practice, seek times and read/write contention add to the duration of the execution.

Although the theoretical assumptions are not met, the differences between the experimental setup and the hypotheses only increase the duration of commissions and decommissions. This ensures that the models keep their property of providing the theoretical minimal duration of the operations even with the relaxed hypotheses.

Assumptions about the components of Pufferbench

With Pufferbench's components, we can make sure that the other assumptions and objectives needed by the models are met.

The DataDistributionGenerator ensures the other three assumptions used to establish the models.

- *Data balance (Assumption 4)*: The nodes host the same (or similar) amount of data.
- *Data replication (Assumption 5)*: Each object stored in the cluster is replicated r times.
- *Uniform data distributions (Assumption 6)*: Each set of r distinct nodes has some data that is replicated only on these r nodes. The amount of such data should be the same for every such set.

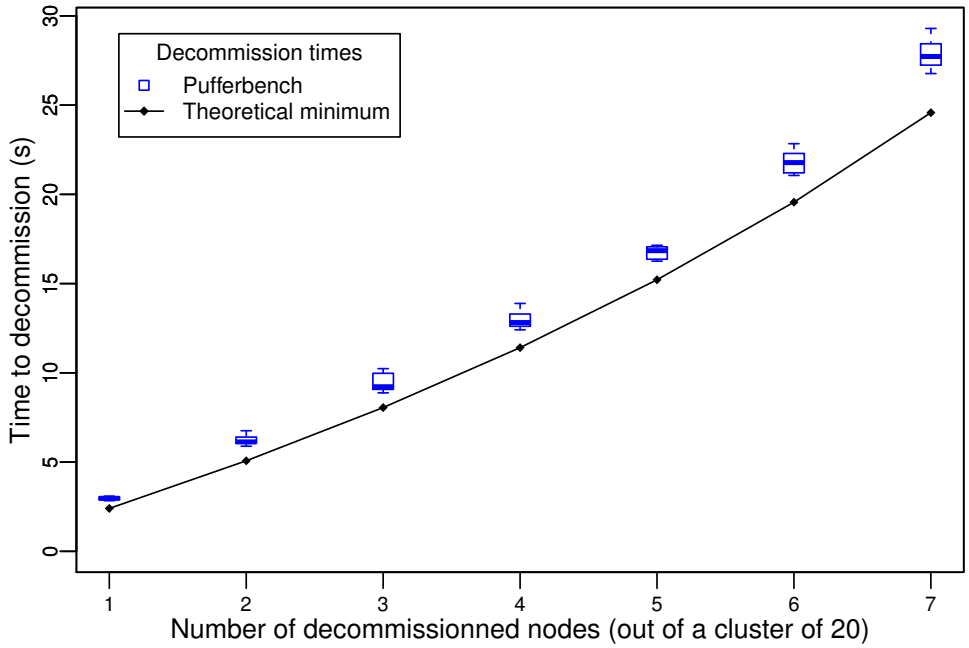


Figure 10.1: Time needed to decommission nodes, with storage in memory. Nodes initially host 50 GiB of data on average. Comparison with theoretical minimum.

The DataTransferScheduler, which implements the migration algorithms, ensures that the following objectives are met at the end of the rescaling operation.

- *No data loss (Objective 1)*: No data is lost during the operations.
- *Maintained data replication (Objective 2)*: The replication factor is the same as the initial one.
- *Data balance (Objective 3)*: The nodes host the same amount of data.

The last objective (Objective 4), which is to have *uniform data distribution* at the end of the rescaling operation, is relaxed since it is achieved with random data placement. This does not impact the models in most cases except when, during the commission, reading from disk becomes the bottleneck (commission of more than 22 nodes in the following measurements). However, the randomness ensures that the data distribution generated is close to the objective.

Overall, the experimental setup guarantees that each duration obtained with the models are a theoretical minimum of the measured duration of the rescaling operations. The models can safely be used to evaluate the performance of Pufferbench configured as detailed in this section.

10.2 Commission and decommission

Figure 10.1 shows the performance of Pufferbench against the theoretical minimum duration when decommissioning nodes from a cluster of 20 nodes. Each node initially hosts 50 GiB of

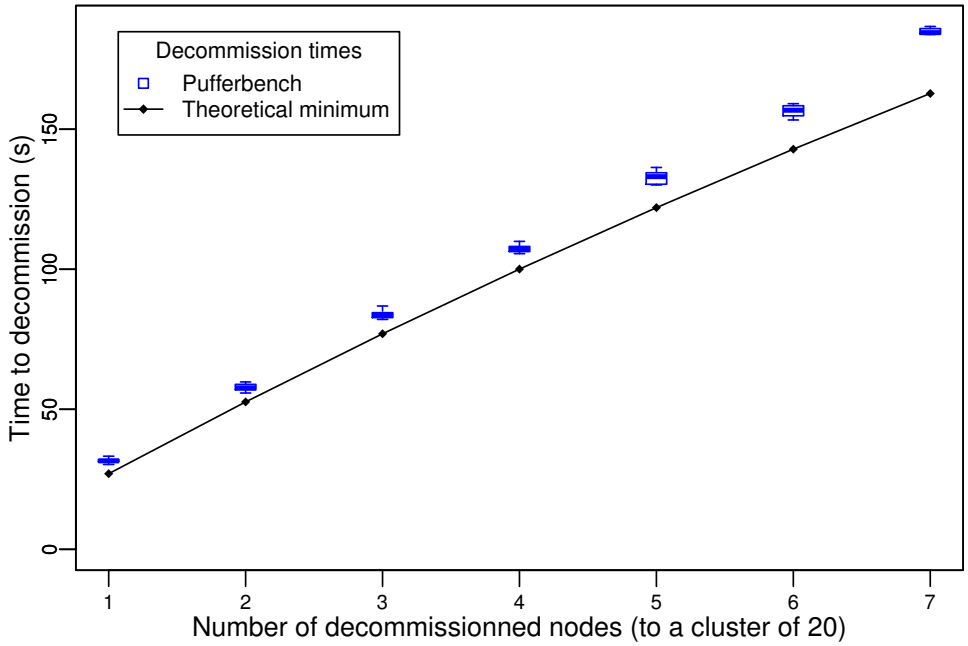


Figure 10.2: Time needed to decommission nodes, with storage on drive. Nodes initially host 50 GiB of data on average. Comparison with theoretical minimum.

data in memory. On average, decommission times measured by Pufferbench are 16% longer than the estimates provided by the models.

Figure 10.2 presents the decommission with storage on disk. On average, Pufferbench is 11% slower than the model.

In both cases, the difference between the theoretical minimum and Pufferbench is due to the fact that Pufferbench runs on real hardware. The models consider only the maximum bandwidth of the network and the storage. They ignore the latency and any interference. Pufferbench takes all this into account since it replays all I/Os needed to decommission nodes.

Figure 10.3 shows the performance of Pufferbench when commissioning nodes into a cluster of initially 10 nodes with storage in memory. On average, Pufferbench is 7% slower than the model.

Figure 10.4 presents the results of the commission when the storage is on disk. In this case, Pufferbench is 16% slower than the theoretical minimum.

The main reason for the difference in performance between the storage in memory and the storage on disk drives is the latency to access data on disks, as well as the disks not exhibiting uniform performance across nodes and across requests (peak read speed varied between 195 MiB/s and 207 MiB/s across nodes).

In both cases, we observe that when 20 nodes are added, the difference between the model and the results of Pufferbench is the largest. This is due to the fact that stragglers appear. In this case, the effect of the stragglers is clear because all the nodes become bottlenecks: the 20 new nodes must finish writing their data in about 15 seconds, but the old nodes must also finish reading and sending their data in 15 seconds. Thus, any node straggling has an impact on the overall performance. Stragglers appear because of the variability of hardware performance.

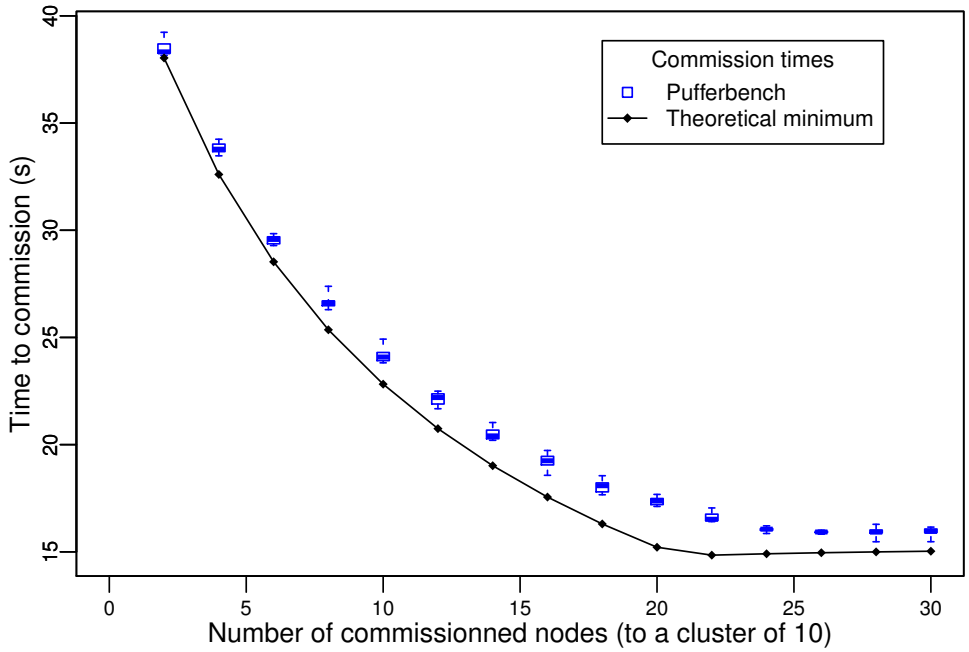


Figure 10.3: Time needed to commission nodes, with a storage in memory. Nodes initially host 50 GiB of data on average. Comparison with theoretical minimum.

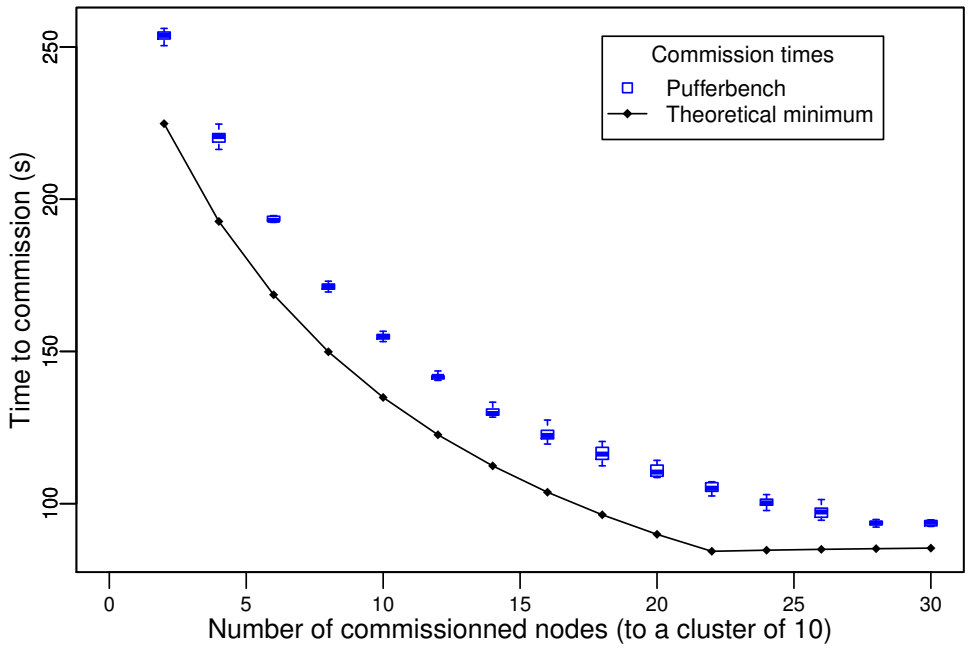


Figure 10.4: Time needed to commission nodes, with a storage on drive. Nodes initially host 50 GiB of data on average. Comparison with theoretical minimum.

> *We show that Pufferbench is able to emulate rescaling operations that are close to the theoretical minimum (at most 16% of difference on average). Such a difference with the theoretical minimum can be attributed to the fact that the hardware does not match the assumptions used to build the model. Moreover, these results also show that the models themselves are realistic.*

10.3 Fast decommission

In Chapter 8, we established a model for the duration of the various phases of the fast decommission. In this section, we use Pufferbench to study the fast decommission in practice.

10.3.1 Implementing fast decommission in Pufferbench

With a fast decommission strategy, the leaving nodes transfer to the remaining ones only the data that is exclusively on them with high priority. The remaining nodes have to recreate the missing replicas; however, the operation is done with a lower priority. The leaving nodes can leave the cluster only after the data is on the storage device; they cannot leave if the data is only buffered in memory.

As in Section 10.1, we make sure that the implementation of the fast decommission matches the assumptions presented in Chapter 4 in order to ensure that the model returns the theoretical minimal duration of the operation. This allows us to safely compare the practical results to the minimal duration of the operation.

10.3.2 Experimental setup

All measurements were done of the *grisou* cluster (see Section 10.1), and were repeated ten times each.

In this evaluation, we use a 20-node cluster as the initial size, to show that the model of the minimal duration matches the behavior observed in practice. For other scales of cluster, the analytical results should be used to determine the relevance of the fast-decommission.

10.3.3 Fast decommission when the network is the bottleneck

Figure 10.5 shows the duration of the data-safekeeping and stabilization phases when the network is the bottleneck. The standard decommission has been added for comparison.

Compared with the theoretical minimum duration, the time to availability is on average 37% slower, while the stabilization time is 32% slower. For the same configurations, the standard decommission is, on average, 22% slower than its theoretical minimum duration. Note that the minimum duration obtained with the models cannot be reached in practice: they assume an absence of latency and permanent maximum throughput from both the storage and the network.

When few nodes are decommissioned (less than 6), the difference in duration between the two strategies is negligible. When many nodes are decommissioned at once, however, there is a large difference between the standard decommission and the time to stabilization.

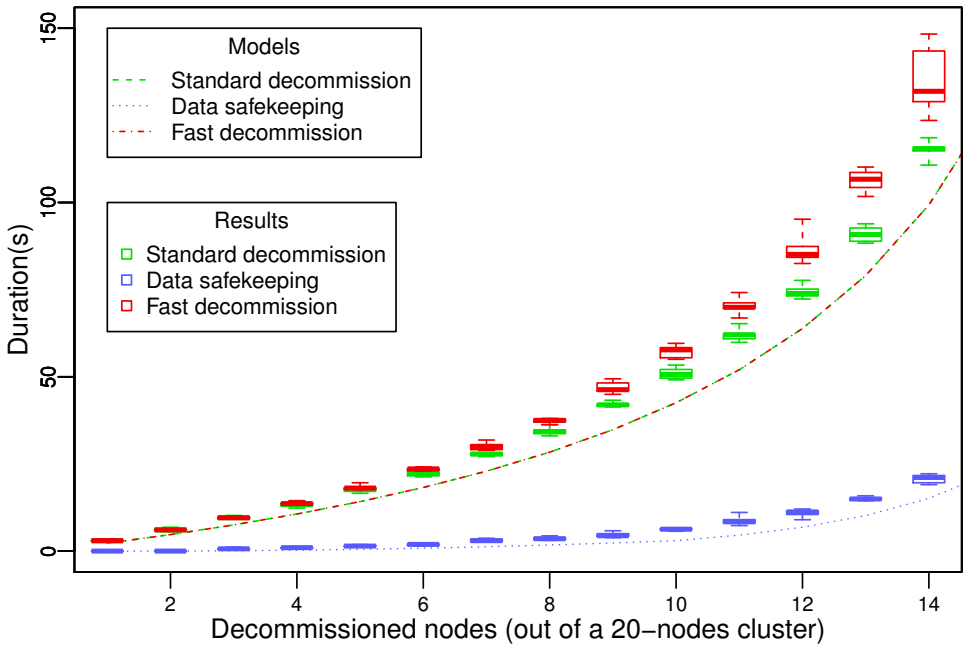


Figure 10.5: Duration of the data-safekeeping phase, fast decommission and normal decommission obtained with Pufferbench and compared with the theoretical minimum duration, in the case of a network bottleneck. Each node initially hosted 50 GiB of data.

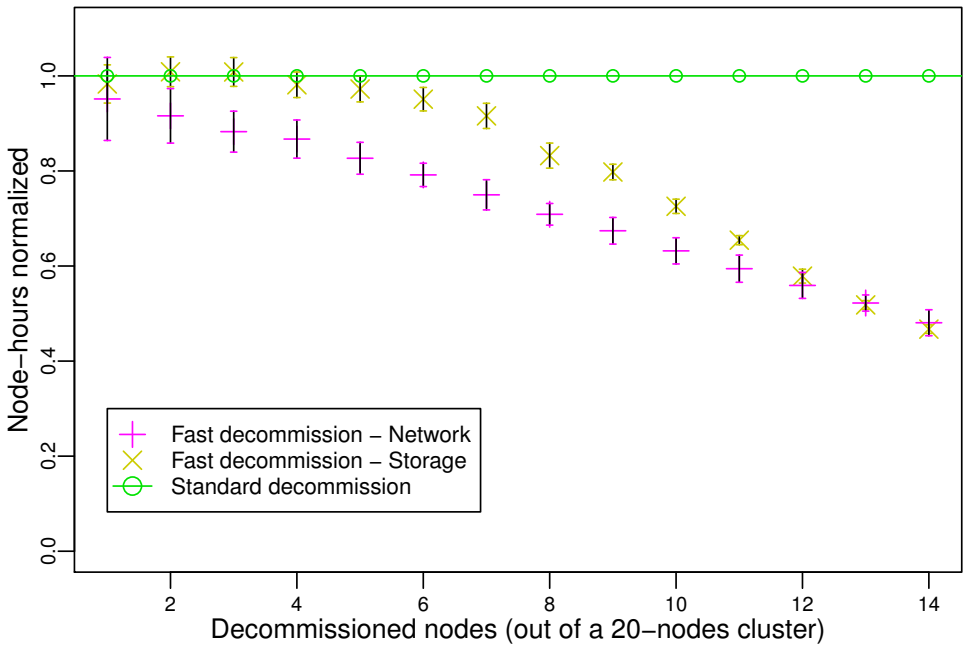


Figure 10.6: Node hours needed to do a fast decommission on a cluster of 20 nodes normalized by the standard decommission with the same bottleneck. The standard deviation has been added.

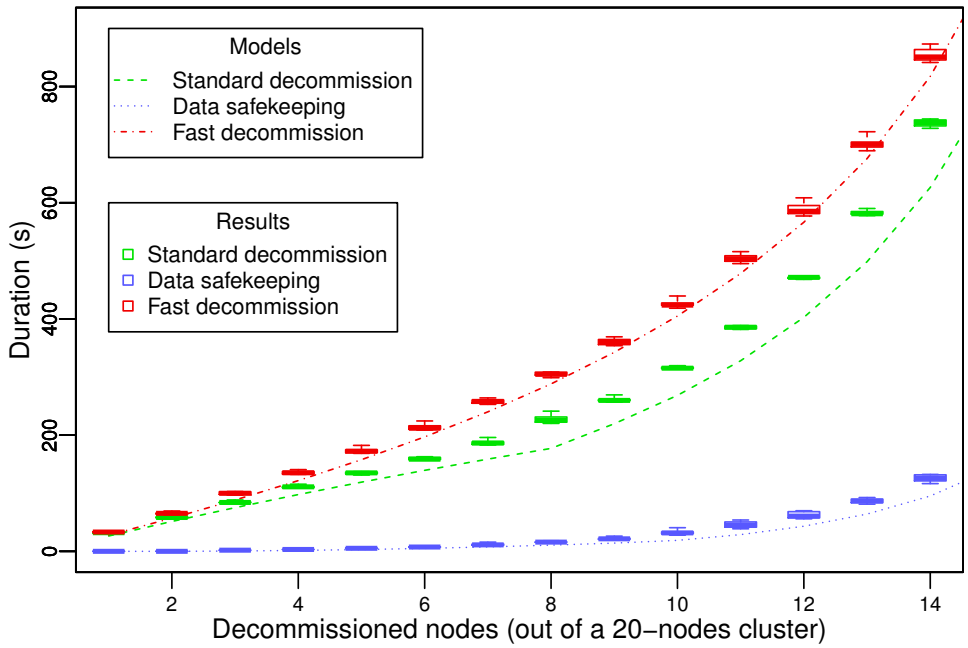


Figure 10.7: Duration of the data-safekeeping phase, fast decommission and normal decommission obtained with Pufferbench and compared with the theoretical minimal duration, in the case of a storage bottleneck. Each node initially hosted 50 GiB of data.

For example, the fast decommission is 12% slower than the standard decommission when 14 nodes are decommissioned. The reason for this difference is the stress on the network induced by the fast decommission. Indeed, during the fast decommission, the remaining nodes have to send and receive data at the maximum bandwidth speed in order to stabilize the system quickly. During the standard decommission, however, the sending load is distributed not only on the remaining nodes but also on the leaving nodes, reducing the overall load on each node. This difference does not appear on the models because we assume that the network is ideal (Assumption 2).

Figure 10.6 shows the number of node-hours consumed by decommission normalized by the consumption of nodes hours during a standard decommission. In most cases, using the fast decommission reduces the usage of node-hours. The gain in node-hours increases with the number of decommissioned nodes. When most of the nodes are decommissioned at once, the fast decommission uses only 50% of the node-hours required by the standard decommission. The theoretical predictions about the node-hour usage (Chapter 8) are confirmed by these results.

10.3.4 Fast decommission when the storage is the bottleneck

The time to availability and the stabilization time obtained with Pufferbench in the case of a storage bottleneck are presented in Figure 10.7. The duration of the standard decommission has been added for reference.

On average, the time to availability is within 10% of its theoretical minimum, while the stabilization time is within 9% of its theoretical minimum. In comparison, the standard decommission is within 17% of its theoretical minimum. From this, we deduce that the models of the minimal duration of the fast decommission are sound and can almost be reached in practice.

Figure 10.6 shows the number of node-hours needed for the whole operation normalized by the node-hours needed for a standard decommission. Using a fast decommission offers no benefit in node-hours when the number of decommissioned nodes is low. In this case, the node-hours needed to stabilize the system are canceling the benefits of releasing the decommissioned nodes earlier. When numerous nodes are decommissioned at once, however, the gain in node-hours can reach 50%. These results are in line with the node-hours that the model established in Chapter 8.

The study of the fast decommission with Pufferbench confirmed the conclusions obtained with the models in Chapter 8.

10.4 Discussion

10.4.1 Ideal setup for the validation of Pufferbench

The experimental setup used to validate Pufferbench is favorable to fast rescaling operations. In particular, all objects had a size of 128 MiB in order to read, send, receive, and write large sequential chunks of data. This optimized I/Os for both the network and the local storage.

Thus, the performance should degrade with smaller data objects, and Pufferbench should then be used to optimize algorithms, storage, and network transfers to efficiently migrate small objects.

10.4.2 Pufferbench and theoretical minimal duration

Compared against the theoretical minimal duration, the results obtained with Pufferbench have the advantage of providing more accurate commission and decommission times. First, it actually replays I/Os so the characteristics of the hardware (network latency, network interference, disk seek times, disk throughput) are taken into account, but it also evaluates an implementation of the operations.

The models have the advantage of fixing what is theoretically possible: provided that the hypotheses are met (in particular the initial load balancing), the operations cannot be faster than the theoretical minimum. They can be a good comparison point when the platform is not available, whereas Pufferbench gives more accurate results when the platform is available.

However, the results show that the models previously determined are realistic and that performance close to the theoretical minimum duration can be reached.

Conclusion

In this chapter, we showed that Pufferbench is able to run rescaling operations with duration close to the theoretical minimum (within 16% on average). This highlights the capability of Pufferbench to measure how fast a rescaling operation can be executed on a given platform. Incidentally, it also shows that the models of the minimal duration of rescaling operations are realistic.
We then used Pufferbench to study, in practice, the fast decommission strategy. We confirmed the observations obtained with the model established in Chapter 8. Using the fast decommission with a network bottleneck has minimal impact of the storage besides the trade-off of fault tolerance for a faster release of leaving nodes; it also reduces the node-hour consumption of the operation compared to a standard decommission. On the contrary, when the bottleneck is at the storage device level, the fast decommission can be detrimental: there are no node-hour gains unless many nodes are simultaneously decommissioned, and the stabilization phase lasts longer than the standard decommission.

CHAPTER 11

USING PUFFERBENCH TO EVALUATE THE RESCALING OPERATIONS OF HDFS

In Chapter 7, we showed that the rescaling operations of HDFS could be greatly accelerated, provided that the theoretical minimal duration can be reached. For example, in the case of decommission, with the storage done on HDDs, HDFS is three times slower than the theoretical minimum. With Pufferbench, we now have a tool to confirm this in practice.

In this chapter, we evaluate the commission and decommission of HDFS by comparing their duration with Pufferbench to assess whether HDFS's rescaling mechanism can be optimized. We evaluate how fast rescaling operations could be under the constraints of HDFS. In particular, we show that, with a different rescaling mechanism (algorithms, data transfers, and disk management), HDFS could be much faster at commissioning and decommissioning nodes.

11.1 Methodology

In this section, we detail the experimental setup used to compare the performance of the rescaling mechanisms of HDFS with Pufferbench.

11.1.1 How Pufferbench emulates HDFS

In order to fairly evaluate HDFS with Pufferbench, we implement in Pufferbench rescaling mechanisms that move the same objects (chunks of 128 MiB) and aims for the same final data distribution as HDFS. By doing so, we can compare the performance of the rescaling operations of HDFS and those of Pufferbench since they produce the same results from the same initial setup.

The initial data placement is generated by the MetadataGenerator and DataDistributionGenerator components in Pufferbench. They are configured to generate 128 MiB chunks of data that are replicated three times across randomly selected nodes. The DataTransferScheduler is also customized to ensure Pufferbench follows the same constraints as HDFS. Since the rescaling mechanisms used by HDFS are not optimized for speed, we implemented our own DataTransferScheduler in Pufferbench to try to achieve the best performance rather than imitating HDFS's algorithm. We detail the commission and decommission algorithms in the following sections.

Note that we did not recreate the commission and decommission mechanisms of HDFS in Pufferbench for a simple reason: the algorithms used by HDFS are dynamic. Data transfers are rescheduled every few seconds to match the actual progress, while Pufferbench assumes that all transfers are scheduled at the beginning of the rescaling operation.

Algorithm for a fast decommission

Compute avg the average amount of data per node (old and new).
Cluster nodes according to the amount of data they store D:
    if $D > avg$ *or if the node is leaving* **then** the node is Over-Utilized
    else the node is Under-Utilized
foreach *Over-Utilized node* n **do**
    Randomly select a chunk of data c hosted on n.
    if *Removing* c *from* n *makes it Under-Utilized and the node is not leaving* **then**
        continue with the next Over-Utilized node
    Mark c as to be removed from n, and consider c to not be hosted by n for the following decisions.
    Find a destination for c:
        Randomly select an Under-Utilized node d that follows the conditions:
            Does not host c (due to replication),
            Is not marked as the destination of c.
        Mark d as a destination of c.
        if d *becomes Over-Utilized* **then** Remove d from Under-Utilized nodes

Algorithm 11.1: Data redistribution used in Pufferbench.

Initialize the loads of each node to 0.
Initialize the amount of data sent of each node to 0.
foreach *Chunk* c *with marked destinations* $\{d_1, .., d_k\}$ **do**
    foreach *Destination* $d \in \{d_1, .., d_k\}$ **do**
        Select the source s that:
            Host c,
            Sends the least amount of data.
        Increase the amount of data sent by s by the size of c.
        Schedule the transfer of c from s to d.
        Add d as a possible source for c.
Remove from their original host the chunks marked for removal.

Algorithm 11.2: Decommission algorithm used in Pufferbench in the case of a network bottleneck.

As shown in Chapter 6, the bottleneck in the decommission is receiving and writing the data to storage. Indeed, thanks to data replication, not only do the nodes being decommissioned have the data, some other nodes have them as well.

The amount of data written on each new node is determined by the data placement, which is random but with some load balancing (Algorithm 11.1). Nodes are classified as either overloaded or underloaded. Nodes being decommissioned are overloaded since they should host no data. Replicas are randomly moved from overloaded nodes to underloaded ones, provided that no two replicas of the same chunk end up on a same node.

When using in-memory storage (Algorithm 11.2), no optimization can be made: each node has data to write and can read data simultaneously without interference. In the case of on-

```
1  Initialize the loads of each node to 0.
2  Initialize the amount of data sent of each node to 0.
3  foreach Chunk c with marked destinations {d_1, .., d_k} do
4  |  foreach Destination d ∈ {d_1, .., d_k} do
5  |  |  Add the size of c to the load of d.
6  foreach Chunk c with marked destinations {d_1, .., d_k} do
7  |  Select the source s that:
8  |  |  Host c,
9  |  |  Has the lowest load.
10 |  Increase the load of s by the size of c.
11 |  Mark s as the single source of c.
12 |  foreach Destination d ∈ {d_1, .., d_k} do
13 |  |  Select the source s among the sources of c that sends the least amount of data.
14 |  |  Increase the amount of data sent by s by the size of c.
15 |  |  Schedule the transfer of c from s to d.
16 |  |  Add d as a possible source for c.
17 Remove from their original host the chunks marked for removal.
```

Algorithm 11.3: Decommission algorithm used in Pufferbench in the case of a storage device bottleneck.

drive storage (Algorithm 11.3), each replica must be read once and then forwarded to nodes on which it should be written. The replica can be read from any node hosting it, not only leaving nodes. Thus, some drive load-balancing is done to choose which node has to read data: all nodes should have balanced amounts of disk I/Os (taking into account the fact that disks from leaving nodes will only read, while others can read and write).

Algorithm for a fast commission

```
1  Initialize the amount of data sent of each node to 0.
2  foreach Chunk c with marked destinations {d_1, .., d_k} do
3  |  Select the source s that:
4  |  |  Host c before the commission,
5  |  |  Sends the least amount of data.
6  |  Select l among {d_1, .., d_k} that sends the most data.
7  |  Increase the amount of data sent by s and {d_1, .., d_k} except l by the size of c.
8  |  Schedule the transfers of c starting from s and then forwarded by all nodes in
   |    {d_1, .., d_k} to finish by l.
9  Remove from their original host the chunks marked for removal.
```

Algorithm 11.4: Commission algorithm used in Pufferbench.

As shown in Chapter 5, two bottlenecks arise during the commission operation: reading the data from the old nodes and writing it to the new nodes. The data placement is the same as for the decommission (Algorithm 11.1).

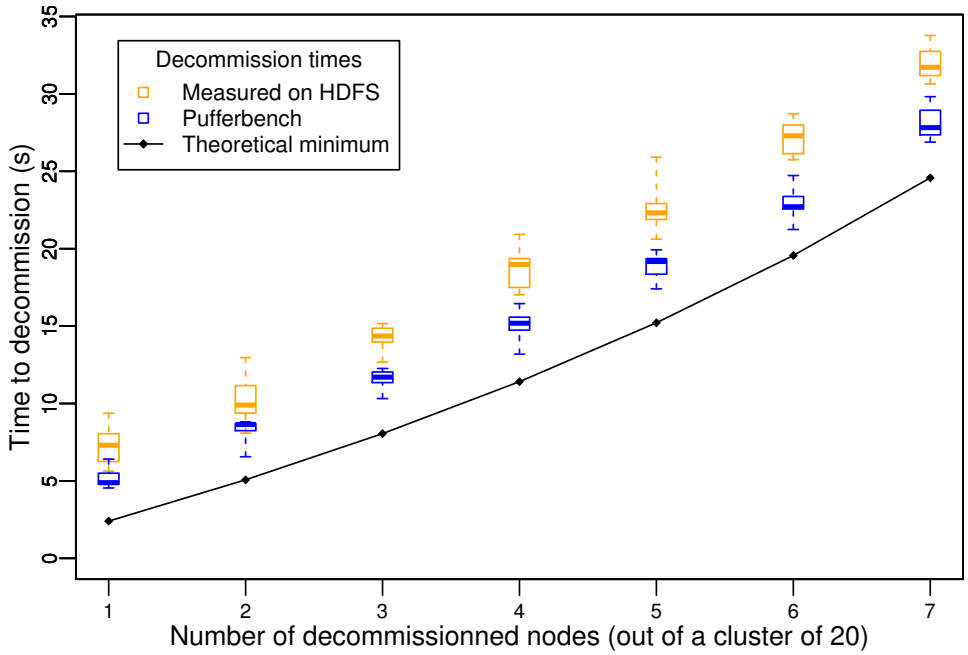


Figure 11.1: Time needed to decommission nodes, with a storage in memory. Nodes initially host 50 GiB of data on average.

The main bottleneck that can be mitigated during the commission is when reading data from the nodes initially present. Since existing nodes initially have all the data, each replica to be moved is read once from an existing node and then exclusively forwarded between new nodes (Algorithm 11.4).

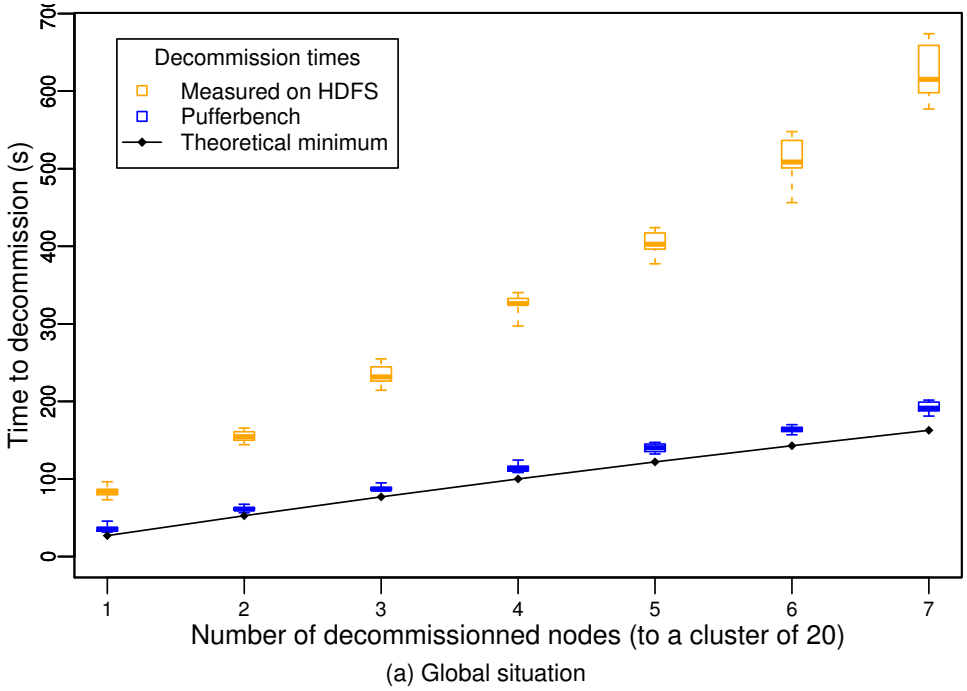
Replay components

The Storage and Network components are the ones provided by default in Pufferbench (in-memory and disk-based for Storage, MPI-based for Network). We disabled caching in the underlying local file system in order to match the configuration used by HDFS.

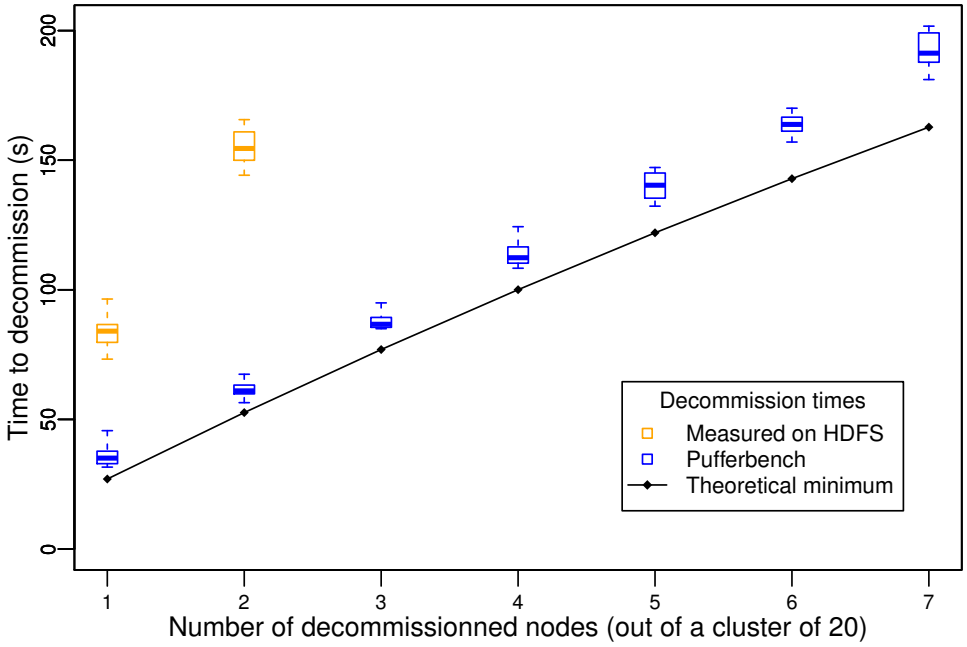
Compared with HDFS, there is a large improvement in the disk I/Os done by the Storage component. HDFS reads and writes data on the drives by blocks of 4 KiB and lets the file system's cache optimize the operations. The Storage component in Pufferbench buffers the data until the data for the whole object is received, then writes it. Overall, Pufferbench's storage component writes and reads larger chunks to/from the drives, optimizing the drive bandwidth usage.

11.2 Rescaling performance of HDFS

In this section, we compare the performance of HDFS and Pufferbench emulating an optimized HDFS during rescaling operations.

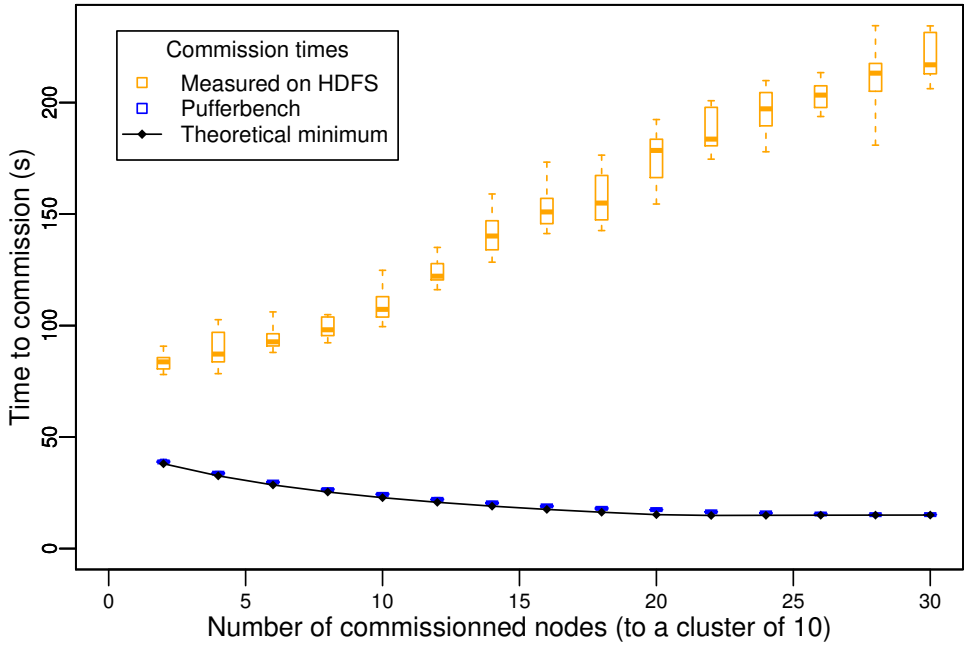


(a) Global situation

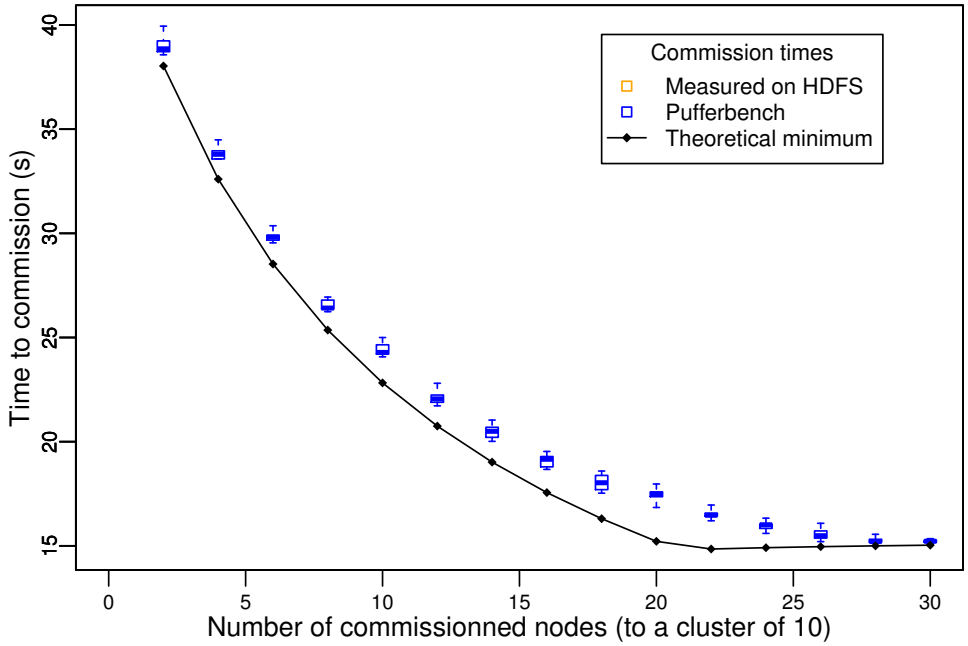


(b) Details of Pufferbench

Figure 11.2: Time needed to decommission nodes, with storage on disk. Nodes initially host 50 GiB of data on average.

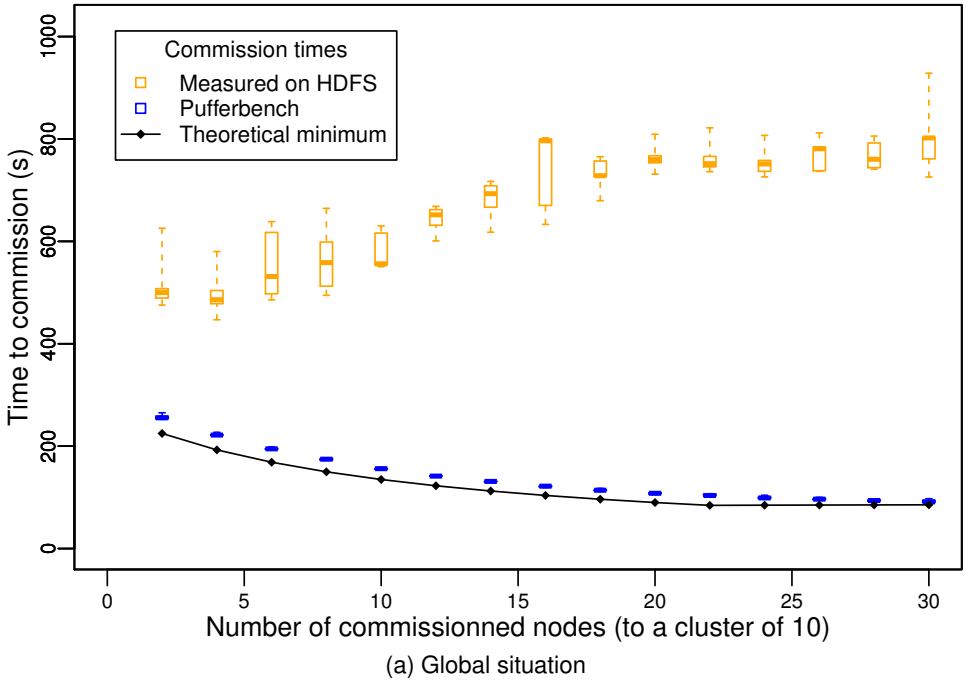


(a) Global situation

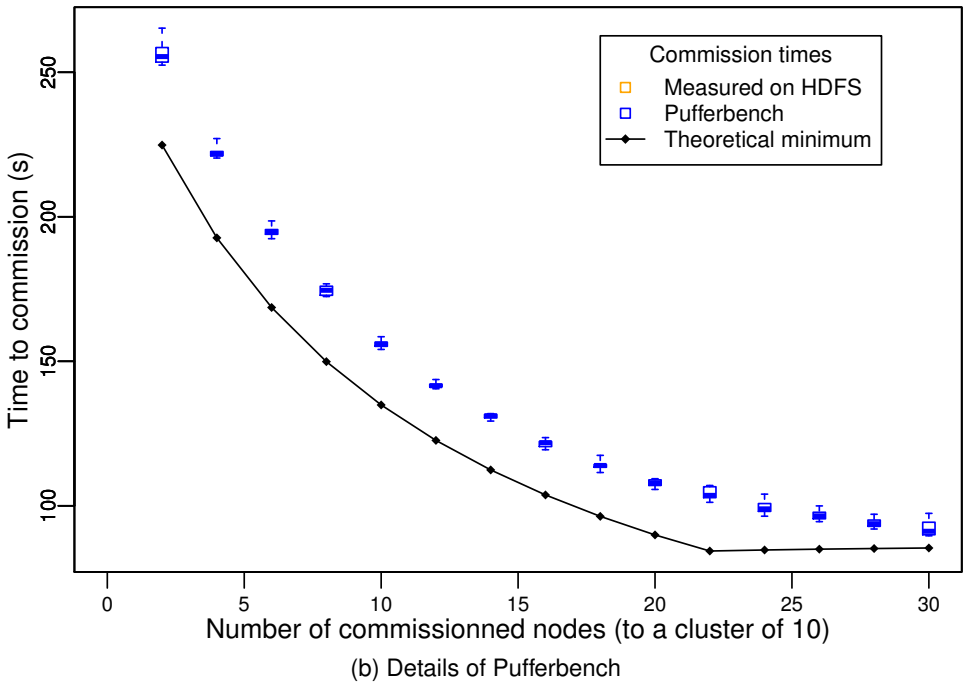


(b) Details of Pufferbench

Figure 11.3: Time needed to commission nodes, with storage in memory. Nodes initially host 50 GiB of data on average.



(a) Global situation



(b) Details of Pufferbench

Figure 11.4: Time needed to commission nodes, with storage on disk. Nodes initially host 50 GiB of data on average.

11.2.1 Experimental setup

The experimental platform is the same as the one presented in Chapter 10.

HDFS and Hadoop 2.7.3 are deployed on the nodes. The replication factor is left unchanged to 3. The configuration of HDFS is adjusted in order to remove the limits on the bandwidth usage of the rescaling operations (details of the parameters can be found in Chapter 7, Section 7.1.1). The commission is divided in two steps. First, new HDFS workers join the cluster; then a rebalancing operation is started. For the decommission, the built-in mechanism is used. The initial size of the cluster is 20 for the decommission experiments and 10 for the commission experiments. All nodes in the initial clusters host 50 GiB of data before each rescaling operation.

The theoretical minimal duration is different depending on whether the network or the storage devices are the bottleneck for the data transfers. In order to create a situation in which HDFS has a storage bottleneck, the data is simply stored on the disks with the file system's cache limited to 64 MiB. For the situation in which the network is the bottleneck, the data of HDFS was stored on a RAMDisk, effectively storing all the data in memory.

Each measurement was repeated 10 times except for the commission with storage on drive, for which time constraints limited the number of repetitions to 5.

11.2.2 Potential speed of decommission in HDFS

Figure 11.1 presents the results of the decommission when the data is stored in memory. Pufferbench is faster than HDFS. HDFS is on average 23% slower than Pufferbench and up to 40% slower in some cases. The difference with the theoretical minimal duration is mainly due to the initial data balance assumption (Assumption 4: the nodes should all host exactly 50 GiB) not being met. Because HDFS and Pufferbench randomly distribute the data on the nodes, initial data balance is indeed not guaranteed. Thus, some nodes receive more data than determined by the model. For instance, a node can initially host 45 GiB of data instead of 50 GiB and have to host 60 GiB at the end of the decommission. Because of the initial data imbalance, this node will have to receive 15 GiB instead of 10 GiB, and the decommission will take longer.

In the case where data is stored on drive, Pufferbench is able to decommission nodes faster than HDFS can(Figure 11.2). On average, HDFS decommissions nodes 2.8 times slower than Pufferbench does. The problem of initial data imbalance observed in Figure 11.1 is still present, but the migration scheduler of Pufferbench configured for these experiments mitigates it when balancing the read load across drives. The drives that have more data to write will spend less time reading.

11.2.3 Potential speed of commission in HDFS

In the case of the commission with storage in memory (Figure 11.3), Pufferbench is on average 8 times faster than HDFS and up to 14 times when numerous nodes are added at the same time.

Note that the commission mechanism in HDFS is not optimized for speed but is made to minimize the impact of the commission on the overall cluster.

Pufferbench also exhibits good performance when commissioning nodes with storage on drive (Figure 11.4). On average, Pufferbench is 5.5 times faster than HDFS.

> *The decommission mechanism of HDFS could realistically be sped up by a factor 2.8 in the case of a bottleneck at the storage level. The commission could be sped up by at least 5.5 times in all cases.*

11.3 Discussion

The comparison of the performance of the rescaling mechanism of HDFS with Pufferbench highlighted some points that we discuss in this section.

11.3.1 Low overhead of Pufferbench

Experiments using Pufferbench have one notable aspect compared to the ones with HDFS: Pufferbench provides results quickly. Its speed is related not only to the improved rescaling operations but also to the reduced overhead.

To measure the duration of the commission with storage in memory in HDFS, one has to start HDFS, start Hadoop, generate enough data, start the commission, and stop the system. The whole operation took 39 hours for the results presented in Figure 11.3, with only 13 hours spent commissioning nodes; an overhead of 26 hours is caused by the setup required to start commission operations.

In contrast, the measurements with Pufferbench lasted for 2 hours and 53 minutes with 2 hours spent commissioning nodes. The overhead of Pufferbench was only 53 minutes, about 30 times lower than the overhead of HDFS. Indeed, Pufferbench only has to allocate some memory in order to be able to replay a commission, and it is able to quickly switch between measurements.

11.3.2 HDFS's case: recommendations

In the case of HDFS, we observe that the optimized algorithms for the scheduling of data transfers during rescaling operations are not the only factor of improvement. Pufferbench's network usage is better than HDFS, but the most important part is the backend drive usage: Pufferbench reads and writes full chunks (128 MiB) sequentially. This approach improves the read and write throughput during the decommission operations by at least a factor of 2 compared with HDFS. The drive management of HDFS can be improved by taking advantage of the amount of memory available on the nodes. HDFS writes and reads data by chunks of 4 KiB. Using larger values (e.g, 128 MiB in the case of Pufferbench) is enough to greatly speed up the decommission of nodes.

Two main aspects of the commission could be improved. First, the algorithm used by HDFS easily accumulates delays: the rebalancing is scheduled by waves of data transfers, and each wave must be completed before the next one starts. This should be replaced with an algorithm that maintains a constant transfer of data between nodes. Second, some buffering should be used in order to read only once each chunk of data from the drives when sending the data to multiple destinations.

11.3.3 Limiting the impact of rescaling operations on application performance

Today rescaling operations of distributed storage systems are used primarily as maintenance operations: permanently increasing the size of a cluster and safely removing faulty nodes. Thus, the rescaling operations are rarely optimized for speed. In HDFS, for example, both the rebalancing (for the commission) and decommission operations have options in the configuration to limit the bandwidth they can use in order to reduce their impact on concurrently running applications.

One can implement such a limitation in Pufferbench simply by implementing a custom Network component that limits the bandwidth usage. The modularity of Pufferbench allows experimenting with multiple limitations (global disk bandwidth, disk read and/or write bandwidth, network bandwidth, network send and/or receive bandwidth, etc.). It also helps to easily test and improve all the relevant components used during the rescaling operations: scheduler, network, and storage.

Conclusion

In this chapter, we used Pufferbench to evaluate the performance of the rescaling mechanisms of HDFS. We showed that the decommission of HDFS could realistically be sped up by 23% on average and the commission by up to 8 times on average and up to 14 times in the case of in-memory storage. When the storage is on drives, the decommission could be 2.8 times faster, and the commission could be on average sped up by a factor 5.5.

PART IV

A Rescaling Manager: Pufferscale

CHAPTER 12

RESCALING TRANSIENT STORAGE SYSTEMS FOR HPC

Since separated storage systems in HPC infrastructures have been an increasing problem for application performance [10], more data services (services taking over some functions of the separated storage and launched on compute nodes) including transient storage systems have emerged. However, these data services are also limited by their rigidity, and cannot fully support evolving and malleable applications.

In this part of the manuscript, we focus on adding malleability to HEPnOS, a transient storage system for high-energy physics applications (see Chapter 3). Adding an efficient support of rescaling operations in HEPnOS presents new challenges since their primary goals are load balance (i.e., ideally, the I/O pressure should be uniformly distributed across those nodes) and speed instead of the objectives of data balance (i.e., ideally, each node should host the same amount of data) and speed that were considered in Part II and III.

In this chapter, we detail the challenges of adding malleability to HEPnOS. We claim the objectives of the rescaling operations of HEPnOS, load balance and speed, cannot be sustained without considering data balance. We then formalize the problem, and highlight the need to consider all three objectives: load balance, speed, and data balance.

12.1 Adding malleability to HEPnOS: Challenges

HEPnOS [11] (presented in Chapter 3, Section 3.3.5) is a transient storage system designed to be co-deployed with high energy physics applications.

An efficient support for malleability in HEPnOS would have two major advantages. HEPnOS could be rescaled before each execution of an application to fit the application's requirements, and thus ensure fast data accesses to the application's data. Moreover, during periods without applications relying on it, HEPnOS would be shrunk to a minimal set of nodes, reducing its resource usage and thereby its energetic and financial cost.

Adding malleability to HEPnOS presents similar challenges to those of the works detailed in Part II and III, however, major differences need to be taken into account. The first difference is the consideration of load balance as one of the main objectives instead of data balance. The second is the absence of a standard fault tolerance mechanism: data considered as important by the applications is marked as such, and then a copy is stored on the separated storage system of the platform. Thus, data replication cannot be leveraged to speed up rescaling operations.

12.1.1 Objectives of the rescaling operations

In light of these differences, the two main goals of the rescaling operations are: speed and load balance.

Speed: Rescaling should be as fast as possible. Whether rescaling is done while the data service is actively being used by applications or when it is idle, fast rescaling will lead to better resource utilization overall by enabling applications to reuse decommissioned resources (when the service is scaled down) and by speeding up the applications that use the service (when the service is scaled up).

Load balance: Part of the data managed by the service may be "hotter" than others (e.g., accessed more frequently). Given that a load metric can be assigned to individual data items or groups of items, such a load should remain balanced across the active nodes on which the service runs. Note that this goal is different from that of *data balance*: in a perfectly data-balanced configuration, nonuniform data accesses (corresponding to having some hot data more frequently accessed than other, colder data) may produce a load imbalance. Having a load-balanced storage system mitigates hotspots, which in turn reduces I/O interference, one of the root causes of performance variability in HPC applications [83, 84, 85, 86].

Contributions previously presented in this manuscript do not fit this problem, and a new approach must be considered to add malleability to HEPnOS.

12.1.2 A need to consider data balance

A possible solution to these challenges would be to use classical load-rebalancing strategies [98], which move data from the most-loaded servers to the least-loaded ones, starting with data with the highest load-to-data-size ratio. This strategy minimizes the amount of data to transfer, while balancing the load on the storage system.

We note, however, that *rescaling* cannot simply be reduced to *load rebalancing*: classical load rebalancing as described above may create data imbalance, leaving some nodes with either much higher or much lower volumes of data than other nodes, for instance when a few (or small) data items have a high load while a large number (or larger) data items have a comparatively lower load. This data imbalance will slow down future rescaling operations. The nodes that are being decommissioned must indeed transfer out all the data they host; thus, if some nodes host more data than others because of data imbalance, some transfers will also be longer than others, globally making the rebalancing operation last longer.

> *We propose an approach considering both load balance and data balance. To enable efficient,* repeated *rescaling operations over a long period, one must jointly (1) optimize load balance, (2) minimize the duration of the current rescaling operation, and (3) ensure data balance to help speed up the following rescaling operations.*

12.2 Formalization

In this section, we formalize the problem of data redistribution during a rescaling operation.

12.2.1 Rescaling operation

Rescaling a distributed storage system consists of either commissioning storage servers or decommissioning storage servers. For simplicity, we do not consider the case of simultaneously commissioning while decommissioning other storage servers, although the results presented in this work can easily be extended to this particular case as discussed in Section 13.4.

Rescaling a distributed storage system has to be done under one main constraint: objects initially stored must be available in the system at the end of the operation. In other words, no data can be lost during the operation.

We assume that the storage service is directed to add or remove specific storage servers, and our challenge is to decide which data to move and where to place it to meet our objective. The definition of this server selection policy is out of the scope of this work since it can depend on external factors, such as the arrival of new jobs that need to take over servers form the existing job. For the same reason, the list of servers to commission or decommission is assumed not to be known in advance.

12.2.2 Parameter description

We consider a homogeneous cluster. Let I be the set of storage servers involved in the operation (including servers that will be commissioned or decommissioned). We denote as I^- the set of storage servers to decommission and as I^+ the set of storage servers to commission. All nodes have a network bandwidth S_{net}. Each storage server has a storage capacity of C. We assume (fast) in-memory storage; *therefore the network is expected to be the bottleneck* of the rescaling operation in this case. The alternative scenario where data storage causes a bottleneck is discussed in Section 12.4.2.

We assume that storage servers can exchange collections of objects — which we call *buckets* — during the rescaling operations. Buckets are considered because rebalancing the storage system with a finer, object-level granularity would take too long due to the sheer number of objects. Since objects are grouped into buckets, few transfer decisions have to be made to move many objects, which reduces the time needed to determine where to transfer each object compared to considering each object individually. Buckets have been used in Ceph [59]. We assume they are not replicated for fault tolerance. We discuss the case of bucket replication in Section 12.4.2. Each bucket has a *size* and a *load*, a generic, user-defined measure of the impact the bucket has on its host. For example, the load of a bucket of objects can be defined as the number of requests for the objects in the bucket over a time period. Let J be the set of buckets. We denote as $Load_j$ and $Size_j$ the load and size of a bucket j, respectively. Each bucket is initially stored on a single storage server. b^0 and b are matrices in $\{0,1\}^{|I|\times|J|}$ representing the placement of the buckets before and (respectively) after the rescaling operation. $b^0_{ij} = 1$ if the bucket j is on the storage server i at the beginning of the rescaling operation. Similarly, $b_{ij} = 1$ if the bucket j is on the storage server i at the end of the rescaling operation.

12.2.3 Problem formalization

Our goal is to minimize at the same time the maximum load per node (load balance), the duration, and the maximum amount of data per node (data balance) (Equation 12.1).

$$\text{Find } b \text{ minimizing } L_{max},\ D_{max}, \text{ and } T_{max} \tag{12.1}$$

$$L_{max} = \max_{i \in I} \sum_{j \in J} b_{ij} Load_j \tag{12.2}$$

$$T_{max} = \max_{i \in I} \left(\sum_{\substack{j \in J \\ b_{ij}=1 \\ b^0_{ij}=0}} \frac{Size_j}{S_{net}}, \sum_{\substack{j \in J \\ b_{ij}=0 \\ b^0_{ij}=1}} \frac{Size_j}{S_{net}} \right) \tag{12.3}$$

$$D_{max} = \max_{i \in I} \sum_{j \in J} b_{ij} Size_j \tag{12.4}$$

$$C \geq \sum_{j \in J} Size_j b_{ij} \qquad \forall i \in I \tag{12.5}$$

$$1 = \sum_{i \in I \setminus I^-} b_{ij} \qquad \forall j \in J \tag{12.6}$$

$$0 = \sum_{j \in J} b_{i^- j} \qquad \forall i^- \in I^-. \tag{12.7}$$

Load balance: The first objective is to reach a distribution of buckets that is load-balanced. We assume that each node is under a load equal to the sum of the load of the buckets that are placed on it. We denote as L_{max} (Equation 12.2) the load of the most loaded node. A cluster is load-balanced when L_{max} is as low as possible.

Although other metrics for load balancing could be considered (*e.g.* variance of the loads across storage servers or the entropy of the load distribution), using the maximum ensures that hotspots will be avoided, while leaving some leeway to optimize the other objectives. Indeed, as long as the maximum load across the cluster does not increase, buckets can be transferred between servers to optimize data balance or left in place to reduce the duration of the operation.

Duration: The second objective is to minimize the duration of the rescaling operation. We denote as T_{max} (Equation 12.3) the duration of the operation. It is the maximum of the time needed to receive data and to send data across all nodes. This reflects that most modern networks are full duplex and thus can send and receive data at the same time without interference. Note that the latency of the network is ignored for simplicity.

Data balance: The third objective is data balance: each node should host comparable amounts of data. We denote as D_{max} (Equation 12.4) the amount of data on the node hosting the most data. A cluster is data-balanced when D_{max} is as low as possible.

Constraints: The above objectives should be minimized while ensuring some constraints. Each node must have the capacity to host the data placed on it (Equation 12.5). Each bucket must be placed on one and only one node (Equation 12.6). No bucket should be hosted by decommissioned nodes (Equation 12.7), since these nodes are leaving the cluster.

12.3 The need for multiobjective optimization

Over-complexifying a problem by considering unneeded parameters and objectives is an easy trap to fall into. In this section, we discuss the relevance of each of the considered objectives and show their impact on the duration of rescaling operations.

12.3.1 Methodology

To study the relevance of each objective, we devise and evaluate several strategies for a rescaling operation and try to minimize various subsets of the objectives.

Problem size and cluster configuration: The optimal solutions for each strategy are obtained by using CPLEX, a solver for mixed-integer programming [129]. Because of the prohibitive compute time needed to get optimal solutions (1 h 40 min on average for one optimal solution on a cluster of up to 8 servers and 32 buckets), we focus on smaller instances of the problem: only 16 buckets on up to 8 nodes. We consider 100 commissions with a cluster of 5 storage servers increased to 8 and 100 decommissions with a cluster of 8 storage servers reduced to 5. Thus, each solution is obtained in 850 ms on average.

Distribution law: We generate buckets with sizes and loads that follow a normal distribution (standard deviation of 40%, for a total of 8 GB hosted on the cluster) to represent a bucket hosting hundreds of objects. Indeed, even if the load induced by a single file in a peer-to-peer setup is known to follow a Mandelbrot-Zipf distribution [130], the central limit theorem shows that the distribution of the load of a bucket can be approximated by a normal distribution with a standard deviation that decreases with the number of files in the collection. The same applies to the size of the buckets.

Initial data placement: The initial placement of the buckets on the nodes is obtained by executing sequentially 9 random rescaling operations (during each operation, the cluster is rescaled to a size randomly selected between 2 and 8 servers) using the same placement strategy as the one studied. This ensures that the initial data placement of the presented results is a consequence of the studied strategy. This way, we study the performance of rescaling operations in a context of successive rescaling operations.

12.3.2 Impact and relevance of each objective

Figure 12.1 respectively presents the load balance, the duration, the data balance, and percentage of the stored data moved during the rescaling, for each of the three following strategies.

- `L`: Data is placed such that L_{max} is minimized. The only objective is to optimize load balance.
- `LT`: The load balancing is relaxed; the data is placed so that L_{max} is within 10% of its optimal value (obtained with `L`) and T_{max} is minimized. Both load balance and speed are optimized.

- `LDT`: The load balancing and data balancing are relaxed, and the duration is minimized. The data is placed so that L_{max} and D_{max} are within 10% of their optimal values, and T_{max} is minimized. Compared to the `LT` approach, this strategy is meant to illustrate the additional impact of data balance.

Load balance

Improving load balance in a distributed storage system helps mitigate I/O interference, which has been shown to be one of the root causes of performance variability in HPC applications [83, 84, 85, 86].

In Figure 12.1, we can observe the performance of `L`, which focuses only on load balancing and is optimal for it. Note that the lower bound is below since it is estimated by the average load per node, and thus it is not a tight lower bound. However, the optimality of the load balance comes at the cost of many data transfers; *at least 85% of the data on the cluster was transferred between nodes in half of the decommissions.* This is also reflected in the duration of operations: the median duration of the decommission is at least 1.8 times longer than with `LDT`.

These results can easily be explained by the fact that `L` solely focuses on finding an optimally load-balanced data placement and is oblivious to the transfers required to create that data placement.

This shows that the duration of the operation should also be taken into account.

Duration

Rescaling a distributed storage system quickly can be important for multiple reasons. If the rescaling is needed to adjust the amount of resources to a varying workload, finishing it quickly ensures that the workload benefits from the newly provisioned resources as soon as possible. Moreover, many platforms have a cost linked to the usage of core-hours, thus; decommissioning and commissioning servers quickly helps minimize the overall cost.

`LT` is a strategy focused on load balancing and fast rescaling.

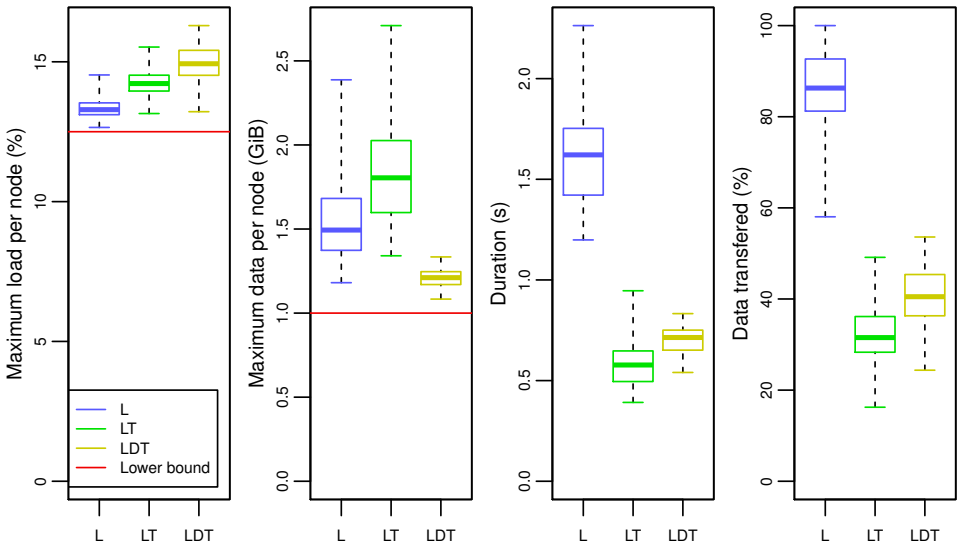
We observe in Figure 12.1a that most of the commissions are done about 3 times faster with `LT` than with `L`, since `LT` moves 3 times less data. Figure 12.1b shows that relaxing the load balancing helps speed decommissions by a factor 1.4 compared with `L`.

However, the strategy does not exhibit stable performance: duration of similar operations (same number of buckets, number of commissioned or decommissioned servers, and initial number of storage servers) can vary by up to 100%. The stability of the duration of rescaling operations can be obtained only by considering a third aspect: *data balance*.

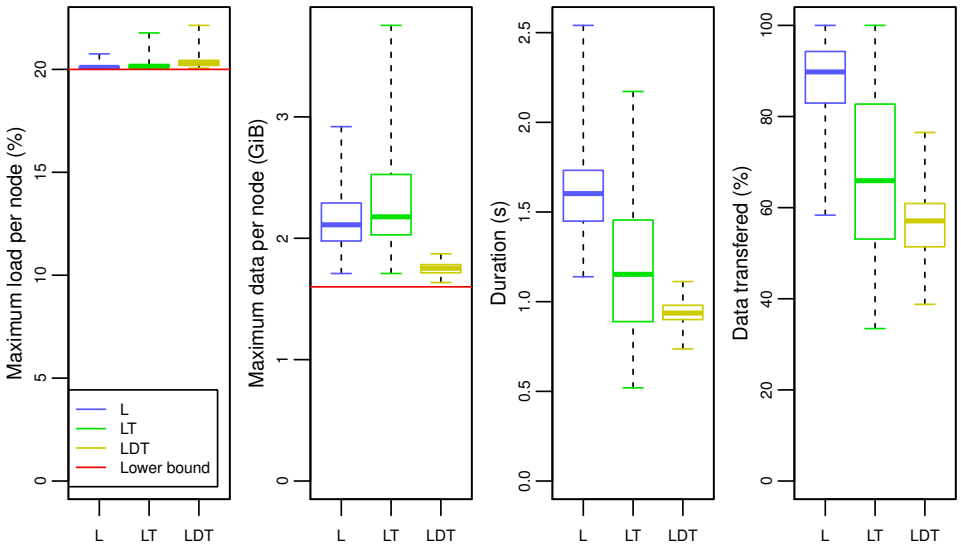
Data balance

Data balancing is needed to speed up decommissions and to stabilize the duration and performance of all rescaling operations.

In the case of a decommission, when the storage is data-balanced, the duration of the operation is independent of the choice of the storage servers that are being removed: all storage servers host the same amount of data. If there is some data imbalance in the storage, the decommission could be faster if only storage servers hosting less data than average are



(a) Commission of 3 servers to a storage system with 5 initial servers



(b) Decommission of 3 servers out of a storage system with 8 initial servers

Figure 12.1: Load balance, data balance, duration, and percentage of data transferred for the strategies `L`, `LT`, and `LDT`. Boxplots represent the minimum, first quartile, median, third quartile, and maximum.

decommissioned. Most likely, however, at least one storage server hosting more data than average is selected to be decommissioned, lengthening the duration of the operation.

In Figure 12.1b we observe that the duration of the decommission with `LT` can be up to 25% shorter than with `LDT`, but it can also be up to 2 times slower in some cases. Overall, `LDT` tries to satisfy all objectives and has less variability in duration, which can greatly help resource managers predict the performance of rescaling operations. However, it comes at the cost of higher load imbalance and longer commissions (Figure. 12.1a): `LDT` requires more data movements to balance the data.

Starting from a data-balanced situation is important for the duration of the rescaling operation, the decommission is faster when data is balanced (`LDT`). However, data balance only has an impact when sending data is the bottleneck: sending tasks are better distributed across servers. To summarize, even if ensuring data balancing during a rescaling operation might deteriorate the duration of the ongoing operation, it is beneficial for the following ones. Enforcing data balancing is a trade-off between short-term speed and long-term efficiency.

12.3.3 Why all three objectives are relevant

The three objectives are often mutually incompatible. For example, the fastest commission duration is not reachable if the cluster must be data-balanced. Besides, there is no objective hierarchy to order these objectives and minimize them one after another. Indeed, depending on the application, some objectives may not be relevant. For instance, a distributed storage that is not a bottleneck for the application using it may not need an ideal load balancing, but the job manager may need stable rescaling duration in order to anticipate the operations.

All three objectives should be taken into account when rescaling a distributed storage system. Thus, the rebalancing algorithm (which is also in charge of moving data out of decommissioned nodes) must consider the load balancing, duration, and data balancing and find an equilibrium between them.

12.4 Related work and discussion

In this section, we present some related work specific to multicriteria optimizations, and comment some aspects of the formalization.

12.4.1 Related work

Rebalancing algorithms: As presented in Chapter 3, many algorithms to rebalance peer-to-peer storage systems have been proposed [98, 99, 100]. However, all these works focus only on one criterion to balance: either the amount of data per node or the load (memory usage, CPU usage, etc.) and intend to do so as quickly as possible. Neither considers both data and load balance.

Consolidation and load balancing of virtual machines: The multicriteria problem studied in this chapter and the next is close to the rebalancing of virtual machines on a cluster [131, 132, 133, 134, 135]. Each virtual machine needs some resources to perform nominally (CPU,

memory, bandwidth, etc.), and each physical machine has limited capacity. Thus, virtual machines should be migrated to avoid overloading physical machines; in addition, as few virtual machines as possible should be migrated to limit the performance degradation. Arzuaga and Kaeli [134] simplify the problem to one dimension by simply adding resource usage metrics and by balancing this sum. However, even if this sum of metrics is well-balanced across the cluster, it may not be the case for each individual metric. Most works on balancing virtual machines [131, 132, 133, 135] are designed to keep the resource usage on each physical machine under a user-defined threshold while migrating as few virtual machines as possible. The work presented in this chapter and the next, however, aims to simultaneously load balance *and* data balance the cluster as fast as possible.

12.4.2 Discussion

Storage bottleneck: The objective T_{max} is written under the assumption of a network bottleneck. However, this formalization can easily be adapted to the case of a storage bottleneck. The particularity of storage devices is that they cannot sustain simultaneous reads and writes at maximum speed thus the duration of the I/O operations has to be modeled as the sum of the time taken to read data and the time taken to write data. Therefore, equation (12.3) should be replaced with equation (12.8).

$$T_{max} = \max_{i \in I} \left(\sum_{\substack{j \in J \\ b_{ij}=0 \\ b^0_{ij}=1}} \frac{Size_j}{S_{write}} + \sum_{\substack{j \in J \\ b_{ij}=1 \\ b^0_{ij}=0}} \frac{Size_j}{S_{read}} \right) \tag{12.8}$$

Replication: In this chapter we focused on the case where buckets are not replicated, which is often the case in state-of-the-art user-level HPC data services, such as HEPnOS. If the storage system replicates buckets for fault tolerance, the focus of the work should be different: instead of ensuring data balance, it should determine what replica to use when transferring buckets. This would be a means to greatly reduce the risk of experiencing bottlenecks on servers sending data, which reduces the importance of data balance. This is an open direction for future work.

Generality: The formalization of the rescaling problem detailed in this chapter is not limited to the malleability of HEPnOS. As long as a system requires load balance and one can define transferable buckets with a load and a size, the rescaling operations of this system can use the proposed formalization. For instance, a data service managing metadata for a storage system could use the same formalization. The metadata itself could be divided in smaller parts (the buckets), and their load can be defined as the number of requests processed. By doing so, rescaling the data service would also distribute the load across the nodes.

Conclusion

In this chapter, we presented the challenge of adding malleability to HPC data services such as but not limited to HEPnOS, a transient storage system for high energy physic applications. We formalized this problem as a multiobjective optimization problem, and we demonstrated the need to consider all three objectives (load balance, data balance, and duration) by showing the consequences for rescaling operations of ignoring one or more of the objectives.

CHAPTER 13

PUFFERSCALE: A RESCALING MANAGER FOR DISTRIBUTED STORAGE SYSTEMS

In the previous chapter, we formalized the problem of rescaling a distributed storage system while balancing the load across the nodes, and presented the challenges to address in order to implement this type of rescaling operations. However, this problem is NP-hard [98] and cannot be solved exactly in a reasonable amount of time in the envisioned situations.

In this chapter we present a heuristic to manage data redistribution during a rescaling operation in a fraction of a second. This heuristic aims to reach a good trade-off between *load balance* and *data balance* for the final data placement and the duration of the rescaling operation.

We design and implement Pufferscale, a generic rescaling manager that can be used in microservice-based distributed storage systems. The roles of Pufferscale are to (1) track the data hosted on each node, (2) schedule the data migrations using the previous heuristic, and (3) start and stop microservices on compute nodes that are being respectively commissioned and decommissioned.

We show the performance and usability of Pufferscale in experiments on the French experimental testbed Grid'5000. We show in practice that one can consider both load balancing and data balancing with negligible impact on the quality of both and with only a 5% slowdown compared with strategies ignoring load balancing. Moreover, we showcase the use of Pufferscale with a combination of microservice components used in the HEPnOS storage systems, with the goal of enabling malleability in HEPnOS.

13.1 Fast approximations with a greedy heuristic

Calculating exact solutions for our multiobjective problem for realistic deployment scales is not feasible: it simply takes too long (see Chapter 12, Section 12.3.1). Thus, a heuristic is needed to get fast approximations that are usable in practice.

13.1.1 Challenges

Traditional load rebalancing is usually a bi-objective optimization problem: the load must be balanced, but it should be done quickly. In the previous section, we showed that, to ensure efficient decommission duration, the data redistribution done during rescaling should consider three objectives simultaneously: load balance, data balance and duration of the operation.

As in most multiobjective optimization problems, there is not just one optimal solution. The goal is to provide an acceptable trade-off. Moreover, because of the expected scale of the storage system (a few hundreds to thousands of nodes on a supercomputer) the rebalancing algorithm must compute a solution quickly. This makes computing an exact solution unusable. We need a fast heuristic that can be parameterized to provide solutions that balance the objectives according to the needs of each application.

13.1.2 Heuristic

The heuristic we design is a greedy algorithm inspired by the longest-processing-time-first rule and usual load-rebalancing mechanisms [136]. A greedy algorithm makes it possible to start transferring buckets before a complete solution is computed.

It works in three steps: (1) estimate the target values for the load balancing, data balancing, and duration; (2) select buckets that will not be moved (which we call "fixed" buckets); and (3) allocate the remaining buckets to destination storage servers (which may be the storage servers they are already on).

Determination of target values for metrics

The heuristic is designed to provide solutions that have their metrics as close as possible to target values; thus the computation of the target values is critical. We do not set the targets to 0 for two reasons, even if 0 is a relevant choice because we aim to minimize the objectives. First, setting targets that are reachable or almost reachable allows us to normalize the metrics with respect to these targets. Second, it prevents an imbalance between the objectives: none of the load balancing and data balancing metrics can reach 0 (since there are buckets on the storage, the lower bounds for these metrics is positive), but the duration can be 0 in case of a commission (leaving new nodes empty is valid). Thus, setting realistic targets helps avoid biases toward some objectives.

$$L_t = \frac{\sum_{j \in J} Load_j}{|I \backslash I^-|} \tag{13.1}$$

$$D_t = \frac{\sum_{j \in J} Size_j}{|I \backslash I^-|} \tag{13.2}$$

$$D_i = \frac{\sum_{j \in J} Size_j}{|I \backslash I^+|} \tag{13.3}$$

$$T_t = \begin{cases} \max\left(\frac{|D_t - D_i|}{S_{net}}, \frac{D_i}{S_{net}}\right) & \text{if } I^- \neq \emptyset \\ \max\left(\frac{|D_i - D_t|}{S_{net}}, \frac{D_t}{S_{net}}\right) & \text{if } I^+ \neq \emptyset \end{cases} \tag{13.4}$$

The target load per storage server L_t and the target amount of data per server D_t at the end of the rescaling operations are respectively the average load per storage server (Equation 13.1) and the average amount of data per storage server (Equation 13.2). The target duration T_t for the rescaling operations is more challenging to estimate. We approximate the initial data placements as perfectly data balanced; that is, each storage server initially hosts the same

amount of data D_i (which can be computed by using Equation 13.3) and will host D_t at the end of the operation. For a decommission (upper part in Equation 13.4), the operation should last at least the time needed to empty the decommissioned servers of their buckets and at least the time needed for the remaining servers to receive those buckets. For a commission (lower part in Equation 13.4), the duration is the maximum of the time needed to add enough buckets to new storage servers and the time required to send those buckets.

Fixing buckets

foreach *Storage server* i **do**
  $\forall j \in J$, set $b_{ij} = 0$;
  Let $J_i = \{j_1, j_2, ...\}$ be the buckets initially on i ordered by decreasing norm N (Eq. 13.7);
  Find the largest n such that $\sum_{k=1}^{n} Size_{j_k} \leq D_t$ and $\sum_{k=1}^{n} Load_{j_k} \leq L_t$;
  Allocate $\{j_1, ..., j_n\}$ to i: $\forall k \in [1, n], b_{ij_k} = 1$;
  Add $\{j_1, ..., j_n\}$ to J^a;

Algorithm 13.1: Fixing buckets

We call *allocated* a bucket for which the algorithm has determined a destination server. Among these buckets, we call *fixed* the ones that will not move from their current location. Let J^a be the set of allocated buckets during the execution of the algorithm. The fixing phase (Algorithm 13.1) consists of determining the buckets that will not be moved and adding them to J^a.

$$S_{Load} = \sum_{j \in J} Load_j \tag{13.5}$$

$$S_{Size} = \sum_{j \in J} Size_j \tag{13.6}$$

$$N(j) = \sqrt{\left(\frac{Load_j}{S_{Load}}\right)^2 + \left(\frac{Size_j}{S_{Size}}\right)^2} \tag{13.7}$$

The idea behind the fixing phase (Algorithm 13.1), is to avoid transferring the “largest” buckets. Buckets are ordered by decreasing norm N (Equation 13.7), which is a combination of their load and size normalized by the total load and total size on the storage system (Equation 13.5, Equation 13.6). The fixing algorithm allocates to the node the largest buckets that fit within the target for load balancing and data balancing. The remaining buckets will be allocated by Algorithm 13.2.

1 Sort unallocated buckets ($J \backslash J^a$) by decreasing N (Eq. 13.7);
2 **foreach** *Bucket* j **do**
3 Let i^0 be the initial host of bucket j, $b^0_{i^0 j} = 1$;
4 Find the server i that minimizes the penalty $P(i) + P(i^0)$ (computed assuming j has been allocated to i) ;
5 Allocate j to i ($b_{ij} = 1$, $b^0_{i^0 j} = 0$);
6 Add j to J^a;

Algorithm 13.2: Allocation of buckets

Allocation of remaining buckets

$$P(i) = \left(\frac{\sum_{j \in J^a} b_{ij} Load_j}{L_t}\right)^3 + \left(\frac{\sum_{j \in J^a} b_{ij} Size_j}{D_t}\right)^3 + \frac{1}{2(S_{net} T_t)^3} * \max\left(\sum_{\substack{j \in J^a \\ b^0_{ij}=0 \\ b_{ij}=1}} Size_j, \sum_{\substack{j \in J^a \\ b^0_{ij}=1 \\ b_{ij}=0}} Size_j\right)^3 \quad (13.8)$$

The allocation phase (Algorithm 13.2) follows a greedy strategy: the buckets, taken in order of decreasing norm N (Equation 13.7) are placed on the servers where they minimize a penalty function (Equation 13.8). The penalty is the sum of the cube of the value of each objective divided by their targeted value. Thus, the bigger the value of an objective compared with its targeted value, the higher the penalty. The effect of this penalty function is to minimize the objective that is the least optimized.

13.1.3 Enabling heuristic tuning

Since no ideal rebalancing exists for all situations, we added weights to provide the possibility to adapt the importance of each objective to the needs of the application. Let W_L, W_D, and W_T (such that $max(W_L, W_D) = 1$ and $W_T > 0, W_L > 0, W_D > 0$) be the weights for the load balancing, data balancing, and duration of the transfers, respectively. The higher the weight, the more important the objective.

$$T_{tw} = \frac{D_T}{W_T} \tag{13.9}$$

$$L_i = \frac{\sum_{j \in J} Load_j}{|I \backslash I^+|} \tag{13.10}$$

$$L_{tw} = \begin{cases} \frac{L_t}{W_L} & \text{if } W_T < 1 \\ \frac{1}{W_L}((1 - \frac{1}{W_T})L_i + \frac{L_t}{W_T}) & \text{otherwise} \end{cases} \tag{13.11}$$

$$D_{tw} = \begin{cases} \frac{D_t}{W_D} & \text{if } W_T < 1 \\ \frac{1}{W_D}((1 - \frac{1}{W_T})D_i + \frac{D_t}{W_T}) & \text{otherwise} \end{cases} \tag{13.12}$$

$$N_w(j) = \sqrt{\left(\frac{Load_j * W_L}{S_{Load}}\right)^2 + \left(\frac{Size_j * W_D}{S_{Size}}\right)^2} \tag{13.13}$$

With these weights, we can adjust the target values in L_{tw}, D_{tw}, and T_{tw} for each objective. The target duration T_{tw} is simply scaled by its weight (Equation 13.9) because the minimum for the duration is 0. But because the needed transfers to reach the targets L_t and D_t are estimated to last at least T_t, the targets for the load balancing and data balancing are adjusted as the weighted mean between the initial balancing and the targeted balancing. Then the data balancing and load balancing each are multiplied by their weights (Equation 13.11 and Equation 13.12). At least one of the weights W_L or W_D is set to 1 so that the heuristic aims to optimize at least one objective. A weighted norm is also used to order the buckets as shown in Equation 13.13.

This heuristic has been designed to be able to start data transfers quickly. Indeed, even if numerous nodes are involved in rescaling operations, the duration of the operations could be very short depending on the size of the buckets. Thus starting the data transfers quickly shorten the duration of rescaling operations. This is why a greedy heuristic was designed: every choice made by the heuristic is final, thus some transfers could start before the heuristic completes. Similarly, relatively few computations are needed to decide the destination of a bucket, which keeps the time needed to run the heuristic low (see Section 13.3.1).

Other approaches could be considered to solve this multi-objective problem. In particular, we plan to study the use of machine learning to replace this heuristic.

> *The greedy heuristic proposed in this section creates a data redistribution with a trade-off between data balance, speed, and load balance. Moreover, this trade-off can be adjusted to the requirements of the user with a careful selection of the weights assigned to each objective.*

13.2 Pufferscale

To evaluate our proposed heuristic, we implemented Pufferscale, a rescaling service developed in the context of the Mochi project [121]. This project aims at boosting the development of

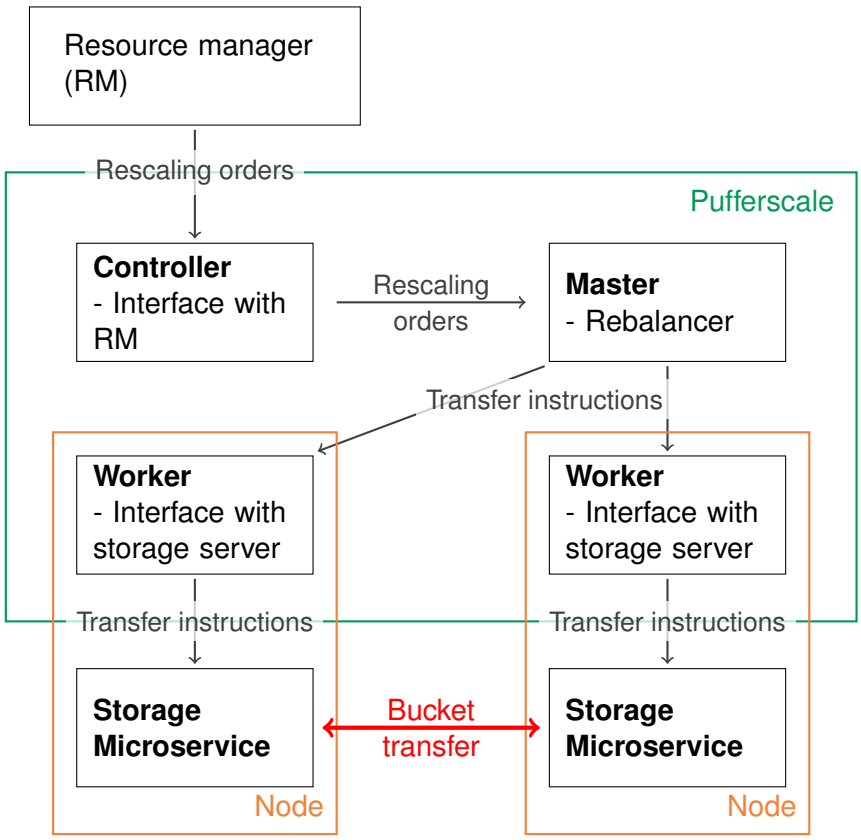


Figure 13.1: Architecture of Pufferscale.

HPC data services thanks to a methodology based on the composition of building blocks that provide a limited set of features, accessible through remote procedure calls (RPC) and remote direct memory accesses (RDMA) and together with a threading/tasking layer [11]. As such, Pufferscale can be composed with other Mochi microservices to be integrated in various larger data services. Pufferscale's roles in such data services are to (1) keep track of the location of buckets, (2) schedule and request the migration of buckets from a node to another using the presented heuristic, and (3) request the deployment and shutdown of microservices on nodes that have to be respectively commissioned or decommissioned.

13.2.1 Overview of Pufferscale

Pufferscale consists of the following components (Figure 13.1). The *controller* acts as an interface to send rescaling orders to the master from outside of the service. The *master* is the component that contains the heuristic and decides where each bucket should be placed during rescaling operations. The *workers* are the interfaces between the master and the *microservices* available on a given node. They are able to start and stop microservices on their node and forward the transfer instructions from the master to the corresponding microservices. The composition with the microservices is done by dependency injection. That is, the microservice

```
1  // Start the thallium engine that wrap Margo, Argobots and Mercury for C++
2  thallium::engine engine(cfg_protocol, THALLIUM_CLIENT_MODE, false, 0);
3
4  // Define the address of the master and its provider identifier
5  MasterInfo masterinfo;
6  masterinfo.m_address = ...;
7  masterinfo.m_provider_id = ...;
8
9  // Create the controller
10 Controller controller = Controller(engine, masterinfo);
11
12 // Retrieve information about all providers managed by Pufferscale
13 unordered_map<WorkerInfo,vector<ProviderInfo>,WorkerInfoHash> all_providers;
14 all_providers = controller.list_all_providers();
15
16 /** Decommission microservices **/
17 // Select microservices to decommission
18 unordered_map<WorkerInfo,vector<ProviderInfo>,WorkerInfoHash> to_decommission;
19 // Fill to_commission as needed
20
21 // Decommission the workers
22 controller.decommission(to_decommission);
23
24 /** Commission microservices **/
25 // Indicate which providers to commission on nodes
26 // Each microservice is identified by its name
27 unordered_map<WorkerInfo,vector<string>,WorkerInfoHash> to_commission;
28 // Fill to_commission as needed
29
30 // Commission the specified microservices
31 controller.commission(to_commission);
```

Listing 13.1: Simple example of a controller of Pufferscale. After its initialization it can exchange informations with the master to get the list of microservices managed by Pufferscale, and issue commission and decommission orders.

registers callbacks that the workers can call to request the migration of a bucket or ask for information about the buckets present in the storage microservice.

Pufferscale is not aware of the nature of the data that it manages. It also does not handle data transfers itself. In the experiments presented in Section 13.3.3, the transfers are performed by REMI, Mochi's REsource Migration Interface [137], another microservice designed specifically to enable efficient file transfers across nodes using RDMA.

Pufferscale is written with about 3500 lines of C++ code and is implemented by using Mercury [138] for remote procedure calls and Argobots [139] for thread management.

The implementation of Pufferscale in itself did not pose any significant technical challenge. There were technical challenges, however, when combining Pufferscale, REMI, and SDSKV (a single-node key-value store) for the evaluation in Section 13.3.3. One of them was reaching good transfer bandwidth. This was done by tuning how the components shared threads; by

```
1  // Start the thallium engine
2  thallium::engine engine(cfg_protocol, THALLIUM_SERVER_MODE,
3      cfg_progress_thread, cfg_rpc_xstreams);
4
5  // Create the master
6  Master* master = new Master(engine, cfg_provider_id_master);
7  // Plan the destruction of the master
8  engine.push_finalize_callback([master]() { delete master; });
9
10 // Configure a microservice called "sdskv"
11 master->configure_provider("sdskv",
12     cfg_weight_load,    // Weight of the heuristic for load balance
13     cfg_weight_data,    // Weight of the heuristic for data balance
14     cfg_weight_xfers,   // Weight of the heuristic for speed
15     cfg_use_disk,       // Wether the microservice stores data on disk
16     cfg_use_memory,     // Wether the microservice stores data in memory
17     cfg_par_migrations);// Maximum number of concurrent transfers per worker
```

Listing 13.2: Example of code required for the master of Pufferscale. The master only requires a light configuration to work.

experimenting with mmap strategies vs read/write strategies, along with different pipelining configurations. We voluntarily omit these technical details in this chapter since they are only relevant to the hardware we used, and since they are not meaningful for the problem studied in this work, for which the real challenge is of algorithmic nature.

13.2.2 Using Pufferscale

Listings 13.1, 13.2, and 13.3 present examples of utilization of Pufferscale taken from the code used for the experiments in Section 13.3.3.

Controller: The controller is the interface used to instruct Pufferscale to execute a rescaling operation. Thus, the controller is the only component of Pufferscale that is not autonomous once initialized. By communicating with the master, it can provide information about the microservices managed, as well as information about the buckets stored on the microservices. Rescaling instruction are given to Pufferscale through the controller that will forward them to the master (see Listing 13.1)

Master: The master is the simplest component to integrate as a user: it only needs to be initialized by providing the configuration of each of the microservices that will be managed by Pufferscale. The configuration parameters are simple, they consist in the weights used to tune the heuristic, whether the microservice stores data on disk and in memory, and lastly, the number of simultaneous bucket transfers that each worker can launch (see Listing 13.2).

```
// Start the thallium engine
thallium::engine engine(cfg_protocol, THALLIUM_SERVER_MODE,
    cfg_progress_thread, cfg_rpc_xstreams);

// Define the address of the master and its provider identifier
MasterInfo masterinfo;
masterinfo.m_address = ...;
masterinfo.m_provider_id = ...;

/** Initialize the worker **/
Worker* worker = new Worker(engine, cfg_provider_id_worker);
// Plan the destruction of the worker
engine.push_finalize_callback([worker]() { delete worker; });

// Configure the worker
worker->configure_worker(cfg_memory_capacity, cfg_network_bandwidth);
worker->configure_disk(cfg_disk_capacity,
    cfg_disk_read_bw, cfg_disk_write_bw);

// Register the provider "sdskv"
worker->register_provider("sdskv",
    nullptr,            // Optional arguments
    initiate_provider,  // Callback to start the microservice
    terminate_provider, // Callback to stop the microservice
    pre_rescaling,      // Callback to run before a rescaling
    post_rescaling);    // Callback to run after a rescaling

/** Indicate microservices already running on the worker **/
// To be done for each microservice running
ProviderInfo pi = {provider_id_microservice, "sdskv", microservice_address};
// Notify this microservice to the worker
worker->manages_provider(pi);

// Notify the worker about the buckets on the microservice
// To be done for each bucket
BucketTag rt;
rt.m_service_name = "sdskv";
rt.m_bucket_id = bucket_id;
worker->manages(pi,       // Identifier of the microservice
    rt,                   // Identifier of the bucket
    migration_callback,   // Callback to migrate the bucket to another worker
    update_meta_callback, // Callback to update the metadata about the bucket
    nullptr);             // Optional arguments

// Once initialized, the worker notify its presence to the master
worker->join(masterInfo);
```

Listing 13.3: Initialization of the workers of Pufferscale. The worker mostly relies on callbacks to follow the instructions of the master. It can create and terminate microservices with the callbacks `initiate_provider` and `terminate_provider`, and gives the order to migrate data with the callback `migration_callback`.

Worker: The workers connect Pufferscale to the microservices running on a node. Most of their interactions are done through callback that are defined during the initialization of the worker (see Listing 13.3). For each microservice managed by Pufferscale, two callbacks must be specified: one to start a microservice on the node managed by the worker, and one to stop a microservice. Then, each bucket stored in microservices on the node must be registered into Pufferbench. During this operation, two callbacks must be defined. The first one returns metadata about the bucket so that Pufferscale can know how much data is stored onto a node or within a microservice. The second callback is called when Pufferscale gives the order to transfer a bucket from a worker to another. Pufferscale does not do the transfer itself, it simply issue instructions for and efficient rescaling operation. Upon reception of a bucket, a microservice must register the bucket into Pufferscale.

13.3 Evaluation

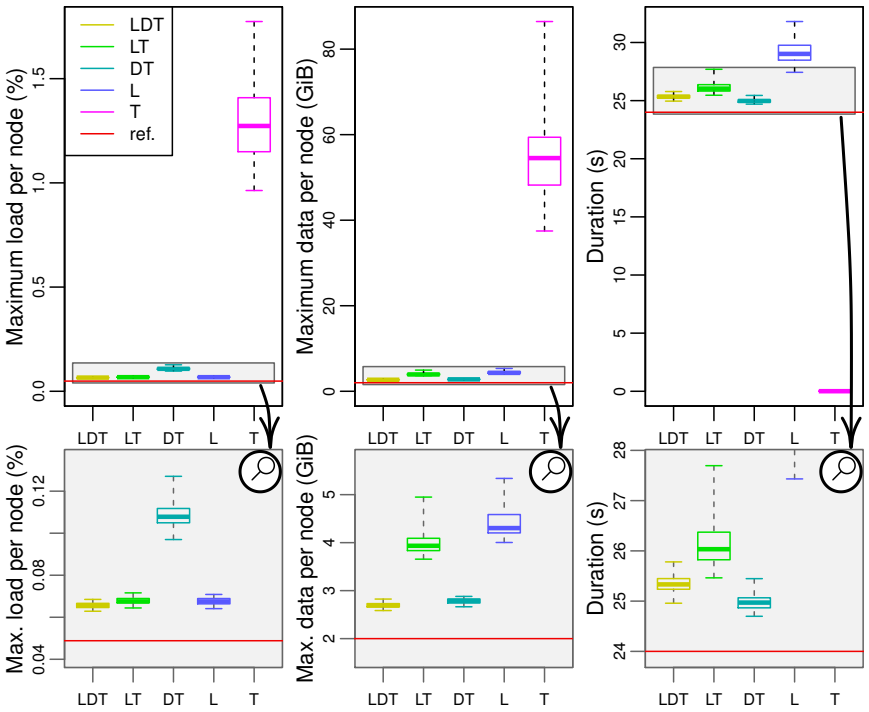
In this section, we evaluate the heuristic at a large scale using emulation. Then we showcase the use of Pufferscale to build the core of a real malleable storage system. For the first goal, we use a "dummy" storage microservice to evaluate Pufferscale's heuristic without making actual data transfers. We then use Pufferscale in a real-world setting, using the set of microservices used in the HEPnOS data service described in Chapters 3 and 12.

13.3.1 Evaluation of the heuristic

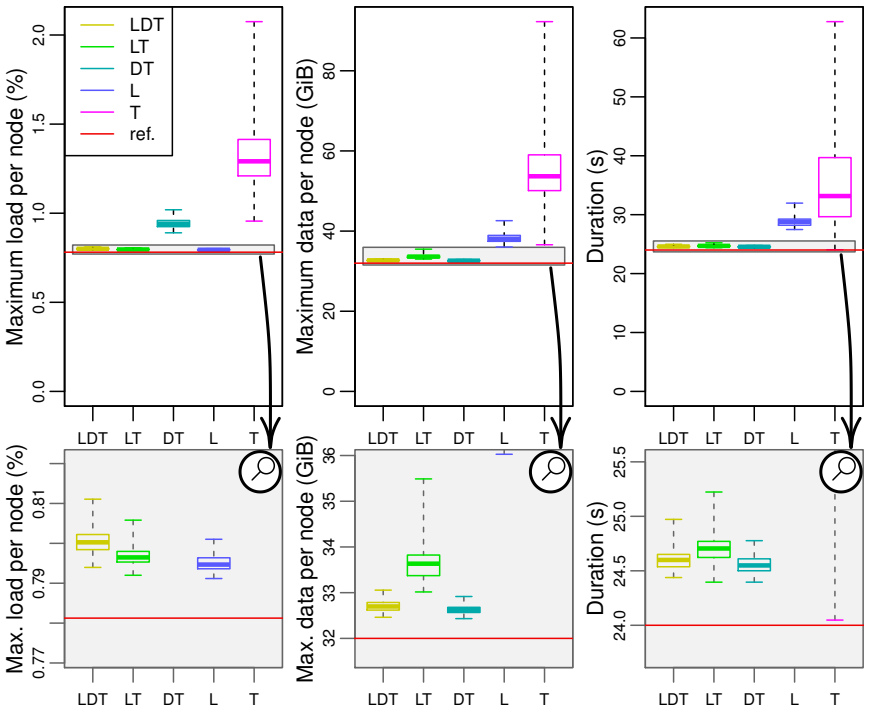
Setup

To evaluate the heuristic and its implementation at large, realistic scales, we emulate a storage system by implementing a "dummy" storage microservice that does not actually store or transfer data. Since no data is actually migrated between any two instances of this microservice, we estimate the duration of the rescaling using Equation 12.3 defined in Chapter 12. This setup allows us to scale up beyond the number of physical nodes available, by emulating multiple virtual storage servers on each physical node.

This distributed service can scale from 128 storage servers up to 2,048 storage servers on a 28-node cluster of the French Grid'5000 testbed [17]. Of the 28 nodes, one acts as a master, and one as a controller, and the other 26 each host up to 80 emulated storage servers. With this setup, we emulate 8,192 buckets with a load distribution following a normal law with 40% standard deviation proportionally adjusted to reach a total load of 100%. Similarly, the amount of data stored in each bucket follows a normal law with 40% standard deviation for a total of 4,096 GiB of data on the storage system. We distribute the buckets by doing 25 random rescaling operations as warm up with the same rescaling strategy as the one studied. Then, we consider the following rescaling scenarios: commission of 1,920 nodes to a storage system of 128 nodes, commission of 64 nodes to a storage system of 256 nodes, decommission of 1,920 nodes out of a storage system of 2,048 nodes, and decommission of 64 nodes out of a storage system of 320 nodes. Each rescaling operation is executed 100 times with 9 random rescaling operations in between two recorded ones.

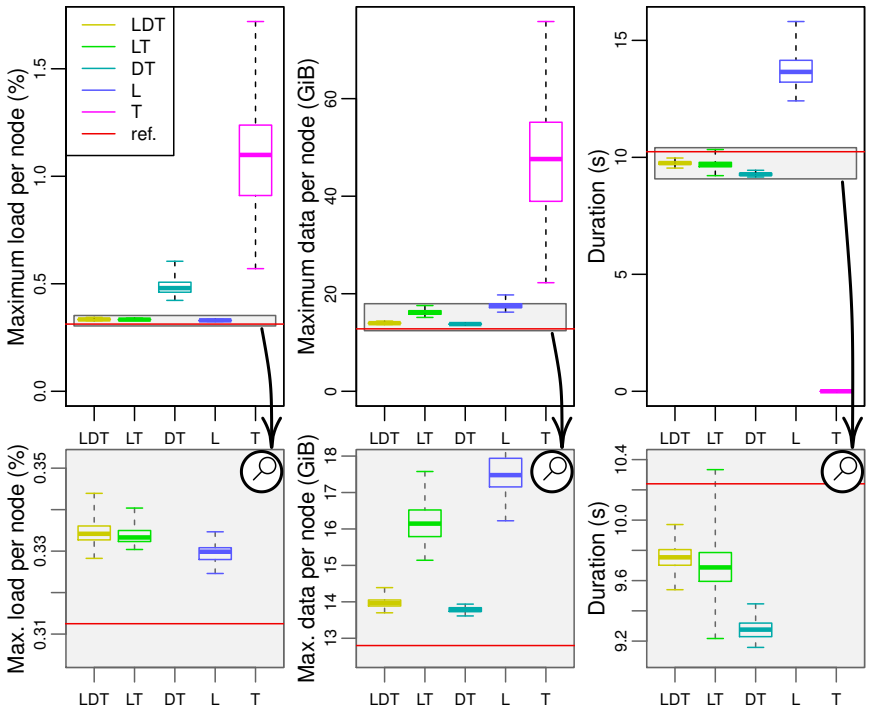


(a) Commission of 1920 to a cluster of 128

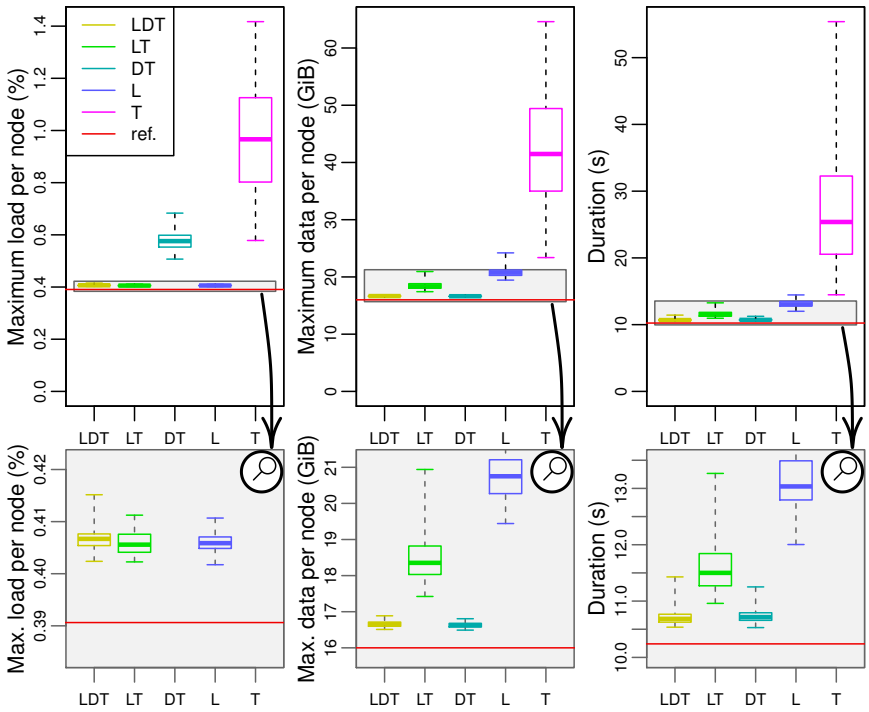


(b) Decommission of 1920 to a cluster of 2048

Figure 13.2: Maximum load, amount of data per node, and duration of the commission of 128 nodes to a cluster of 1920 nodes and their decommission.



(a) Commission of 64 to a cluster of 256



(b) Decommission of 64 to a cluster of 320

Figure 13.3: Maximum load, amount of data per node, and duration of the commission of 64 nodes to a cluster of 256 nodes and their decommission.

Metrics and baseline: We record the load balance, the data balance, and the amount of data received and sent that allows us to estimate the duration of the data transfers under the best conditions.

In Figure 13.2 we compare the load balance, data balance, and duration of 5 strategies: `LDT` (optimization of all three objectives), `LT` (optimization of the load balance and of the duration of transfers), `DT` (optimization of data balance and of the duration of transfers), `L` (sole optimization of the load balance), and `T` (optimization of the duration of the current rescaling operation only). Each of the strategies is obtained by modifying the weights in our heuristic.

We could not compare fairly our work with other works on rebalancing, because the latter do not comply with the strong constraints of the decommissions: nodes being decommissioned must have all their data transferred out to the other nodes, a constraint that is not enforced by other rebalancing algorithms. Moreover, to the best of our knowledge, there is no distributed storage system designed to be co-located with HPC applications that could serve as reference. Distributed storage systems designed to be co-located with applications exist in the cloud, like HDFS, but their rescaling mechanism rely on data replication (that is unneeded for our use-case) and are not optimized for speed (see Chapter 7).

Instead, we added computed comparison points (*ref.*) on each of the graphs: for load balance and data balance we added the lower bounds (L_t, Equation 13.1, and D_t, Equation 13.2). For duration, we added the lower bound of the duration of the transfers needed to transition from a perfectly data-balanced storage system to another perfectly data-balanced storage system (T_t, Equation 13.4).

Results

`T`, the strategy optimizing the duration of the current rescaling operation, optimizing solely for the short-term has a negative impact for the following operations. The `T` strategy, as expected, does not transfer data during the commissions (Figure 13.2a and Figure 13.3a) and thus exhibits instantaneous commissions. However, the decommission duration (Figure 13.2b and Figure 13.3b) can vary by up to 300%. The `T` strategy focuses only on minimizing the duration of the rescaling operation taking place, and thus exhibits a high variability on all metrics. During commission operations, no data is moved, leading to an instantaneous operation, but empty new nodes. During a decommission, only the data on the nodes leaving is moved to other nodes without considering the load and data hosted by the destination. Because of this, the load and data stored on each node increase until the node is decommissioned. Thus, depending on which nodes get decommissioned, the load-imbalance (*resp.* data-balance) can increase if the most loaded node (*resp.* node hosting the most data) is not decommissioned or potentially decrease if the most loaded node is decommissioned. Because the selection of nodes to decommission is random, this creates a high variability for the load and data balance metrics. Similarly, the duration of the decommission depends mostly on the maximum amount of data hosted per decommissioned node, and thus varies because of the data-imbalance.

Overall, the quality of the load balancing for `LT` and `L`, and of the data balancing for `DT` compared with their respective lower-bounds depends on the average number of buckets on the servers at the end of the operation. It is on average within 2% when there are 128 servers after the rescaling operation (Figure 13.2b), within 4% when 256 servers are left (Figure 13.3b), within 8% when 320 servers are left (Figure 13.3a), and within 40% when there are 2,048

servers at the end of the operation (Figure 13.2a). Such a difference between the lower bounds and the obtained results is explained by the granularity of the load and data stored per node: the 8,192 buckets cannot be subdivided to perfectly balance the corresponding objective. The lower bounds, however, are the average per node, ignoring the granularity of the metrics.

`LDT` combines the benefits of the `LT` and `DT` strategies, without any major drawbacks. Its load balance is not significantly different from that of `L` or `LT`, and its data balance is close to that of `DT`. It can be up to 5% slower than `DT` since `LDT` has more transfers to do in order to maintain both the data balance and the load balance.

Moreover, compared with `LT`, the duration range (difference between the longest and shortest operation) of `LDT` is shorter by 32% to 70%, highlighting the fact that data balancing is needed to make operations faster (Figure 13.2a and Figure 13.3b) and have a more stable, and thus more predictable rescaling duration.

Computation time of the heuristic

The computation needed for the heuristic to compute the necessary data transfers lasts between 89 ms for the commission of 64 nodes to a storage cluster of 256 initial nodes to compare with the 10 seconds needed for the execution of the rescaling operation, and 612 ms for the commission of 1,920 nodes to a storage cluster of 128 initial nodes while the operation lasts 25 seconds. The impact of the duration of this computation can be reduced by starting the transfers of buckets as soon as their destination is decided by the heuristic.

> *We showed in the previous chapter that considering both data balance and load balance is needed to ensure fast rescaling operations on the long term. Experiments showed that it is possible to consider both, without major drawbacks compared to only considering one of them.*

13.3.2 Impact of the initial data balance

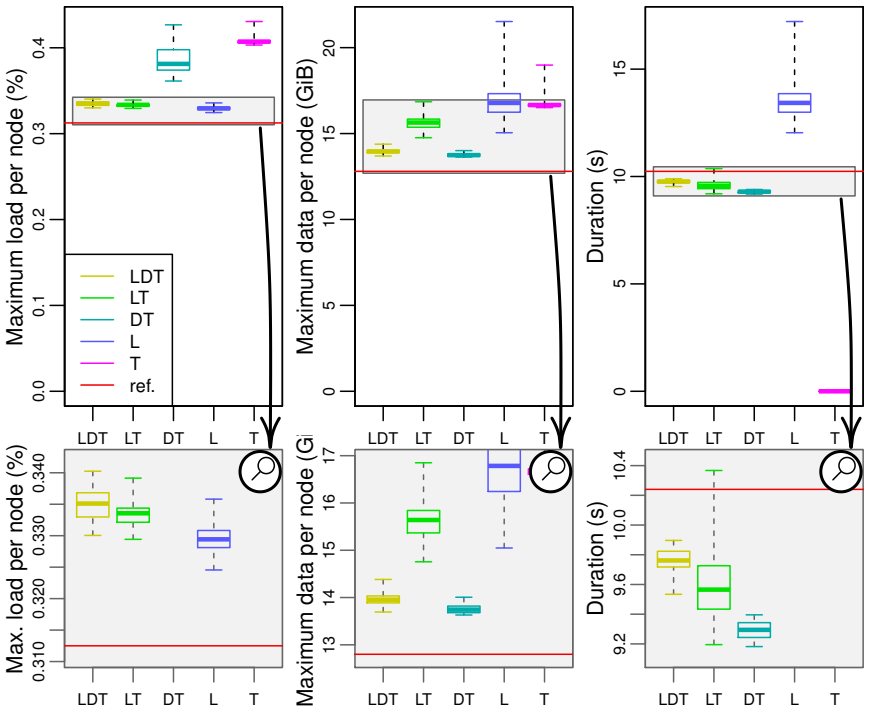
In this section, we study how the initial data balance impacts the various strategies for rescaling.

Setup

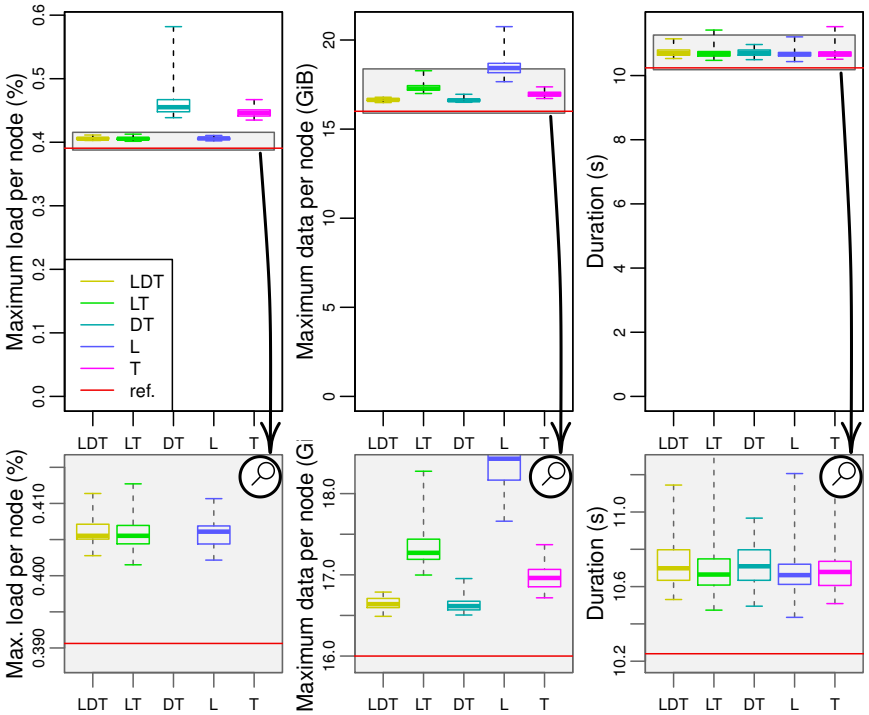
Using the same setup as in the previous experiment, we conduct a different set of measurements. After a warm-up of 25 operations that follow the `LDT` strategy, we switch the strategy and perform one rescaling operation. This ensures that the bucket placement before the last operation is data-balanced and load-balanced. Each measure is repeated 50 times with newly generated bucket sizes and loads.

Results

By looking at the results of the commission (Figure 13.4a) and comparing them with those of the previous experiment (Figure 13.3a), we can see that starting from a data-balanced bucket placement has no impact on the commission operation for all strategies except `T`. `T` has better



(a) Commission of 64 to a cluster of 256



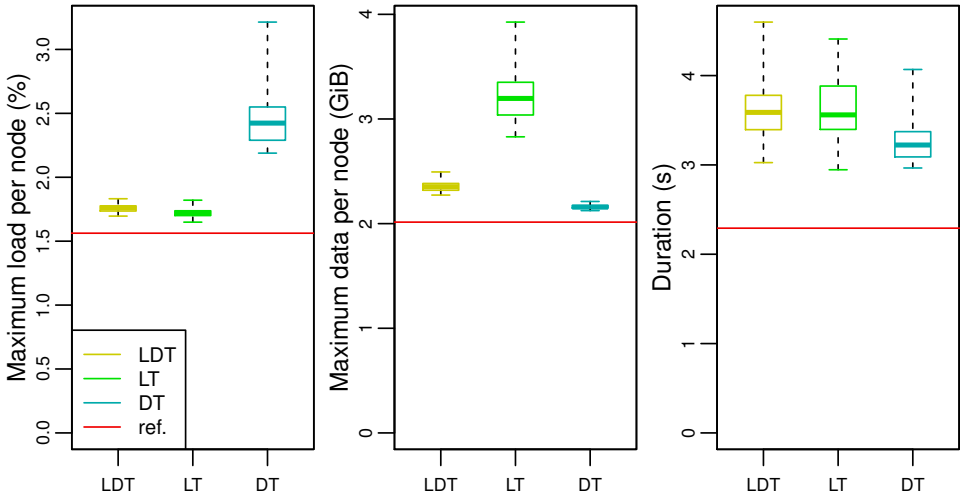
(b) Decommission of 64 to a cluster of 320

Figure 13.4: Maximum load, amount of data per node, and duration of the commission of 64 nodes to a cluster of 256 nodes and their decommission starting from a load and data-balanced data placement.

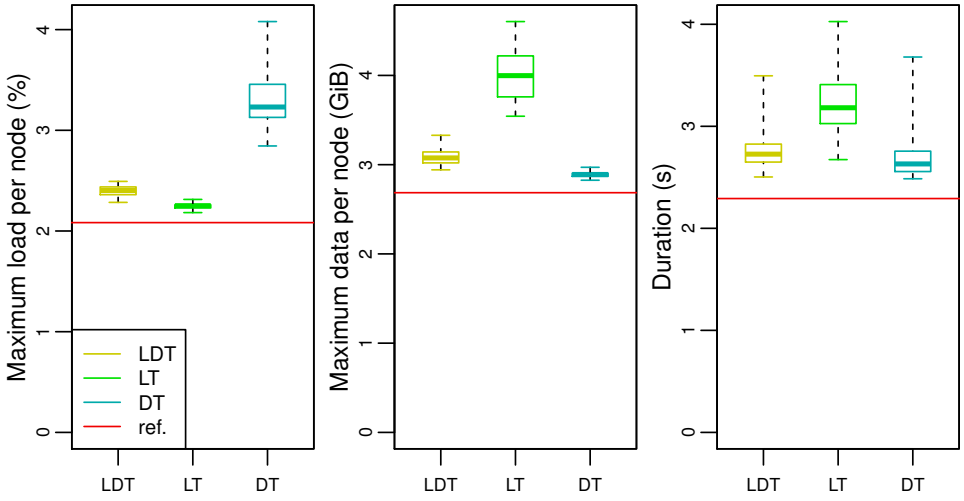
load balancing and data balancing compared with the previous experiments because the imbalance created by a single operation is less than the one accumulated over successive rescaling operations.

In contrast, we can observe that an initially data-balanced bucket placement has an important impact on the duration of the decommission (Figure 13.4b and Figure 13.3b): all strategies have a similar duration.

> *These results show that, independently of any other value, enforcing data balance is required to improve the duration of rescaling operation on the long term.*



(a) Commission of 16 to a cluster of 48



(b) Decommission of 16 from a cluster of 64 servers

Figure 13.5: Maximum load, amount of data per node, and duration for two rescaling operations of the composition of Pufferscale, SDSKV, and REMI.

13.3.3 Pufferscale in HEPnOS

In this section, we showcase the use of Pufferscale with a composition of microservices corresponding to the HEPnOS use case described in the introduction. We composed SDSKV [140], an in-memory, single-node key-value store acting as a storage microservice, REMI, a microservice designed to efficiently transfer files between nodes, and Pufferscale, to build the basis of an elastic version of HEPnOS. Contrary to the previous experiments, databases are transferred between nodes, and the duration of the rescaling operations is recorded.

The rescaling operations of this composition were evaluated on the *paravance* cluster of the Grid'5000 testbed. This cluster is composed of 72 nodes, each with 16 cores, 128 GiB of RAM, and a 10 Gbps network interface. At the maximum size, 64 nodes were used to host databases, and another one node served as controller to issue rescaling orders. Hosted by the SDSKV microservice instance were 256 databases, acting as buckets, each with a load following a normal distribution (with 40% standard deviation) and a size following a normal distribution with a mean of 512 MiB and a standard deviation of 40%. Similarly to the previous evaluation (Section 13.3.1), we distributed the databases by doing 25 random rescaling operations as warm up. Then, each rescaling operation was executed 50 times with 9 random rescalings in between two recorded ones. We add the same references on the figures, with the difference that the network bandwidth is the one recorded when benchmarking RDMAs on the cluster: 900 MiB/s. The network bandwidth is not maximized because of the emulation of RDMAs over a TCP network as well as some overhead in libfabric's socket provider used by Mercury.[1]

The load balance, data balance, and duration of the rescaling operations are presented in Figure 13.5. Comparing `LT` and `LDT`, we observe trends similar to that of Chapter 12 Section 12.3: because `LDT` considers the data balance, its decommissions are on average 13% faster than `LT`, but there are no significant changes for the commissions. However, the duration variability is larger with the composition because of actual data transfers (neither the network nor the scheduling of the databases transfers is perfect, which adds interference). The variability is likely to be increased by the simulation of RDMAs over TCP network. The data balancing of `LDT` is on average 10% worse than `DT`, but it does not have a significant impact on the duration of the decommissions: both strategies ensure similar duration. In the case of the commission, `LDT` is slower than `DT` because of a higher number of database transfers that induces more network interference.

> *An overall conclusion of all these experiments is the following: it is worth considering data balancing in addition to load balancing for rescaling storage systems. This helps reduce the rescaling duration with a negligible impact on load balance when sending data is the bottleneck of the rescaling, without any negative impact on the duration in the other cases.*

[1] `https://ofiwg.github.io/libfabric/master/man/fi_provider.7.html`: "Socket [...] This provider is not intended to provide performance improvements over regular TCP/UDP sockets, but rather to allow developers to write, test, and debug application code even on platforms that do not have high-speed networking."

13.4 Discussion

In this section, we discuss some assumptions made on Pufferscale, as well as some aspects related to generalizing the approach.

13.4.1 Other features of Pufferscale

The implementation of Pufferscale includes aspects that were not detailed in this chapter: Pufferscale is able to manage buckets that are stored both in memory and on drive, can manage multiple microservices at the same time, and can cope with some heterogeneity in the cluster (presence of drives, bandwidths, storage capacities).

13.4.2 Generality of Pufferscale

Pufferscale is a rescaling manager that has been designed and implemented independently of any storage system. Its interface with the microservices it manages relies on dependency injection using callbacks and thus could be integrated in many storage systems. Thus, Pufferscale can be used to add malleability to many systems: it can start and stop microservices as needed and organize the data transfers needed to have a load balanced data placement during rescaling operations.

Its generality allows it to easily be used in various situations. For instance, for the evaluation performed in Section 13.3.1, Pufferscale has been composed with a simple microservice that records and sends the metadata of the bucket to its destination, allowing the evaluation of Pufferscale on its own, without actual data transfers. On the other hand, in Section 13.3.3 Pufferscale was composed with REMI and SDSKV to showcase its performances in a real situation.

Like the formalization of the problem presented in Chapter 12, Pufferscale is not restricted to storage: as long as one can define transferable buckets with a load and a size, any service can have rescaling operations managed by Pufferscale.

13.4.3 Adjusting the weights

Giving the user the possibility to adjust the weights of the heuristic enables the user to tune the heuristic to the needs of the application. Users could also fine-tune these weights using some training runs in which the weights are adjusted in some outer loop, e.g., using derivative-free optimization solvers (DFO solvers) such as POUNDERS [141].

For instance, data balance is required mostly for decommissions. Thus, if few decommissions will be required, it makes sense to reduce the importance of the data balance.

If the load and the size of buckets are volatile, the load balance and data balance could be relaxed, since they will quickly change after the rescaling operation.

13.4.4 Reconfiguring a distributed storage system

Pufferscale supports commissions and decommissions, but is not designed to simultaneously commission and decommission nodes. The heuristic would work for the problem if the target

T_t for duration of the operation is modified to include the duration of the data transfers of nodes that are not commissioned or decommissioned as well as the duration of the data transfers on the commissioned and decommissioned nodes (Equation 13.14).

$$T_t = \max\left(\frac{|D_t - D_i|}{S_{net}}, \frac{D_i}{S_{net}}, \frac{D_t}{S_{net}}\right) \tag{13.14}$$

Using Pufferscale to simply rebalance a storage system without commissioning nor decommissioning nodes would be possible by changing the target for the duration of the operation, which could be a parameter left to the user.

Conclusion

In this chapter, we proposed a heuristic that helps find a good approximate solution in a short time to the problem of maintaining load balance during rescaling operations of distributed storage systems that do not use data replication. Users are provided with the possibility to weight each criterion as needed, in order to reach the desired trade-off between load balance, speed, and data balance. To evaluate our heuristic, we introduced Pufferscale, a generic rescaling manager for microservice-based distributed storage systems. Our large-scale evaluation of the proposed heuristic with Pufferscale exhibits the importance of maintaining data balance in order to systematically ensure fast rescaling operations when data reading generates a bottleneck, with no drawbacks in the other cases. We showcase the use of Pufferscale as a means to enable storage malleability in HEPnOS in the future.

PART V

Conclusion and Future Work

CHAPTER 14

CONCLUSION AND FUTURE WORK

Large scale applications running on shared platforms such as the Cloud have the possibility to adapt themselves to varying workloads by requesting more resources and releasing unneeded resources when necessary. However, many applications rely on a distributed storage system co-located with the computing resources. Changing the processing capabilities of the application without changing the storage can easily create an I/O-bound situation, in which the storage system is not able to cope with the new load.

One solution to avoid this situation consists of using malleable distributed storage systems, which can have nodes commissioned and decommissioned. With such systems, new storage nodes can be added to increase both the capacity and the number of simultaneous requests that can be processed, and nodes can be released once the need for simultaneous request processing decreases.

Such storage systems need to have efficient rescaling operations. In particular, one of the important aspects is the speed of these operations. Indeed, a fast commission makes newly added resources operational quickly and thus helps to cope with varying workloads. It also helps to reduce the cost of running the system by shortening the duration during which resources are rented but nonoperational. Similarly, a fast decommission reduces costs since resources can be released quickly after they become unneeded.

Designing fast rescaling operations raises many questions, which we addressed in this thesis.

- *What is the theoretical maximum speed for these operations?* Knowing beforehand that these operations are fast enough in theory motivates concrete implementations. Moreover, the same information helps to evaluate the implementations of rescaling mechanisms in existing distributed storage systems.

- *How fast can rescaling operations be in practice?* Assumptions used to reach theoretical results are rarely, if ever, met in practice. Thus, having a tool able to measure the duration of rescaling operations in practice on any given platform gives a more accurate estimation of the potential speed of these operations.

- *How to manage the data transfers required by rescaling operations?* The placement of the data at the end of rescaling operations and the scheduling of data transfers required are critical to minimize the duration of rescaling operations. Tools managing data transfers during rescaling operations are required to build malleable distributed storage systems.

The specific contributions of this thesis towards answering the above questions are detailed in the next section. We then discuss the prospects opened for future work.

14.1 Achievements

Modeling the minimal duration of rescaling operations: Rescaling operations of distributed storage systems have been assumed to be too slow to be used to match resources to a workload, and are thus reserved for maintenance purposes. As a result, little has been done in existing distributed storage systems to implement them. To show that this assumption is incorrect, our first step was the elaboration of models of the minimal duration of such rescaling operations.

We modeled the minimal duration of both commissions and decommissions. These models have been used to evaluate the implementation of rescaling operations in HDFS. We showed that the decommission of HDFS has a duration almost optimal when the network is the bottleneck, but poorly optimized disk access patterns hinder the operation when the storage device is the bottleneck. However, in the case of the commission, HDFS could be sped up by up to 14 times.

> *Rescaling operations are, in theory, fast enough to be useful.*

Studying of the trade-off between fault tolerance and duration of the decommission: We studied a strategy for a fast decommission, that is, a decommission mechanism that makes the released nodes available to the resource manager as soon as possible by relaxing the fault tolerance. We provided a model of the minimal duration of the various steps required for this operation.

We demonstrated that the fast decommission allows to return the decommissioned nodes to the resource manager in a fraction of the time required by a standard decommission at the cost of the fault tolerance. The operation has the same duration as a standard decommission and reduces the usage of node-hours in the case of a network bottleneck. However, in the case of a storage device bottleneck, the decommission lasts longer and the consumption of node-hours only decreases when many nodes are decommissioned simultaneously.

> *Relaxing fault tolerance to release decommissioned nodes faster does not speed up the operations for the distributed storage system and can even lengthen them in the case of a storage bottleneck.*

Benchmarking the duration of rescaling operations on a target platform: While the models can give an estimation of the duration of rescaling operations, they rely on strong assumptions that may not be applicable to all platforms. It is therefore needed to evaluate the duration of these operations in practice. Yet, using an existing distributed storage system may not yield accurate results since rescaling operations in current distributed storage systems are not optimized for speed.

We introduced Pufferbench, a modular benchmark, to measure the duration of rescaling operations on a given platform. With this benchmark, users of the platform can decide whether using distributed storage system malleability benefits performance, cost, energy, or any metrics they may be interested in. It can be used to fine-tune all components involved in data migration (scheduler, storage, and network). Pufferbench enables an easy prototyping and testing of the data migration mechanisms before implementing them in any existing storage system.

We showed that the minimal duration of rescaling operations can be closely approached: the rescaling operations emulated with Pufferbench have a duration within at most 16% of the models. Thus, using Pufferbench, we showed the performances obtained with the models can be achieved in practice.

We also used Pufferbench to show that, under the same constraints of data object sizes, replication, and data object placement, HDFS's rescaling mechanisms could be sped up by a factor 3 in the case of the decommission and by up to a factor 14 in the case of the commission.

> *The theoretical minimal duration of rescaling operations can be closely approached in practice.*

Implementing a generic rescaling manager: User-space HPC data services are developed to support data-intensive applications running on HPC infrastructures. Since data services are developed in support of specific applications, they can be tailored to precisely fit the needs of these applications. In particular, the load on the system can be carefully balanced to avoid I/O interference. Thus, the load balance of these systems must be considered when such data services are rescaled to optimize resource utilization.

We formalized the problem of rescaling a data service by simultaneously considering three optimization criteria: load balancing, data balancing, and duration of the rescaling operation. Since computing an exact solution to this multiobjective optimization problem takes too long, we introduced a heuristic to find a good approximate solution in a much shorter time. To evaluate our heuristic, we introduced Pufferscale, a generic rescaling manager for microservice-based distributed storage systems. Our large-scale evaluation of the proposed heuristic with Pufferscale highlights the importance of maintaining data balancing in order to systematically ensure fast rescaling when reading data generates a bottleneck, with no drawbacks in the other cases.

> *Pufferscale is a rescaling manager that organizes data transfers during rescaling operations while balancing the load on the cluster.*

14.2 Prospects

The contributions presented in this thesis show that rescaling a distributed storage system can be done fast enough to be useful in practice, making distributed storage system malleability worthy of further investigation. In this section we discuss multiple potential courses of actions to continue this work.

Implementing efficient rescaling operations: In this thesis, we focused only on the speed of rescaling operations of distributed storage systems and assumed all resources were available to the rescaling operation. It directly implies the absence of data accesses from external applications during these operations. This, however, is an unrealistic situation: many applications are sensitive to latency and simply preempting all data accesses and waiting for rescaling operations to terminate is too restrictive.

Thus, the follow-up challenge to integrating malleability in distributed storage systems is the design and implementation of efficient rescaling operations that also enables simultaneous data accesses. A trade-off between the speed of rescaling operations and the performance of data accesses is required, but some properties of rescaling operations can be leveraged. For instance, in the case of the commission of only a few nodes, the new nodes are saturated and thus, old nodes can manage most of the data accesses. We foresee two main problems to address. The first one is the development of strategies to determine which node can serve each request. The second one is the correct handling of write operations on data objects that are being transferred between nodes to ensure the consistency of the data.

Determining which use cases would benefit from distributed storage malleability: Distributed storage system malleability is clearly beneficial in some cases such as HEPnOS: using rescaling operations to reconfigure the storage between application executions ensures that the storage system is perfectly dimensioned for each application while minimizing resource usage. However, for many use cases, malleability may not have such clear benefits. Since resources are used to commission and decommission nodes, the cost of rescaling operations needs to be offset by their benefits. Moreover, the relevance of using a co-located storage system also need to be considered since the use of a remote storage system instead cancels the need for distributed storage system malleability.

One of the important steps for distributed storage malleability would be a characterization of the workloads that benefit from distributed storage malleability. Moreover, having an estimation of the gains brought by using a malleable distributed storage system would help users to decide whether to invest time in improving the malleability of their application.

Towards an elastic storage system: Optimized rescaling operations are useful only if the system deciding when to rescale and which nodes to add or remove efficiently supports rescaling operations. The design of this resource manager is critical to make an elastic distributed storage system.

This research topic includes two directions. The first one is the automatic detection of situations in which commissioning and decommissioning storage nodes would improve the performances of an application while keeping the cost of the used resources as low as possible. The second direction is jointly leveraging the malleability of an application and that of its storage in order to cope with a varying workload. Since the cost of rescaling operations is different for the application and its storage, they should probably not rely on the same rescaling decisions. However, the rescaling of either the application or the storage has an impact on the performance and needs for resources of the other. Thus, there is an interest in having a single system managing both.

BIBLIOGRAPHY

[1] *Top500*, www.top500.org, Accessed May 20, 2019.

[2] *Amazon Web Services*, aws.amazon.com, Accessed June 18, 2019.

[3] *AWS Cloud Computing Ops, Data Centers, 1.3 Million Servers Creating Efficiency Flywheel*, www.zdnet.com/article/aws-cloud-computing-ops-data-centers-1-3-million-servers-creating-efficiency-flywheel/, Accessed June 18, 2019.

[4] *Supercomputing and the Weather: How far We've Come, and Where We're Going*, www.hpcwire.com/solution_content/hpe/weather-climate/supercomputing-weather-far-weve-come-going/, Accessed June 18, 2019.

[5] Simson Garfinkel, *Architects of the Information Society: 35 Years of the Laboratory for Computer Science at MIT*, MIT press, 1999.

[6] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al., "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective", *in*: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629.

[7] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", *in*: *Communication of the ACM* 51.*1* (2008), pp. 107–113.

[8] Tom White, *Hadoop: The Definitive Guide*, O'Reilly Media, inc., 2009.

[9] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica, "Spark: Cluster Computing with Working Sets", *in*: *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* 10.*10* (2010), p. 95.

[10] Ioan Raicu, Ian T Foster, and Pete Beckman, "Making a Case for Distributed File Systems at Exascale", *in*: *3rd ACM International Workshop on Large-scale System and Application Performance (LSAP)*, 2011, pp. 11–18.

[11] Matthieu Dorier, Philip Carns, Kevin Harms, Robert Latham, Robert Ross, Shane Snyder, Justin Wozniak, Samuel Gutierrez, Bob Robey, Brad Settlemyer, Galen Shipman, Jerome Soumagne, James Kowalkowski, Marc Paterno, and Saba Sehrish, "Methodology for the Rapid Development of Scalable HPC Data Services", *in*: *3rd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2018.

[12] Nathanaël Cheriere and Gabriel Antoniu, "How Fast Can One Scale Down a Distributed File System?", *in*: *IEEE International Conference on Big Data (BigData)*, 2017.

[13] Nathanaël Cheriere, Matthieu Dorier, and Gabriel Antoniu, "How Fast can One Resize a Distributed File System?", *in*: *Journal of Parallel and Distributed Computing (JPDC)* (2019), Submitted.

[14] Nathanaël Cheriere, Matthieu Dorier, and Gabriel Antoniu, "Is it Worth Relaxing Fault Tolerance to Speed Up Decommission in Distributed Storage Systems?", *in*: *19th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2019.

[15] Nathanaël Cheriere and Matthieu Dorier, “Design and Evaluation of Topology-aware Scatter and AllGather Algorithms for Dragonfly Networks”, *in*: *IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[16] Nathanaël Cheriere, Matthieu Dorier, and Gabriel Antoniu, “Pufferbench: Evaluating and Optimizing Malleability of Distributed Storage”, *in*: *3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2018, pp. 35–44.

[17] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec, “Adding Virtualization Capabilities to the Grid’5000 Testbed”, *in*: *Cloud Computing and Services Science*, vol. 367, 2013, pp. 3–20.

[18] *Bebop*, `www.lcrc.anl.gov/systems/resources/bebop/`, Accessed June 17, 2019.

[19] Jacek Blazewicz, Mikhail Y Kovalyov, Maciej Machowiak, Denis Trystram, and Jan Weglarz, “Preemptable Malleable Task Scheduling Problem”, *in*: *IEEE Transactions on Computers* 55.*4* (2006), pp. 486–490.

[20] Jacek Blazewicz, “Malleable Tasks Scheduling to Minimize the Makespan”, *in*: *Annals of Operations Research* 129 (2004), pp. 65–80.

[21] Pierre-François Dutot, Grégory Mounié, and Denis Trystram, “Scheduling Parallel Tasks: Approximation Algorithms”, *in*: *Handbook of Scheduling: Algorithms, Models, and Performance Analysis* (2004).

[22] Dror G Feitelson and Larry Rudolph, “Toward Convergence in Job Schedulers for Parallel Supercomputers”, *in*: *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1996, pp. 1–26.

[23] André B Bondi, “Characteristics of Scalability and Their Impact on Performance”, *in*: *2nd ACM International Workshop on Software and Performance (WOSP)*, 2000, pp. 195–203.

[24] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner, “Elasticity in Cloud Computing: What it is, and What it is not”, *in*: *10th International Conference on Autonomic Computing (ICAC)*, 2013, pp. 23–27.

[25] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong, “Theory and Practice in Parallel Job Scheduling”, *in*: *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1997, pp. 1–34.

[26] Jan Hungershofer, “On the Combined Scheduling of Malleable and Rigid Jobs”, *in*: *16th IEEE Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2004, pp. 206–213.

[27] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kale, “A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications”, *in*: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2015), pp. 429–438.

[28] Michael Mercier, David Glesser, Yiannis Georgiou, and Olivier Richard, “Big Data and HPC Collocation: Using HPC Idle Resources for Big Data Analytics”, *in*: *IEEE International Conference on Big Data (Big Data)*, 2017, pp. 347–352.

[29] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale, “Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting”, *in*: *4th IEEE International Conference on Cloud Computing (CLOUD)*, 2011, pp. 500–507.

[30] Di Niu, Hong Xu, Baochun Li, and Shuqiao Zhao, “Quality-Assured Cloud Bandwidth Auto-Scaling for Video-on-Demand Applications”, *in*: *IEEE International Conference on Computer Communications (INFOCOM)*, 2012, pp. 460–468.

[31] Ming Mao and Marty Humphrey, “Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows”, *in*: *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12.

[32] Brian Dougherty, Jules White, and Douglas C Schmidt, “Model-Driven Auto-Scaling of Green Cloud Computing Infrastructure”, *in*: *Future Generation Computer Systems (FGCS)* 28.*2* (2012), pp. 371–378.

[33] Aleksandra Kuzmanovska, Rudolf H. Mak, and Dick Epema, “KOALA-F: A Resource Manager for Scheduling Frameworks in Clusters”, *in*: *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, 2016, pp. 592–595.

[34] Ionut David, Bojan Orlic, Rudolf H Mak, and Johan J Lukkien, “Towards Resource-Aware Runtime Reconfigurable Component-Based Systems”, *in*: *6th IEEE World Congress on Services (SERVICES)*, 2010, pp. 465–466.

[35] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Roa, “Morpheus: Towards Automated SLOs for Enterprise Clusters”, *in*: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016), pp. 117–134.

[36] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, et al., “Mapping Abstract Complex Workflows onto Grid Environments”, *in*: *Journal of Grid Computing* 1.*1* (2003), pp. 25–39.

[37] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao, “Scientific Workflow Management and the Kepler System”, *in*: *Concurrency and Computation: Practice and Experience (CCPE)* 18.*10* (2006), pp. 1039–1065.

[38] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al., “Taverna: a Tool for the Composition and Enactment of Bioinformatics Workflows”, *in*: *Bioinformatics* 20.*17* (2004), pp. 3045–3054.

[39] J Buisson, F André, and JL Pazat, “A Framework for Dynamic Adaptation of Parallel Components”, *in*: *International Conference on Parallel Computing (ParCo)* (2005), pp. 1–8.

[40] Laxmikant V. Kale, Sameer Kumar, and Jayant Desouza, “A Malleable-Job System for Timeshared Parallel Machines”, *in*: *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)* (2002).

[41] Sathish S. Vadhiyar and Jack J. Dongarra, “SRS: A Framework for Developing Malleable and Migratable Parallel Applications For Distributed Systems”, *in*: *Parallel Processing Letters* 13.*2* (2003), pp. 291–312.

[42] Philip Maechling, Ewa Deelman, Li Zhao, Robert Graves, Gaurang Mehta, Nitin Gupta, John Mehringer, Carl Kesselman, Scott Callaghan, David Okaya, and Others, “SCEC CyberShake Workflow - Automating Probabilistic Seismic Hazard Analysis Calculations”, *in*: *Workflows for e-Science*, 2007, pp. 143–163.

[43] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi, “Characterizing and Profiling Scientific Workflows”, *in*: *Future Generation Computer Systems (FGCS)* 29.*3* (2013), pp. 682–692.

[44] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright, “Modular HPC I/O Characterization with Darshan”, *in*: *5th IEEE Workshop on Extreme-Scale Programming Tools (ESPT)*, 2016, pp. 9–17.

[45] *HMMER: Biosequence Analysis Using Profile Hidden Markov Models*, `hmmer.org`, Accessed June 5, 2019.

[46] *Summit*, `www.olcf.ornl.gov/olcf-resources/compute-systems/summit/`, Accessed May 20, 2019.

[47] *Summit: Data Storage & Transfers*, `www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/`, Accessed May 28, 2019.

[48] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, Wei Zhao, Xunqiang Yin, Chaofeng Hou, Chenglong Zhang, Wei Ge, Jian Zhang, Yangang Wang, Chunbo Zhou, and Guangwen Yang, “The Sunway TaihuLight Supercomputer: System and Applications”, *in*: *Science China Information Sciences* 59.*7* (2016).

[49] Jack Dongarra, *Report on the Tianhe-2A System*, 2017.

[50] *Supercomputer Titan to Get World’s Fastest Storage System*, `phys.org/news/2013-04-supercomputer-titan-world-fastest-storage.html`, Accessed June 12, 2019.

[51] *Using the Sequoia and Vulcan BG/Q Systems*, `computing.llnl.gov/tutorials/bgq/`, Accessed June 12, 2019.

[52] Shinji Sumimoto, *Current Status of FEFS for the K computer*, 2012.

[53] Jeffrey S Vetter, *Contemporary High Performance Computing: from Petascale Toward Exascale*, Chapman and Hall/CRC, 2013.

[54] Arthur S Bland, Wayne Joubert, Ricky A Kendall, Douglas B Kothe, James H Rogers, and Galen M Shipman, “Jaguar: The World’s Most Powerful Computer System–an Update”, *in*: *Cray Users Group* (2010).

[55] Silverton Consulting, *IBM Spectrum Scale 5.0.0 IO performance*, 2018.

[56] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, “The Hadoop Distributed File System”, *in*: *IEEE Symposium on Mass Storage Systems and Technologies (MSST)* (2010), pp. 1–10.

[57] *The Infrastructure Behind Twitter: Scale*, `blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html`, Accessed June 12, 2019.

[58] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman, “Project Adam: Building an Efficient and Scalable Deep Learning Training System”, *in*: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 571–582.

[59] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn, “Ceph: A Scalable, High-Performance Distributed File System”, *in*: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 307–320.

[60] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos Maltzahn, “Rados: a Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters”, *in*: *2nd International Workshop on Petascale Data Storage (PDSW)*, 2007, pp. 35–44.

[61] Philip Schwan, “Lustre: Building a File System for 1000-node Clusters”, *in*: *Linux symposium*, 2003, pp. 380–386.

[62] Robert B Ross, Rajeev Thakur, et al., “PVFS: A Parallel File System for Linux Clusters”, *in*: *4th Annual Linux Showcase and Conference*, 2000, pp. 391–430.

[63] Frank B Schmuck and Roger L Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters.”, *in*: *USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[64] Frank Herold, Sven Breuner, and J Heichler, “An Introduction to BeeGFS”, *in*: (2014).

[65] Ohad Rodeh and Avi Teperman, “zFS-a Scalable Distributed File System Using Object Disks”, *in*: *11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2003, pp. 207–218.

[66] Michael Moore David Bonnie, Becky Ligon, Mike Marshall, Walt Ligon, Nicholas Mills, Elaine Quarles Sam Sampson, Shuangyang Yang, and Boyd Wilson, “OrangeFS: Advancing PVFS”, *in*: *USENIX Conference on File and Storage Technologies (FAST)* (2011).

[67] *MooseFS*, `moosefs.com`, Accessed June 5, 2019.

[68] *Amazon S3*, `aws.amazon.com/s3`, Accessed May 29, 2019.

[69] *Microsoft Azure*, `azure.microsoft.com/en-gb/services/storage/`, Accessed May 29, 2019.

[70] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin, “Erasure coding in windows azure storage”, *in*: *USENIX Annual Technical Conference (ATC)*, 2012, pp. 15–26.

[71] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam Mckelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim, Muhammad Ikram, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin Mcnett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas, "Windows Azure Storage : A Highly Available Cloud Storage Service with Strong Consistency", *in*: *ACM Symposium on Operating Systems Principles (SOSP)* 20 (2011), pp. 143–157.

[72] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System", *in*: *ACM SIGOPS Operating Systems Review (OSR)* 37.*5* (2003), p. 29.

[73] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M Rumble, Eric Stratmann, and Ryan Stutsman, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM", *in*: *ACM SIGOPS Operating Systems Review (OSR)* 43.*4* (2010), pp. 92–105.

[74] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine", *in*: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.*4* (2015).

[75] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica, "Reliable, Memory Speed Storage for Cluster Computing Frameworks", *in*: *ACM Symposium on Cloud Computing (SoCC)*, 2014, pp. 1 –15.

[76] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J Lang, Garth Gibson, and Robert B Ross, "On the Duality of Data-Intensive File System Design: Reconciling HDFS and PVFS", *in*: *ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, p. 67.

[77] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing", *in*: *9th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 2–2.

[78] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao, "A Multiplatform Study of I/O Behavior on Petascale Supercomputers", *in*: *24th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2015, pp. 33–44.

[79] *Intrepid*, `www.alcf.anl.gov/intrepid`, Accessed June 6, 2019.

[80] *Mira*, `www.alcf.anl.gov/mira`, Accessed June 6, 2019.

[81] *Edison*, `www.nersc.gov/users/computational-systems/retired-systems/edison-retired-on-5132019/`, Accessed June 6, 2019.

[82] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross, "Understanding and Improving Computational Science Storage Access Through Continuous Characterization", *in*: *ACM Transactions on Storage (TOS)* 7.*3* (2011), p. 8.

[83] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf, "Managing Variability in the IO Performance of Petascale Storage Systems", *in*: *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010, pp. 1–12.

[84] Qing Liu, Norbert Podhorszki, Jeremy Logan, and Scott Klasky, "Runtime I/O Re-Routing+ Throttling on HPC Storage", *in*: *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[85] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim, "CALCioM: Mitigating I/O Interference in HPC Systems Through Cross-Application Coordination", *in*: *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 155–164.

[86] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems", *in*: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 750–759.

[87] Sudipto Das, Amr El Abbadi, and Divyakant Agrawal, "ElasTraS: An Elastic Transactional Data Store in the Cloud.", *in*: *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009, pp. 131–142.

[88] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi, "ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for the Cloud", *in*: *ACM Transactions on Database Systems (TODS)* 38.*1* (2013), p. 5.

[89] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J Elmore, "Database Scalability, Elasticity, and Autonomy in the Cloud", *in*: *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2011, pp. 2–15.

[90] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, "Dynamo: Amazon's Highly Available Key-Value Store", *in*: *ACM SIGOPS Operating Systems Review (OSR)*, vol. 41, 6, 2007, pp. 205–220.

[91] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform", *in*: *Proceedings of the VLDB Endowment* 1.*2* (2008), pp. 1277–1288.

[92] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan, "Sierra: Practical Power-Proportionality for Data center Storage", *in*: *6th Conference on Computer Systems (EuroSys)* (2011), p. 169.

[93] Hrishikesh Amur, James Cipar, Varun Gupta, Gregory R. Ganger, Michael A. Kozuch, and Karsten Schwan, "Robust and Flexible Power-Proportional Storage", *in*: *ACM Symposium on Cloud Computing (SoCC)* (2010), pp. 217–228.

[94] Xu Lianghong, Cipar James, Krevat Elie, Tumanov Alexey, Gupta Nitin, Kozuch Michael, and Ganger Gregory, "SpringFS: Bridging Agility and Performance in Elastic Distributed Storage", *in*: *USENIX Conference on File and Storage Technologies (FAST)*, 2014, pp. 243–255.

[95] Harold C Lim, Shivnath Babu, and Jeffrey S Chase, "Automated Control for Elastic Storage", *in*: *International Conference on Autonomic Computing (ICAC)* (2010), pp. 1–10.

[96] Beth Trushkowsky, Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson, "The SCADS Director: Scaling a Distributed Storage System under Stringent Performance Requirements", *in*: *USENIX Conference on File and Storage Technologies (FAST)* (2011), pp. 163–176.

[97] Ahmad Al-Shishtawy and Vladimir Vlassov, "Elastman: Elasticity Manager for Elastic Key-Value Stores in the Cloud", *in*: *ACM Cloud and Autonomic Computing Conference (CAC)*, 2013, p. 7.

[98] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica, "Load Balancing in Structured P2P Systems", *in*: *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003, pp. 68–79.

[99] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina, "Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems", *in*: *13th International Conference on Very Large Data Bases (VLDB)*, 2004, pp. 444–455.

[100] Yingwu Zhu and Yiming Hu, "Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems", *in*: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 16.*4* (2005), pp. 349–361.

[101] Hung-Chang Hsiao, Hsueh-Yi Chung, Haiying Shen, and Yu-Chang Chao, "Load Rebalancing for Distributed File Systems in Clouds", *in*: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 24.*5* (2013), pp. 951–962.

[102] Alberto Miranda and Toni Cortes, "CRAID: Online RAID Upgrades Using Dynamic Hot Data Reorganization.", *in*: *USENIX Conference on File and Storage Technologies (FAST)*, vol. 14, 2014, pp. 133–146.

[103] Tim Anderson, *Who Needs a Supercomputer When You Can Get a Couple of Petaflops on AWS?*, `www.theregister.co.uk/2019/06/20/supercomputer_aws/`, Accessed June 21, 2019.

[104] Dean Hildebrand and James Coomer, *Competing with Supercomputers: HPC in the Cloud Becomes Reality*, `cloud.google.com/blog/products/storage-data-transfer/competing-with-supercomputers-hpc-in-the-cloud-becomes-reality`, Accessed June 21, 2019.

[105] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and Ponnuswamy Sadayappan, "Scalable I/O Forwarding Framework for High-Performance Computing Systems", *in*: *IEEE International Conference on Cluster Computing and Workshops (Cluster)*, 2009, pp. 1–10.

[106] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig, "Small-File Access in Parallel File Systems", *in*: *IEEE International Symposium on Parallel and Distributed Processing (ISPDC)*, 2009, pp. 1–11.

[107] Erez Zadok, Dean Hildebrand, Geoff Kuenning, and Keith A Smith, "POSIX is Dead! Long Live... errr... What Exactly?", *in*: *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[108] Russ Rew and Glenn Davis, “NetCDF: an Interface for Scientific Data Access”, *in*: *IEEE Computer Graphics and Applications* 10.*4* (1990), pp. 76–82.

[109] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson, “An Overview of the HDF5 Technology Suite and its Applications”, *in*: *ACM Workshop on Array Databases (AD)*, 2011, pp. 36–47.

[110] David A Boyuka, Sriram Lakshminarasimham, Xiaocheng Zou, Zhenhuan Gong, John Jenkins, Eric R Schendel, Norbert Podhorszki, Qing Liu, Scott Klasky, and Nagiza F Samatova, “Transparent in Situ Data Transformations in ADIOS”, *in*: *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2014, pp. 256–266.

[111] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, Orcun Yildiz, Shadi Ibrahim, Tom Peterka, and Leigh Orf, “Damaris: Addressing Performance Variability in Data Management for Post-Petascale Simulations”, *in*: *ACM Transactions on Parallel Computing (TOPC)* 3.*3* (2016), p. 15.

[112] Matthieu Dorier, Matthieu Dreher, Tom Peterka, and Robert Ross, “CoSS: Proposing a Contract-Based Storage System for HPC”, *in*: *2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2017, pp. 13–18.

[113] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn, “On the Role of Burst Buffers in Leadership-Class Storage Systems”, *in*: *28th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2012, pp. 1–11.

[114] J Bent and G Grider, “Usability at Los Alamos National Lab”, *in*: *5th DOE Workshop on HPC Best Practices: File Systems and Archives*, 2011.

[115] Qing Zheng, Kai Ren, Garth Gibson, Bradley W. Settlemyer, and Gary Grider, “DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers”, *in*: *International Workshop on Parallel Data Storage (PDSW)* (2015), pp. 1–6.

[116] Philippe Cudré-Mauroux, Hideaki Kimura, K-T Lim, Jennie Rogers, Roman Simakov, Emad Soroush, Pavel Velikhov, Daniel L Wang, Magdalena Balazinska, Jacek Becla, et al., “A Demonstration of SciDB: a Science-Oriented DBMS”, *in*: *Proceedings of the VLDB Endowment* 2.*2* (2009), pp. 1534–1537.

[117] Ciprian Docan, Manish Parashar, and Scott Klasky, “Dataspaces: an Interaction and Coordination Framework for Coupled Simulation Workflows”, *in*: *Cluster Computing* 15.*2* (2012), pp. 163–181.

[118] Hugh Greenberg, John Bent, and Gary Grider, “MDHIM: A Parallel Key/Value Framework for HPC”, *in*: *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2015.

[119] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka, “FTI: High Performance Fault Tolerance Interface for Hybrid Systems”, *in*: *IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12.

[120] Wolfgang Frings, Dong H Ahn, Matthew LeGendre, Todd Gamblin, Bronis R de Supinski, and Felix Wolf, "Massively Parallel Loading", *in*: *27th ACM International Conference on Supercomputing (ICS)*, 2013, pp. 389–398.

[121] *Mochi*, `https://www.mcs.anl.gov/research/projects/mochi/`, Accessed March 29, 2019.

[122] *HEP on HPC*, `http://computing.fnal.gov/hep-on-hpc/`, Accessed March 29, 2019.

[123] Rene Brun and Fons Rademakers, "ROOT—an Object Oriented Data Analysis Framework", *in*: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 389.*1-2* (1997), pp. 81–86.

[124] *ASCI Red*, `web.archive.org/web/20110926225845/http://www.sandia.gov/ASCI/Red/`, Accessed May 20, 2019.

[125] Alexey Ilyushkin and Dick Epema, "The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows", *in*: *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2018, pp. 331–341.

[126] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron, "Pelican: A Building Block for Exascale Cold Data Storage", *in*: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 351–365.

[127] *Cray XC Series*, `www.cray.com/sites/default/files/Cray-XC-Series-Brochure.pdf`, Accessed June 19, 2017.

[128] *NVMe SSD 960 PRO/EVO*, `www.samsung.com/semiconductor/minisite/ssd/downloads/document/NVMe_SSD_960_PRO_EVO_Brochure_Rev_1_1.pdf`, Accessed June 19, 2017.

[129] *IBM ILOG CPLEX Optimization Studio (version 12.8.0.0)*, `https://www.ibm.com/products/ilog-cplex-optimization-studio`, 2018.

[130] Mohamed Hefeeda and Osama Saleh, "Traffic Modeling and Proportional Partial Caching for Peer-to-Peer Systems", *in*: *IEEE/ACM Transactions on Networking* 16.*6* (2008), pp. 1447–1460.

[131] Gunjan Khanna, Kirk Beaty, Gautam Kar, and Andrzej Kochut, "Application Performance Management in Virtualized Server Environments", *in*: *10th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2006, pp. 373–381.

[132] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, Mazin S Yousif, et al., "Black-box and Gray-box Strategies for Virtual Machine Migration", *in*: *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 7, 2007, pp. 17–17.

[133] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra, "Server-Storage Virtualization: Integration and Load Balancing in Data Centers", *in*: *ACM/IEEE Conference on Supercomputing (SC)*, 2008, p. 53.

[134] Emmanuel Arzuaga and David R Kaeli, "Quantifying Load Imbalance on Virtualized Enterprise Servers", *in*: *1st Joint WOSP/SIPEW International Conference on Performance Engineering*, 2010, pp. 235–242.

[135] Haiying Shen, "RIAL: Resource Intensity Aware Load Balancing in Clouds", *in*: *IEEE Transactions on Cloud Computing (ToCC)* (2017).

[136] Ronald L. Graham, “Bounds on Multiprocessing Timing Anomalies”, *in*: *SIAM Journal on Applied Mathematics* 17.*2* (1969), pp. 416–429.

[137] *REMI*, `https://xgitlab.cels.anl.gov/sds/remi`, Accessed March 29, 2019.

[138] *Mercury*, `https://mercury-hpc.github.io/`, Accessed March 29, 2019.

[139] *Argobots*, `http://argobots.org`, Accessed March 29, 2019.

[140] *SDSKV*, `https://xgitlab.cels.anl.gov/sds/sds-keyval`, Accessed March 29, 2019.

[141] Stefan M Wild, “POUNDERS in TAO: Solving Derivative-Free Nonlinear Least-Squares Problems with POUNDERS”, *in*: *SIAM Advances and Trends in Optimization with Engineering Applications*, 2017, pp. 529–539.

APPENDIX A

PROOFS

Appendix

Property 4.1:

The storage is a bottleneck if $\frac{S_{Read}S_{Write}}{S_{Read}+S_{Write}} < S_{Net}$.

Proof:

The storage is a bottleneck if it cannot read and write all the data that can be sent and received on the network.

During a lapse of time t, at most $t * S_{Net}$ is sent and $t * S_{Net}$ is received. The storage must share its data accesses between reads and writes (Assumption 3).

Let us denote as $t_{storage}$ the time needed for the storage to read and write these amounts of data.

$$t_{storage} = t \cdot \left(\frac{S_{Net}}{S_{Read}} + \frac{S_{Net}}{S_{write}} \right)$$

The storage is a bottleneck if $t_{storage} > t$.

$$\begin{aligned} t_{storage} &> t \\ \frac{S_{Net}}{S_{Read}} + \frac{S_{Net}}{S_{write}} &> 1 \\ \frac{1}{S_{Read}} + \frac{1}{S_{write}} &> \frac{1}{S_{Net}} \\ \frac{S_{Read} \cdot S_{Write}}{S_{Read}+S_{Write}} &< S_{Net} \end{aligned}$$

QED

Lemma 4.1:

The probability of finding a specific object on a node is $\frac{r}{N}$ *with* N *the number of nodes in the cluster.*

Proof:

Let us compute the probability of finding an object O on a node A.

The number of sets of r nodes that contain node A is $\binom{N-1}{r-1}$. The probability of one of these sets to host O is $\frac{1}{\binom{N}{r}}$.

Thus, the probability of finding O on A is $\frac{r}{N}$.

QED

Property 5.1:

Each node must host $D' = \frac{ND}{N+x}$ *of data at the end of the commission.*

Proof:

Objectives 1 and 2 ensure that there is as much data on the cluster at the end of the commission as there was in the initial situation.

Objective 3 ensures that each node hosts the same amount of data.

Thus, the amount of data on a node at the end of the commission D' is the total amount of data on the cluster divided by the number of nodes:

$$D' = \frac{ND}{N+x}.$$

QED

Property 5.2:

$$D_{\rightarrow new} = xD' = \frac{xND}{N+x}$$

Proof:

There are x new nodes, and each hosts D' of data. Thus,

$$D_{\rightarrow new} = xD' = \frac{xND}{N+x}.$$

QED

Definition 5.1:

$$p_i = \begin{cases} \frac{\binom{x}{i}\binom{N}{r-i}}{\binom{N+x}{r}} & \forall 0 \leq i \leq r \\ 0 & \forall i > r \end{cases}.$$

Detail:

The problem is modeled as an urn problem: x new nodes, N old nodes. We select a combination of r of them (Assumption 5) and compute the probability that exactly i new nodes are selected.

QED

Lemma 5.1:

$\sum_{i=0}^{r} p_i = 1$

Proof:

All files have between 0 and r replicas on the new nodes.

QED

Lemma 5.2:

$\sum_{i=0}^{r} ip_i = \frac{xr}{N+x}$.

Proof:

The data stored on the new nodes at the end of the commission D_{new} can be expressed in two different manners:

- With the amount of data per node:

$$D_{new} = x\frac{ND}{N+x}$$

- With the probability of finding a replica on them:

$$D_{new} = \frac{ND}{r}\sum_{i=0}^{r} ip_i$$

Thus,

$$\sum_{i=0}^{r} ip_i = \frac{xr}{N+x}.$$

QED

Property 5.3:

$D_{old \to new} = \frac{ND}{r}(1 - p_0)$.

Proof:

All the unique data that must be transferred to new nodes must be read from the old nodes.

$$D_{old \to new} = \frac{ND}{r}\sum_{i=1}^{r} p_i$$

$$D_{old \to new} = \frac{ND}{r}(1 - p_0)$$

QED

Property 5.4:

$$D_{old/new->new} = \frac{ND}{rx}(\frac{rx}{N+x} + p_0 - 1).$$

Proof:

$D_{old/new\to new}$ is the amount of data that must be stored on new nodes $D_{\to new}$ minus the replicas that can be read only from old nodes $D_{old\to new}$.

$$\begin{aligned} D_{old/new\to new} &= D_{\to new} - D_{old\to new} \\ &= \frac{xND}{N+x} - \frac{ND}{r}(1-p_0) \\ &= \frac{ND}{rx}(\frac{rx}{N+x} + p_0 - 1) \end{aligned}$$

QED

Property 5.5:

Assuming that objects can always be divided into multiple objects of any smaller size, Algorithm 5.1 avoids all data transfers between old nodes and satisfies all the objectives.

Proof:

Objectives 1 and 2 are satisfied by design since data is transferred from node to node.
No data transfers occur between old nodes by design.

Quantifying data transfers

Let S^r_{old} be the set of sets of r distinct old nodes.
S^r_{old} contains exactly $\binom{N}{r}$ elements.
Let A be a set of r distinct old nodes ($A \in S^r_{old}$).
Let D_A be the amount of data exclusive to A.

$$D_A = \frac{ND}{r\binom{N}{r}}$$

The second step of Algorithm 5.1 divides the exclusive data of A into $r+1$ distinct subsets of sizes D^0_A, ..., D^r_A.

$$\forall 0 \leq i \leq r, D^i_A = p_i D_A$$

Then, during the third step of Algorithm 5.1, for all $i \in [0, r]$, each new node receives a part d^i_A of the exclusive data stored on D.

$$\forall 0 \leq i \leq r, d^i_A = \frac{iD^i_A}{x}$$

During the same phase, each node in A loses dr^i_A of the exclusive data initially storage on A.

$$\forall 0 \leq i \leq r, dr^i_A = \frac{iD^i_A}{r}$$

Data balance (Objective 3)

Algorithm 5.1 assigns D'_{new} data to each new node. D'_{new} is the sum of all d^i_A for all possible i and A.

$$\begin{aligned}
D'_{new} &= \sum_{A \in S^r_{old}} \sum_{i=0}^{r} d^i_A \\
&= \sum_{A \in S^r_{old}} \sum_{i=0}^{r} \frac{iD^i_A}{x} \\
&= \binom{N}{r} \sum_{i=0}^{r} \frac{NDip_i}{rx\binom{N}{r}} \\
&= \frac{ND}{xr} \sum_{i=0}^{r} ip_i \\
&= \frac{xD}{N+x} \textit{(with Property 5.2)} \\
&= D'
\end{aligned}$$

Algorithm 5.1 leaves D'_{old} data to an old node n. D'_{old} is D minus the sum of all dr^i_A for all possible i and all A that include n.

Let $S^r_{old}(n)$ be the set of sets of r distinct nodes that include n. $S^r_{old}(n)$ contains $\binom{N-1}{r-1}$ sets.

$$\begin{aligned}
D'_{old} &= D - \sum_{A \in S^r_{old}(n)} \sum_{i=0}^{r} dr^i_A \\
&= D - \sum_{A \in S^r_{old}(n)} \sum_{i=0}^{r} \frac{iD^i_A}{r} \\
&= D - \sum_{A \in S^r_{old}(n)} \sum_{i=0}^{r} \frac{NDip_i}{r^2\binom{N}{r}} \\
&= D - \binom{N-1}{r-1} \frac{ND}{r^2\binom{N}{r}} \sum_{i=0}^{r} ip_i \\
&= D - \binom{N}{r} \frac{r}{N} \frac{ND}{r^2\binom{N}{r}} \frac{xr}{N+x} \textit{(with Property 5.2)} \\
&= D - \frac{xD}{N+x} \\
&= D'
\end{aligned}$$

With this, the objective of data balance is satisfied.

Exclusive data (Objective 4)

Let A be a set of r distinct nodes.
Let k be the number of new nodes in A.
Let D_{ex} be the amount of exclusive data on A.
In order to show that the distribution satisfies objective 4, D_{ex} should be equal to $\frac{ND}{r\binom{N+x}{r}}$.

The amount of exclusive data on A is composed of objects that have $r-k$ replicas on old nodes and k replicas on new nodes. Before being assigned to the new nodes, the k replicas could have been on any other two old nodes. Sicne the algorithm does not move data between old nodes, however, the data present on the old nodes after the redistribution was initially on the same old nodes.

From this, we deduce that D_{ex} is the product of the following.

1. nb, the number of sets of r distinct nodes containing the $r-k$ old nodes of A;
2. D_A^k, the amount of data from a set of r distinct nodes that was assigned to exactly k new nodes by Algorithm 5.1;
3. p_{remain}, the proportion of that data that remains on the $r-k$ old nodes of A;
4. p_{distr}, the proportion of that data that is assigned to the k new nodes of A;

Using the urn problem, we have

$$p_{remain} = \frac{\binom{r-k}{r-k}}{\binom{k}{0}}\binom{r}{r-k} = \frac{1}{\binom{r}{r-k}}$$

$$p_{distr} = \frac{\binom{k}{k}}{\binom{x-k}{0}}\binom{x}{k} = \frac{1}{\binom{x}{k}}$$

$$nb = \binom{N-r+k}{k}.$$

Thus,

$$D_{ex} = nb \times D_A^k \times p_{remain} \times p_{distr}.$$

After simplification,

$$D_{ex} = \frac{ND}{r\binom{N+x}{r}}.$$

Thus, the objective of uniformity is satisfied.

QED

Property 5.6:

$$T_{recv} = \frac{ND}{(N+x)S_{Net}}$$

Proof:

$$T_{recv} = \frac{D'}{S_{Net}}.$$

Since $D' = \frac{ND}{N+x}$,

$$T_{recv} = \frac{ND}{(N+x)S_{Net}}.$$

QED

Property 5.7:

$$T_{old \to new} = \frac{D}{rS_{Net}}(1 - p_0)$$

Proof:

$$T_{old \to new} = \frac{D_{old \to new}}{S_{Net}^{old}}$$

where $S_{Net}^{old} = NS_{Net}$, the aggregated network speed of the old nodes.
Thus,

$$T_{old \to new} = \frac{D}{rS_{Net}}(1 - p_0).$$

QED

Property 5.8:

$$T_{new \to new} = \frac{ND}{rxS_{Net}} \sum_{i=2}^{r}(i-1)p_i$$

Proof:

$$T_{new \to new} = \frac{D_{old/new \to new}}{S_{Net}^{new}}$$

where $S_{Net}^{new} = xS_{Net}$, the aggregate network speed of the new nodes. Thus,

$$T_{new \to new} = \frac{ND}{rxS_{Net}} \sum_{i=2}^{r}(i-1)p_i$$

QED

Property 5.9:

$$T_{\to new}^{balanced} = \frac{xND}{(N+x)^2 S_{Net}}$$

Proof:

Let us denote as Y the amount of data that must be transferred between new nodes to balance send times for transfers from old to new nodes and between new nodes.

$$T_{old \to new}^{balanced} = \frac{D_{\to new} - Y}{S_{Net}^{old}} = \frac{1}{NS_{Net}}(x\frac{ND}{N+x} - Y)$$

$$T_{new \to new}^{balanced} = \frac{Y}{S_{Net}^{new}} = \frac{1}{xS_{Net}}Y$$

Then $T_{old \to new}^{balanced} = T_{new \to new}^{balanced}$, and thus $Y = \frac{x^2 ND}{(N+x)^2}$

QED

Property 5.10:

If $T_{old\to new} \leq T_{new\to new}$, *then* $T_{old\to new} \leq T^{balanced}_{\to new} \leq T_{new\to new}$.

Proof:

Assuming $T_{old\to new} \leq T_{new\to new}$, we have the following.

$$
\begin{aligned}
T_{old\to new} &\leq T_{new\to new} \\
1 - p_0 &\leq \frac{N}{x}\sum_{i=2}^{r}(i-1)p_i \\
\sum_{i=1}^{r} p_i &\leq \frac{N}{x}\sum_{i=1}^{r} ip_i - \frac{N}{x}\sum_{i=1}^{r} p_i \\
\frac{x}{N}(1+\frac{N}{x})\sum_{i=1}^{r} p_i &\leq \sum_{i=1}^{r} ip_i \\
(\frac{x}{N}+1)\sum_{i=1}^{r} p_i &\leq \sum_{i=1}^{r} ip_i
\end{aligned}
$$

$$
\begin{aligned}
&T^{balanced}_{\to new} - T_{old\to new} \\
&= \frac{D}{S_{Net}}\left(\frac{xN}{(N+x)^2} - \frac{1-p_0}{r}\right) \\
&= \frac{D}{rS_{Net}}\left(\frac{N}{N+x}\sum_{i=1}^{r} ip_i - \sum_{i=1}^{r} p_i\right) \text{ using the properties on } p_i \\
&\geq \frac{D}{rS_{Net}}\left(\frac{N}{N+x}\frac{N+x}{x}\sum_{i=1}^{r} p_i - \sum_{i=1}^{r} p_i\right) \text{ using the assumption} \\
&\geq 0
\end{aligned}
$$

$$
\begin{aligned}
&T_{new\to new} - T^{balanced}_{\to new} \\
&= \frac{D}{S_{Net}}\left(\frac{N}{rx}\sum_{i=2}^{r}(i-1)p_i - \frac{xN}{(N+x)^2}\right) \\
&= \frac{D}{S_{Net}}\left(\frac{N}{rx}\sum_{i=2}^{r}(i-1)p_i - \frac{N}{r(N+x)}\sum_{i=0}^{r} ip_i\right) \text{ with Lemma 5.2} \\
&= \frac{ND}{xrS_{Net}}\left(\sum_{i=1}^{r} ip_i - \sum_{i=1}^{r} p_i - \frac{x}{N+x}\sum_{i=1}^{r} ip_i\right) \\
&= \frac{ND}{xrS_{Net}}\left(\frac{N}{N+x}\sum_{i=1}^{r} ip_i - \sum_{i=1}^{r} p_i\right) \\
&\geq \frac{ND}{xrS_{Net}}\left(\frac{N}{N+x}\frac{N+x}{N}\sum_{i=1}^{r} p_i - \sum_{i=1}^{r} p_i\right) \text{ using the assumption} \\
&\geq 0
\end{aligned}
$$

QED

Property 5.11:

If $T_{old \to new} \geq T_{new \to new}$, *then* $T_{old \to new} \geq T_{\to new}^{balanced} \geq T_{new \to new}$.

Proof:

Basically the same as Property 5.10.

QED

Property 5.12:

$T_{recv} \geq T_{\to new}^{balanced}$

Proof:

$$T_{recv} - T_{\to new}^{balanced} = \frac{N^2 D}{(N+x)^2 S_{Net}} \geq 0$$

QED

Property 5.13:

$t_{com} = max(T_{old \to new}, T_{recv})$

Proof:

The commission time is the maximum between T_{recv} (time to receive data) and the time to send the data: $T_{old \to new}$ (if $T_{new \to new} \leq T_{old \to new}$) or $T_{old \to new}^{balanced}$ (if $T_{new \to new} \geq T_{old \to new}$).

After applying Properties 5.10, 5.11, and 5.12, we have

$$t_{com} = max(T_{old \to new}, T_{recv}).$$

QED

Property 5.14:

$$T_{write} = \frac{ND}{(N+x)S_{Write}}$$

Proof:

$$T_{write} = \frac{D'}{S_{Write}}.$$

$$T_{write} = \frac{ND}{(N+x)S_{Write}}.$$

QED

Property 5.15:

$$T_{old \to new} = \frac{D(1-p_0)}{rS_{Read}}$$

Proof:

$$T_{old \to new} = \frac{D_{old \to new}}{S^{old}_{Read}}.$$

where $S^{old}_{Read} = NS_{Read}$ is the aggregated reading speed of the old nodes.
Thus,

$$T_{old \to new} = \frac{D(1-p_0)}{rS_{Read}}.$$

QED

Property 5.16:

$$t_{com} = max(T_{write}, T_{old \to new}).$$

Proof:

The commission cannot be faster than reading all unique data and writing it.

QED

Property 5.17:

$$T^{alldata}_{old \to new} = \frac{xD}{(N+x)S_{Read}}$$

Proof:

$$\begin{aligned} T^{alldata}_{old \to new} &= \frac{D'}{S_{Read}} \\ &= \frac{xD}{(N+x)S_{Read}}. \end{aligned}$$

QED

Property 5.18:

If $x \geq \frac{NS_{Read}}{S_{Write}}$*, new nodes can forward data, and* $T^{balanced}_{old \to new} = \frac{xND}{(N+x)^2} \frac{S_{Read} + S_{Write}}{S_{Read} S_{Write}}$.

Proof:

Let y be the proportion of time used to write data on new nodes.
During the time t,
$D^R_o = tNS_{Read}$ data is read from the old nodes,
$D^R_n = t(1-y)xS_{Read}$ data is read from the new nodes,
$D^W_o = 0$ data is written on old nodes,

$D_n^W = tyxS_{Write}$ data is written on new nodes.

$$D_n^W + D_o^W = D_o^R + D_n^R$$

Thus, $y = \frac{(N+x)}{x}\frac{S_{Read}}{S_{Read}+S_{Write}}$ and $T_{old\to new}^{balanced} = \frac{D_{\to new}}{xyS_{Write}}$

$$T_{old\to new}^{balanced} = \frac{xND}{(N+x)^2}\frac{S_{Read}+S_{Write}}{S_{Read}S_{Write}}.$$

This is possible if and only if $y \leq 1$, thus $x \geq \frac{NS_{Read}}{S_{Write}}$.

QED

Property 5.19:

$$t_{com} = \begin{cases} T_{write} & \text{if } x \leq \dfrac{NS_{Read}}{S_{Write}}, \\ max(T_{old\to new}, T_{old\to new}^{balanced}) & \text{otherwise} \end{cases}$$

Proof:

If $T_{Write} \leq T_{old\to new}^{alldata}$, then the balanced strategy is used. Similarly to Property 5.10, we have

$$T_{Write} \leq T_{old\to new} \leq T_{old\to new}^{alldata}.$$

However, the old nodes still have to send the minimum amount of data for a duration of $T_{old\to new}$.

In the other case, writing is the bottleneck, and T_{Write} is the time needed for the commission.

$$t_{com} = max(T_{Write}, T_{old\to new}, T_{old\to new}^{balanced})$$

QED

Property 6.1:

$$D_{write} = xD$$

Proof:

Here x nodes are leaving the cluster, and each host D data (Assumption 4). Thus,

$$D_{write} = xD.$$

QED

Definition 6.1:

$$t_{decom} = \frac{D_{write}}{S_{write}^{cluster}}$$

Property 6.2:

$$S_{write}^{cluster} = S_{Net}(N-x)$$

Property 6.3:

$$t_{decom} = \frac{xD}{S_{Net}(N-x)}$$

Proof:

Using Definition 6.1 as well as Properties 6.2 and 6.1, we obtain the result.

QED

Definition 6.2:

$$p_i = \begin{cases} 0 & \text{if } i > r, \\ \frac{\binom{r}{i}\binom{N-r}{x-i}}{\binom{N}{x}} & \text{for } i \leq r. \end{cases}$$

Detail:

This is the probability for each object to have exactly i replicas on the x leaving nodes. This is modeled as a classical urn problem.

QED

Property 6.4:

$$R(N,x) = \begin{cases} 1 & \text{when buffering is not possible,} \\ \frac{\sum_{i=1}^{r} i p_i}{\sum_{i=1}^{r} p_i} & \text{otherwise.} \end{cases}$$

Proof:

If an object has $k > 0$ replicas on leaving nodes, it needs to be read once using the buffering and written k times on remaining nodes.

The data to write $D_{towrite}$ is, for each possible number of replicas on leaving nodes, the probability for an object to have that number of replicas on leaving nodes multiplied by the total amount of data to move and the number of times this object has to be written.

$$D_{towrite} = xD \sum_{i=1}^{r} i p_i$$

Similarly, D_{toread} is the data to read: for each possible number of replicas on leaving nodes, the probability of the data having that number of replicas on leaving nodes multiplied by the total amount of data and the number of times the data has to be read, which is one.

$$D_{toread} = xD \sum_{i=1}^{r} p_i$$

With $D_{towrite}$ and D_{toread}, we can deduce the ratio of data to write with respect to the data to read as $R(N,x)$. The ratio is 1 when no buffering is possible, since data must be read as many times as it is written.

$$R(N,x) = \begin{cases} 1 & \text{when buffering is not possible,} \\ \frac{\sum_{i=1}^{r} i * p_i}{\sum_{i=1}^{r} p_i} & \text{otherwise.} \end{cases}$$

QED

Property 6.5:

$$S_{write}^{cluster} = S_{Write}(N - x).$$

Proof:

Each of the remaining nodes can receive data with a throughput of S_{Write}

$$S_{write}^{cluster} = S_{Write}(N - x)$$

QED

Property 6.6:

$$S_{write}^{cluster} = \frac{NR(N,x)S_{Read}S_{Write}}{S_{Write} + R(N,x)S_{Read}}.$$

Proof:

To obtain the writing speed of the cluster in case of storage bottleneck, we first consider the amount of data read and written during duration t. Leaving nodes can read at full speed, and remaining nodes can read and write. We denote as d the proportion of time that remaining nodes spend writing.

$$\begin{cases} d_{written} = & t \cdot (N - x) \cdot d \cdot S_{Write} \\ d_{read} = & t \cdot (N - x) \cdot (1 - d) \cdot S_{Read} + t \cdot x \cdot S_{Read} \end{cases}$$

$$R(N,x) = \frac{(N - x) \cdot d \cdot S_{Write}}{(N - x) \cdot (1 - d) \cdot S_{Read} + x \cdot S_{Read}}$$

Thus,

$$d = \frac{R(N,x) \cdot S_{Read} \cdot N}{(N - x)(S_{Write} + R(N,x) \cdot S_{Read})}$$

We find the following.

$$\begin{aligned} S_{write}^{cluster} &= (N - x) \cdot d \cdot S_{Write} \\ &= \frac{NR(N,x)S_{Read}S_{Write}}{S_{Write} + R(N,x)S_{Read}} \end{aligned}$$

QED

Property 6.7:

$$T(N,x) = \frac{NS_{Write}}{R(N,x)S_{Read} + S_{Write}}$$

Proof:

We consider d from the proof of Property 6.6.

Here d has two constraints: $0 \leq d \leq 1$. $d \geq 0$ means $S_{Read} \geq 0$ which is implicit, and $d \leq 1$ results in $T(N,x)$.

QED

Property 6.8:

$$t_{decom} = \begin{cases} \dfrac{xD}{S_{Write}(N-x)} & \text{if } x \geq T(N,x), \\ \dfrac{x \cdot D \cdot (S_{Write} + R(N,x)S_{Read})}{N \cdot R(N,x) \cdot S_{Read} \cdot S_{Write}} & \text{in other cases.} \end{cases}$$

Proof:

Using Definition 6.1 and Properties 6.1, 6.5, 6.6, and 6.7, and after simplification, we have the result.

QED

Property 6.9:

$$t_{decom} = \max\left(\frac{xD}{(N-x)S_{Net}}, \frac{D}{S_{Net}}\right)$$

Proof:

During the decommission, the total amount of data to transfer is xD.

$N-x$ remaining nodes can receive it at a throughput of S_{Net}.

x leaving nodes can send it at a throughput of S_{Net}.

QED

Property 6.10:

$$t_{decom} = \max\left(\frac{xD}{(N-x)S_{Write}}, \frac{D}{S_{Read}}\right)$$

Proof:

During the decommission, the total amount of data to transfer is xD.

$N-x$ remaining nodes can write it at a throughput of S_{Write}.

x leaving nodes can read it at a throughput of S_{Read}.

QED

Property 6.11:

In the case of a network bottleneck, decommissioning a set of nodes in k consecutive steps takes as much time as decommissioning the same nodes all at once.

Proof:

We demonstrate that the time to decommission in k steps is $t_{decom}(k) = \frac{xD}{S_{Net}(N-x)}$ by recurrence.

For $k = 1$, decommission in one step is a simple decommission so they have the same duration.

For $k > 1$, we assume that decommissioning in $k-1$ steps takes $t_{decom}(k-1) = \frac{xD}{S_{Net}(N-x)}$. Then, we denote as y the number of nodes decommissioned in the first step on the total of N nodes. Thus,

$$t_{decom}(k) = \frac{y \cdot D}{S_{Net}(N-y)} + t_{decom}(k-1)$$

Then, for the $k-1$ other steps, the cluster as a size of $N-y$ and each node hosts $\frac{N \cdot D}{N-y}$. Thus,

$$\begin{aligned} t_{decom}(k) &= \frac{yD}{S_{Net}(N-y)} + \frac{(x-y)\frac{ND}{N-y}}{S_{Net}(N-y-(x-y))} \\ &= \frac{1}{S_{Net}}(\frac{yD}{N-y} + \frac{(x-y)\frac{ND}{N-y}}{N-x}) \\ &= \frac{1}{S_{Net}}\frac{(N-x)yD + (x-y)ND}{(N-y)(N-x)} \\ &= \frac{1}{S_{Net}}\frac{xD}{(N-x)} \end{aligned}$$

Thus, the time to decommission in k steps is $t_{decom}(k) = \frac{xD}{S_{Net}(N-x)}$.

QED

Definition 8.1:

$$p_i = \begin{cases} 0 & \text{if } i > r, \\ \frac{\binom{r}{i}\binom{N-r}{x-i}}{\binom{N}{x}} & \text{for } i \leq r. \end{cases}$$

Detail:

This is the probability of each object having exactly i replicas on the x leaving nodes. It is modeled as a classical urn problem.

QED

Definition 8.2:

$$D_{avail} = \begin{cases} NDp_r/r & \text{if } x \geq r \\ 0 & \text{in other cases.} \end{cases}$$

Detail:

If $x < r$, no data is present on the leaving nodes because two replicas cannot be placed on a single node.

If $x >= r$, only the objects that have r replicas on the leaving nodes need to be secured by moving one replica to remaining nodes.

QED

Definition 8.3:

$$D_{stab} = xD$$

Detail:

All the data that was present on the leaving nodes needs to be recreated in order to stabilize the system.

$D_{stab} = xD = \sum_{i=1}^{r} ip_i \frac{ND}{r}$

QED

Property 8.1:

$$t_{avail} = \begin{cases} \dfrac{NDp_r}{rxS_{Net}} & \text{if } x \leq N/2 \\ \dfrac{NDp_r}{r(N-x)S_{Net}} & \text{otherwise.} \end{cases}$$

Proof:

If the sending nodes are the bottleneck, their throughput is $S_{Send}^{avail} = xS_{Net}$.

In this case the time to availability is $t_{avail} = \frac{p_r ND}{rxS_{Net}}$.

If the receiving nodes are the bottleneck, their throughput is $S_{Receiving}^{avail} = (N-x)S_{Net}$

In this case the time to availability is $t_{avail} = \frac{p_r ND}{r(N-x)S_{Net}}$.

Overall, $t_{avail} = \begin{cases} \frac{NDp_r}{rxS_{Net}} & \text{if } x \leq N/2 \\ \frac{NDp_r}{r(N-x)S_{Net}} & \text{otherwise.} \end{cases}$

QED

Property 8.2:

$$t_{over} = \begin{cases} \dfrac{(N-2x)NDp_r}{rx(N-x)S_{Net}} & \text{if } x \leq N/2 \\ 0 & \text{otherwise.} \end{cases}$$

Proof:

In case the data sent by the leaving nodes cannot saturate the network of the remaining nodes, the remaining nodes can exchange data before the data-safekeeping phase is over.

Let S_{Recv} be throughput at which the remaining nodes can receive the data.

$$\begin{aligned} t_{over} &= t_{avail} - D_{avail}/S_{Recv} \\ &= \frac{p_r ND}{rxS_{Net}} - \frac{p_r ND}{r(N-x)S_{Net}} \\ &= \frac{(N-2x)p_r ND}{rx(N-x)S_{Net}} \end{aligned}$$

QED

Property 8.3:

$$\begin{aligned} t_{stab} &= t_{avail} - t_{over} + \frac{D_{stab} - D_{avail}}{S_{Recv}} \\ &= \frac{xD}{(N-x)S_{Net}}. \end{aligned}$$

Proof:

Let S_{Recv} be throughput at which the remaining nodes can receive the data. S_{Recv} is also the rate at which data can be exchanged during the stabilization phase as nodes can send and receive data at the same time.

$$\begin{aligned} t_{stab} &= t_{avail} + \frac{D_{stab} - D_{avail}}{S_{Recv}} - t_{over} \\ &= t_{avail} + \frac{D_{stab} - D_{avail}}{S_{Recv}} - t_{avail} + \frac{D_{avail}}{S_{Recv}} \\ &= \frac{D_{stab}}{S_{Recv}} \\ &= \frac{xD}{(N-x)S_{Net}} \end{aligned}$$

QED

Property 8.4:

$$t_{avail} = \begin{cases} \dfrac{NDp_r}{rxS_{Read}} & \text{if } x \leq \dfrac{NS_{Write}}{S_{Read} + S_{Write}} \\ \dfrac{NDp_r}{r(N-x)S_{Write}} & \text{otherwise.} \end{cases}$$

Proof:

In the case of a reading bottleneck,

$$t_{avail} = \frac{D_{avail}}{xS_{Read}}$$

In case of a writing bottleneck,

$$t_{avail} = \frac{D_{avail}}{(N-x)S_{Write}}$$

A reading bottleneck occurs if

$$\begin{aligned} \frac{D_{avail}}{xS_{Read}} &> \frac{D_{avail}}{(N-x)S_{Write}} \\ (N-x)S_{Write} &> xS_{Read} \\ x &< \frac{NS_{Write}}{S_{Read}+S_{Write}} \end{aligned}$$

QED

Property 8.5:

$$t_{over} = \begin{cases} \dfrac{(N-x)S_{Write} - xS_{Read}}{x(N-x)S_{Read}S_{Write}} & \text{if } x \leq \dfrac{NS_{Write}}{S_{Read}+S_{Write}} \\ 0 & \text{otherwise.} \end{cases}$$

Proof:

The available time to exchange data during the data-safekeeping phase is the time to availability nodes minus the time needed to write the data onto storage.

In the case of a reading bottleneck:

$$\begin{aligned} t_{over} &= t_{avail} - D_{avail}/(S_{Write} * (N-x)) \\ &= \frac{NS_{Write} - x(S_{Write}+S_{Read})}{x(N-x)S_{Read}S_{Write}}. \end{aligned}$$

QED

Property 8.6:

$$R = \begin{cases} 1 & \text{in case of in-memory storage,} \\ \dfrac{\sum_{i=1}^{r-1} p_i}{(r-1)p_r + \sum_{i=1}^{r-1} ip_i} & \text{otherwise.} \end{cases}$$

Proof:

The data that must be read is a replica from each of the objects that will be lost when the leaving nodes leave except for the ones that have been transferred by the leaving nodes (when objects have all their replicas on leaving nodes); those replicas can be buffered upon reception from leaving nodes.

The data that must be written is all the data that was on the leaving nodes except for the data that was written during the data-safekeeping phase: one replica for each of the objects that were entirely stored on leaving nodes.

QED

Property 8.7:

$$S_{eff} = \frac{(N-x)S_{Write}S_{Read}}{S_{Read} + RS_{Write}}$$

Proof:

During a time t, data is read and written with a ratio R.
Let $t = t_{Read} + t_{Write}$.

$$R = \frac{t_{Read}(N-x)S_{Read}}{t_{Write}(N-x)S_{Write}}$$
$$t_{Read} = t\frac{RS_{Write}}{S_{Read} + RS_{Write}}$$

S_{eff} is the amount of data written during t divided by t.

$$\begin{aligned} S_{eff} &= \frac{t_{Write}(N-x)S_{Write}}{t} \\ &= \frac{(N-x)S_{Write}S_{Read}}{S_{Read} + RS_{Write}} \end{aligned}$$

QED

Property 8.8:

$$t_{stab} = \frac{D}{N-x}\left(\frac{R}{S_{Read}} + \frac{1}{S_{Write}}\right)\left(x - \frac{Np_r}{r}\right) + \frac{NDp_r}{r(N-x)Sw}$$

Proof:

$$\begin{aligned} t_{stab} &= t_{avail} + \frac{D_{stab} - D_{avail}}{S_{eff}} - t_{over} \\ &= t_{avail} + \frac{D_{stab} - D_{avail}}{S_{eff}} - t_{avail} + \frac{D_{avail}}{(N-x)S_{Write}} \\ &= \frac{D}{N-x}\left(\frac{R}{S_{Read}} + \frac{1}{S_{Write}}\right)(x - \frac{Np_r}{r}) + \frac{NDp_r}{r(N-x)Sw} \end{aligned}$$

QED

Property 8.9:

$$t_{avail} = \sum_{i=1}^{k} ip_{r-k+i}\frac{ND}{r(N-x)S_{Net}}$$
$$t_{stab} = \frac{xD}{(N-x)S_{Net}}$$

Proof:

Since the data is distributed uniformly on all nodes, all nodes host some data that must be replicated during the data-safekeeping phase. The bottleneck is the reception of the data (it holds as long as $r \geq 2$). From this, we deduce t_{avail}.

The rest of the data to transfer for the stabilization can be sent and received with the same throughput by the remaining nodes. From this, we can deduce t_{stab}.

QED

Property 8.10:

Let R_{avail} be the ratio of the amount of data to read on the data to write during the data-safekeeping phase.

$$R_{avail} = \begin{cases} 1 & \text{for in-memory storage} \\ \frac{\sum_{i=1}^{k} p_{r-k+i}}{\sum_{i=1}^{k} p_{r-k+i}} & \text{in other cases.} \end{cases}$$

$$t_{avail} = \begin{cases} \sum_{i=1}^{k} ip_{r-k+i} \frac{D}{r} \frac{S_{Read} + R_{avail} S_{Write}}{S_{Write} S_{Read}} & \text{if } x < \frac{R_{avail}(N-x) S_{Write}}{S_{Read}} \\ \sum_{i=1}^{k} ip_{r-k+i} \frac{ND}{r(N-x) S_{Write}} & \text{in other cases.} \end{cases}$$

Proof:

The ratio of data read on the amount of data to be written is determined with p_r: if an object does not have enough replicas on the remaining nodes, a comparable number of replicas must be moved. Thanks to buffering, however, each object can be read once.

Then, the transfer rate in the cluster can be determined with R_{avail} and with the fact that the drives cannot read and write at the same time. Moreover, leaving nodes do not have to write, so they can read data all the time. t_{avail} is deduced from these.

QED

Property 8.11:

Let R_{stab} be the ratio of the amount of data to read on the amount of data to write during the stabilization phase.

$$R_{stab} = \begin{cases} 0 & \text{in case of storage in-memory} \\ \frac{\sum_{i=1}^{r-k} p_i}{\sum_{i=1}^{r} ip_i - \sum_{i=1}^{k} ip_{r-k+i}} & \text{in other cases.} \end{cases}$$

$$t_{stab} = t_{avail} + \left(\sum_{i=1}^{r} ip_i - \sum_{i=1}^{k} ip_{r-k+i} \right) \frac{ND}{r} \frac{R_{stab} S_{Write} + S_{Read}}{(N-x) S_{Read} S_{Write}}$$

Proof:

Similar to the proof of Property 8.10, the ratio of data read on the amount of data written is determined with the p_r. However, the data that has been read during the data-safekeeping phase is assumed to have been buffered and does not need to be read a second time.

Then, the transfer rate in the cluster can be determined with R_{stab}, and the fact that the drives cannot read and write at the same time. t_{stab} is deduced from these and t_{avail}.

QED

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

école normale supérieure rennes

Titre: Malléabilité des Systèmes de Stockage Distribués : Des Modèles à la Pratique

Mot clés : Systèmes de Stockage Distribués, Malléabilité, Élasticité, Modélisation, Benchmark

Résumé : Le Cloud, avec son modèle économique, offre la possibilité d'un gestion élastique des ressources; les utilisateurs peuvent louer des ressources selon leurs besoins. Cette élasticité permet de réduire les coûts énergétiques et financiers, et aide les applications à s'adapter aux charges de travail variables.

Les applications manipulant de grandes quantités de données exécutées dans le Cloud ou sur des supercalculateurs sont souvent colocalisées avec un système de stockage distribué pour garantir un accès rapide aux données. Bien que de nombreux travaux aient été proposés pour redimensionner dynamiquement les capacités de calcul pour s'ajuster à la charge de travail, le stockage n'est pas considéré comme malléable (capable d'être redimensionné dynamiquement) puisque les transferts de grandes quantités de données nécessaires sont considérés trop lents. Cependant, le matériel et les techniques de stockage ont évolué et cette hypothèse doit être réévaluée.

Dans cette thèse, nous présentons une étude sous différents angles des opérations de redimensionnement des systèmes de stockage distribués. Nous commençons par modéliser la durée minimale de ces opérations pour évaluer leur vitesse potentielle. Puis, nous développons un benchmark conçu pour mesurer la viabilité de la malléabilité d'un système de stockage sur une plateforme donnée. Finalement, nous implémentons un gestionnaire d'opérations de redimensionnement pour systèmes de stockage distribués qui décide et organise les transferts de données requis par ces opérations.

Title: Towards Malleable Distributed Storage Systems: From Models to Practice

Keywords : Distributed Storage Systems, Malleability, Elasticity, Modeling, Benchmarking

Abstract: The Cloud, with its pay-as-you-go model, gives the possibility of elastic resource management; users can claim and release resources as needed. This elasticity leads to financial and energetical cost reductions, and helps applications to cope with varying workloads.

Distributed cloud and HPC applications processing large amounts of data are often co-located with a distributed storage system in order to ensure fast data accesses. Although many works have been proposed to dynamically rescale the processing part of such systems to match their workload, the storage is never considered as malleable (able to be dynamically rescaled) since moving massive amounts of data around is assumed to be too slow in practice. However, in recent years hardware and storage techniques have evolved and this assumption needs to be revisited.

In this thesis, we present a study of the rescaling operations in distributed storage systems approached from different angles. We start by modeling the minimal duration of rescaling operations to estimate their potential speed. Then, we develop a benchmark to measure the viability of distributed storage system malleability on a given platform. Last, we implement a rescaling manager for distributed storage systems that decides and organizes the data transfers required during a rescaling operation.