Elasticity in Distributed File Systems

Cyril Séguin, Gaël Le Mahec, Benjamin Depardon

▶ **To cite this version:**

Cyril Séguin, Gaël Le Mahec, Benjamin Depardon. Elasticity in Distributed File Systems. [Research Report] MIS. 2016. hal-01335978

HAL Id: hal-01335978
https://inria.hal.science/hal-01335978

Submitted on 22 Jun 2016

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

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

Dynamic Elasticity in Distributed File Systems

Cyril Séguin[1], Gaël Le Mahec[1], and Benjamin Depardon[2]

[1] University of Picardie Jules Verne - MIS Laboratory, Amiens, FRANCE,
cyril.seguin@u-picardie.fr, gael.le.mahec@u-picardie.fr
[2] b.depardon@gmail.com

Abstract. In IaaS Cloud Computing platforms, *elasticity* offers to users the possibility to adjust the number of resources to the current workload, taking into account peak and trough periods (high and low activity) by powering down/up some resources. In previous work, we solved the challenges raised by the extension of elasticity to storage resources in the context of *static data popularities*. In this paper, we address the same challenges in the context of *dynamic data popularities*. We propose 3 solutions that estimate the data popularities, and an algorithm that reduces or increases the number of used server according to the period activity. Based on the *platform cost* (the resources are paid for the time they are used), our simulations on top of SimGrid show that we could either improve HDFS' performance by up to 99% while providing similar cost, or we could reduce cost up to 50% while providing similar performance.

1 Introduction

Since about a decade, Cloud Computing platforms have become very popular. At their beginning, they were mainly used for business or commercial applications, but they have quickly extended to almost all IT activities. From Customers Relation Management to High Performance Computing, Cloud Computing is now an unavoidable technology. Among the different Cloud models, Infrastructure as a Service (IaaS) is one of the most used for High Performance Computing. Indeed, these platforms can punctually offer very large computing infrastructures to researchers who do not have the needs or the funds to acquire their own computing resources. On these platforms it is possible to get access to thousands of resources for some hours, paying exactly what is needed for a particular problem to solve. An essential characteristic of this kind of platform is the *elasticity*, i.e., the capability to be dynamically extended or reduced by adding or removing resources. Applications take into account these variations of resources availability and distribute computing tasks among the computing resources to achieve a good load balancing. Cloud elasticity offers to users the possibility to adapt the computing platform to workloads with peak (high activity) and trough periods (low activity) by powering down/up some resources.

Today, data growth being a major concern that administrators of datacenters have to deal with, the main challenge for large scale computing platforms is to give access to efficient and reliable storage systems. Some frameworks have been developed to deal with such data intensive applications. Nowadays, the main storage solution used by supercomputers, clusters and datacenters is Distributed File System (Ceph [10], Gluster [5], HDFS [7], . . .). However, extending elasticity to the DFSs represents some challenges. Indeed, these DFSs allow for voluntarily

adding or removing resources but it usually implies data movements, introducing delays, bandwidth consumption, and sometimes loss of data.

In what follows, we call *popularity* of a data the number of requests on this data for a given time interval. In our previous work [8], we addressed these challenges in a context of *static data popularity*, that is the number of requests on each data is known and regular during the workload. In this context, we were able to remove or add storage resources to face with a workload activity while maintaining data availability and good performance. In this paper, we address the same challenges in a context of *dynamic data popularity*, that is we cannot make assumption about the number of requests on each data before the requests' submissions. We focus on reducing the number of used resources and on increasing read access performance facing with data popularities variations. These two conflicting goals are grouped into the notion of *platform cost* that we aim to minimize. Our solutions rely on a dynamic adaptation of the replication factor (number of replicas per data) to the data popularities. We provide several algorithms that create or remove replicas that are distributed across the fewest possible servers.

2 Related work

Distributed file systems, such as Ceph [10], HDFS [7], Gluster [5], Lustre [6], divide data into fixed-size blocks and replicate them across multiple storage resources. Each data has the same replication factor and each storage resource almost stores the same amount of data. These systems provide good performance by using as many servers as possible and by aggregating their throughput. However, they are not compatible with the cloud platform's elasticity. The data placement allows them to safely remove only $r - 1$ servers, where r is the replication factor, without loss of data. Removing more servers is possible but is likely to be hard. Indeed, it is a special case of the set covering problem which is an NP-complete problem [9]. Moreover, the data placement and the replication factor are not based on data popularities. Multiple data with a lot of requests may be stored on the same server and may have the same replication factor as data with fewer requests. This server can quickly become a performance bottleneck.

Some DFSs, such as ERMS [4], address this last problem. They focus on adapting the replication factor to the data popularities that they try to estimate based on historic popularities. Based on these popularities, data are ranked into two sets: *hot* and *cold*. ERMS respectively add or remove replicas of hot or cold data. However these systems focus on performance and fault tolerance but not on platform cost (i.e. the number of resources used and how long they are used). Moreover they do not provide a data placement based on their popularities. It exists some DFSs that address the problem of distributed file system elasticity in a context of dynamic resources extension/reduction (Rabbit [1], Sierra [9], SpringFS [11]). But, they address this problem by ensuring the *power proportionality*. That is the power used by the platform should be proportional to the performance at any time. These systems try to estimate the future load based on historic load. According to this future load, they add or remove resources to keep the proportionality. However, they do not take into account the popularities of the data to adapt their placements and replication factors.

3 Challenges

Data intensive applications frequently and concurrently request data. These requests can concern the same data on different servers or different data on the same server. According to data popularities and to reach good performance, the replication factor of each data can differ. In peak periods, by creating several replicas, the requests can be distributed among these replicas to balance the servers load. In trough periods, by removing replicas, some servers can be stopped. For our problem, defining peak and trough periods, and choosing the replicas to be created or removed is critical. But removing and/or creating replicas is not always a good solution. Reducing the number of active servers may have impact on performance, as well as performance improvements may have impact on the number of active servers. So, one of our challenges is to find a good trade-off between reducing the number of storage servers and improving the platform responsivity by adding more replicas. Finally, data popularities change over time. Estimating these data popularities variations is critical to get close to optimal performance while using the fewest possible servers. In the next sections, we introduce the solutions we propose to rise to these challenges.

3.1 Assumptions

In a DFS, data are divided into fixed-size blocks. For the rest of this paper, let the term *data* denotes a data block. We consider the DFSs are deployed on a cloud platform composed of homogeneous resources (same cpu, memory, storage capacity, network bandwidth...). Clients applications and servers are all located inside the platform (no access from the outer world). Each machine have a unique network interface and they are all connected to a unique network switch. So, the bandwidth available for the communications depends only on the number of concurent accesses to the same server. We use the same definition of data popularity that in ERMS [4].

In cloud platform, the resources are paid for the time they are used. This is what we call *platform cost*. Different platform cost models exist: per second-based billing, per minute-based billing or per hour-based billing. The platform cost differs depending on the billing model. In the rest of this paper, we consider the per second-based billing model. Our goal is to minimize this platform cost. In other words, our goal is to find a trade-off between performance, and the number of servers used to reach this performance. Optimal performance can be reached by using as many servers and by creating as many replicas as there are requests. Our goal can be defined again as: get as close as possible to optimal performance by using the fewest possible servers.

Let denote tw the time window representing the billing model, and Tw_{exec} the number of time windows needed to complete all the requests. Let denote S_i the number of active servers (servers that are powered on) during a time window i. Let define the platform cost to minimize:

$$Minimize : C = \sum_{i=1}^{Tw_{exec}} (S_i \times tw)$$

The platform cost can be minimized in two ways: reducing the number of servers in trough periods or reducing the number of time windows in peak periods.

3.2 Trough and peak periods

Trough periods are the periods in which all the requests can be completed in only one time window tw. Peak periods are the periods that need more than one time window tw to complete all the requests. Using the ERMS notation [4], we define a *cold server* as a server that can complete the requests it received in one time window. And we define a *hot server* as a server that needs several time windows to complete its requests. To minimize the platform cost, we try to remove the cold servers, and relieve the hot servers by creating new replicas.

3.3 Replicas removal

The *cold set* and the *hot set* are respectively the set of cold servers and the set of hot servers. At trough periods, once the cold set is created, we try to remove the servers one by one. To maintain the data availability, we prevent from removing the last remaining data's replica.

Formally, let denote D a set of fixed-size t data and S_a the set of active servers (servers that are powered up). Let denote r_d the data d replication factor, and d_i the i^{th} replica of d. Let denote $\delta_s^{d_i} \in \{0, 1\}$ a distribution function that output 1 if replica i of data d is on server s, 0 otherwise. The data availability (1) challenge, called DAv, can be formalize as the following constraint:

$$\forall d \in D, \sum_{s \in S_a} \sum_{i=1}^{r_d} \delta_s^{d_i} \geq 1 \tag{1}$$

Removing a server implies removing data's replica. But removing a data's replica can increase the number of requests on other servers that store the data, impacting on their performance to serve the requests. So, a server can be shut down only if it does not hold the last replica of a data, and if the servers impacted by this removal can complete their requests during the same number of time windows as before the removal.

Let denote $Rold\_tw_s$ and $Rnew\_tw_s$ the number of time windows needed for a server s to complete its requests respectively before and after the removal. The performance removal challenge (2), called PRC, can be formalized as the following constraint:

$$\forall s \in S_a, Rnew\_tw_s \leq Rold\_tw_s \tag{2}$$

3.4 Replicas creation

Creating replicas to relieve hot servers allows to distribute the data's requests across more replicas. So, a replica is created if the number of time windows needed by a hot server to complete its requests after the creation is less than before the creation. So, we ensure that adding a server always have a positive impact on the general performance of the platform.

Let denote $Aold\_tw_s$ and $Anew\_tw_s$ the number of time windows needed for the hot server s to complete its requests respectively before and after the

creation. The performance creation challenge (2), called *PCC*, can be formalize for each hot server s as the following constraint:

$$Anew\_tw_s < Aold\_tw_s \tag{3}$$

New replicas must be stored either on an active server or on a new server (i.e. a server that was inactive before the replica creation). Storing a new replica on an active server impacts its performance. Storing a replica on an inactive server impacts the number of active servers. To use the fewest possible servers, we choose as a priority to create a new replica on an existing server. This must respect the following constraints: first, there is enough storage capacity on the *destination* server; second, the number of time windows needed by the destination server to complete its request after the creation is the same as before the creation. Otherwise, a server is activated and the replica is stored on this server. Furthermore, a new replica must be created by copying the data from an active server. This impacts the *source* server performance. Consequently, we choose to copy the data from the less loaded server holding it. This must respect the following constraint: the number of time windows needed by the source server to complete its request after the creation is the same as before the creation.

Formally, let denote T the storage capacity of the destination server. Using $Aold\_tw_s$ and $Anew\_tw_s$ for the destination server or the source server s, the performance source and destination challenge , called *PSDC*, can be formalized as the following constraints:

$$\sum_{d \in D} \left(\sum_{i=1}^{r_d} \left(\delta_s^{d_i} \times t \right) \right) \leq T \tag{4}$$

$$Anew\_tw_s \leq Aold\_tw_s \tag{5}$$

Removal and creation processes are described in Algorithm 1 and Algorithm 2.

3.5 Data popularity variations

Data popularity can change over time, and depends on the workload generated by data intensive applications. In some workloads, number of requests to the data can be constant, while in other workloads, these number can vary. These variations can be predictable or not. So, the replication factor of each data should be dynamically updated. This implies that replicas removal/creation process should be applied several times during the workloads execution. A workload execution can be split into several periods of requests. For each period tw_i the data popularity $pop_d^{tw_i}$ is computed and can be used to define the data popularity for the next periods. Consequently, we have defined 3 functions, that estimate the popularities of each data that are likely to be the data popularities for the next periods. The first one called *current* uses the data popularities measured during the current period. The second one called *mean* computes for each data the average popularity of all completed periods. Finally, the last function called *median* computes for each data the median popularity of all completed periods.

Algorithm 1 REMOVE SERVER

```
1: for all s ∈ S_a do
2:     if s is cold then
3:         for all d stored on s do
4:             if DAv then
5:                 Remove d from s
6:             else
7:                 Abort, try another s
8:             end if
9:         end for
10:        if PRC then
11:            Remove s
12:        else
13:            Abort, try another s
14:        end if
15:    end if
16: end for
```

Algorithm 2 ADD REPLICAS

```
1: for all s ∈ S_a do
2:     do
3:         if s is hot then
                                  ▷ Existing server
4:             Find d on s with max pop.
5:             Find source server s'
6:             Find destination server s''
7:             Create d from s' to s''
8:             if ! PCC OR ! PSDC then
                                  ▷ New server
9:                 Abort
10:                Add new server s'''
11:                Create d from s' to s'''
12:            end if
13:        end if
14:    while ! PCC OR ! PSDC
15: end for
```

4 Evaluation

Our goal is to minimize the platform cost. Consequently, our solutions are analyzed comparing the platform cost but also comparing the platform cost components: the average access time and the number of used server per time window . Our solutions rely on the functions used to estimate the data popularities. These solutions are evaluated facing with data access patterns composed of different data popularities evolutions. Our solutions are also evaluated using workload traces provided by the Statistical Workload Injector for MapReduce (SWIM) [3].

4.1 Experimental setup

To evaluate our solutions, we simulate an homogeneous cloud platform using the SimGrid 3.11 framework [2], a "Versatile Simulation of Distributed Systems". This platform is composed of 60 storage servers, each offering 700GB of storage capacity. To simulate multiple requesters, we use one client linked to each storage server with a 2GB/s bandwidth. Two requests on two different servers use two distinct links. Two requests on two different servers are processed in parallel, but two requests on a single server are sequentially processed using a FIFO queue. Finally, we use 30 seconds time windows.

To compare our algorithms with the existing solutions, we have reproduced the HDFS behaviour. HDFS is one of the most popular DFS and is especially used with map-reduce by the data intensive applications. When using only one rack of servers, HDFS distributes the replicas among the servers in such a way that each server stores almost the same amount of data.

4.2 Data sets

To evaluate our algorithms, we built different patterns of data popularities. In the *constant* pattern all the data have the same popularity during the workload. In the *increasing* and *decreasing* patterns the data popularities respectively constantly increase/decrease during the workload. In the *random* pattern the data

popularities are varying randomly. In the *gaussian* pattern, the data popularities are varying following a gaussian distribution along the time (starting low, quickly increasing to reach a peak, then quickly decreasing). To simulate a perfect periodical variation of requests number for each data, we define a *sinusoidal* pattern where data popularities gradually increase then gradually decrease and so on. We also defined a pattern with sinusoidal variation of the popularity of each data but with a different phase for each data (*singap* pattern). These data sets are composed of 50 data requested during 33 time windows and are shown in Figure 1. To confront our algorithms to *real world*, we have used the workload traces provided by the Statistical Workload Injector for MapReduce (SWIM) [3]. These workloads consist of a one month job trace of a Facebook production cluster (3000 machines). These workloads are composed of about 2,000 data, and are shown in Figure 2. To correspond to the other patterns and make easier the comparisons, the popularity of each data have been multiplied by 3 and the workload have been repeated as required to reach the same amount of time windows.

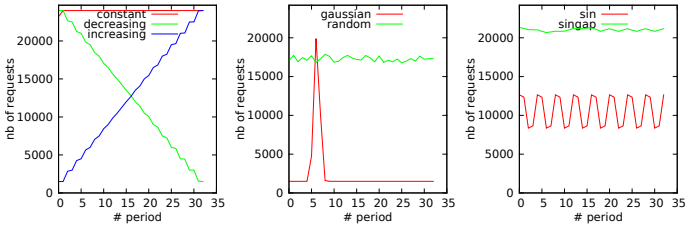

Fig. 1: Total number of requests occurring at each time window for different pattern

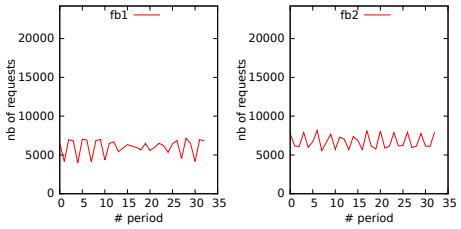

Fig. 2: Total number of requests occurring at each time window for SWIM pattern

4.3 Data access pattern

We use the different patterns to first identify our best solution facing with different data popularities evolutions. The results are shown from Figure 3 to Figure 7. The *current* method is our best solution. It has a cost lower than or equal to the *mean* and *median* methods and provides lower or equal data access time for all the patterns except the *gaussian* one. It relies on the current time period to estimate the future data popularities and is adapted to a gradual evolution. It

reduces the platform cost by removing servers in trough periods and by reducing data access time in peak periods. But, it is vulnerable when this evolution quickly increases or decreases (as our *gaussian* pattern Figure 6) because it acts with one period delay. In that case, the *mean* and *median* methods are more adapted. They slowly reduce or increase the number of servers as seen in the *decreasing* pattern (Figure 4) and the *increasing* pattern (Figure 5). They are also more adapted to periodic patterns (Figure 7).

Then, we compare our *current* method to the HDFS that has similar results: the one which uses 20 servers. Two cases can be noted. First, our *current* method has similar cost, between -2% and +3%, and better data access time, up to -99% (*constant* and *sinus gap*). It also has better cost, up to 20%, and either similar data access time, which represents about 1/6 of the time window (*sinus* and beginning of *increasing*), or better data access time, up to 99% (*decreasing* and end of *increasing*). Second, facing with the *gaussian* pattern, the *current* method has better cost, about 50%, but has higher data access time in peak periods. The first patterns gradually evolve, and as seen before, our *current* method is able to quickly remove and add servers facing with this kind of patterns. With the *gaussian* pattern, data popularities quickly increase and quickly decrease, and as seen before, the *current* method is not adapted to this kind of pattern. The *constant* and *sinus gap* patterns provide similar results. Consequently, only the *constant* results are shown. About the *random* pattern, results are similar whatever the used solution. So, we do not show the results we obtained.

These results show that in trough periods the *current* solution is able to reduce the platform cost by removing servers without impacting data access time. In peak periods, this method is able to reduce data access time by adding servers without impacting the platform cost. However, in case of patterns with which data popularities quickly increase and then quickly decrease, the *current* method is able to reduce the platform cost but data access times are impacted. With this kind of patterns, the *mean* and *median* method are more adapted.

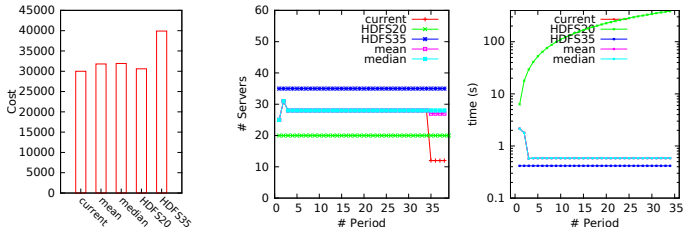

Fig. 3: Platform cost, number of used servers and average time for *constant*

4.4 Workload traces

About the workload traces, the results are similar whatever the used workload. So, we only show the *fb1* workload (Figure 8). Our solutions have better cost

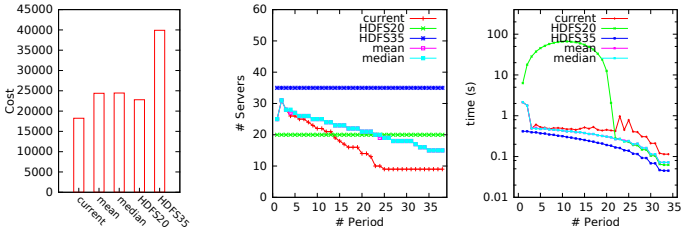

Fig. 4: Platform cost, number of used servers and average time for *decreasing*

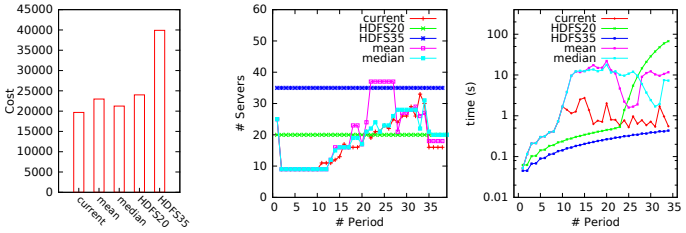

Fig. 5: Platform cost, number of used servers and average time for *increasing*

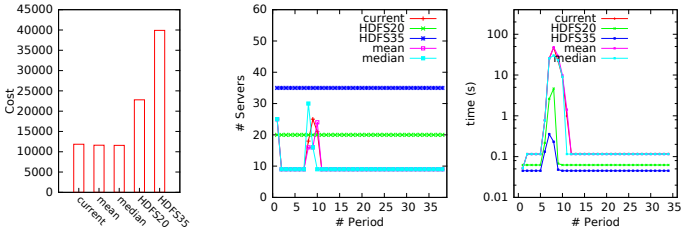

Fig. 6: Platform cost, number of used servers and average time for *gaussian*

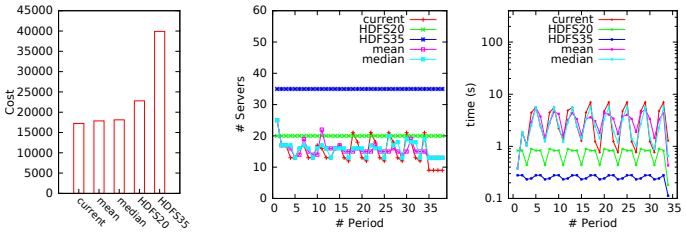

Fig. 7: Platform cost, number of used servers and average time for *sin*

than HDFS, about -50%. The *current* and the *mean* methods have data access time similar to HDFS, whereas the *median* method have higher data access time. These workloads have few requests per time window, it explains the low data access time. These results confirm that in trough periods our solutions can reduce the platform cost by removing servers without impacting data access time.

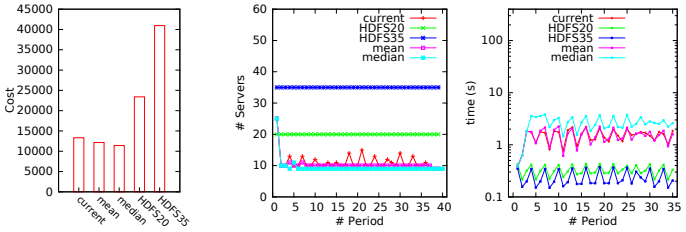

Fig. 8: Platform cost, number of used servers and average time for *fb1*

5 Conclusion and future works

Deploying a DFS on a cloud platform can provide high performance computing applications with quick data access. Extending cloud elasticity at a storage level is still a challenge: powering down or up resources in trough or peak periods, while maintaining high I/O performance is a complex task.

In this paper, we show that dynamically adapting the replication factor to the data popularities by removing or adding replicas and servers allows to obtain either better performance (up to 99%) while keeping cost similar, or obtain better cost (up to 50%) while keeping performance similar. The platform cost is reduced in trough periods whereas performance are improved in peak periods. We also show that the method used to estimate the data popularities evolution, are impacted by the kind of evolution. A method based on the current popularities to estimate the future popularities is efficient in case of gradual evolution while methods based on the mean or median popularities are more adapted with patterns where data popularities quickly increase and then decrease.

For our future works, we plan to take into account the agility of the platform (i.e. its capabilities to react quickly to the changes). In our current model, we did not take into account the time needed to start a new instance (considered negligible comparing to the time to access the data) of storage server and the possibility to stop a server without deleting all the data it stored. Indeed, the public cloud platforms generally offer the possibility to attach a persistent block storage image to a running instance. As the cost to store data on such a service is a fraction of the cost of a running instance, it should be possible to increase the agility and to reduce the data transfers when starting a new server without impacting negatively the total cost.

References

1. H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 217–228, New York, NY, USA, 2010. ACM.
2. H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.
3. Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 390–399, Washington, DC, USA, 2011. IEEE Computer Society.
4. Z. Cheng, Z. Luan, Y. Meng, Y. Xu, D. Qian, A. Roy, N. Zhang, and G. Guan. Erms: An elastic replication management system for hdfs. In *CLUSTER Workshops*, pages 32–40. IEEE, 2012.
5. Gluster. An Introduction to Gluster Architecture, 2011. `http://moo.nac.uci.edu/~hjm/fs/An_Introduction_To_Gluster_ArchitectureV7_110708.pdf`.
6. Lustre. A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, Nov 2002.
7. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symp. on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Washington, DC, USA. IEEE Computer Society.
8. C. Séguin, G. Le Mahec, and B. Depardon. Towards elasticity in distributed file systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, SCRAMBLE '15, pages 1047–1056. IEEE, 2015.
9. E. Thereska, A. Donnelly, and D. Narayanan. Sierra: a power-proportional, distributed storage system. Technical report, Microsoft Research, 2009.
10. S. A. Weil. *Ceph: reliable, scalable, and high-performance distributed storage*. PhD thesis, Santa Cruz, CA, USA, 2007.
11. L. Xu, J. Cipar, E. Krevat, A. Tumanov, N. Gupta, M. A. Kozuch, and G. R. Ganger. Springfs: Bridging agility and performance in elastic distributed storage. In *Proc. of the 12th USENIX FAST*, pages 243–255, Berkeley, CA, 2014. USENIX.