BACOS: A Dynamic Load Balancing Strategy for Cloud Object Storage

Manoel Rui P. Paula[1], **Eduardo Rodrigues**[1],
Victor A. E. Farias[1], **Flávio R. C. Sousa**[1], **Javam C. Machado**[1]

[1]LSBD – Departamento de Computação – Universidade Federal do Ceará (UFC)
Campus do Pici – Bloco 952 – 60.020-181 – Fortaleza – CE – Brazil

{manoel.rui,eduardo.rodrigues}@lsbd.ufc.br
{victor.farias,flavio.Sousa,javam.machado}@lsbd.ufc.br

Abstract. *Cloud Computing is an efficient model for processing and storing large amounts of data. The cloud is composed by heterogeneous resources and has a variable workload. Cloud object storage systems arise as an efficient manager for data using heterogeneous devices, regarding storage capacity and performance. In the cloud, since workload changes dynamically, the dynamic reconfiguration is needed to improve resource utilization. Thus, load balancing techniques are crucial to redistribute workload among the processing nodes to avoid underloading or overloading. Conventional load balancing strategies are only aware of storage devices' capacity, resulting in system performance degradation. To address these limitations, this paper presents a non-intrusive approach to load balancing in the cloud which considers storage devices with heterogeneous performance. Experimental results confirm that our approach improves performance in terms of response time and throughput when compared to the strategy employed by Openstack Swift object storage.*

1. Introduction

Cloud computing is a paradigm of remarkable success for service-oriented computing. Modern data-driven applications in the cloud demand large computing resources. In this scenario, distributed data storages are crucial components on the software stack. For this purpose, cloud object storage has emerged to support such high requirements [Mesnier et al. 2003]. Cloud object storage systems have scalable architecture and are composed by distributed nodes responsible for storing and retrieving data distributed among several servers and their storage devices. The communication among those nodes is typically through a high performance network. These object storage systems provide a high level interface to abstract low level layers of storage devices such as local file system. This high level layer is commonly used to read and write unstructured data as objects, often being multimedia data like documents, images, videos, audio etc. [Mesnier et al. 2003]

Commodity storage devices are used on cloud object storage systems to store data. Therefore, it is possible to extend the total storage space capacity with reduced cost. Most of commodity storage devices are not reliable, but data durability is achieved using replication mechanisms that can be provided by a cloud object storage. However, those devices may be a bottleneck because of their poor performance. [Gunawi et al. 2005] In order to improve system's throughput and latency in storage layer, providers offer storage services

with high performance storage devices as Solid State Drive (SSD) and Serial Attached SCSI (SAS) hard drives. In general, high performance storage devices are composed by costly components or are difficult to be produced, for this reason, their storage capacity is reduced and they are expensive. Consequently, it is infeasible to maintain a storage system where most of the storage devices are high performance and it is interesting to not overload them, avoiding frequent replacements.

Commercial object storage systems as Ceph [Weil et al. 2006], OpenStack Swift [OpenStack 2017], and GlusterFS [GlusterFS 2017] are mainly designed to store large amounts of data on a pool of storage devices based only on the device capacity while not focusing on the device performance aspect. So, they fail in leveraging performance of these devices, especially for devices with distinct (heterogeneous) performance.

In cloud environment, load imbalance happens when a cloud object store dynamically handle different types of operations like read, write and delete from several clients. Eventually, some system components can not manage requests due to lack of resources in that moment, hence compromising system performance [Deshmukh and Deshmukh 2015]. It is a non-trivial problem because is hard to know future system load to adjust the current system resource utilization, especially in a heterogeneous system containing some device components faster then others. Works in state-of-art such as [Tan et al. 2013] and [Hsiao et al. 2013] try to solve this problem rebalancing current storage system load through a data migration scheme between system components to readjust underloaded and overloaded resources. In this scenario, load balancing policies arise as an effective strategy to tackle the performance aspect of object storage systems and improving utilization of heterogeneous devices.

In this paper, we address the load balancing problem to cloud object storage. The term *balance* and *rebalance* are interchangeable in this paper. We introduce BACOS: a dynamic load BAlancing strategy to Cloud Object Storage. BACOS uses previous knowledge about the workload behavior and handle distinct storage device performance to improve system's overall performance. Specifically, we can highlight the following contributions:

- Non-intrusive model that handle storage devices with heterogeneous performance for distributed object storage systems.
- A strategy to adapt to workload changes.
- Testbed for testing production-ready systems. Specifically, we tested with a commercial object storage system.

Organization: The remaining of this paper is organized as follows. Section 2 surveys related work. Section 3 gives a brief introduction on the architecture of an object storage system suitable to BACOS. Section 4 explains BACOS strategy. The implementation and experimental evaluation of BACOS is presented in Section 5. Finally, Section 6 presents the conclusions and future works.

2. Related Work

The work presented in [Wang et al. 2015] proposes a lightweight framework of workload balancing and adaptive resource management for Swift object storage. The framework was designed to balance the workload regularizing virtual and physical nodes. To relieve

the overloaded resources in the distributed physical nodes the framework, instead of migrating data between the nodes, migrates virtual machines between physical machines while the system is running. Additionally, the approach is not intrusive and is designed to be used in a virtual environment through a well-known API. However, how to configure the values of parameters based on a workload analysis is not explored. Also, the authors didn't include storage capabilities to their model.

The work presented in [Chung et al. 2012, Hsiao et al. 2013] proposes a fully distributed load rebalancing algorithm where each storage node is responsible for balance its own load spontaneously, eliminating the need for a central coordinator. The approach implements a DHT protocol to get a fast lookup and handle the load imbalance problem by migrating data load from the heavier nodes to the lighter ones, so that after few interactions all file chunks would be as uniformly distributed as possible among all storage nodes. Additionally, the proposed algorithm aims to reduce network traffic caused by the rebalancing process as much as possible, although it does not take advantage of performance qualities from the storage backend since the main objective is balance the data among storage servers in regard to their storage capacity.

The Adaptive Loading Data Migration in Distributed File System (ALDM) [Tan et al. 2013] strategy is a dynamic algorithm to balance the load of distributed file systems using data migration to mitigate the effect of frequent access to popular data. This approach proposes a measure mechanism to represent the system load status by pricing the system resources against their degradation along time, like network, disk I/O and disk capacity. A central controller node is responsible to dynamically decide when to execute the data migration, which files will be migrated, and where to migrate, to minimize the migration costs. The experiments show that the proposed algorithm can effectively balance the load of data servers, increasing the bandwidth after the migration process. Furthermore, ALDM seems a good adaptability approach to various loads, although it has been tested only in an environment with homogeneous resources.

3. Object Storage Architecture

In this work we assume a cloud object storage architecture as depicted in Figure 1. A cloud object storage is composed of one or more proxy nodes and a set of storage nodes. A storage node is a server that maintains at least a storage device such as a HDD or SSD, which is able to store objects in non-volatile memory. A proxy node is responsible for receiving requests from clients and redirect them to storage nodes.

A Distributed Hash Table (DHT) is a distributed system that provides the functionality of a hash table, mapping a key k to a node n [Felber et al. 2014]. The implementation of DHT is crucial to a object storage performance since the dispersion of the keys among nodes defines the proportion of objects in a storage device. They can improve system performance using a variant of DHT called consistent hash. Where a consistent hash is a hash function such that when a hash table is resized, only K/N keys need to be remapped on average, where K is the number of keys, and N is the number of nodes. An object storage system can also support different types of storage policy, for example, erasure code [Li and Li 2013] or partial replication policy [Nuaimi et al. 2013]. Examples of commercial cloud objects storages are Ceph [Weil et al. 2006], OpenStack Swift [OpenStack 2017], and GlusterFS [GlusterFS 2017].

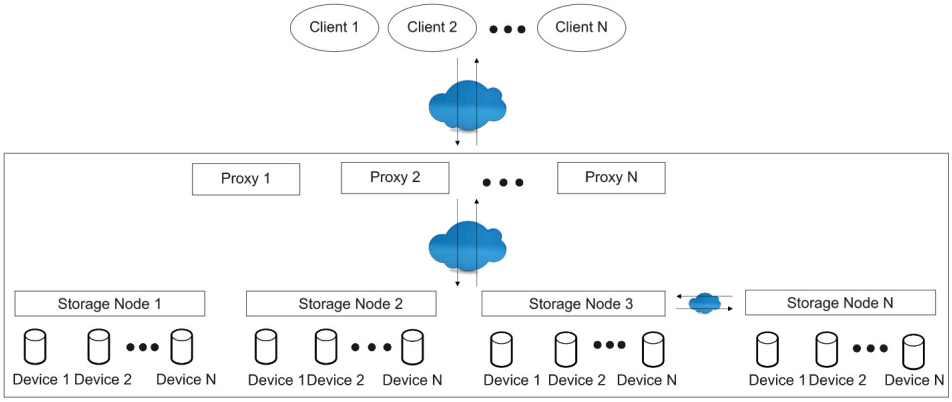


Figure 1. Object Storage Architecture.

In this work, we focus on Openstack Swift, which is an open source object storage system widely used in production by many companies, e.g. Rackspace [Rackspace 2017] and SwiftStack [SwiftStack 2017]. Swift ensures the data is replicated across the cluster to increase data availability and durability. The location where the data should reside in the cluster is determined by a ring, which is responsible to map the data and its replica from the logical locations to their physical locations. The Swift's ring uses a consistent hashing. During the creation of the ring, which is not an automated process, it is possible to set up values like replication count and weight for each device in the cluster. The weight of each device determines the proportion of keys over its partitions. Thus, it causes a direct impact in the distribution of the data among the storage nodes. Note that the data distribution inducts the proportion of requests toward devices.

Swift architecture is composed by different web service nodes and background processes where each component is scalable. The web service nodes are classified in two main categories: proxy server type and storage node type (composed by an account server, a container server and an object server). Each proxy server node and storage node contain a copy of the ring. The background processes are responsible for data replication, data reconstruction, data updating and data auditing. Read and write requests from/to Swift obey the following rules: for reading requests, only one replica node is enough to retrieve the object and return success to the client; for writing requests, the quorum of half of total replica nodes plus one are needed to persist the object and return success to the client.

4. The BACOS Approach

The main purpose of our strategy is to balance requests toward an object storage system by redistributing data proportionally to the performance of the storage devices. Moreover, it redistributes data dynamically according to the workload changes to ensure that the storage system operates in optimal performance. BACOS is based on previous knowledge obtained by a set of offline experiments in a test environment isolated from production system.

Figure 2 shows BACOS's load balancing flow. The storage device classification step measure the performance of devices, identify devices with similar performance and assign them to performance classes; In the system performance measurement step, several workloads are issued to the storage system to measure system performance results

under distinct data distributions among storage devices. In last step, the load balancing method uses a lookup table to find the best data distribution that improves the storage system performance while avoiding overload faster storage devices and minimizing data migration.

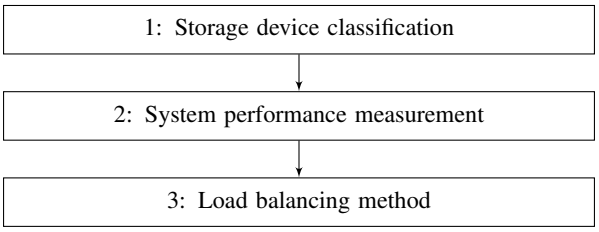


Figure 2. BACOS's load balancing flow.

4.1. Storage Device Classification

To propose a storage device classification, we define a storage device as $d_k \in \{d_1 \dots d_Z\}$, where Z is the total number of storage devices in storage system. Furthermore, we assume the performance of d_k as the max number of Input/Output Operations Per Seconds (IOPS). Then, a benchmark tool is used to issue read and write operations toward the device and is measured the resultant IOPS in order to discover the supported IOPS by a storage device. We recommend using 50% for read and write operations and $1GB$ file for benchmark parameters. This process is done only when a new storage device is attached to storage system.

Once discovered the IOPS of each device, the devices are grouped into performance classes C_j, for $1 \leq j \leq L$, such that devices belonging to the same class have similar performance. As we are dealing with heterogeneous device performance, at least two performance classes should exist, where L is the total number of classes. These classes are constructed by an user-defined threshold for IOPS T_{C_j} per class, that defines which performance class a device is assigned. Assume, without any loss, that $T_{C_1} < T_{C_2} < \cdots < T_{C_L}$. Thus a device d_k is assigned to class C_j where j is the largest $j \in \{1, 2, \cdots, L\}$ such that the IOPS of d_k is larger than T_{C_j}. Note that by the construction of the performance classes, devices in class C_i are faster than devices in class C_j if $1 \leq j < i \leq L$. These performance classes are useful for simplifying the attributes of heterogeneous devices in the storage system and reducing the management complexity of number of devices in the next steps of our approach.

4.2. System Performance Measurement

In this step, we execute several experiments in order to understand the behavior of the storage system. These experiments are run in a test environment isolated from the production environment. In each experiment, we vary workload parameters and system parameters. However, it is important to highlight that BACOS uses high level interfaces from storage system to setup the proportion of data and requests that a device should manage as a system parameter. For this reason, we define class weight $W = (w_1, w_2, \cdots, w_L)$, where the weight w_j specifies the proportion of data and requests that a performance class C_j manage. Hence, w_j defines the proportion of data and requests that a device manage in a performance class. Therefore, in an experiment, we use only the class weight W as

system parameter. The workload parameters that we employ are number of clients N and read/write ratio RW. Thereby, we assign a list of values for each parameter, shown in section 5.2. For each parameter combination (N, RW, W), we execute an experiment of 5 minutes with 30 seconds of warm up and 30 seconds of cool down. Thus we collect and record the mean response time RT, and the total number of errors N_{error} that reflects bad resource utilization or lack of resources by storage system.

4.3. Load Balancing Method

In this step, BACOS generates a lookup table to hold the optimal W for a given workload configuration that maximize the system performance. The optimal class weight is defined by W_{opt}. The lookup table is based on the experiments done in a previous step. In production environment, BACOS monitor the current workload and consult the table to get the W_{opt} for the current moment.

We divide our load balancing method in two substeps: i) the first substep in section 4.3.1, criteria for optimal class weight, presents a score function F as main criterion to evaluate the W_{opt} for each workload configuration and ii) the second substep in section 4.3.2, lookup table construction, shows how the lookup table is created and how to use it in a production system.

4.3.1. Criteria for optimal weights

Since we aim to construct a lookup table in the next substep (section 4.3.2), we use metrics collected in step 2 (section 4.2) to choose an optimal class weight W_{opt} that improves system performance without overloading storage systems. Then, we define a cost function F in Equation 1 which scores the performance of an experiment given its parameters (N, RW, W) with response time denoted by RT and the total error number denoted by N_{error}. Consider $Error_{mean}$ the mean number of error, RT_{max} the maximum mean response time and RT_{min} the minimum mean response time of all experiments executed using a specific workload configuration (N, RW) with the set of class weights in list L_w.

$$F = \frac{RT + (RT_{min} \times P)}{RT_{max} + RT_{min}} \tag{1}$$

Where P is a penalty function for errors in the experiments as depicted in Equation 2. Function P penalizes experiments in which a large number errors occurred. These errors are failed operations, which indicate unbalancing in the storage system, producing overloaded or underloaded storage devices.

$$P = \frac{N_{error}}{N_{error} + Error_{mean}} \tag{2}$$

Note that F is normalized by RT_{max} to generate a final score value between 0 and 1. The smallest the value of F, the best is the performance of the experiment.

4.3.2. Lookup Table Construction

In order to construct the lookup table T, it is necessary to find the optimal class weights W_{opt} that maximize the performance of the system for each workload configuration. In step 2 (section 4.2), for each tested workload configuration, i.e., a pair (N, RW), we did many experiments with a set of class weights W in list L_w. To choose the W_{opt} for a given pair (N, RW), we pick up the best experimented $W \in L_w$ in terms of performance that minimized function F. In the end of process, the lookup table mapped a pair (N_i, RW_j) to a W_{opt_k}. Once the table T is created, BACOS load balance method is ready to query T to get W_{opt} even to a unknown N and RW. BACOS use the nearest neighbor interpolation method [Lehmann et al. 1999] in T to discover the desirable W_{opt_k} from (N_i, RW_j).

```
   input : T ← {(N_1, RW_1) : W_opt_1, ..., (N_i, RW_j) : W_opt_k}
           Dev ← {C_1 : [d_1, ..., d_Z], ..., C_L : [d_1, ..., d_Z]}
           α
   output: 0: no need for load balancing
           1: load balancing success
 1 N^t ← collectCurrentNumberOfClients();
 2 RW^t ← collectCurrentReadWriteRatio();
 3 W^t ← collectCurrentClassWeight();
 4 W_opt ← T[(N^t, RW^t)];
 5 W^t ← W^t + α × (W^t − W_opt);
 6 if dataMigration() or W^t ≈ W_opt then
 7 |   return 0;
 8 else
 9 |   while W^t ≠ W_opt do
10 |   |   for l ← 1 to L do
11 |   |   |   deviceList ← Dev[C_l];
12 |   |   |   foreach dev in deviceList do
13 |   |   |   |   w_dev ← getDeviceWeight(W^t, len(deviceList));
14 |   |   |   |   apply w_dev to dev;
15 |   |   |   end
16 |   |   end
17 |   |   W^t ← W^t + α × (W^t − W_opt);
18 |   end
19 |   return 1
20 end
```

Algorithm 1: BACOS load Balance Algorithm.

The Algorithm 1 presents a pseudo code of BACOS in production system. The first input is the lookup table T containing pairs (N_i, RW_j) as access key and W_{opt_k} as result value. The second input Dev is a dictionary containing a set of storage devices per performance class derived from first step of our strategy (section 4.1). The last input α is the transfer migration factor that represents the proportion of the whole data intended to be migrated per interaction, where $0 \leq \alpha \leq 1$. This parameter is intended to avoid a system collapse during the migration process while new requests continue to arrive. To choose the α value, a trade-off between fast convergence regarding the number of interactions and system performance during object migration process must be considered. The α parameter should be computed based on utilized storage system capacity to mitigate the problem of transferring a huge volume of data through the network and overload storage

system resources. Parameter estimation is out of scope of this work. On lines 1 to 3, current workload and system parameters in time t are collected. On lines 4 to 5, W_{opt} from T is acquired and the new class weight W^t from α are estimated to reach W_{opt}. On lines 6 and 7, it is verified if there are any data migrations in execution or if W^t is very close to the expected W_{opt} at the current moment. In both cases, there is no need to do load balancing. On lines 9 to 19, the object migration between storage devices is done, starting from current W^t to W_{opt}. When W_t is close enough to W_{opt}, the load balance process converge and is finished with success. On lines 13 and 14 the individual proportion of data that each storage device should manage is computed, denoted by weight w_{dev}, and the data proportion of a performance class is spread across all devices in the class.

5. Performance Evaluation

In this section, we describe the experiments used to evaluate the performance of our strategy. The main goal is to show that our strategy can improve storage system performance in a environment with storage devices with heterogeneous performance. We measure performance in terms of three metrics: response time, throughput, and success rate. These experiments are conducted in a private cloud using Openstack Swift Object Storage [OpenStack 2017]. We compare our approach to a baseline strategy recommended in Swift documentation where is suggested to set static weights proportionally to the storage device capacity. For instance, two devices with storage capacity of 1 TB and 2 TB would have weights set as 1000 and 2000, respectively.

The chosen evaluation method was the confidence interval of the mean of the difference between two performance distributions [Jain 1991]. To assess this confidence interval, the samples from the two distributions (BACOS and Swift default strategy) are paired, as they are collected under the same conditions, and their difference results in another distribution. If the confidence interval includes zero, the performance distributions are not significantly different. Otherwise, by analyzing the values of the resulting distribution, it is possible to know how much one strategy has improved the other. To build the confidence interval, we consider a Student's t-distribution with 95% confidence level. This procedure was applied to read and write operations in response time and throughput.

5.1. Environment

To measure the performance of our approach, we used Cloud Object Storage Benchmark (COSBench) tool version 0.4.2 rc2 [Zheng et al. 2013]. COSBench has supported many cloud object storage solutions on the market like Swift, Amazon S3, and Ceph thus making easier any future comparison among those cloud object storage solutions.

We deployed and tested a BACOS prototype in a private cloud on Openstack environment configuring the ring DHT with 2 object replicas. All virtual machines in our experiments were provided by Openstack Nova and storage devices by Openstack Cinder. The virtual machine hosting COSBench was a 4 vCPUs, 8 GB main memory and 80 GB storage capacity. The proxy node configuration had a VM instance with 16 vCPUs, 16 GB main memory and 160 GB storage capacity. We deployed 2 storage nodes, $sn1$ and $sn2$, each one provided with 3 storage devices and with 4 vCPUs, 8 GB main memory and 80 GB storage capacity. The storage node $sn1$ had heterogeneous devices, one with 10 GB and 100 IOPS, another device with 20 GB and 500 IOPS and the last one with 40

GB and 1000 IOPS. The storage node $sn2$ was provided with homogeneous devices in terms of storage capacity and performance, where each one had a device with 40 GB and 100 IOPS.

5.2. Evaluation scenario

The evaluation scenario consists in issuing a workload from COSBench to object storage Swift and collecting metrics like response time, throughput and success ratio to further analysis. We created 32 containers with 5000 objects per container resulting a total of 160000 objects in our experiments, where 2500 objects per container were reserved to read operations and another 2500 objects to write operations. We inserted half of the total number of objects in Swift to simulate an object storage system already in production and required to start concurrent read and write operations. The objects are selected following a uniform distribution. From now on, we will refer to the approach suggested by Swift's documentation, which consists of statically configuring the weights of storage devices proportionally to the device storage capacity [OpenStack 2017] as the baseline approach.

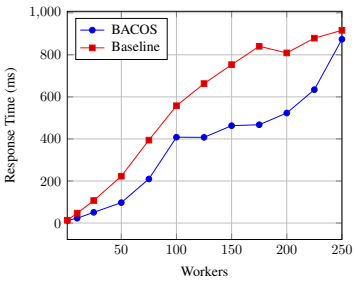
The workload and system parameter values for system performance measurement step (section 4.2) of our approach are: The number of clients N or workers to COSBench tool takes values in $[10, 50, 100, 150, 200, 250]$; we fixed the read/write ratio at $80/20$ and 4KB object size; we define empirically the parameter $\alpha = 1$ for our environment; $W = (W_1, W_2, W_3)$ as a three-sized tuple for weights of 3 performance classes containing integers W_1, W_2, W_3, where we vary W_i from 0 to 1000 with step 50, $W_1 + W_2 + W_3 = 1000$ and $W_1 \leq W_2 \leq W_3$. We used similar parameters from performance measurement step to evaluation, however, the difference when N assumes values in $[1, 10, 25, 50, 100, 125, 150, 175, 200, 225, 250]$ to compare BACOS and baseline are in different scales. The resulting performance metrics (response time and throughput) are collected every second during 300s of observation time for a given N, only discarding 30s from the start and 30s from the end to avoid noises. The collected performance metric values for a given N were aggregated in a mean and the set of mean values of each N represents a distribution of a performance metric.

5.2.1. Response Time

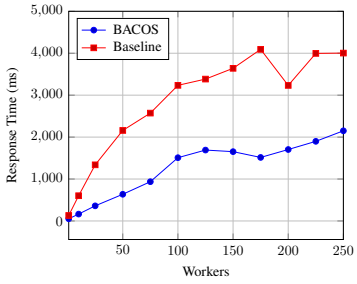
Regarding the response time for read and write operations, BACOS provided low response time values compared to the baseline as depicted in Figure 3. We computed 95% of confidence interval for the mean of difference of response time distributions between the baseline and BACOS for read and write operations as depicted in Figure 4. Thus, we can conclude that our approach was better than the baseline since the zero value was not included in the interval and positive sign of mean denoted that the baseline response times were higher than ours. The Figures 3(a) and 4(a) show the comparison of the response time for read operations between our approach and the baseline. The response time differences between the two approaches increases as the number of workers increases too, since few read operations underutilize storage device resources. Although there is a clear difference between performance distributions of both strategies for read response time, this difference could be greater because of the default replication choice approach adopted by Openstack Swift and considered for both strategies in evaluation. The replica choice approach chooses a replica with same probability between storage nodes and its

decision is not aware of storage device performance. Even so, since in our strategy the largest proportion of replicas are stored in the faster storage devices, choosing a replica from these devices makes the read operation faster than the baseline, where the proportion of replicas are based in the storage capacity and not in the performance.

The same behavior of read response times happens with write response times depicted in Figures 3(b) and 4(b) when the number of workers increases. However, the large difference in response time values between our strategy and baseline in write operations was due to the quorum decision of Openstack Swift. As we worked with two replicas per object, just one replica is necessary to confirm the successful operation. Consequently, there is a high probability that the first replica of an object has confirmed by a fast storage device.

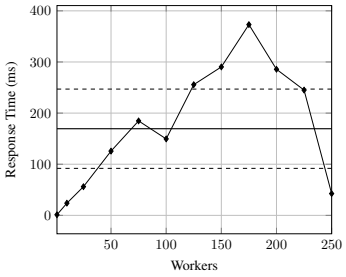


(a) Read response time.

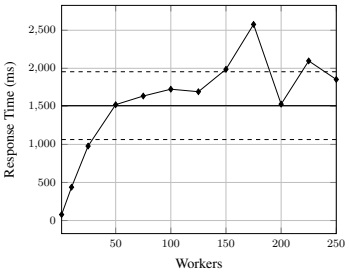


(b) Write response time.

Figure 3. Response time distributions.



(a) Difference of read response time between baseline and BACOS.



(b) Difference of write response time between baseline and BACOS.

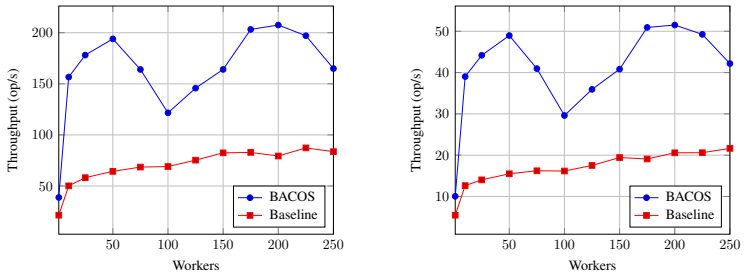
Figure 4. Result of 95% of confidence interval for the mean of difference of response time distributions between baseline and BACOS for read and write operations.

5.2.2. Throughput

To compare the throughput for read and write operations in Figure 5, we computed 95% of confidence interval for the mean of difference of throughput distributions between BACOS and baseline for read and write operations as shown in Figure 6. We can conclude that in both types of operations, our approach was better than the baseline since the zero value was not included in the interval and the sign of the mean was positive, indicating that the BACOS throughput rates are better than baseline. Also, Figures 5 and 6 show

that read and write operations have a similar behavior. As the proportion of number of operation has been set up with 80% to read and 20% to write, it was expected that the results show a throughput much bigger in read operations than in write ones. The best throughput rates of read and write operations were reached by BACOS when the storage handled 200 workers, however the system could not keep providing similar values since it had to deal with a high demand and lack of storage device resources. On the other hand, with same system resources, the baseline strategy wasn't able to reach similar throughput rates because of its inefficiency to manage storage device performance.

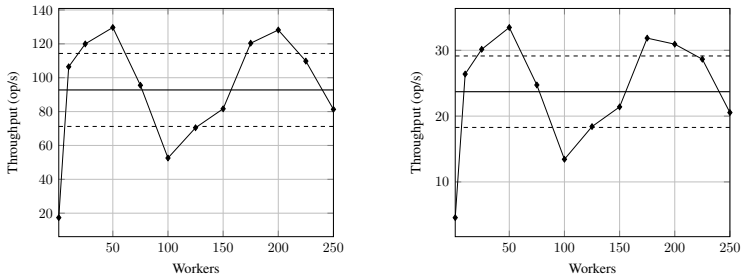
Analyzing read throughput in Figure 5(a) and write throughput in Figure 5(b) side by side, the experiments have shown that our strategy was not able to keep providing high throughput rates when the number of workers was between 50 and 150. This behavior was expected, since our approach made the decision to change weight of performance classes, hence it had to execute migration of a large number of object to ensure further performance. While this process is running, it uses more system resources to inner management than to handle the users requests, therefore the system performance decreases not only in number of operations but also in response time as shown in Figure 3. Although the throughput behavior of our strategy was not so smooth compared with the baseline, the experiments still have shown good results regarding to difference of throughput rates for all number of workers as depicted in Figures 6(a) and 6(b).



(a) Read throughput.

(b) Write throughput.

Figure 5. Throughput distributions.



(a) Difference of read throughput between BACOS an baseline.

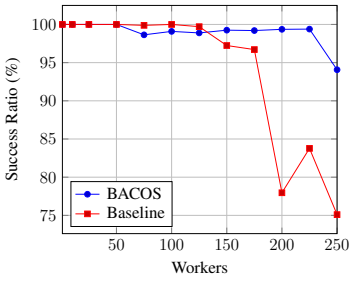
(b) Difference of write throughput between BACOS an baseline.

Figure 6. Result of 95% of confidence interval for the mean of difference of throughput distributions between BACOS and baseline for read and write operations.

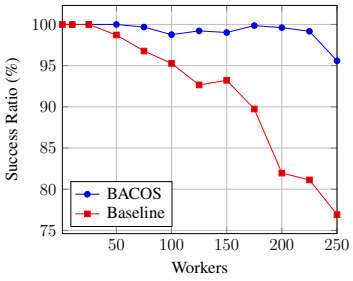
5.2.3. Success Ratio

To compare success ratio in Figure 7 with respect to read and write operations, our strategy was much better than the baseline strategy because in the worst case (workload configured with 250 workers) it kept the success ratio above 94% against 74% from the baseline. From analysis only of the success ratio of read operations in Figure 7(a), when the number of workers was between 50 and 125, we see that the baseline strategy was better than BACOS since it kept 100% of success ratio, although our strategy got very close to reach 100% of success ratio while objects were migrating between storage devices. Once the number of workers rises, the baseline strategy was not able to keep the same success ratio. Even on high demand, BACOS still provide a reasonable success ratio, but the success ratio of baseline decreases drastically since the system could not conclude read operations for more than 175 workers.

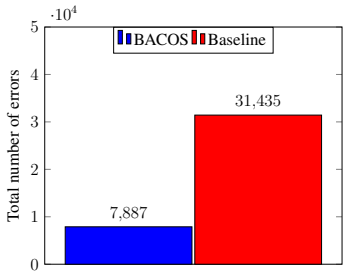
A similar behavior happens to the write operations in Figure 7(b), although the success ratio decreases earlier compared to read operations. The precocious decline of success ratio of write operations in the baseline is justified by the fact that the baseline algorithm considers only capacity, so the strategy will decide to write in storage devices with high available space but not necessarily with best performance. Thus, writing replicas delays new write operations that will be waiting long time in request list of object storage system.



(a) Read Success Ratio.



(b) Write Success Ratio.



(c) Error statistic.

Figure 7. Success ratio comparison.

Note that Figure 7(a) and Figure 7(b) show that as the number of workers increases, it becomes more difficult to maintain the success ratio due to the lack of resources or the bad resource utilization for both strategies. Figure 7(c) show the total number of errors and complements the results of success ratio for read and write operations. The number of errors represents failed operations and express unpredictable and undesirable behaviors. The total number of errors for BACOS was much smaller than baseline strat-

egy once our strategy takes it into consideration.

6. Conclusion and Future Work

This paper presented BACOS, an approach to load balancing in cloud data storage systems that uses previous knowledge of workload behaviour and improves the system performance taking advantage of heterogeneous storage devices. BACOS designates a reasonable amount of requests to faster storage devices without overloading its performance resource. Also, BACOS is a non-intrusive approach that consumes high level interfaces from the storage system to change storage device demand according to a dynamic variation in workload.

In order to evaluate BACOS, experiments that measured response time, throughput and success ratio were conducted in Openstack Swift, a popular commercial object storage. BACOS improved response time and throughput to read/write operations compared to the strategy recommended in official Swift documentation. Additionally, BACOS could provide high success rates even in high demands. Although the experiment evaluated our strategy in a small cluster, our strategy is not limited by the number of the storage nodes. Results corroborate that BACOS improves performance to object storage systems in cloud environments.

There is a number of research opportunities that derive from this work, including: execute experiments with large objects and different ratio of read and write; add support for workload prediction; and incorporate forecast models. Additionally, BACOS can be combined with our previous work about replica selection [Almeida et al. 2016] to improve storage overall system performance. Finally, we intend to conduct a study changing of the number of replicas and consistency issues.

Acknowledgements

This research was partially supported by Hitachi Data Systems (HDS), Funcap/Brazil and LSBD/UFC.

References

Almeida, A. M. R., Cavalcante, D. M., Sousa, F. R. C., and Machado, J. C. (2016). LB-RLT approach for load balancing heterogeneous storage nodes. In *SBRC*.

Chung, H. Y., Chang, C. W., Hsiao, H. C., and Chao, Y. C. (2012). The load rebalancing problem in distributed file systems. In *IEEE Int. Conf. on Cluster Computing*, pages 117–125.

Deshmukh, S. C. and Deshmukh, S. S. (2015). A survey: Load balancing for distributed file system. *International Journal of Computer Applications*, 111(5).

Felber, P., Kropf, P., Schiller, E., and Serbu, S. (2014). Survey on load balancing in peer-to-peer distributed hash tables. *IEEE Communications Surveys & Tutorials*, 16(1):473 – 492.

GlusterFS (2017). Glusterfs. `https://www.gluster.org`. Accessed: 2017-04-03.

Gunawi, H. S., Agrawal, N., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Schindler, J. (2005). Deconstructing commodity storage clusters. *SIGARCH Comput. Archit. News*, 33(2):60–71.

Hsiao, H.-C., Chung, H.-Y., Shen, H., and Chao, Y.-C. (2013). Load rebalancing for distributed file systems in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):951–962.

Jain, R. (1991). *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. J. Wiley & sons, New York.

Lehmann, T. M., Gonner, C., and Spitzer, K. (1999). Survey: interpolation methods in medical image processing. *IEEE Transactions on Medical Imaging*, 18(11):1049–1075.

Li, J. and Li, B. (2013). Erasure coding for cloud storage systems: A survey. *Tsinghua Science and Technology*, 18(3):259–272.

Mesnier, M., Ganger, G. R., and Riedel, E. (2003). Object-based storage. *IEEE Communications Magazine*, 41(8):84–90.

Nuaimi, K. A., Mohamed, N., Nuaimi, M. A., and Al-Jaroodi, J. (2013). A partial replication load balancing algorithm for distributed data as a service (daas). In *Int. Conf. on High Performance Computing and Simulation (HPCS)*, pages 35–40.

OpenStack (2017). Openstack swift. `http://docs.openstack.org/developer/swift/`. Accessed: 2017-04-03.

Rackspace (2017). Rackspace. `https://www.rackspace.com/`. Accessed: 2017-04-03.

SwiftStack (2017). Swiftstack. `https://www.swiftstack.com/`. Accessed: 2017-04-03.

Tan, Z., Zhou, W., Feng, D., and Zhang, W. (2013). Aldm: Adaptive loading data migration in distributed file systems. *IEEE Transactions on Magnetics*, 49(6):2645–2652.

Wang, Z., Chen, H., Fu, Y., Liu, D., and Ban, Y. (2015). Workload balancing and adaptive resource management for the swift storage system on cloud. *Future Gener. Comput. Syst.*, 51(C):120–131.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C. (2006). Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association.

Zheng, Q., Chen, H., Wang, Y., Zhang, J., and Duan, J. (2013). COSBench: cloud object storage benchmark. In *Proceedings of the 4th ACM/SPEC Int. Conf. on Performance Engineering*, pages 199–210. ACM.