How Fast Can One Scale Down a Distributed File System?

Nathanaël Cheriere, Gabriel Antoniu

How Fast Can One Scale Down a Distributed File System?

Nathanaël Cheriere
ENS Rennes/IRISA,
Rennes, France,
nathanael.cheriere@irisa.fr

Gabriel Antoniu
Inria, Rennes Bretagne Atlantique Research Centre,
Rennes, France,
gabriel.antoniu@inria.fr

***Abstract*—For efficient Big Data processing, efficient resource utilization becomes a major concern as large-scale computing infrastructures such as supercomputers or clouds keep growing in size. Naturally, energy and cost savings can be obtained by reducing idle resources. Malleability, which is the possibility for resource managers to *dynamically* increase or reduce the resources of jobs, appears as a promising means to progress towards this goal.**

However, state-of-the-art parallel and distributed file systems have not been designed with malleability in mind. This is mainly due to the supposedly high cost of storage decommission, which is considered to involve expensive data transfers. Nevertheless, as network and storage technologies evolve, old assumptions on potential bottlenecks can be revisited.

In this study, we evaluate the viability of malleability as a design principle for a distributed file system. We specifically model the duration of the decommission operation, for which we obtain a theoretical lower bound. Then we consider HDFS as a use case and we show that our model can explain the measured decommission times. The existing decommission mechanism of HDFS is good when the network is the bottleneck, but could be accelerated by up to a factor 3 when the storage is the limiting factor. With the highlights provided by our model, we suggest improvements to speed up decommission in HDFS and we discuss open perspectives for the design of efficient malleable distributed file systems.

***Keywords*-Elastic Storage, Distributed File System, Malleable File System, Model, Decommission**

I. Introduction

For efficient Big Data processing, reducing idle resources is a major goal for the operation of large-scale infrastructures such as clouds and supercomputers. It directly leads to a reduction of the energy consumption and in a lower cost for the platform user; it optimizes productivity and results in a higher cost effectiveness for the platform operation.

Job malleability (i.e., the possibility for jobs to have their amount of resources changed by the resource manager), is a means to decrease the number of idle resources: any resource (idle ones, in particular), can be taken from a job by the resource manager and given to another one when needed.

Resource managers for malleable jobs [1], [2] and malleable frameworks [3], [4], [5] have been proposed in earlier work, however there is a missing piece: a distributed file system optimized for malleability - i.e., a file system that can efficiently be resized dynamically according to the data processing needs. Importantly, we focus on the case when the file system is deployed on the compute nodes in order to have as many as possible local accesses to data. This is in line with the current design of state-of-the-art file systems for Big Data processing, such as HDFS, as well as with the trends we witness both in cloud environments and in emerging high-end supercomputer designs.

Actually, most distributed file systems are malleable to some extent as the operations of *commission* (adding nodes) and *decommission* (removing nodes) often exist for maintenance purposes. However, they are rarely used in practice for optimizing resource usage, as they are known to have high resource requirements. Yet, this tends to change: networks get faster and so does storage, thanks to SSDs, NVRAM, or to the ever more popular in-memory file systems [6].

Fast decommission favors a quick response to new requests for resources or to sudden variations in workload. In light of this, we focus on the cost of the decommission operation. We devise a theoretical yet realistic, implementation-independent lower bound for this operation. Indeed, it is important to understand the bottlenecks of the decommission mechanism, its costs, and how fast is can be executed. Based on this, enabling efficient dynamic storage resizing can be implemented, thereby allowing malleability to become an efficient means to optimize resource usage on large-scale infrastructures used for Big Data processing.

The contribution of this paper (Section IV) consists in proposing a general model for the time needed to decommission storage nodes for distributed file systems based on data replication. It gives a realistic lower bound for the duration of the operation, thus providing a baseline for evaluating the optimality of alternative implementations of this operation. As a case study, we apply it to HDFS (Section V), a representative state-of-the-art distributed file system with a decommission mechanism already implemented. We show that the decommission mechanism of HDFS is efficient when data are stored in memory, but could be improved by up to a factor 3 when data are on secondary storage. We also show how the model can be used to predict decommission times when it is instantiated based on real measurements. With the insights provided by the model, we suggest modifications that would further improve decommission times in HDFS. The generality and usefulness of the proposed model are discussed in details in Section VI.

II. Context and motivation

A. Relevance of malleability

Malleable jobs are jobs that can increase or decrease the amount of computing resources in reaction to an external order, without needing to be restarted. Malleability is an effective means to reduce the number of idle resources on a platform. It helps users save money on cloud resource rental or make a better usage of the core hours allocated to them on supercomputers. On the other side, the platform operator has more resources available to rent, while cutting down the energy wasted by idle resources.

Some frameworks [3], [4], [5] provide support for malleability, however, few applications are malleable in practice. Many workflow execution engines such as [7] make workflows malleable: each job can be executed on any resource, and once the jobs running on a node are finished, the node can be given back to the resource manager. However, efficient support for malleable storage is missing.

Note that the *malleability*, coming from the scheduling field, differs from *horizontal scalability*. Malleability focuses on dynamically resizing operations, while scalability refers to the behavior of a system at large scales.

B. Relevance of distributed file system malleability

Having an efficient malleable distributed file system, a file system for which storage can be efficiently added and removed dynamically, could benefit both systems in the Cloud as well as HPC systems.

In the cloud, one of the key selling points of the cloud is its elasticity: one can get almost as many resources as wanted. However, this is actually still rigid in practice: most applications must be stopped and restarted to use newly allocated resources. This lack of dynamism is mainly explained by the inefficiency of the existing mechanisms for resource commission and especially decommission.

A malleable distributed file system able to add and release nodes *dynamically and efficiently, without having to be restarted*, would enable truly dynamic elasticity on the cloud through on-the-fly resource reallocation as needed. The efficiency gain would be even larger when compute and storage resources are colocated on the same nodes; efficient dynamic commission and decommission would then operate for computation and storage at the same time.

On HPC systems, similar benefits can be expected. Past efforts to bring malleability to HPC jobs [4], [8] could be completed by providing a malleable file system able to efficiently leverage the presence of the drives on the compute nodes (e.g., Theta [9], at Argonne National Laboratory features such a configuration).

C. Baseline scenario

There are resource managers that have been designed to work with malleable jobs: they add or decommission compute nodes in ranges varying from seconds to hours to match the volatility of the workload.

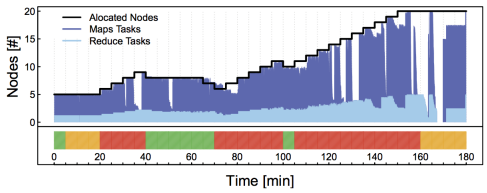

Figure 1. KOALA-F managing the resources of Hadoop in order for it to always be right-provisioned: KOALA-F adds and removes nodes according to the workload (from [1]).

In this paper, we consider the case presented for KOALA-F [1], where there are 20 nodes available and the size of each job is evaluated every 5 minutes and, in practice, up to 2 nodes are added or decommissioned. Figure 1 is an example of such a scenario. However, KOALA-F does not resize the file system as this is considered too expensive. This gives us a comparison point in order to evaluate if the performance of a malleable file system are useful in practice.

D. Focus on decommission

The malleability of a file system is composed by two operations: *commission* (i.e. adding resources) and *decommission* (i.e., resource removal).

When a storage node is added to the cluster, data need to be transferred to it before it can start serving requests. When a node is removed, the data hosted by this node must be sent to other nodes that will not leave the cluster, to satisfy data availability constraints. Once this data transfer is done, the node can safely be removed. In both cases there are data transfers and data must not be lost.

Both commission and decommission operations are important. In this paper, we focus on the decommission.

III. Related Work

Malleability has been explored in past work under various angles. Some work focused on implementing malleable applications [3], [4], [5], some on resource managers able to exploit optimization opportunities available with malleable jobs [1], [2]; many efforts were dedicated to scheduling malleable jobs [10], [11]. However, rare are the papers focused on the malleability of file systems.

Among them, the SCADS Director [12] is a resource manager that aims to ensure some service level objective by managing data: it chooses when and where to move data, when and if some nodes can be added or removed, and the number of replicas needed for each file, thanks to the interface proposed by the SCADS file system. The authors of this work propose an algorithm for node decommission but do not study its efficiency. Their algorithm includes a decision mechanism on which nodes to remove. In contrast,

we build on assumption that the file system must follow commands from an independent resource manager and we propose a model for the decommission time the can be integrated in external scheduling strategies.

Lim, Babu, and Chase [13] propose a resource manager based on HDFS. It chooses when to add or remove nodes and the parameters of the rebalancing operations. However, it simply uses HDFS without considering the time of the decommission operation. Both [12], [13] focus on ways to leverage malleability rather that on improving it. They are complementary to our work.

Another class of file system implement malleability to some extent: Rabbit [14], Sierra [15], or SpringFS [16] shutdown nodes to save energy. Because of the fault tolerance mechanism, these nodes must be ready to rejoin the cluster quickly and the data cannot be erased from them. This restricts the reallocation of these machines to other jobs. In our work we consider a more general case of node decommission, where the decommissioned nodes can be shutdown, but also allocated to new jobs without restrictions. This gives more freedom to the resource manager to reach its objective, should it concern energy saving, improving resource utilization, or maximizing gains.

IV. A model for decommission

A. Scope of our model: which type of file system?

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

With this in mind, we reduce the scope of our target model to file systems using *data replication* as their fault tolerance mechanism. This 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 it to restore the replication level to its original level. Moreover, this technique does not need much CPU power.

We do not consider *full node replication*, used in systems where sets of nodes host exactly the same data, as the recovery mechanism is fundamentally different from the one used with data replication. We also exclude from our scope systems using on erasure coding for fault tolerance, such as Pelican [17]. The mechanism however requires some CPU power to regenerate missing data. Finally, another major fault tolerance mechanism for storage systems is lineage, used in Tachyon [18] for Spark [19]. We do not consider lineage in this paper as the base principles differ greatly from the ones needed for efficient decommission. With lineage, the path used to generate the data is saved safely and, in case of crash, the missing data is regenerated. Consequently, a file system using lineage must be tightly coupled to a framework and a lot of CPU power is needed to recover data.

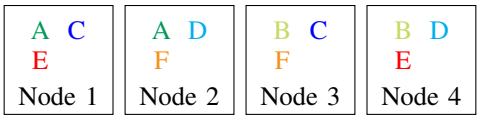

Figure 2. Example of uniform data distribution with 4 nodes and a replication factor of 2, every two nodes share 1/3 of their data (A to F).

B. Problem definition

In this section, we model a realistic lower bound for the decommission time on a given platform. Such a bound provides a potential target for optimal implementations. In particular, we are interested in the *time to decommission*: the time between the reception of the order given by the resource manager to free a set of nodes (called *in-decommission* nodes), and the time when they are effectively removed from the cluster. During that time, all nodes are present and can send and receive data but only the *non-decommissioned* nodes will remain in the cluster after this operation. Moreover, we assume that **no data may be lost** during the decommission.

Even if this work is done in the context of collocation of tasks and data, we will assume for now there is no workload on the nodes involved in the decommission operation. Will discuss this hypothesis in Section VI.

Last, there is a choice to make about fault tolerance during decommission. One option is to guarantee that the system is able to support the same number of faults during the decommission, by recreating the same number of available data replicas *before* releasing the in-decommission nodes. We choose this approach. Alternatively, faster decommission could be achieved by releasing the nodes earlier and letting the fault-tolerance mechanism recreate the missing replicas. This would temporarily weaken fault tolerance and increase the risk for data loss. Such strategy could be the focus of future work as discussion in Section VI-F.

C. Hypotheses on the system

We made several assumptions to create a model that is not too complex, yet practical:

1) Homogeneity and capacity of the cluster: We consider a cluster consisting of identical nodes: same storage capacity S, network bandwidth S_{net}, and storage characteristics (write speed S_{write} and read speed S_{read}).

Moreover, the decommission cannot be started if the data do not fit in the shrunk cluster: a cluster of 10 nodes with a capacity of 100 GB each hosting 50 GB (replicas included), can not be shrunk to 4 nodes because each node would have to host 125 GB each which is above their capacity.

To formalize this, we assume a cluster composed of N nodes, each of which has a capacity C and hosts an amount of data D, among which x are to be decommissioned. The decommission is possible only if the capacity of the shrunk cluster $(N-x)\cdot C$ is larger than the hosted data $(N\cdot D)$.

2) Uniform data distribution: We consider a perfect load balancing of the data, each node stores exactly D data. Moreover, with a cluster of N nodes and a replication factor of r, any two nodes share exactly $\frac{r-1}{N-1}$ of their data. Figure 2 is an example of such data distribution.

Both hypotheses above reflect the goal of the load balancing policies implemented in existing file systems, such as HDFS [20] or RAMCloud[6].

3) Hypotheses on the network : We consider that the network is full-duplex: data can be sent and received at the maximum speed at the same time. We assume that the network is ideal. Each device is always used at its maximum bandwidth and there are no interferences.

As no network is ideal, the bisection bandwidth is a better metric to use for the network, and the interferences can be taken into account by using a bandwidth measured in the presence of interferences (or provided by some model that is interference-aware). Doing so would provide a better estimation of the time needed for the decommission, but the model provided would not be a lower-bound anymore.

4) Hypotheses on the storage devices: First, we assume the writing speed is not faster than the reading speed. Second, the hardware must share its I/O time between reads and writes and thus can not sustain reads and writes at maximum speed. Both hold for state-of-the-art hardware.

5) Conservative replication factor: We assume that the replication factor r is greater or equal to 2. We ensure that the replication factor is the same before and after the decommission and is never below its original level.

6) Avoiding unneeded reads during decommission: By using buffering in RAM, data can be read from the persistent storage and sent to multiple destination, with only one read operation on the storage. This reduces the load on the storage device and save time as reading from memory is faster. It is already used by HDFS to optimize data writing.

The assumption that only one read is needed holds for most persistent storage systems. However it does not hold for in-memory storage as the buffering would be as fast as the storage itself. In that case, there are as many reads as there are write operations.

D. Modeling the decommission time

To establish a model of the decommission time, it is important to notice that data writing is the bottleneck of this operation, for three reasons.

First, each of the non-decommissioned nodes and each of the in-decommission nodes share $\frac{r-1}{N-1}$ of their data (Hyp. IV-C2), thus any non-decommissioned node can read and send data as the same rate as in-decommission nodes. Second, only the non-decommissioned nodes can receive and write some data on their storage: as in-decommission nodes will leave the cluster, it is pointless to have them store more data. Last, storage devices have a lower writing speed than their reading speed (Hyp. IV-C4).

With this, the general model for the decommission time is quite simple: it equals the amount of data to write ($Data\_to\_write$) divided by the writing speed of the whole cluster $S_{write\_cluster}$ (eq. 1), to which we add an initialization time t_0 which should be negligible with an optimized implementation.

$$t_{decom} = \frac{Data\_to\_write}{S_{write\_cluster}} + t_0 \tag{1}$$

E. Data to write

As the replication factor must be left unchanged after the decommission (Hyp. IV-C5), all data present on the in-decommission nodes must be written on non-decommissioned nodes.

Thus the data to write is exactly:

$$Data\_to\_write = x \cdot D. \tag{2}$$

F. Writing speeds

Determining the writing speed of the cluster is more complex. There are two main cases, depending on the relative speed of the network with respect to that of the storage. In one case, a slow network is the bottleneck and the nodes do not receive enough data to saturate the storage's bandwidth. In the second case, storage is slow and becomes a bottleneck (i.e., storage can not write at the speed at which the data is received from the network).

1) First case: the network is the bottleneck: We assume the network is full duplex, and without interferences (Hyp. IV-C3). In this case, the non-decommissioned nodes can receive data at the network speed S_{net}, even if they send data at the same time. Each one of the $N-x$ non-decommissioned node can receive and write data at the network speed S_{net}. Thus the writing speed is $S_{write\_cluster} = S_{net}(N-x)$ and the overall decommission time is

$$t_{decom} = \frac{x \cdot D}{S_{net}(N-x)} + t_0. \tag{3}$$

2) Second case: the storage is 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) can not read and write at the same time (Hyp. IV-C4).However, what is read can be written more than once (Hyp. IV-C6), thus we denote as $R(N,x)$ the ratio data written divided by data read.

$$R(N,x) = \begin{cases} 1 & \text{for RAM-based storage,} \\ \frac{\sum_{i=1}^{r} i * P(X=i)}{\sum_{i=1}^{r} P(X=i)} & \text{if other cases.} \end{cases} \tag{4}$$

where

$$P(X=k) = \begin{cases} 0 & \text{if } k > r, \\ \frac{\binom{r}{k}\binom{N-r}{x-k}}{\binom{N}{x}} & \text{for } k \leq r. \end{cases} \tag{5}$$

The ratio $R(N,x)$ (eq. 4) is expressed with the probability $P(X=k)$ of a chunk of data to have k replicas on the

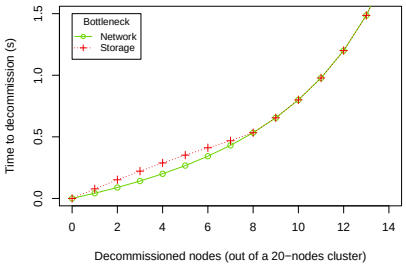

Figure 3. Time needed to transfer 1 GB from each of the in-decommission nodes on two different settings: either the network is the bottleneck and have a bandwidth of 10 Gb/s (1.25 GB/s), either the storage is the bottleneck and has read and write speeds of 1.25 GB/s. The cluster is composed of 20 nodes, and $t_0 = 0$.

in-decommission nodes (eq. 5), which is a classical urn problem. In that case, the data is read once, but is written k times to ensure the replication factor.

There are two cases for the writing speed of the cluster due to the fact that in-decommission nodes can only read (they will leave the cluster at the end of the decommission): either the in-decommission nodes can read enough data to saturate the writing on the non-decommissioned nodes, either they can not.

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

$$S_{write\_cluster} = S_{net}(N-x). \tag{6}$$

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

$$S_{write\_cluster} = \frac{N \cdot R(N,x) \cdot S_{read} \cdot S_{write}}{S_{write} + R(N,x) \cdot S_{read}}. \tag{7}$$

The in-decommission nodes are able to saturate the non-decommissioned nodes when more than $T(N,x)$ nodes are decommissioned at once. The threshold $T(N,x)$ can be expressed as

$$T(N,x) = \frac{NS_{write}}{R(N,x) \cdot S_{read} + S_{write}}. \tag{8}$$

With equations 1, 2, 6, and 7, the time to decommission in the case of storage as a bottleneck is modeled with

$$t_{decom} = \begin{cases} \frac{xD}{S_{write}(N-x)} + t_0 & \text{if } x \geq T(N,x), \\ \frac{x \cdot D \cdot (S_{write} + R(N,x) S_{read})}{N \cdot R(N,x) \cdot S_{read} \cdot S_{write}} + t_0 & \text{in other cases.} \end{cases} \tag{9}$$

G. Observations

In the case of an implementation with a negligible t_0, three interesting observations can be made with the model.

1) Impact of the data hosted per node: The decommission time is proportional to the amount of data hosted per node. With this, the decommission scales linearly with the amount of data hosted for a given platform.

Figure 3 summarizes the minimal decommission times that can be reached for both kinds of bottlenecks, on an artificial platform that exposes the differences in behavior between both bottleneck. The decommission times expected for existing hardware can be found in section VI-C.

2) Impact of the proportion of decommissioned nodes: In case of a network bottleneck, the decommission time only depends on the proportion of nodes in-decommission. In this situation, decommissioning 20 nodes in a cluster of 100 or 4 in a cluster of 20 would take the same time if each node hosts the same amount of data.

For instance, we note that in the case of a network bottleneck with a speed of 10 Gb/s, with 20% of the nodes decommissioned at once, it only takes 0.2 s to transfer 1 GB from each in-decommission node. This means that if each node hosts 100 GB of data, the decommission could take as little as 20 s if the implementation is properly optimized.

3) Decommission one by one or in batch: Last, in the case where the network is the bottleneck, there is no difference if the decommission is done in one batch or in successive steps. Indeed, equation 10 giving the time needed to decommission in k steps, is the same as equation 3 when $t_0 = 0$.

$$t_{decom} = \frac{x \cdot D}{S_{net}(N-x)} + k \cdot t_0 \tag{10}$$

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 amount of nodes that can write, which is higher in the first step than in the last steps. Note that this results can not be found in the case where the bottleneck is at storage level: the storage compensate for the nodes that can not write and have a constant speed.

V. The case of HDFS

In this section, we use the previous model to study the decommission mechanism of HDFS in two cases: HDFS with its storage in RAM (bottleneck at the network level); HDFS with storage on drives (bottleneck at storage level).

In both cases, we compare experimental measurements to the model and propose improvements for the transfer scheduler of HDFS that would decrease decommission time.

A. Experimental setup

1) Testbed: The experiments presented in this section have been performed on the Grid'5000 [21] experimental testbed. The *paravance* cluster from Rennes was used. Each node has 16 cores, 128 GB of RAM, a 10 Gbps network interface, and two hard drives. The file system's cache has

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

Table 4. Parameters used for the experiments.

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.

2) HDFS: We deployed HDFS and Hadoop 2.7.3. One node was acting as both DataNode (slave of HDFS) and NameNode (master of HDFS) while the other were only used as DataNodes. One drive is reserved for HDFS to store its data. Most of the configuration is left to its default values, including the replication factor left unchanged to 3.

However, some parameters were adjusted for the experiments: HDFS checks the decommission status every second (instead of 30 s by default) with the parameter *dfs.namenode.decommission.interval*. This gives us the decommission times with a precision of 1 s. Besides, as HDFS schedules data transfers every 3 s, we used the parameters presented in Table 4 to schedule enough transfers to maximise the bandwidth utilization while avoiding unbalanced work distribution. This has been confirmed experimentally.

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

3) HDFS in memory: To experiment RAM-based storage with HDFS, we used the same setup as in paper introducing Tachyon[18]: a tmpfs partition of 96 GB is mounted and HDFS uses 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 a lot higher (6 GB/s reading and 3 GB/s writing) moving the bottleneck from the drives to the network.

4) Experiment protocol: To measure the decommission time of HDFS, a random subset of nodes is selected among the DataNodes except the one hosting the NameNode, and the command to decommission those nodes is given to the NameNode. The recorded time is the time elapsed between the moment the NameNode receives the command and the moment when 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.

B. HDFS: when the bottleneck is at the network level

To create a setup with a bottleneck at the network level, we configure HDFS with RAM-based storage (we could measure writing at 3 GB/s in memory, including an overhead induced by the file system, while transfers on the network are done at 1.1 GB/s (i.e. 8.8 Gb/s)).

1) How close is HDFS to the model?: Figure 5 displays the decommission times observed for multiple amounts of data hosted per node and various numbers of nodes to decommission. In addition, the figure also shows the theoretical minimum decommission time for this platform computed with the model presented in section IV. 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 55s. If we consider the scenario of KOALA-F detailed in Section II-C, the decommission of 1 or 2 nodes would take less than 13 s every 5 min, 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 also observe that the measured values are close to the theoretical minimum provided by the model: the decommission mechanism of HDFS is close to the lower bound in this case, but can be improved.

2) Fitting the model to HDFS: In Figure 6, 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 $t_0 = 4.4$ s and $S_{net} = 0.98$ GB/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 to receive data on the non-decommissioned nodes that is the main bottleneck, and that there is a flat cost of 4.6 s.

The network bandwidth determined by the regression matches the observations as we can see on Figure 7 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 s (on average 1.5 s delay), or the verification of the status of the decommission every second (on average 0.5 s delay). It also includes the imbalance in the scheduling that appears at the end of the transfers: due to the scheduler, some nodes have the maximum amount of transfers to do while other have none.

Note that the model explains the decommission times of HDFS well even if some of the hypotheses needed by the model are not fulfilled by HDFS: the data are not evenly distributed (see Figure 8), and the transfer speeds are not constant (see Figure 7, especially the reception speed that should not change). That explains why the value of t_0 is higher than expected: it compensates for the lower transfer speeds for small amounts of data transferred.

3) The practical cost of HDFS: Figure 7 shows the network bandwidth during the decommission. As expected, the in-decommission nodes do not receive any data as they are going to leave the cluster after the operation. The nodes

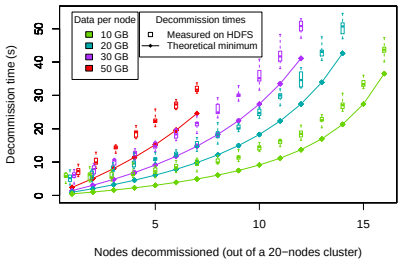

Figure 5. Decommission time measured on the platform presented in sec. V-B. The minimum theoretical time obtained with the model on this platform is added.

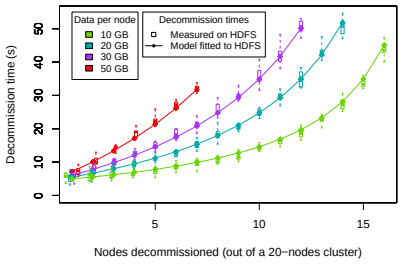

Figure 6. Model fitted to HDFS on the platform presented in sec. V-B. The model fits the data with a coefficient of determination r^2 of 0.98.

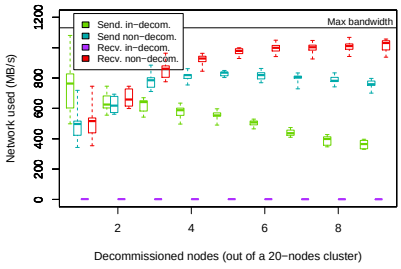

Figure 7. Average bandwidth measured for in-decommission and non-decommissioned nodes for both reception and emission on the platform presented in sec. V-B. Each node hosts 40 GB of data, and the maximum bandwidth was measured with a benchmark.

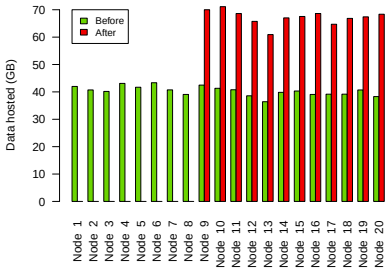

Figure 8. Amount of data before and after the decommission on the nodes of the cluster on the platform presented in sec. V-B. Data was generated for 40 GB per node, and 8 nodes were decommissioned.

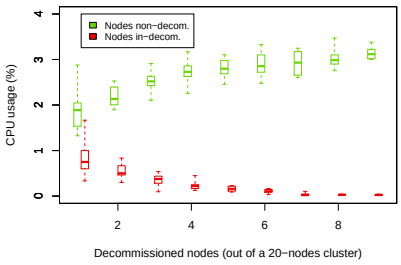

Figure 9. Average CPU usage measured for in-decommission and non-decommissioned nodes on the platform presented in sec. V-B. Each node hosts 40 GB of data.

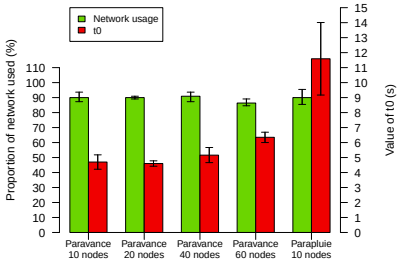

Figure 10. Value of t_0 and proportion of network utilization obtained by regression for multiple setups with a network bottleneck.

send data at a lower bandwidth as the bottleneck is the reception of the in-decommission nodes. Figures 9, shows that the CPU usage is low during the decommission: it is used only for the metadata operations. As shown in Figure 8, the storage on non-decommissioned nodes does increase.

4) How to improve decommission time in HDFS: Although the performance is already good, it can still be improved. Parameter tuning by reducing the heartbeat rate, increasing the transfer scheduling rate, 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 very 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 (this is beyond the scope of this paper).

5) Different fits for different platforms: Lastly, in Figure 10, we present 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 Gbps network). We observe that the network utilization stays roughly at 90 % of the maximum bandwidth thanks to a good configuration of HDFS. On the other hand, t_0 changes, not due to the larger amount of work to schedule transfers, but mainly due to the impact of scheduling mistakes made by the scheduler of data transfers of HDFS.

C. HDFS: when the bottleneck is at storage level

To create a setup where the bottleneck is at storage level, we configure HDFS to store data on drive (read spead: 180 MB/s, write speed: 160 MB/s), a lot slower than the network (1.1 GB/s).

1) How close is HDFS to the model?: Figure 11 shows the decommission times observed and the minimal theoretical time to do so. As we can observe, even if the measures follow the same trends as the model, HDFS is about 3 times slower than what could be achieved on the platform.

In Figure 11 we present the measurements with the same configuration (number of nodes and data hosts per node) as the ones presented in Section V-B for better comparison, even if the technical constraint would allow larger experiments. In particular, when comparing to Figure 5, we observe that the decommission times are up to 20 times slower when using the drive. However, the drive should only 13 times slower than the network in the worst case (reading

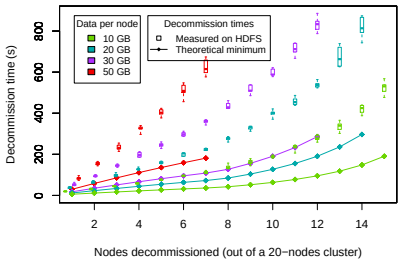

Figure 11. Decommission time measured on the platform presented in sec. V-C. The minimum theoretical time obtained with the model is added.

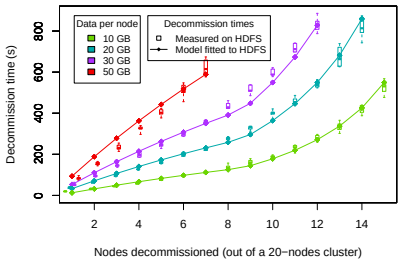

Figure 12. Model fitted to HDFS on the platform presented in sec. V-C. The model is enough to explain the performance of HDFS and has a coefficient of determination r^2 of 0.983.

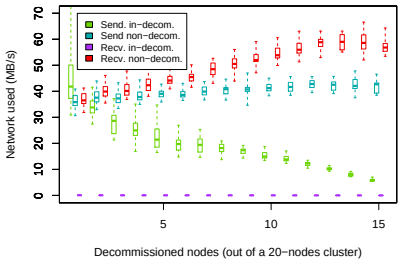

Figure 13. Average network usage measured for in-decommission and non-decommissioned nodes on the platform presented in sec. V-C. Each node hosts 40 GB of data.

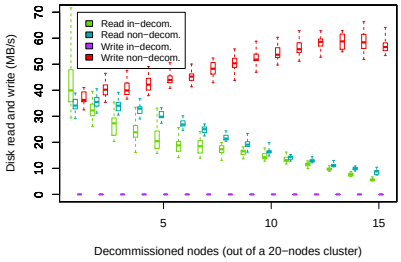

Figure 14. Average disk usage measured for in-decommission and non-decommissioned nodes on the platform presented in sec. V-C. Each node hosts 40 GB of data.

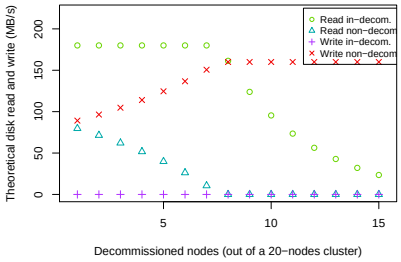

Figure 15. Theoretical disk utilization for in-decommission and non-decommissioned nodes obtained with the model on the platform presented in sec. V-C.

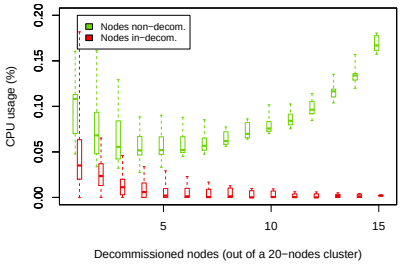

Figure 16. Average CPU usage measured for in-decommission and non-decommissioned nodes on the platform presented in sec. V-C. Each node hosts 40 GB of data.

and writing at the same time). This is a confirmation that the decommission in this configuration is a lot less efficient than the one presented in section V-B.

2) Fitting the model to HDFS: As the pattern of the measures follows the model, we use regression to fit the model for the results. The decommission of HDFS matches the realistic lower bounds obtained on a platform with reading speed of 50.7 MB/s, writing speed of 55.1 MB/s, with an initialization time t_0 of -3.55 s, with a coefficient of determination of 0.983 as shown in Figure 12. The negative initialization time is due to the fact that the transfer scheduler of HDFS balances reads instead of writes: this does not match the scheduling strategy expected by the model.

3) How to improve decommission in HDFS: HDFS schedules data transfers by balancing the reads and send operations, but the bottleneck are the receive and write operations. It can be observed on Figure 14: all nodes read data at approximately the same speed. This results in high competition for the drive accesses on non-decommissioned nodes that must read and write data, while the disk of in-decommission nodes are underloaded as they do not write.

A scheduling strategy leveraging the model is quite simple: the scheduler should balance the writing operations, prioritize them, then maximize reading from in-decommission nodes. This would lead to read and write patterns like those presented in Figure 15. If non-decommissioned nodes can write all that is read by in-decommission nodes, then they also read to accelerate the decommission. If they can not, the in-decommission nodes have their reading speed reduced while non-decommissioned nodes simply stop reading.

4) Cost of the decommission: Figure 13 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 very low (See Fig. 16): the CPU is only used for metadata operations, that are rare due to the reading and writing speeds of the storage.

VI. Discussion

A. How dependent is this model on HDFS?

The model is generic and does not rely on HDFS. Thus, the hints given to improve the transfer scheduler of HDFS can also be used to improve the decommission time in any distributed file system using data replication.

B. Usefulness of the model

The model helps particularly in understanding the bottlenecks of the decommission operation and thus gives hints to optimize it in distributed file systems such as HDFS.

But a more interesting utilization we foresee would consist in using the estimated time for the decommission provided by the model to efficiently schedule dynamically resources

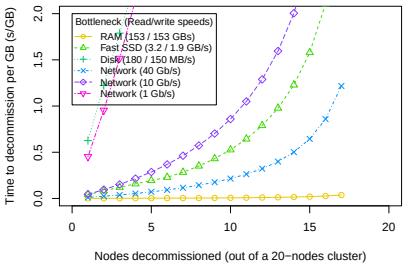

Figure 17. Minimal decommission time (per GB) for different existing technologies on a 20 nodes cluster.

for malleable applications. With this model of the decommission time, resource schedulers can more easily anticipate the decommission of nodes, or even estimate if it is interesting to add nodes that will soon be taken back.

Last, the model provides a realistic lower bound of the decommission time, thus it can be used as a baseline to evaluate the implementation of decommission mechanisms in distributed file systems in general.

C. Predicting decommission times for various technologies

As the model is generic, it is possible to use it to predict the decommission times that could be reached when other storage technologies, existing or emerging, are used. As an example, Figure 17 illustrates expected decommission times for various settings: storage bottlneck with RAM (from the Cray XC series [22]), drive (see Section V-A), and one of the fastest SSDs [23] and network bottleneck with different bandwidths. From this figure it is obvious that the decommission times decrease with newer technologies, strengthening the idea that malleable file systems can currently be useful as the cost of the malleability is decreasing.

D. The case for improving file system malleability

A malleable distributed file system that stores its data on HDDs can be considered too slow to be useful, especially with the network bandwidth available that negates the need for local storage. However, if we consider faster storage (that would make the network the bottleneck) such as some of the fast SSDs, or even the RAM, the local storage improves the performance of IO-intensive applications. On top of that, having malleability allows applications to easily dig into unused yet available resources, or to avoid having idle resources. Thus a malleable distributed file system on fast storage in such a setting would make an application able to exploit all available resources to their maximum.

The Cybershake workflow [24] is a good example that would greatly benefit from a malleable execution engine and file system. This workflow is basically malleable as each of its 815,000 short jobs can be launched on any resources. Moreover, it only writes 920 GB of data, but reads 217 TB of it [25]. This amount of data could quite easily be stored on a few nodes with large amounts of RAM, and the read operations would be greatly accelerated using local storage.

To conclude, malleability is getting increasingly interesting as a means to better use resources on shared platforms. Thus, even if the malleability of distributed file systems is currently limited, we confirm in this paper that its cost is low (except for HDDs), and provide a model that can be integrated in scheduling strategies for malleable jobs.

E. Would the model work if a workload is present?

The model presented in section IV assumes that all resources are available for the decommission but this is sometimes not the case. Some implementations might limit the bandwidth used for the decommission, or give a lower priority to the decommission operation to favor the execution of applications. In both cases, this choice is made when implementing the distributed file system. Thus, it is possible to add trade-offs such as limiting the bandwidth available for the decommission, and thus to reduce the S_{net} accordingly.

F. Relaxing hypotheses

Our proposed model is based on many hypotheses, many of which are common among file systems, but one in particular could be relaxed. If the user prioritizes the decommission time over fault tolerance, it is no longer necessary to always maintain the replication factor (hypothesis IV-C5).

The replication factor of the files present on the in-decommission nodes can be lowered to free the nodes faster, and then brought back to its initial level once the nodes have left. This is faster as only the data that is exclusively on in-decommission 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. We will study this in future work.

G. The uniformity hypothesis

The model is based on the assumption that the data is uniformly distributed among the nodes. This is almost impossible to do in practice. Systems like HDFS place data using randomness in order to have a distribution of the data close to uniformity. We have seen in section V that HDFS has performances close to the model (Fig. 5), even if the data distribution (Fig. 8) is not uniform among the nodes.

If one does not want to use this hypothesis, the amount of data D can be set to the minimum amount of data hosted by a single node, which guarantees that the model provides a lower bound.

H. How to determine where is the bottleneck?

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 storage level if the read/write speeds can not keep up with the speed at which

data are sent and received through the network: $\frac{S_{read} \cdot S_{write}}{S_{read}+S_{write}} < S_{net}$. By combining the two possibilities, it appears that there is also a possibility of having bottlenecks both at the storage and the network level, at the same time. We leave this less intuitive situation outside the scope of this study.

VII. Conclusion

Efficient decommission of nodes is essential in order to enable the design of malleable distributed file systems. To the best of our knowledge, this is the first study that aims to model this operation present in distributed file systems, regardless of its implementation. Using this generic model, we evaluate the decommission of HDFS, highlight potential improvements thanks to the better understanding of the operation, and show that the only resources needed to decommission nodes are the network and drive bandwidth. We also exploit the model to predict the decommission times on emerging technologies. Finally, we discuss how this model can serve as a base for a malleable distributed file system.

The modeling of the commission operation is similar to the one presented but raises other challenges, it will be subject to a future work.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] A. Kuzmanovska, R. H. Mak, and D. Epema, "KOALA-F: A Resource Manager for Scheduling Frameworks in Clusters," *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 592–595, 2016.

[2] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv *et al.*, "Morpheus: Towards Automated SLOs for Enterprise Clusters," *USENIX Symposium on Operating Systems Design and Implementation*, pp. 117–134, 2016.

[3] S. S. Vadhiyar and J. J. Dongarra, "SRS: A Framework for Developing Malleable and Migratable Parallel Applications For Distributed Systems," *Parallel Processing Letters*, vol. 13, no. 2, pp. 291–312, 2003.

[4] L. V. Kale, S. Kumar, and J. Desouza, "A Malleable-Job System for Timeshared Parallel Machines," *IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[5] J. Buisson, F. André, and J. Pazat, "A Framework for Dynamic Adaptation of Parallel Components," *International Conference Parallel Computing*, pp. 1–8, 2005.

[6] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières *et al.*, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2010.

[7] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa *et al.*, "Parallel Scripting for Applications at the PetaScale and Beyond," *Computer*, vol. 10, pp. 50–60, 2009.

[8] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale, "A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications," *International Parallel and Distributed Processing Symposium*, pp. 429–438, 2015.

[9] "Theta," www.alcf.anl.gov/theta, Accessed 19/06/17.

[10] K. Jansen and L. Porkolab, "Linear-Time Approximation Schemes for Scheduling Malleable Parallel Tasks," *Algorithmica*, vol. 32, pp. 507–520, 2002.

[11] G. Mounie, C. Rapine, and D. Trystram, "Efficient Approximation Algorithms for Scheduling Malleable Tasks," *ACM symposium on Parallel algorithms and architectures*, vol. 3, pp. 23–32, 1999.

[12] B. Trushkowsky, P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements," *USENIX conference on File and Storage Technologies*, pp. 163–176, 2011.

[13] H. C. Lim, S. Babu, and J. S. Chase, "Automated Control for Elastic Storage," *International Conference on Autonomic Computing*, pp. 1–10, 2010.

[14] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan, "Robust and Flexible Power-Proportional Storage," *ACM Symposium on Cloud Computing*, pp. 217–228, 2010.

[15] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: Practical Power-Proportionality for Data center Storage," *Conference on Computer Systems*, p. 169, 2011.

[16] L. Xu, J. Cipar, E. Krevat, A. Tumanov, N. Gupta, C. Mellon *et al.*, "SpringFS : Bridging Agility and Performance in Elastic Distributed Storage This paper is included in the Proceedings of the," *USENIX conference on File and Storage Technologies*, pp. 243–255, 2014.

[17] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper *et al.*, "Pelican : A Building Block for Exascale Cold Data Storage," in *Operating Systems Design and Implementation*, 2014, pp. 351–365.

[18] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Reliable , Memory Speed Storage for Cluster Computing Frameworks," in *ACM Symposium on Cloud Computing*, 2014, pp. 1 – 15.

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," *HotCloud*, vol. 10, no. 10, p. 95, 2010.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *IEEE Symposium on Mass Storage Systems and Technologies*, pp. 1–10, 2010.

[21] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine *et al.*, "Adding Virtualization Capabilities to the Grid'5000 Testbed," in *Cloud Computing and Services Science*, 2013, vol. 367, pp. 3–20.

[22] "Cray XC Series," www.cray.com/sites/default/files/Cray-XC-Series-Brochure.pdf, Accessed 19/06/17.

[23] "NVMe SSD 960 PRO/EVO," www.samsung.com/semiconductor/minisite/ssd/downloads/document/NVMe_SSD_960_-PRO_EVO_Brochure_Rev_1_1.pdf, Accessed 19/06/17.

[24] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta *et al.*, "SCEC CyberShake Workflow - Automating Probabilistic Seismic Hazard Analysis Calculations," in *Workflows for e-Science*. Springer, 2007, pp. 143–163.

[25] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and Profiling Scientific Workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.