LB-RLT Approach for Load Balancing Heterogeneous Storage Nodes

Antonio M. R. Almeida, Denis M. Cavalcante, Flávio R. C. Sousa e Javam C. Machado [1]

[1] Mestrado e Doutorado em Ciencia da Computacao (MDCC)
Universidade Federal do Ceará (UFC) – Fortaleza, CE – Brasil

{manoel.ribeiro,denis.cavalcante}@lsbd.ufc.br {sousa,javam}@ufc.br

Abstract. *Cloud computing is a paradigm of service-oriented computing and has changed the way computing infrastructure is abstracted and used. The cloud is composed by heterogeneous resources and has a variable workload. Thus, load balancing techniques are crucial to distribute workload to processing nodes for enhancing the overall system performance. Conventional algorithms for load balancing have limitations in this environment, or they do not consider specific aspects of the resources simultaneously, e.g., response time and throughput. To address these limitations, this paper presents an approach to load balancing in the cloud. This approach considers cloud infrastructure storage throughput and a heterogeneous storage nodes environment. In order to evaluate this approach, we have conducted experiments that measure the response time, throughput and success rate and compared our results against conventional algorithms. Experimental results confirm that our approach ensures quality of service agreement, while using resources more efficiently.*

1. Introduction

Cloud computing is a paradigm of service-oriented computing and has changed the way infrastructure is abstracted and used. Infrastructure as a service (IaaS) is the most basic cloud model for provisioning compute, storage, and networking resources. In cloud storage, there are major types of storage resources, layered on the following order bottom-up: (a) Block storage: low level I/O access to records through a fixed size (i.e. volumes); (b) File storage: files are composed by its data and are organized through a hierarchical directory. A file is associated to its directory by a well-defined system meta-data that also provides some basic information such as access time, file size etc; (c) Object storage: objects are composed by data and user-accessible attributes (i.e user meta-data) [Mesnier et al. 2003]. User meta-data can be included into objects using any custom key/value, thus making data and user meta-data physically together.

Object storage systems are well suited to scaling-out by replicating the object itself several times for redundancy as well as for responding to a higher demand. During a higher demand, a common necessity of cloud providers is to be able to meet the most strict Quality of Service (QoS) levels defined in a Service Level Agreement (SLA) better using the available resources before allocating more replicas/servers. However, it is still a challenging issue for dynamic application providers how to efficiently tackle scenarios of gradual load variations and load peaks from the workloads seen by dynamic applications.

For this challenge, the proper load balance of replicas are a key point to avoid violations of SLA requirements and to reduce infrastructure costs. [Xie et al. 2015]

Many works have been developed in order to optimize the load balancing of replica placement in systems based on centralized naming such as Lightweight Directory Access Protocol (LDAP) or distributed ones such as distributed hash tables (DHT) (i.e. hash functions) that avoids searching for available nodes in a central directory [Brinkmann et al. 2000] [Allcock et al. 2002] [Godfrey et al. 2004]. HUSH and CHUSH algorithms have proposed hashing functions for reducing look-up response time as well as load balancing the replicas according to server weighting [Honicky and Miller 2004] [Weil et al. 2006].

Another important front of research has focused on load balancing of replica selection, i.e., which replica is the most suitable for being used taking into account some aspects such as performance, cost, security, number of accesses, and replica crashes [Rajalakshmi et al. 2014]. By this means, replica selection is part of load balancing techniques that aim to distribute workload to processing nodes for enhancing the overall performance of system.

Many common aspects of cloud systems leveraged by load balancing approaches have been classified according to the type of algorithm, knowledge base, issues to be addressed, usage and drawbacks [Katyal and Mishra 2013]. Even though some authors have not yet approached fine-grained load balancing by distributing the read workload over data replicas [Weil et al. 2006]. Other authors optimized replica selection by considering disk head scheduling, cache utilization and inter-brick load balancing [Lumb et al. 2003]. It is clear that these load balancing approaches could be considered by replica selection optimization in order to improve some specific aspect of a cloud system.

An ideal load balancing algorithm for homogeneous resources should avoid overloading or under loading of any specific node. However, the cloud is composed by heterogeneous resources and variable workloads, thus making harder the implementation of load balancing strategy because it involves additional processing to efficiently detect how much one replica should be selected more than others while still avoiding starvation. Conventional algorithms for load balancing have limitations in this environment, or they do not consider specific aspects of the resources simultaneously, e.g., response time and throughput. To address these limitations, this paper presents an approach for load balancing replica selection in a cloud object storage. The major contributions of this paper are as follows:

- It proposes an approach to load balance replica selection of read requests for heterogeneous nodes by combining and comparing their last read data throughput and their total of read data.
- It presents a full prototype implementation in a real cloud object storage system, an experimental evaluation and the results showing that our approach is effective in balancing replica selection of read requests for heterogeneous storage nodes environment.

Organization: This paper is organized as follows: Section II explains our approach. The environment is described in Section III. The evaluation of our approach is presented in Section IV. Section V surveys related work and, finally, Section VI presents the conclusions.

2. LB-RLT Approach

Our approach, Load Balance by Round-robin, Last Read Data Chunk Throughput and Total of Read Data Chunks (LB-RLT), distributes dynamically the read requests of objects through node replicas. LB-RLT takes advantage of an architecture that consists of a proxy node accounted for spreading the workload through multiple node replicas. Every node replica is a storage node that is able to read/write one object replica requested or submitted by a proxy node in order to support a client demand for an object, thus making the object access transparently for the client as shown in the Figure 1.

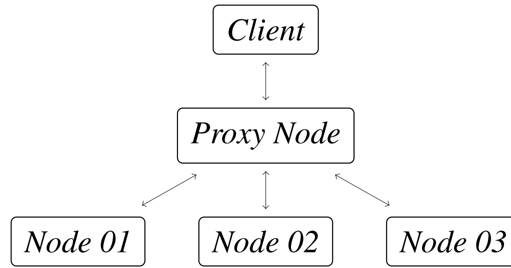

Figure 1. Architecture

To improve understanding, Table 1 shows the definitions. In the Figure 1, we can think about how an object is requested or submitted by the client until it reaches the Node 01, Node 02 or Node 03. It is a common technique for transferring objects to split and transfer the object in small units called dataChunk. These dataChunks have a fixed size, thus the client sends/reads multiples dataChunks to/from the proxy node that forwards/retrieves them until an object is totally transferred. For our replicated system, one replica is sufficient to process a read request.

Table 1. Definitions

Term	Description
DataChunk	A fragment of an object
DataChunkSize	Standard size of a dataChunk
DataChunkTransferTime	Total transfer time in seconds of a dataChunk between a proxy node and a storage node
Throughput	Transfer rate per object or DataChunk (bytes / seconds)
LastReadDataChunkThroughput	The throughput measured by the last reading of a dataChunk from a storage node
TotalReadDataChunks	The number of dataChunks sucessfully read from a storage node until present moment

It is important to note that the transfer time for each dataChunk to be transferred is very sensitive to the network as well as the node performance, i.e., dataChunkTransferTime gets lower when less requests are demanded to a storage node or gets higher when more requests are demanded to a storage node. By this means, we define the Equation 1:

$$Throughput = \frac{dataChunkSize}{dataChunkTransferTime} \quad (1)$$

Considering the previous architecture showed at Figure 1 and the Equation 1 used to measure throughput during the reading of objects, LB-RLT adds to the proxy node the capability of collecting in runtime the following metrics for every storage node:

- *LastReadDataChunkThroughput:* The measurement of a throughput evaluated in the last reading of dataChunk from a storage node by a proxy node as described by the Equation 1. In this manner, the lastReadDataChunkThroughput metric is updated frequently instead of being collected only when an object is totally read. Once the proxy node holds a lastReadDataChunkThroughput for each storage node regardless of which object is being read, every new read object request will consider the last throughput performed for each storage node.
- *TotalReadDataChunks:* The measurement of how many dataChunks were read from each storage until present moment, i.e., totalReadDataChunks metric is incremented only for dataChunks received with success by a proxy node.

By having both metrics being evaluated in runtime and using round-robin sorting to spread equally when the previous metrics are the same, LB-RLT adds to the proxy node, the capability of load balancing the new coming read request by evaluating and comparing the storage nodes using the Equation 2.

$$Performance = lastReadDataChunkThroughput \times \ totalReadDataChunks \tag{2}$$

Our performance model is able to evaluate the node performance when throughput metric is much higher/lower than the totalReadDataChunks, thus being the important value for distinguishing performance among replica nodes as well as when throughput is relatively similar among the replica nodes. In this second case, the totalReadDataChunks is able to evaluate which storage node has performed more reading requests with success in order to distinguish the best node. Our approach considers different types of workloads according to the following:

- Low/medium workloads: any replica node would respond the requests with success, but the best replica nodes would be more used because of their highest throughput;
- High workloads: best replica nodes would still respond with success, but their throughput would get a little bit down. In this scenario, our model would let the worse replica nodes be chosen a little bit more due the fact that it is still better to use the bad nodes than otherwise;
- Extreme High workloads / Above cloud capacity: any storage node replica would be used to attend the extreme workload stress.

LB-RLT aims to load balance heterogeneous storage node by targeting high throughput, low response time and longer high success rate with no profiling information required. Table 2 summarizes the features of the approach.

2.1. LB-RLT Algorithm

LB-RLT is described by two algorithms: (1) the LB-RLT Performance Score Evaluation Algorithm 1 that updates the performance score of each replica node according to our approach and (2) the LB-RLT Read Object Request Algorithm 2 that sorts the replica nodes using our approach and choose the best one to read the data object.

In Algorithm 1, on line 1, the input parameter replicasNodeList is all the storage nodes that contain a replica of the requested object. On line 2, the output parameter is

Table 2. LB-RLT Approach

Type of Algorithm	Dynamic and Centralized
Prerequisite	Replicated Data
Knowledge Base	Locally and in runtime measured
Issue to be addressed	LastReadDataChunkThroughput, TotalReadDataChunks
Usage	Heterogeneous Nodes / No profiling needed
Measure expiration	yes

the replicaNodeList with its performance score updated for each replica node. On line 5, LB-RLT evaluates the performance score of the current node.

Algorithm 1: LB-RLT PERFORMANCE SCORE EVALUATION

```
1  Input: replicasNodeList
2  Output: replicasNodeList
3  foreach node in the replicasNodeList do
4  |   node.performanceScore ← node.lastReadDataChunkThroughput ×
5  |_  node.totalReadDataChunks
6  return replicasNodeList
```

Analyzing the Algorithm 2, on line 1, the input objectData is the data object to be read by the proxy node from a replica node. On line 2, the input replicasNodeList is all the replica nodes that can be used to retrieve the data object. On line 3, the output dataChunk[] is the data chunks reads of the object data. On line 4, the Algorithm 1 is performed to evaluate the performance score of all the replica nodes. On line 5, the list of storage nodes is sorted according to our load balancing approach. In case of similar performance score among the replica nodes, the replicaNodeList is sorted using round-robin. On line 6, the replica node with best performance score is chosen to start the reading of the data object. On line 7, proxy node begins to read all the data object dataChunk by dataChunk from the storage node. On line 9, it is verified if the reading of the dataChunk has succeeded. In positive case, on line 10 and on line 11, for the current best node, the lastDataChunkThroughput and the totalReadDataChunks metrics are updated.

Algorithm 2: LB-RLT - OBJECT READ REQUEST

```
1  Input: objectData
2  Input: replicasNodeList
3  Output: dataChunk[]
4  replicasNodeList ← evaluateLBRLTPerformanceScore(replicasNodeList)
5  replicasNodeList ← sort(replicasNodeList)
6  bestNode ← getFirstNode(ReplicasNodeList)
7  foreach dataChunkIter in the objectData do
8  |   dataChunk[i] ← readDataChunkFrom(bestNode)
9  |   if ReadDataChunkSucceeded then
10 |   |   bestNode.lastDataChunkThroughput ← dataChunkSize /dataChunkTransferTime
11 |   |_  bestNode.totalReadDataChunks += 1
   |_
12 return dataChunk[]
```

3. Environment

There are many cloud object storage solutions available, such as Ceph, GlusterFS and OpenStack-Swift. In this work, we used OpenStack-Swift that is a highly available, distributed, eventually consistent object/blob store [Swift 2016]. For evaluating our experiment, we chose Swift due its simplicity, once it seemed easier to implement our load balancing approach. We use the Ubuntu 12.04 operating system and Swift Kilo version.

3.1. OpenStack-Swift

Swift architecture can hold many different web service nodes and background processes whereas each component can be scaled multiple times. The web service nodes are classified in two main categories: proxy server type and storage node type (composed by account server, container server and object server). The background processes are responsible for data replication, data reconstruction, data updating and data auditing. The proxy server node takes requests from a client and forwards them to the account, container and object server nodes in order to persist/retrieve data objects and its metadata to/from disks. Swift implements a ring (consistent hashing) in order to map data and its replicas from the logical locations to their physical locations.

Swift implements partial replication model and flexible configuration for number of nodes and number of replicas. Read and write requests from/to Swift obeys the following rules: for reading requests, it is enough only one replica node to retrieve the object and return success to the client; for writing requests, it is enough the quorum of half of total replica nodes plus one to save the object and return success to the client.

3.2. Benchmark

In order to evaluate our approach, we used *Cloud Object Storage Benchmark* (COSBench) [Zheng et al. 2013]. COSBench has support to many different 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 use the COSBench 0.4.0.1 version.

For our workload, we fixed read/write ratio at 80/20. We varied tiny sizes for object data size: 4KB and 16KB. For each object data size, we varied the number of threads/workers: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 700 in order to compare the approaches in different scales. We did not vary the delay between requests. The performance of the load balancing approaches was compared using the average values of the benchmark metrics collected in a interval of 5 seconds by COSBench with transaction phase of 300s and the default values for other COSbench parameters. In this paper, we considered three metrics for evaluating the approaches: average response time, throughput and success ratio.

3.3. Hardware

For our hardware configuration, we deployed five different physical machine nodes. One node for COSBench and four nodes for a minimalist Swift deployment. For our Swift deployment, we configured one proxy node with more processing power and three heterogeneous nodes. Storage Node 02 (SN2) and Storage Node 03 (SN3) presented best performance, whereas Storage Node 01 (SN1) presented poor performance during our initial performance profiling. Hardware details are described at the Tables 3 and 4.

Storage Node (SN1) was chosen on purpose, with a disk architecture that offers less IO performance than we, as Table 4. This aspect tries to play a scenario with heterogeneous nodes in a data center.

For our swift deployment with three storage nodes, we configured three replicas for each data object just because it is a common setup and it does not make sense to have more replicas than the number of nodes even though our approach works is agnostic to the number of replicas and nodes.

Table 3. CPU Configuration

Node	Cores	MHz	L1 / L2 / L3 (Cache)
COSBench	8	800	32 / 256 / 8192
Swift Proxy	8	800	32 / 256 / 8192
SN1	2	**1995**	32 / **2048** / N/A
SN2	4	1600	32 / 256 / 6144
SN3	8	800	32 / 256 / 8192

Table 4. RAM and Disk Configuration

Node	RAM(GB)	R/RW(IOPS)	Filesystem
COSBench	8	133 / 100,4	xfs
Proxy Node	16	161,4 - 123,8	xfs
SN1	3	**98,4 / 80,4**	xfs
SN2	8	147,8 / 119,4	xfs
SN3	4	142,6 / 121,4	xfs

IOPS benchmark performed by random disk operations for a workload of 4KB by 16 threads simultaneously using FIO - Flexible I/O Tester Synthetic Benchmark.

4. Evaluation

In this section, we describe the results we have obtained from it. According to the available Swift configuration, we verified three different load balancing approaches implemented by the Swift object storage, thus making our approach comparable against production-like approaches described in the Table 5.

Approaches under evaluation are classified according to their types and needed for profiling as shown in the Table 6. For LB-RLT evaluation against other approaches, we focused on analyzing which load balance approach obtained best results from the premise described in COSBench paper which considers a better system one that is able to engage more clients with same resources [Zheng et al. 2013]. In our experiments, we configured the LB-RW approach to give preference to the best two nodes (SN2 and SN3).

4.1. Replica Selection Monitoring

The replica selection behavior of the load balancing approaches was monitored enabling ”statsd” time-series metrics [Malpass and Malpass 2011] via UDP protocol for every Swift node in order to confirm which storage nodes were being chosen for every load balancing approach. Below, we describe what happened in the experiment for 16KB as object size since the overall behavior of the approaches were similar for 4KB as object size.

Table 5. Approaches performed

Notation	Description
LB-R	Load Balance by **Round-robin** distribution. In the Openstack Swift, this is called the shuffling method (tested) [Ying et al. 2015].
LB-RW	Load Balance by Round-robin with static **Weights**. Profiling of nodes performance is needed. In the Openstack Swift, this is called the affinity method (tested).
LB-RCT	Load Balance by Round-robin and **Connection Timing**. In the Openstack Swift, this is called the timing method (tested).
LB-RL	Load Balance by Round-robin and **LastReadDataChunkThroughput** performed by each storage node (proposed and tested).
LB-RLT	Load Balance by Round-robin, **LastReadDataChunkThroughput** and **TotalReadDataChunks** performed by each storage node, i.e., total of dataChunks successfully read (proposed, tested and the best choice).

Table 6. Compared Approaches

Approach	Profiling	Run-time measurement	distribution criteria	Standard
LB-R	N/A	No	Round-robin	Yes
LB-RW	Needed	No	Weights	Yes
LB-RCT	N/A	Yes	Response-time	Yes
LB-RL	N/A	Yes	Throughput	Custom
LB-RLT	N/A	Yes	Throughput and Total succeeded Reads	Custom

The LB-R approach balanced equally the workload for all the three storage nodes. Since it is the obvious load balancing behavior, we did not show its monitoring in the Figure 2. LB-RCT had similar results in comparing with LB-R, but it got worse even though LB-RCT also used round-robin technique. In the beginning of the experiment, LB-RCT gave preference to the worse node (SN1). As LB-RCT strategy uses no disk access, since it performs only an HTTP connection test, this happened mainly because SN1 has the largest L2 memory cache. This can be verified at the Item C of the Figure 2, for the LB-RTC approach, the continuous line (SN1) stayed above the dashed lines (SN2 and SN3). According to [Foong et al. 2003], when transfer sizes are less than L2, the cache is large enough to accommodate all of the application buffers. In LB-RCT case, L2 cache memory made TCP connection faster for the worse storage node (SN1).

LB-RW balanced the workload only to the SN2 and SN3 according to the previously configured weights with exception when the benchmark tool began to overload the cloud object storage in such a way that was impossible its proper usage. By this means, Swift implementation tries any available replica node. So, it is clear, at the Figure 2, that LB-RW used SN1 (worse storage node) only when the system was already unable to respond properly the requests.

LB-RL balanced almost equally the workload to all storage nodes. We believe this happened because the way we implemented LB-RL caused the replica selection to prefer nodes with higher throughput values without considering which nodes were processing more successful readings, thus not being enough to evaluate storage node performance dynamically.

Our approach LB-RLT load balanced the workload with three behaviors: (1) at the very beginning, SN2 or SN3 was used because only one was able to respond the requests,

(2) after this very beginning and before the half of the experiment, both SN2 and SN3 were used and (3) before a little bit the half of the experiment, both SN2 and SN3 continued to be used, but LB-RLT started to use the SN1. This last behavior happened because the two best nodes were not enough for processing the increased workload as described by the continuous line representing the SN1 at the Figure 2.

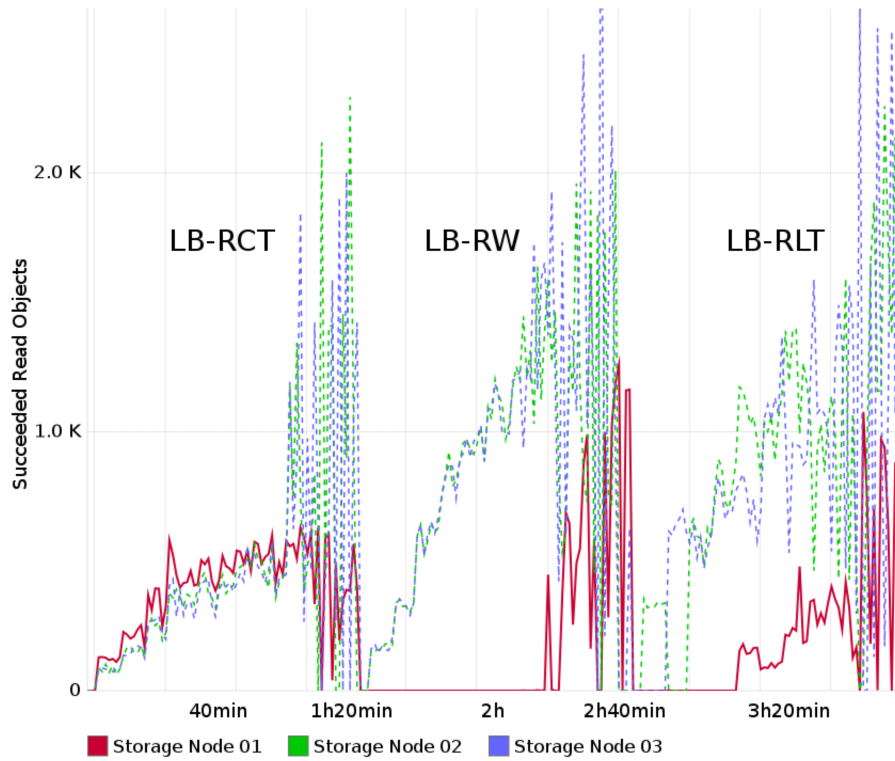

Figure 2. Load Balancing Monitoring for 16KB.

4.2. Performance Comparison

LB-RLT and LB-RW performed more operations with success than the other approaches. This average of total of operations performed with success is shown at the Table 7.

Table 7. Number of operations (avg. ops) for 100% Success Rate

Approach	4KB read	16KB read	4KB write	16KB write
LB-R	1.66×10^4	1.10×10^4	3.70×10^3	2.10×10^3
LB-W	1.90×10^4	1.44×10^4	4.46×10^3	2.85×10^3
LB-RCT	1.50×10^4	1.09×10^4	3.16×10^3	2.06×10^3
LB-RL	1.85×10^4	1.08×10^4	4.30×10^3	1.99×10^3
LB-RLT	1.96×10^4	1.45×10^4	4.65×10^3	2.87×10^3

Comparing the success rate metric with 4KB as the object size, only LB-RLT performed all the read requests with 100% success rate as shown by the continuous line at the Figure 3(a). A similar behavior happened with 16KB as the object size because LB-RLT lasted longer in 80% of success rate as shown by the continuous line in the Figure 4(a). LB-RCT and LB-R were totally worse than the others. Our environment was deployed to not support purposely a high number of workers with 16KB as object size (above cloud capacity scenario).

Comparing the throughput metric with 4KB as the object size, LB-RW could not maintain the highest throughput after 200 workers. LB-RL had similar results with LB-RW. From the middle to the end, LB-RLT bypassed LB-RW as shown by the continuous line in the Figure 3(b). In the comparison with 16KB as the object size, LB-RW could not maintain again the best throughput after 200 workers as shown in the Figure 4(b). Considering the lower success rates during the rest of the experiment, LB-RLT maintained a little bit better throughput than all the other approaches. LB-RL had a bad throughput similar to LB-RCT and LB-R because LB-RL approach could not give preference to the best nodes as discussed at Section 4.1.

Comparing the response time metric with 4KB as the object size, LB-RLT performed the lowest response time during all experiment until in the end when the success rate 4(a) of the other approaches began to fall as shown at the Figure 3(c). The comparison with 16KB as object size, LB-RLT had the best response time until near after success rate began to fall from 100% 4(a) as shown in the Figure 4(c). The response time metric for LB-RLT increased in the end because LB-RLT processed more requests than the other approaches, once the others were losing requests. LB-RL again did not perform well for the same reason described at the throughput comparison. The response time metric is available and is statistically analyzed at the Section 4.3

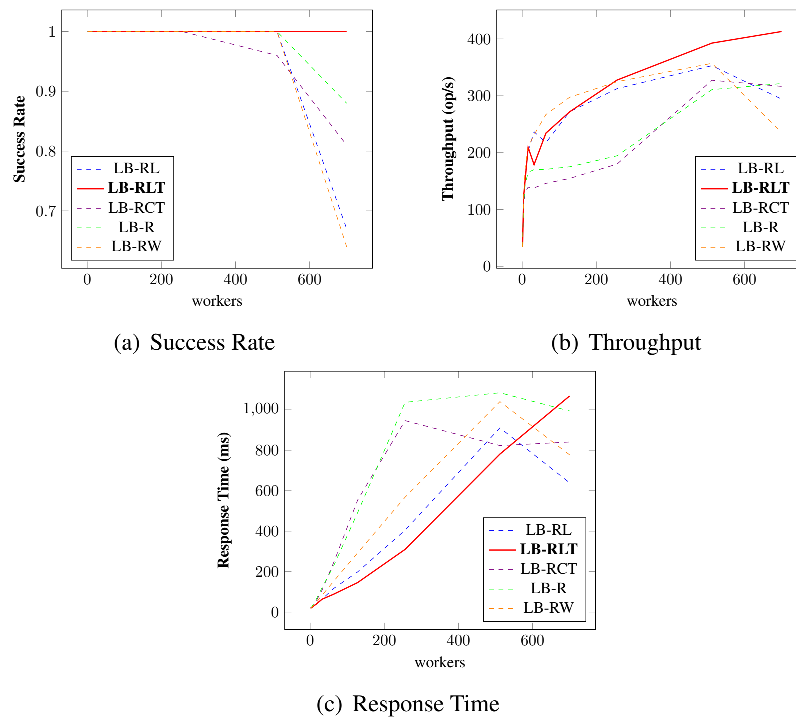

Figure 3. Load Balance Approaches Comparison For 4KB object sizes

4.3. Response Time Analysis

For 4KB as object size, at Table 8, LB-RLT performed on average a response time 11.58% better than any other approach. In the best case, the response time was 36.11% better and in the worst case only 1,93% worse. The experiment with 256, 512 and 700 workers were not considered because not all the approaches have completed 100% of successful operations.

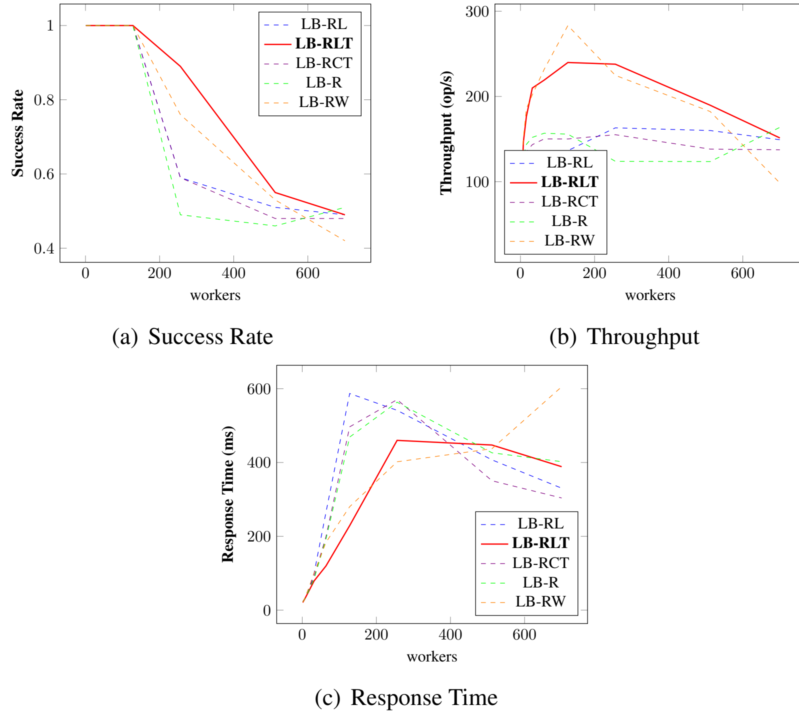

Figure 4. Load Balance Approaches Comparison For 16KB object sizes

Table 8. Response time analyses for 4KB object size

Workers	**LB-RTL**	LB-RL	LB-RCT	LB-R	LB-RW	Best others	(Best others/LB-RLT) - 1
1	20.53	20.97	20.78	20.72	20.88	20.72	.93%
2	20.33	20.08	21	20.09	20.22	20.08	-1.23%
4	21.71	21.29	23.01	25.38	21.5	21.29	-1.93%
8	28.62	32.67	32.45	36.48	31.08	31.08	8.6%
16	38.74	40.36	53.99	61.74	47.06	40.36	4.18%
32	63.46	64.04	108.48	120.28	82.83	64.04	.91%
64	88.81	117.44	247.5	231.19	153.72	117.44	32.24%
128	145.79	198.43	553.5	491.46	292.49	198.43	36.11%
256	309.73	404.59	946.49	1036.82	567.81	404.59	30.63%
512	781.11	910.85	823.13	1084.23	1040.42	823.13	5.38%
						mean	11,58%
						best	36,11%
						worse	-1,93%

For 16KB as object size, at Table 9, LB-RLT performed on average a response time 10.89% better than any other approach. In the best case, the response time was 53.46% better and in the worst case 3.57% worse. The experiment with 256, 512 and 700 workers were not considered because none approach has completed 100% of successful operations.

Table 9. Response time analyses for 16KB object size

Workers	**LB-RLT**	LB-RL	LB-RCT	LB-R	LB-RW	Best others	(Best others/LB-RLT) - 1
1	22.19	23.1	22.68	22.78	22.96	22.68	2.21%
2	21.95	23.93	23.52	23.46	22.93	22.93	4.46%
4	25.47	26.48	25.21	25.01	24.56	24.56	-3.57%
8	32.6	32.36	32.82	32.9	35.82	32.36	-.74%
16	47.99	49.41	50.52	48.92	55.84	48.92	1.94%
32	79.16	100.09	90.82	84.67	100.54	84.67	6.96%
64	119.63	264.74	198.83	192.32	183.59	183.59	53.46%
128	229	586.96	496.63	468.89	280.23	280.23	22.37%
						mean	10.89%
						best	53.46%
						worse	-3.57%

5. Related Work

LB-R and LB-RW can be classified as static approaches according to [Khiyaita et al. 2012] and [Katyal and Mishra 2013]. LB-R approach is not able to adapt to any change during runtime, thus making it a bad choice for heterogeneous storage nodes environment. LB-RW approach allows finer control according to prior knowledge of nodes' capacity. It basically sets a weight for each node. In case of same weights for storage nodes, equally sorting by round-robin is used to spread the workload. A drawback for LB-RW is mainly due to its static behavior for high loads, once the approach does not collect any metric of the system current performance.

LB-RCT is mentioned as Central Load Balancing Decision Model (CLBDM) by [Katyal and Mishra 2013] [Radojević and Žagar 2011] to be used in a static environment, but it uses a runtime metric to load balance the nodes. Basically, LB-RCT measures the duration required to establish a TCP connection between a proxy node and a storage node in order to give preference to the storage nodes with lower connection latency. It also implements measure expiration protection and equally sorts nodes in case of same latency environment. A drawback for this approach is due to the fact that latency of connection does not consider other system aspects such as disk performance.

LB-RL approach is cited by [Katyal and Mishra 2013] as a throughput issue to be addressed for static, dynamic, centralized and distributed types of algorithms. We analyzed our LB-RL implementation on OpenStack-Swift as a dynamic load balance approach by using the lastReadDataChunkThroughput measured between the proxy node and the storage nodes with measure expiration protection and equally-sorting nodes in case of same throughput. A drawback for this approach is due to the fact that, according to our experiments, using only the lastReadDataChunkThroughput is not enough because the bad node continued to be selected for reading more than it was needed.

Our approach aimed to bypass limitations from the previous approaches by using the lastReadDataChunkThroughput that considers a last storage node performance and the

totalReadDataChunks as a long-term performance of storage node until present moment. By this means, in runtime, LB-RLT is able to give preference to the best nodes without neglecting the bad nodes when needed.

6. Conclusion and Future Work

This work presented LB-RLT, an approach for load balancing replica selection in a cloud object storage heterogeneous system. LB-RLT collects two metrics: lastReadDataChunk-Throughput and totalReadDataChunks, and combines both metrics in order to adapt not only for storage nodes heterogeneous environments but also for when load requirement changes dynamically. We evaluated the LB-RLT approach by considering different performance metrics, obtaining the lowest response time, the highest throughput rate and lasting success rate, without any prior performance profiling. Considering our results, LB-RLT improves the overall performance of cloud object storage systems.

As future work, we intend to conduct new experiments with different proportion between read and write requests as well as more scenarios for workloads. Other important issues to be addressed are related to experiments with a greater number of nodes and the analysis of expiration time when there is more delay between the coming requests.

Acknowledgments

This work is partially supported by Hitachi Data Systems (HDS) and Laboratório de Sistemas e Banco de Dados (LSBD) - UFC.

References

Allcock, B., Bester, J., Bresnahan, J., Chervenak, A. L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., and Tuecke, S. (2002). Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771.

Brinkmann, A., Salzwedel, K., and Scheideler, C. (2000). Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 119–128. ACM.

Foong, A. P., Huff, T. R., Hum, H. H., Patwardhan, J. P., and Regnier, G. J. (2003). Tcp performance re-visited. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 70–79. IEEE.

Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., and Stoica, I. (2004). Load balancing in dynamic structured p2p systems. In *INFOCOM 2004*, volume 4, pages 2253–2262. IEEE.

Honicky, R. and Miller, E. L. (2004). Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 96. IEEE.

Katyal, M. and Mishra, A. (2013). A comparative study of load balancing algorithms in cloud computing environment. *International Journal of Distributed and Cloud Computing*, 1(2).

Khiyaita, A., Zbakh, M., El Bakkali, H., and El Kettani, D. (2012). Load balancing cloud computing: State of art. In *Network Security and Systems (JNS2), 2012 National Days of*, pages 106–109.

Lumb, C. R., Ganger, G. R., and Golding, R. (2003). D-sptf: Decentralized request distribution in brick-based storage (cmu-cs-03-202).

Malpass, I. and Malpass, I. (2011). Measure anything, measure everything.

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

Radojević, B. and Žagar, M. (2011). Analysis of issues with load balancing algorithms in hosted (cloud) environments. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 416–420. IEEE.

Rajalakshmi, A., Vijayakumar, D., and Srinivasagan, K. (2014). An improved dynamic data replica selection and placement in cloud. In *Recent Trends in Information Technology (ICRTIT), 2014 International Conference on*, pages 1–6. IEEE.

Swift (2016). *OpenStack*. `http://docs.openstack.org/developer/swift/`.

Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. (2006). Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM.

Xie, Q., Dong, X., Lu, Y., and Srikant, R. (2015). Power of d choices for large-scale bin packing: a loss model. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 321–334. ACM.

Ying, L., Srikant, R., and Kang, X. (2015). The power of slightly more than one sample in randomized load balancing. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*, pages 1131–1139. IEEE.

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