Workload Evaluation Tool for Metadata Distribution Method

Éloïse Billa[1,2(✉)], Philippe Deniel[1], and Soraya Zertal[2]

[1] CEA/DAM, Arpajon, France
[2] Li-PaRAD, UVSQ, Paris Saclay, France

Abstract. In the *High Performance Computing* field (HPC), metadata server cluster is a critical aspect of a storage system performance and with object storage growth, systems must now be able to distribute metadata across servers thanks to distributed metadata servers. Storage systems reach better performances if the workload remains balanced over time. Indeed, an unbalanced distribution can lead to frequent requests to a subset of servers while other servers are completely idle. To avoid this issue, different metadata distribution methods exist and each one has its best use cases. Moreover, each system has different usages and different workloads, which means that one distribution method could fit to a specific kind of storage system and not to another one. To this end, we propose a tool to evaluate metadata distribution methods with different workloads. In this paper, we describe this tool and we use it to compare state-of-the-art methods and one method we developed. We also show how outputs generated by our tool enable us to deduce distribution weakness and chose the most adapted method.

Keywords: Metadata distribution · Load-balancing · Evaluation tool

1 Introduction

To reach the highest accuracy as possible, scientific applications and numerical simulations continuously look for more resources. It means huge data storage capacities since a high level of accuracy is mainly achieved by resolving more and more complex models. Dealing with larger datasets induce to get better computation units in order to run simulation codes in a human-admissible time. The High Performance Computing (HPC) field is permanently evolving to provide larger supercomputers and having a better computation power. As data sets rapidly grow in volume, the need for an efficient storage system becomes critical. Indeed, the well-known File System paradigm is not expected to handle the number of access requests related to massive data incoming in a near future [1]. This scalability limitation is due to the hierarchical organization of files in POSIX norms [2]. Hence, clusters need storage systems based on structures that are independent of POSIX norms and provide a higher level of parallelism. Object storage systems [3] have emerged in response to these issues. They distinguish

H. Song and D. Jiang (Eds.): SIMUtools 2020, LNICST 369, pp. 796–810, 2021.
https://doi.org/10.1007/978-3-030-72792-5_63

data flows from metadata flows, enabling parallel access and higher throughput. To reach another step in the requests parallelization, they also detach themselves from the hierarchical tree constraints by using a flat namespace to store data and metadata.

This kind of storage opens new opportunities to scalability challenges such as a highly distributed metadata service. A distributed and dedicated metadata service represents an important factor in the performance of a storage system [4] and the ability to manage distributed metadata is an arduous point to solve [5]. Managing several metadata servers (MDS) induces to know which is the server to ask for any metadata. For this purpose, the metadata namespace is split and distributed across all metadata servers using a namespace distribution method.

In this paper, we focus on the evaluation of those distribution methods. Each metadata distribution method has weaknesses and strengths and is adapted to different request flows. In the same manner, each storage system has its own requests flow depending on the system usage. A distribution method could be adapted to a particular storage system while being unable to cope with another system's workload. Finding the metadata distribution method which fits with a specific storage system workflow is not easy: It is necessary to evaluate the method with the system workload before choosing it.

Main Contribution: In order to choose a method adapted to a particular workflow, we provide a tool to evaluate metadata distribution methods and to test their robustness with representative HPC application traces. This tool provides a unified context of evaluation for all methods and allows to analyse and compare their behavior on different workflows. We propose also the implementation of three distribution methods to show the comparison process.

The structure of the paper is the following: In Sect. 2, we discuss current state-of-the-art and related works. Section 3 introduces the tool we developed and Sect. 4 presents the environment we have used to evaluate distribution methods. Using this ecosystem, Sect. 5 describes the evaluated methods. Section 6 compares them and presents the outputs our tool generates. Finally, we conclude in Sect. 7.

2 Related Work

Distributing metadata across all metadata servers in an efficient way is mandatory to make the global system scalable and efficient [4]. Indeed, distributing the metadata namespace to the MDS set in an unsuitable manner generates hot spots. In practice, these hot spots will be massively accessed and servers in charge of them will be overloaded, causing bottlenecks and slowing down the whole system. Well known metadata distribution methods are proposed such as the Dynamic Subtree Partitionning [6], hashing techniques used in Dynamo [7] or DROP [8] or table-based methods as the Dynamic Hashing method [9] or in Someta system [10].

Each new proposed method induces performance evaluation and usually comparisons with previous distribution methods. Benchmarking tools such as

MDtest [11] or *PostMark* [12] enables to evaluate performance for File Systems operations. They are used if the method is POSIX compliant as for Xing et al. in their adaptive method [13] and Yang et al. in PPMS [14], or for Tang et al. [10] which use *MDtest* to compare with Lustre File System [15]. However, lots of distribution methods are detached from POSIX norms and hence these tools are not suitable. Some tools exist for object storage systems, such as *COSBENCH* [16] and enable to evaluate overall system performance, but they do not specialize in metadata evaluation and could not provide a distribution specific analysis with for example a per-server workload.

The lack of specific comparison tools bring authors to implement their own method in their system and evaluate performance with a process they themselves design, as done in Landstore [17]. Each evaluated method has so its own metrics or measurements and does not follow one standard evaluation process. Then, to compare these methods to previous ones, most of the state-of-the-art authors choose to reimplement algorithms [6,18,19] to integrate them in their own evaluation process. Indeed, there are different parameters which affect evaluations such as workflow or number of servers, and to have a fair performance comparison, they have to reimplement state-of-the-art methods in the same system as the one which evaluates their own method. That means for each new method, previous ones had to be reimplemented to fit with their own evaluation processes, which is a real waste of time. Moreover, some reimplemented methods fit not well with the system in which they are integrated and adaptations are needed. It can generate performance variations with the original algorithm and so alters the comparison.

3 Framework

In order to fairly compare the different distribution methods, we decided to develop a complete framework that allows us to instrument and compare methods on many various ways. This framework allows us to run and evaluate distribution methods for various types of traces in the same context.

3.1 Architecture Overview

The distribution method choice depends on the storage system: each system has its particular workflow with different ratios of metadata creations/old metadata accesses or period of burst / period of low workload, or even different object naming policies. Similarly, every method has its main strengths and troublesome use cases. In order to select the namespace distribution method the most suitable for a particular workflow, it is mandatory to define an evaluation process based on practical experiments. We propose a framework to evaluate distribution policies on a same workload and so compare them on equal terms.

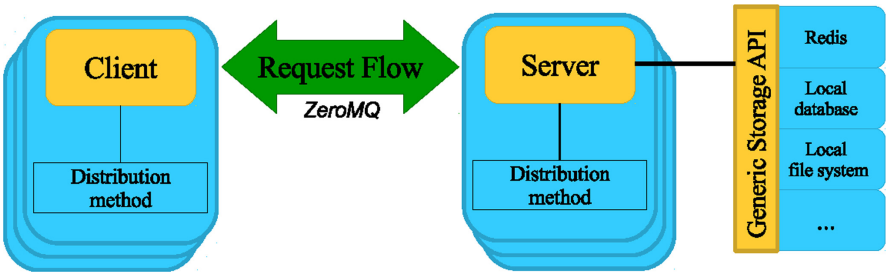


Fig. 1. Overview of framework architechture

Our framework allows to simulate a metadata service with a user-defined number of servers. It is a C-implemented project with 7000 lines of code in 80 files, which will be open-source [20]. An overview of the framework structure is shown on Fig. 1. There is a client-server interaction which is achieved through a *TCP* connection using the ZeroMQ library [21]. This interaction enables clients to establish requests to servers and servers to reply. The main specificity of our client-server interaction is the genericity: Each client and server have generic calls to the distribution method, allowing to switch methods without a whole reimplementation. The behavior of the distribution method on the client side is independent of the one on the server side. This allows us to mix the client behavior from one distribution policy and the server behavior of another one. The generic storage API we can see in Fig. 1 is another generic part of the proposed system which is associated with each server. This storage backend enables to easily adapt the framework to different storage systems.

3.2 Specific Features

We will now describe in more details some specific characteristics of our tool. To better understand these features, we should go deeper into the tool explanation. A more detailed view of the architecture is given in Fig. 2 with a class diagram.

Switching Namespace Distribution: Our tool allows to switch transparently the distribution method and to measure the impact of different policies on the server workload and the overall performances. The flexibility in the choice of the distribution method enables to highlight the hot spots in systems and to compare how different methods handle them. It is also possible to create new policies and to easily integrate them into our framework for evaluation.

As we can see in Fig. 2, there are two classes dedicated to the distribution method: the *Client Distribution* and the *Server Distribution*. The *Server Distribution* has an extra function named `end_epoch` which defines the behavior of the distribution when time is passing (this will be more detailled below). Both distributions have functions named `pre_send` and `post_receive`, which allows to perform distribution specific operations before sending the request (or the

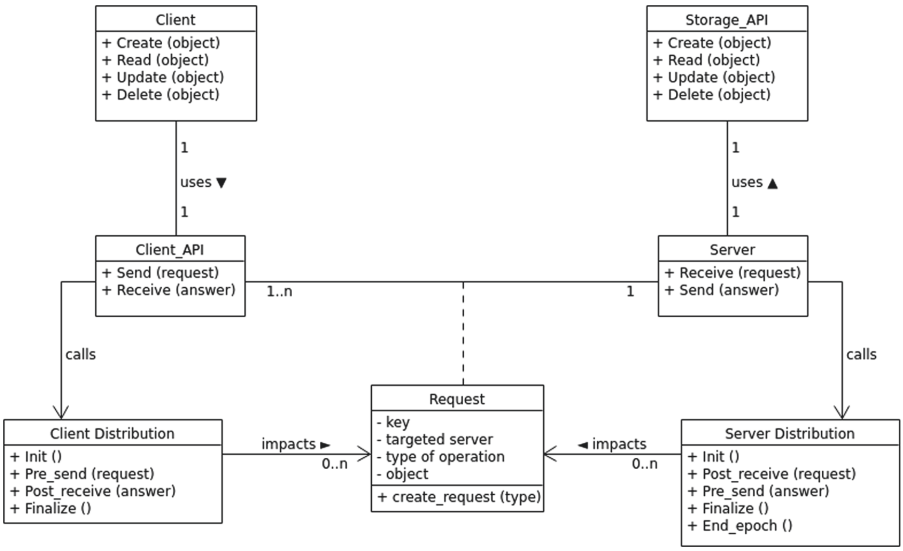


Fig. 2. Class diagram for the framework architechture

answer) and after receiving the answer (or the request). The complete requesting process is the following:

1. *Client* class (which represents users which need to request metadata) gives information about the request to the *Client_API*. As all our requests follow the CRUD semantic [22], *Client* only needs to provide the requested metadata key and the operation to execute.
2. The *Client_API* creates an object of type *Request* and ask to the *Client Distribution* class for filling in the targeted server field, i.e. to compute the distribution method algorithm to find the server to request.
3. Before sending the *Request*, the *Client Distribution* can add information that is specific to the distribution method using the `pre_send` function.
4. Once the *Server* class receives the *Request*, it calls the `post_receive` function in the *Server Distribution* class to check distribution specific information. For example, it could check if the client cache is up-to-date and if it asks for the real server in charge of the metadata.
5. The *Server* executes the operation using the *Storage_API*.
6. Before sending the answer to the client, the *Server Distribution* is allowed to add other distribution specific information. For example, it could inform the client that the distribution has changed and it is no longer in charge of this metadata.
7. The *Client_API* receives the answer and the *Client Distribution* collects distribution specific information provided by the server distribution with the `post_receive` function
8. *Client_API* returns an acknowledgment for the request execution to the *Client* class.

Federating Storage Devices: The Storage_API, shown in Fig. 2, could be connected to any storage media as long as it accepts CRUD semantic too. It enables to use our tool without taking care of the storage media type, but also to federate different media types such as magnetic tapes, databases or local file systems in the same storage system. We can for example have a heterogenous system with some servers linked to a local disk on POSIX [2] and others accessing to a *REDIS* cloud [23]. The API is a wrapper for access functions of each type of media linked to the server.

Input Traces: Our simulator framework accepts different input traces in a simple format: all operations are gathered in a CSV formatted file following the {*timestamp, operation, key, jobid*} pattern. A simple format enables for each user to artificially create workflow that is characteristic of the targeted storage system. If user has no input traces available, one default trace is proposed. It is a real trace extracted from 24h of a supercalculator (further explanation in Subsect. 4.2). Our framework also provides an embedded tool to generate traces representative of a particular workflow depending on user-defined parameters. Synthetic workflow used in the evaluation part (again, see Subsect. 4.2) is generated using this feature.

Controlling Time Passing: We designed a simulation process which controls and simulates time passing in order to run the operations of a trace in an accelerate fashion. Compressing a run and playing it in a limited time enables to limit potential failures during the execution. As our workload measures are not impacted by time, our simulator can play operations in an accelerated mode without any care. Moreover, our time control process ensures load metrics reproducibility: For one distribution method and one particular trace file, each run of these traces will provide same load values. It allows us to avoid noise in load comparison of two distribution methods.

To control time passing, we use a configurable time step mechanism: we play operations of a time step in concurrency and we take measures at the end of the step. It is possible to configure time steps small enough to run the traces in real time: a small time step enables to consider that operations are executed at the timestamp, allowing to measure other metrics influenced by time compression. A small time step takes more time and increase failures probabilities during the run whereas a big time step enables a faster run but could induce approximations.

In the first step of our simulation process, we split the trace file according to the time step. For each step, we spread requests across a user-defined number of clients and then each client executes its operation list concurrently. Once all clients finished, we take measurements for each server, and then we let the distribution method the possibility to make time-depending actions with the `end_epoch` function. Some distribution methods could have periodic checks or rebalancing processes. After this action, we can finally process the next step.

3.3 Metrics and Outputs

We choose as a principal metric of comparison the load of each server: each 5 min, we measure the number of requests received by each server and compute the percentage of load of each server at each step. We also recorded various information during the run such as information which belongs to the distribution method. At the end of the run of traces, the framework generates gnuplot curves such as the display of the received requests over time for the whole metadata service or per server. It also provides per server workload evolution in percentages and some statistics such as the maximum and the average distance between the load on a server and the ideal load it should receive.

4 Evaluation Environnement

4.1 Test Environnement and Experimentals Assumptions

All our tests have been conducted on a cluster composed of Intel Xeon Platinium 8167 processors and 187 GB memory. Nodes are interconnected by Infiniband EDR. Each server and client process runs on independent virtual machines coordinated by a VM-manager named *PCOCC* [24], which enables to host clusters of VMs on compute nodes, alongside regular job allocation. Each virtual machine in the cluster runs on 48 cores with *Centos 7.4* as operating system. Each test represents a new start of the system and storage disks are initially empty.

For evaluated dynamic methods, we assume the redistribution is immediate and does not influence the workload. We measure the load without taking the redistribution computation into account. The load evaluation of each server is negligible (4 arithmetic operations) and the redistribution algorithm needs few computations that can be neglected too. The most expensive part of the rebalancing stage corresponds to the metadata transfers. We choose to ignore this cost because our implementation is not optimized, and it is not the research purpose.

4.2 Workload Specification

To fairly compare methods, we have to evaluate them on different kinds of workloads. We choose on the first hand real traces, recorded on a cluster, that model typical HPC needs, and on the other hand, synthetic traces generated with our tool from a particular I/O pattern to challenge the methods.

Real Traces: We work with real traces extracted from a run of the industrial and academic supercalculator *Joliot Curie* [25]. With 1 656 *Skylake* nodes (Intel Xeon 8168 bi-processors nodes) and 828 *Knight Landing* nodes (many cores Intel Xeon Phi 7250 nodes), the supercalculator has a computational power of 9,4 Pflops. The dedicated storage system of 5 PB is a Lustre system with 2 MDS and 42 OST. Our traces reflect 24h of daily usage on this cluster and are representative of a wide set of HPC applications. Indeed, the usage of this

cluster gathers different scientific fields including machine learning, chemistry, physics and climatology with well-known simulation code such as abinit, cp2k, gromacs or tensorflow. Initially, traces were recorded on the Lustre File System, then tree-specific operations (*mkdir, rmdir, unlink...*) have been adapted into object operations (*Create, Read, Update, Delete*). Figure 3 shows some requests extracted from the traces file.

```
1534555546.812093123,update,15,7
1534555546.812093123,delete,16,0
1534555546.812093124,update,13,1
1534555546.812093129,update,17,3
1534555546.812093131,create,18,4
```

Fig. 3. requests extracted from real traces

Synthetic Traces: The workload we generate follows the temporal pattern given in Fig. 4: on the first 15 min, it has a low level, which represents less than 50% that servers could handle. The workload intensifies linearly during 45 min to achieve a high level, which is 90% that servers could handle. It remains at the high level during 1 h. The number of accessed keys during the run is 10% of the total requests. To simulate a less favorable case, we artificially forced the repartition of the requests across servers in order to have one server overloaded in comparison with the others. The distribution across servers is computed with the initial distribution for the whole traces. So, we set the percentage distribution across our 4 servers at [70,10,10,10].

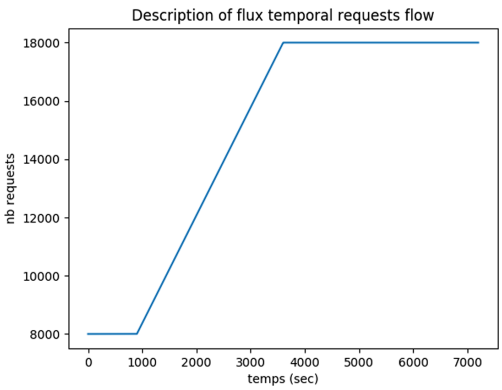


Fig. 4. Temporal request flow created for the metadata service in synthetic traces

5 Evaluated Methods

To have a base of comparison in our tool, we have implemented two state-of-the-art methods: the Static hashing method, which is a basic method and a dynamic version of hashing, to observe evolving behavior. The last described method is an adaptation of the Dynamic Hashing method [9].

Static Hashing: We implement the Static hashing method to distribute metadata across MDS. To know which server request, clients have to calculate hash modulo the number of available servers. To have a better balanced workload, we choose the Murmur3 hash function [26] because of the randomness of the distribution. The hash computation is performed by clients, so servers are fully engaged to process requests. Servers accept all the requests they receive as the spreading policy is done on the client side and the distribution does not change with time.

Dynamic Hashing: To assess the benefit of a dynamic distribution, we choose to implement a dynamic version of the already implemented hashing distribution. We implement a version of the Dynamic hashing first presented by Li et al. [9].

To distribute the namespace, metadata are first computed with a hash function (MurMur3 as said above), split between N entries of an index table (here we take $N = 100$) and each entry is spread across all the MDS. Every new client or server build an index table once and could then refer to the table for all the following requests. Periodically, a workload rebalancing algorithm named RElative LoAd Balancing (RELAB) is performed (we choose every hour). Between each rebalancing, servers compute for every entry a Synthetically Access Information (SAI), which are then summed to obtain the server load. With the complete set of workloads, the RELAB algorithm find a subset of entries from overloaded servers to fill in the underload in each underloaded server. Once these load transfers are defined, we update the index table and execute transfers to achieve the workload rebalancing.

Load-Adaptive: This method is a method we develop, inspired by the Dynamic hashing [9], shown above. Structures from the Dynamic hashing method such as the index table (with $N = 100$ entries) to split the namespace and the Synthetically Access Information (SAI) to evaluate an entry load are reused here. The main idea of this method is to dynamically trigger the workload rebalancing when it is required, limiting useless rebalancing and trying to better fit the overall workload.

For this purpose, a workload checker is integrated in every server. This workload checker regularly computes the server load and compare it to a critical threshold. To obtain the load level for a single server, we sum the SAI of each entry assigned to this server, as in the Dynamic hashing method. The threshold for rebalancing is initially set to 0 and is updated at each rebalancing, allowing to better fit to the current workflow. It is defined by the redistribution algorithm, and is the mean load each server should handle ideally with a margin of 15%. The margin is configurable and enables to rebalance in an aggressive way or not.

When a server reaches the load threshold, a rebalancing phase with a new redistribution algorithm inspired by the RELAB one [9] is requested. In this new algorithm, we try to spread overload across all underloaded servers. If the overload is too big, it is split before spreading. Once the index table is updated, we execute the transfers and the new ideal load threshold is also updated. Having a dynamic period to rebalance enables to have a system more tolerant to heavy workload changes, rebalancing the workload before a server become overloaded.

6 Evaluations and Outputs

We evaluate these three methods with our tool and we take as input the two workflows described in Subsect. 4.2: the real traces and the generated one. We present here outputs curves and an example of how we can analyse them. With real traces, loads are globally balanced with curves oscillating between 20 and 30 % for every method. With 4 servers, the ideal load value that every curve should approach is 25%. Even if it seems to be negligible, a tiny difference of load could lead to heavy imbalance of load if a high burst occurs.

Figure 5 shows the total workflow of requests over time with a measure each 5 min. This curve is about the metadata service in its entirety and not per server, that means it does not change depending on the distribution method: all methods are evaluated with this same request flow. This curve enables us to see the temporal profile of the input traces and see periods with few requests and other periods with high burst. It is easier to evaluate a method's behavior if we can know periods which could be problematic, i.e. periods of burst, or significative load changes.

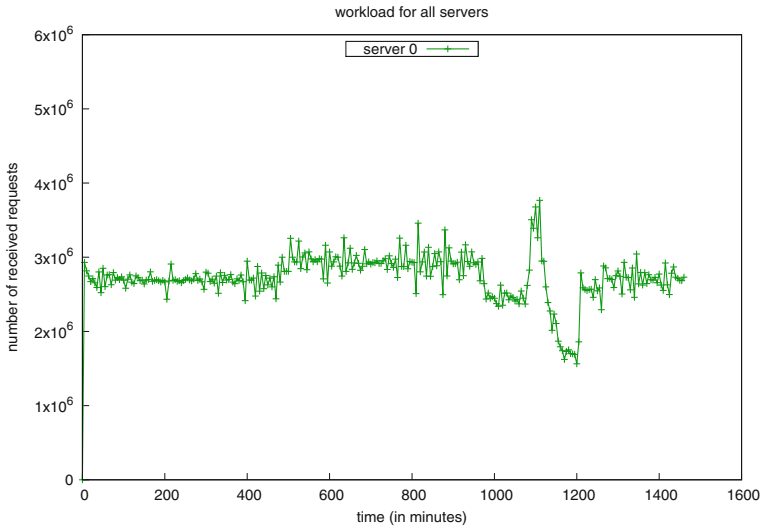


Fig. 5. Total number of requests received with real traces

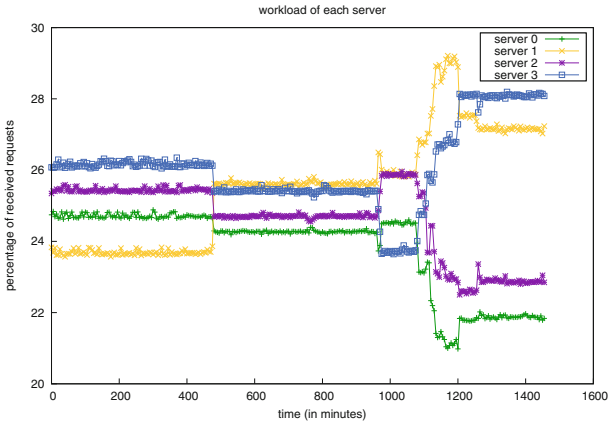
In Fig. 6, we present how requests are distributed across the four servers with a percentage of received requests during 5 min, according respectively to the Static hashing (Subfig. 6a), the Dynamic hashing (Subfig. 6b) and the Load-adaptive method (Subfig. 6c). This enables us to spot imbalance period and see which server is in difficulty. We can use these curves in correlation with the first one, for example to see if a potential unbalanced period matches with a high requesting period, which highlight a big distribution method failing. In the current case, Subfig. 6a shows, during the first 450 min, an imbalance between all servers, while the workflow remain stable (in Fig. 5). This imbalance is fixed with the Dynamic hashing, as we can see in Subfig. 6b: after the first redistribution, illustrated by pink vertical lines, the percentages of workload of each server become nearer.

Again in the Fig. 6, we can see that from 450 min to 1000 min, a real imbalance happens: the server one (in yellow) is really more loaded than the three others. The metadata workflow during this time (in Fig. 5) is bigger than during the other periods, which means Dynamic hashing has difficulties to keep the workload balanced. Indeed, even if rebalancing are performed, the gap remains the same because of the RELAB algorithm: the server one overload could not fit in one idle server, hence there is no metadata transfer due to these redistributions. If we check how the third method behaves in Subfig. 6c, we can see that two rebalancing are performed around 500 min (again illustrated by pink vertical lines) and after these redistributions, the imbalance becomes significantly smaller, which means redistributions were effectives.

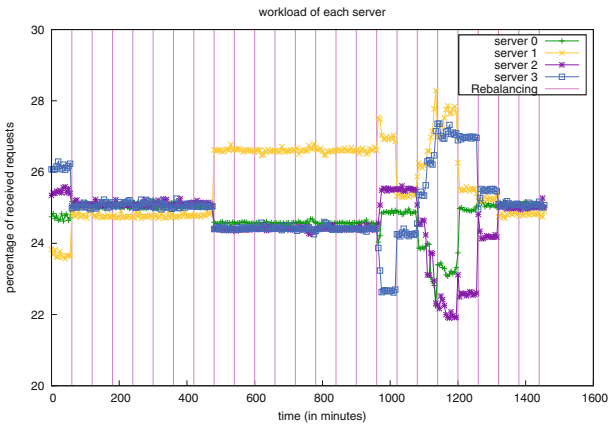
We also generate this kind of curves with the synthetic workload: Fig. 7 shows the requests workflows and Fig. 8 presents the percentage of loads per servers for the Dynamic hashing method (Subfig. 8a) and the Load-adaptive method (Subfig. 8b). We do not present the Static hashing curves here, because it is the same curves as with Dynamic hashing without the redistributions vertical lines. If we look at these curves, we can assert that the Dynamic hashing had failed its rebalancing (for the same reasons than with real traces), while the third method shows a per server workload approaching over time 25%.

In addition of generating curves, our tool could also provide useful information about the run and even about the distribution method. For example, during each dynamic method testing, we record time value when a rebalancing is performed which is an information specific to the distribution method. This allows us to know how many redistributions are performed during a run, but also to evaluate in comparison with the percentage workload curves if a rebalancing had an impact on the workload or not. With the real traces, the Dynamic hashing performs 24 rebalancing (one per hour), while the Load-adaptive method executes 15. For a better comfort, we add these kind of information on top of generated curves, as we can see in Subfigs. 6b, 6c, 8a and 8b with pink vertical lines.

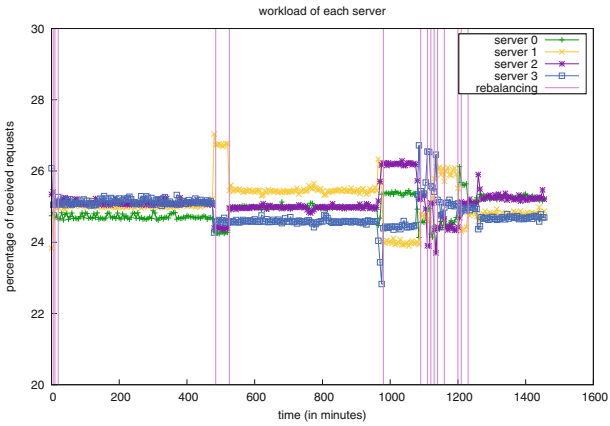
Our tool also provides some useful statistics, such as the maximum and the average distance in percentage between a server load and the ideal server load (here 25%). For example, with the synthetic traces and the Dynamic hashing



(a) For the Static hashing method



(b) For the Dynamic hashing method



(c) For the Load-adaptive method

Fig. 6. Percentage of received requests by servers with real traces

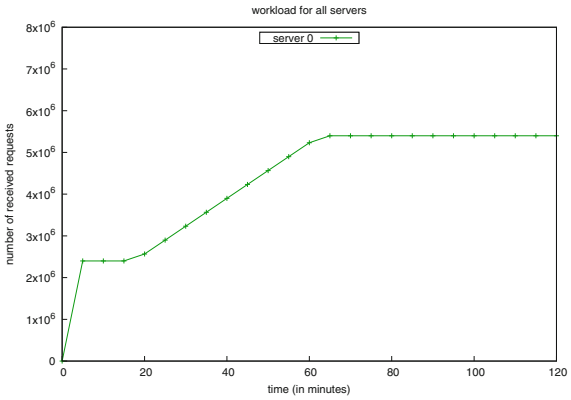
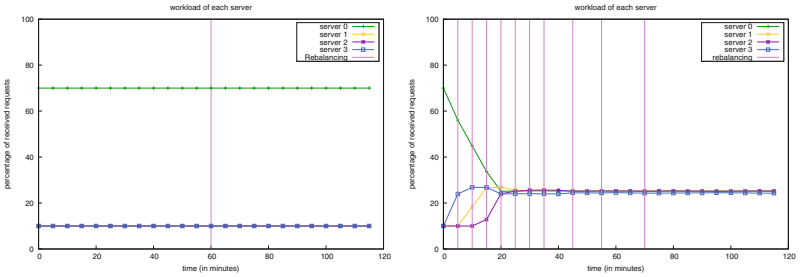


Fig. 7. Number total of requests received with synthetic traces



(a) For the Dynamic hashing method (b) For the Load-adaptive method

Fig. 8. Percentage of received requests by servers with synthetic traces

method, the average distance between the percentage of load on a server and the ideal percentage of load it should reach is 22.50 points which is really high. This means that servers are 22.50% more or less loaded than they should be. The maximum distance is 45.00 points, which is the distance between the load on server one and the ideal load. The Load-adaptive method evaluation shows the average distance between the percentage of load on a server and the ideal percentage of load it should reach is 1.74 points, which is way less than before. The maximum distance is 45.00 points, which is the imbalance we obtain before the first redistribution at the initial repartition.

7 Conclusion

Because of the lack of evaluation tools, state of the art methods should be reimplemented in the storage system for each new method to have a fair comparison. To avoid this waste of time, we presented, in this work, a framework, which allows to evaluate and compare fairly various distribution methods on any kind

of workflows. We proposed an evaluation by our tool of three distribution methods on two specific workflows: first with a real execution case and then with a synthetic unfavorable use case, generated by our tool. We have shown that curves and statistics generated by our tool enable to highlight imbalance points over time. We saw which server was overloaded in comparison of the others, or how many rebalancing was performed during the run. This kind of analyse allows to deduce which kind of workflows puts in trouble a distribution method and so which distribution method better fit to a storage system.

For now, our tool enables to measure workload metrics and highlight overloads due to metadata distribution method. It could be improved in different ways: first we plan to add more metrics to evaluate, as the response time of a request for a client or the cost in time of a redistribution. These metrics could help to better measure the impact of a distribution method which not fit to the workflow. Secondly, we could extend our model to simulate in a more realistic way a storage system: for example, we could add replication or fault tolerance protocols. This would allow us to evaluate distribution method behaviors more completely and even in other unfavorable use cases. Finally, it is also possible to adapt our tool to use it in other research field. The nearest example field is data distribution.

References

1. Raicu, I., Foster, I.T., Beckman, P.: Making a case for distributed file systems at exascale. In: Proceedings of the Third International Workshop on Large-Scale System and Application Performance (2011)
2. IEEE. IEEE 1003 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R)) (1988)
3. Mesnier, M., Ganger, G.R., Riedel, E.: Object-based storage. IEEE Commun. Mag. **41**(8), 84–90 (2003)
4. Meshram, V., Besseron, X., Ouyang, X., Rajachandrasekar, R., Darbha, R.P., Panda, D.K.: Can a decentralized metadata service layer benefit parallel filesystems? In: 2011 IEEE International Conference on Cluster Computing (2011)
5. Singh, H.J., Bawa, S.: Scalable metadata management techniques for ultra-large distributed storage systems-a systematic review. ACM Comput. Surv. (CSUR) **51**(4), 1–37 (2018)
6. Weil, S.J., Pollack, K.T., Brandt, S.A., Miller, E.L.: Dynamic metadata management for petabyte-scale file systems. In: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (2004)
7. DeCandia, G., et al.: Dynamo: amazon's highly available key-value store. ACM SIGOPS Oper. Syst. Rev. **41**(6), 205–220 (2007)
8. Xu, Q., Arumugam, R.V., Yong, K.L., Mahadevan, S.: Efficient and scalable metadata management in eb-scale file systems. IEEE Transactions on Parallel and Distributed Systems (2014)
9. Li, W., Xue, W., Shu, J., Zheng, W.: Dynamic hashing: adaptive metadata management for petabyte-scale file systems. In: 23rd IEEE/14th NASA Goddard Conference on Mass Storage System and Technologies (2006)

10. Tang, H., Byna, S., Dong, B., Liu, J., Koziol, Q.: Someta: scalable object-centric metadata management for high performance computing. In: Cluster Computing (CLUSTER), 2017 IEEE International Conference on (2017)
11. Morrone, C.J., Loewe, B., McLarty, T., Kroiss, R.: Hpc io benchmark repository (2011). https://github.com/hpc/ior
12. Katcher, J.: Postmark: a new file system benchmark. Technical report TR3022, Network Appliance (1997)
13. Xing, J., Xiong, J., Sun, N., Ma, J.: Adaptive and scalable metadata management to support a trillion files. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (2009)
14. Yang, D., Wu, W., Li, Z., Yu, J., Li, Y.: PPMS: a peer to peer metadata management strategy for distributed file systems. In: Hsu, C.-H., Shi, X., Salapura, V. (eds.) NPC 2014. LNCS, vol. 8707, pp. 435–445. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44917-2_36
15. Braam, P.: The lustre storage architecture. arXiv preprint arXiv:1903.01955 (2019)
16. Zheng, Q., Chen, H., Wang, Y., Duan, J., Huang, Z.: Cosbench: a benchmark tool for cloud object storage services. In: 2012 IEEE Fifth International Conference on Cloud Computing (2012)
17. Xue, W., Zhu, M.: Efficient dynamic management of distributed metadata. In: Zhu, R., Ma, Y. (eds.) Information Engineering and Applications. LNEE, vol. 154, pp. 354–362. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2386-6_46
18. Xiong, J., Hu, Y., Li, G., Tang, R., Fan, Z.: Metadata distribution and consistency techniques for large-scalecluster file systems. IEEE Trans. Parallel Distrib. Syst. **22**(5), 803–816 (2010)
19. Wang, J., Feng, D., Wang, F., Lu, C.: MHS: a distributed metadata management strategy. J. Syst. Softw. **82**(12), 2004–2011 (2009)
20. Billa, B.: Medie: a metadata distribution evaluator for object storage systems (2020). https://github.com/Billae/MeDiE
21. Hintjens, P.: ZeroMQ, Messaging for Many Applications. O'Reilly Media, Sebastopol (2013)
22. Battle, R., Benson, E.: Bridging the semantic web and web 2.0 with representational state transfer (rest). J. Web Semant. **6**(1), 61–69 (2008)
23. Sanfilippo, S., Noordhuis, P., Stancliff, M.: Redis, an in-memory data structure store, used as a database, cache and message broker (2009). https://github.com/antirez/redis
24. Diakhate, F., Besnard, J.-B.: Pcocc: Run vms on an hpc cluster (2016). https://github.com/cea-hpc/pcocc
25. CEA. Inauguration of joliot-curie, the french supercomputer dedicated to french and european research (2019). http://www.cea.fr/english/Pages/News/Inauguration-of-Joliot-Curie,-the-French-supercomputer-dedicated-to-French-and-European-research.aspx
26. Scott, P.: C port of murmur3 hash (2011). https://github.com/PeterScott/murmur3