Received: 1 May 2019 | Revised: 1 January 2020 | Accepted: 3 January 2020
DOI: 10.1002/cpe.5675

WILEY

RESEARCH ARTICLE

An adaptive replica placement approach for distributed key-value stores

José S. Costa Filho | Denis M. Cavalcante | Leonardo O. Moreira | Javam C. Machado

Departamento de Computação, Universidade Federal do Ceará, Fortaleza, Ceará, Brazil

Correspondence
José S. Costa Filho, Departamento de Computaçã, Universidade Federal do Ceará, Fortaleza, Ceará 60020-181, Brazil
Email: serafim.costa@lsbd.ufc.br

Summary

The use of distributed key-value stores (KVS) has experienced fast adoption by various applications in recent years due to key advantages such as hypertext transfer protocol-based RESTful application programming interface, high availability and elasticity. Due to great scalability characteristics, KVS systems commonly use consistent hashing as data placement mechanism. Although KVS systems offer many advantages, they were not designed to dynamically adapt to changing workloads which often include data access skew. Furthermore, the underlying physical storage nodes may be heterogeneous and do not expose their performance capabilities to higher level data placement layers. In this paper, we address those issues and propose an essential step toward a dynamic autonomous solution by leveraging deep reinforcement learning. We design a self-learning approach that incrementally changes the data placement, improving the load balancing. Our approach is dynamic in the sense that is capable of avoiding hot spots, that is, overloaded storage nodes when facing different workloads. Also, we design our solution to be pluggable. It assumes no previous knowledge of the storage nodes capabilities, thus different KVS deployments may make use of it. Our experiments show that our method performs well on changing workloads including data access skew aspects. We demonstrate the effectiveness of our approach through experiments in a distributed KVS deployment.

KEYWORDS

consistent hashing, deep reinforcement learning, key-value stores, load balancing, replica placement

1 | INTRODUCTION

Distributed key-value stores (KVS) are a well-established approach for cloud data-intensive applications[1] mainly because they are capable of successfully managing huge data traffic driven by the explosive growth of different applications such as social networks, e-commerce, and enterprise. In this work, the focus is on a particular type of KVS, also known as Object Store, which can store and serve any type of data (eg, photo, image, and video).[2] Object Store such as Dynamo[3] and OpenStack-Swift[4] have become widely accepted due to its scalability, high capacity, cost-effective storage and reliable REST programming interface. These systems take advantage of peer-to-peer architecture[5] and replication techniques in order to guarantee high availability and scalability. The data placement of KVS systems are commonly based on a distributed hash table (DHT) and consistent hashing (CHT)[6] with virtual nodes. Consistent hashing is a type of hashing that minimizes the amount of data that needs to move when adding or removing storage nodes. Using only the hash of the id of the data one can determine exactly where that data should be. This mapping of hashes to locations is usually known as "ring." While this strategy provides item-balancing guarantees, it may be not very efficient in balancing the actual workload of the system for the following reasons.

First, it assumes uniformity for data access, all data items are equally popular, thus workload is equally split among storage nodes.[3] Data access skew is mainly a consequence of popular data, also referred to as hot data, due to high request frequency. Popular data is one of the key reasons for high data access latency and/or data unavailability in cloud storage systems.[7] Second, it assumes that storage nodes are homogeneous, having the same performance capabilities (ie, same capacity of serving demand) which is not always the case. Although a cloud infrastructure can start with near-homogeneous resources, it will likely grow more heterogeneous over time due to upgrades and replacement.[8] With heterogeneous machines, the system designers have the option of quickly adding newer hardware that is more powerful than the existing hardware.[9] When this is done, the assignment of equal load among nodes results in suboptimal performance.[10] The main objective of load-balancing methods is to speed up the execution of applications on resources whose workload varies at run time in unpredictable way.[11] It is pivotal that KVS systems balance the load, optimizing the use of heterogeneous resources while also taking into consideration data access skew in different workloads in order to improve the quality of service and resource utilization.

In this paper, we address those two issues and propose a load-balancing strategy in KVS systems based on CHT that considers dynamically changing workloads with data access skew and storage node heterogeneity. In real-world applications, workloads changes dynamically making it very difficult to predict which data will become hotspot data and which will not[12].[13] Thus it is necessary to adjust the load dynamically while the application is being executed.[14] We employ a smart approach to dynamically migrate data replicas that is robust to workload changes which occur due to natural variations in data popularity across different periods of time. It detects data hotspots and adjusts the replica placement among storage nodes in order to avoid loss in performance.

The authors of Reference 15, found that the ratio of *Get*/*Put* requests in large distributed KVS systems is 30:1. Being *Get* requests the majority, our strategy focus on balancing those type of requests. To that end, we define a KVS system's load as a function of the number of *Get* requests per time frame. More specifically, we monitor the number of *Get* requests for each data replica in the system per time frame. By changing the replica placement mapping scheme, we can trigger data replica migrations, modifying the location of data replicas in the storage nodes, thus changing the amount of *Get* requests each storage node receives. Since the storage nodes may have different performance capacities, each storage node can serve a different number of *Get* requests without becoming overloaded. A common way to address nodes heterogeneity is to partition the load among nodes according to their performance capacity.[9] As shown in Reference 16, every system has its maximum capacity: once it is saturated, more clients will have to compete with each other for resources, resulting in unacceptable response times. Our approach monitors the system's load and the latency of each storage node. The latency is an important measurement to know when storage nodes become overloaded. Our strategy aims to aid system administrators by reducing system load imbalance autonomously to improve the overall resource utilization.

Whenever deciding to migrate data replicas in the system, there are three essential questions to answer. Which storage nodes are overloaded? Which replicas on overloaded storage nodes should be migrated? Which storage node is best suitable destination to allocate specific data replicas? The latter question is particularly difficult to answer since it is not known a priori the performance impact that a particular replica might cause in each storage node. For instance, a storage node equipped with a solid state drive (SSD) may be less affected by the additional load that replica will draw than a storage node equipped with a hard disk drive (HDD). It is crucial to alleviate hotspots without overloading other nodes.

To tackle this issue, we propose the use of reinforcement learning (RL).[17] RL can model complex systems and decision-making policies. Also, it allows to train for objectives that are difficult to optimize directly because they lack precise models if there exist reward signals that correlate with the objective.[18] By continuing to learn, RL offers robustness. An RL agent can optimize for a specific workload and still performs well under varying conditions. However, there are challenges in using it to obtain practical success in real-world applications. First, RL can suffer from poor scalability in large state spaces, particularly in its simplest and best-understood form in which a lookup table is used to store a separate value for every possible state-action pair.[19] Since the size of such a table increases exponentially with the number of state variables, it can quickly become prohibitively large in many real applications. Fortunately, recent advances have shown success in combining the RL and deep learning techniques enabling RL agent to achieve great results in large state-action spaces.[20] Our approach leverages deep RL technique to calculate the impact that migrating data replicas would cause to each storage node in the system. That mechanism allows us to find the most suitable storage node, that is, the one that its latency is less affected. Our strategy is designed to be pluggable, it assumes no knowledge of the storage nodes heterogeneity. In fact, it is capable of adjusting when hardware upgrades happen to storage nodes in the system or new storage nodes are added.

Figure 1 summarizes the general workflow of our strategy. First, information on data popularity and on each storage node latency (SNL) is queried. That information will be formatted and served as input to our RL agent. Next, the current ring in the KVS system is copied. The agent then takes an action and modifies the ring(ie, modifies the replica placement). The new ring is then deployed in the KVS system, triggering the migration of data replica(s). After the migration process is completed, it is given a timeout so the effects of the migration can take place. Then, the agent queries for new latency measurements to evaluate if the action it took reflected on reducing or not the overall latency and by how much. Even though our approach focus on KVS system, the strategy can be generalized to other systems as well. The major contributions of this paper are identified as the following:

- We propose new machine learning-based approach for replica placement that is adaptable and robust to dynamic workloads. The main idea is to treat the replica placement as a “black-box” mechanism, and try to learn its relation to the system overall latency by using a deep neural network (DNN).

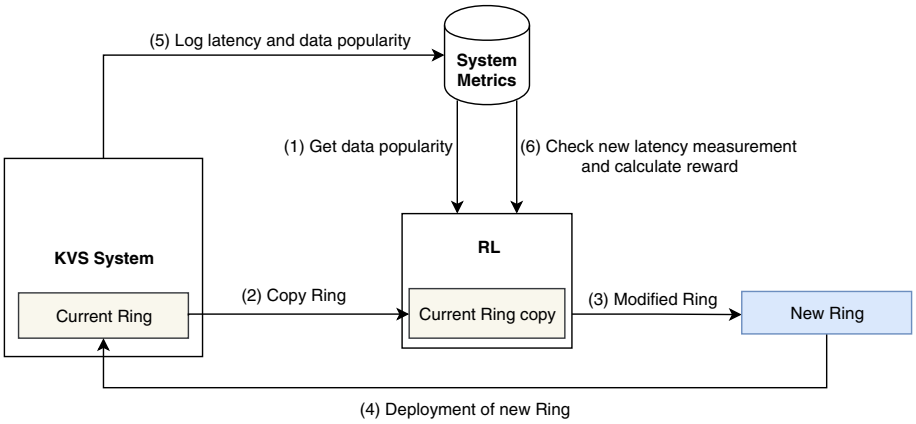


FIGURE 1 Workflow of our strategy deployed in the key-value stores system

- We evaluate the performance of our approach through experiments. We deploy it in a real-world distributed KVS system and analyze the results in terms of system overall latency reduction and resource utilization.

The remainder of this paper is organized as follows. Section 2 introduces the related works comparing them to our work. Section 3 provides important background knowledge about RL. In Section 4, we detail our approach by defining the system architecture, specify the RL formulation and present our replica placement algorithm. Then, in Section 5, we present and discuss the experiments results comparing our strategy to the baselines. Section 6 concludes the paper and indicates the future work.

2 | RELATED WORK

The problem of replica placement concerning load imbalance in KVS systems is not new. Several papers have already described different solutions and have shown the effectiveness of their approach. In this section, we briefly describe the most similar to our strategy while we highlight their how they differ from our work.

Wang et al[12] proposed a framework of workload balancing and resource management for OpenStack-Swift. In their framework, they implemented workload monitoring and analysis algorithms to detect overloaded and underloaded nodes in the cluster. The work focuses on the workload regulation of virtual nodes and physical nodes. Through dynamic changing the mapping of virtual machines (VM) to physical machines, the workload is balanced in physical machines. Even though the proposed framework improved resource utilization and response time, there are a few limitations to their work that are not present in our work. For instance, their framework is only applied to OpenStack-Swift. Our approach is more general since we focus on a common aspect of KVS systems that is the replica placement problem. Also, another limitation of their framework is that it only works when the KVS system is deployed in an architecture with virtualization layer. Our approach is more general purpose as it does not depend on a particular infrastructure or how the KVS system is deployed.

In another previous work, Cavalcante et al[1] improved the load-balancing for workloads with data access skew. They defined the replica placement as a multi-objective optimization that takes into account data access distribution, data redundancy and data movement. PopRing was presented, an approach based on a genetic algorithm to find a new replica placement scheme (RPS). The work presented promising results on distributing the load among storage nodes. However, unlike our work, their algorithm is designed for homogeneous environments only, thus limiting load-balancing opportunities.

Makris et al[7,21] reported that response time of *Get* requests quickly degrade in the presence of workloads with power-law distributions for data access. The authors defined their objective as the minimization of the average response time of the system under workloads with data access skew. Their approach employs data migration when a decreased performance is detected. Thresholds are given to nodes and data is migrated to the nodes following a first fit decreasing algorithm. The process of obtaining the thresholds is very important, but it is clear how it is done. Their algorithm was able to reduce response time under workloads with skewed data access. However, unlike our work, the authors did not evaluate their algorithm in heterogeneous environments.

Long et al[22] developed a multiobjective strategy to find the number of replicas and the location of replicas based on a mathematical model with five objectives. Through simulation, their algorithm helped to improve load balancing and latency but in contrast to our work, it does not perform data migration to deal with the dynamic changes in file access characteristics. Also, another difference from our work is that they do not consider DHT/CHT techniques in their proposed solution, thus whether their solution performs well on KVS based on DHT/CHT is unknown.

Paula et al[23] proposed a nonintrusive approach to load balancing in cloud object storage considering storage devices with different performances. The approach described in the paper classifies storage devices based on their input/output operations per second capacities.

The work adjusts the weights of disks taking into consideration their performance so that disks with higher weights store more data. That strategy achieves improvements in response time and throughput, but unlike our work, it does not consider data access skew which is known to degrade performance.

Naiouf et al[14] discussed dynamic and static balancing on heterogeneous cluster architectures. Three forms of load distribution (Direct Static, Predictive Static, and Dynamic by Demand) were studied and analyzed. The evaluation showed that distributing workload taking into account the processing power improves load balancing and algorithms that distribute workload dynamically can assign work in a more balancing way among the machines. Their work differ from ours in two important ways. Their algorithms are designed for a different context and whether they performs effectively in a KVS system based on DHT/CHT is unknown. Moreover, their algorithms require the machine's processing power to be known a priori. Our work, abstracts the machine performance capacity and learns to better distribute workload autonomously.

Mseddi et al[24] argue that replica placement systems usually need to migrate and create a large number of data replicas over time. The authors proposed a replica migration scheme called CRANE. They focused their work on minimizing the time needed to copy the data to the new replica location by avoiding network congestion and ensuring minimal replica unavailability. Unlike our work, their work does not tackle the replication placement problem in KVS systems. In fact, CRANE is supposed to complement replica placement algorithms.

In summary, our work is more complete and robust and differs from the related works because in its design it takes into account all the following features:

- tackles the Replica Placement problem;
- employs data migration to support dynamic workloads;
- consider heterogeneity aspects;
- consider data access skews;
- evaluation on a real KVS deployment;

3 | RL BASICS

In this section, we briefly discuss about RL[25] to identify its characteristics and capabilities. Our approach makes use of this technique and its algorithms in order to solve the problem of replica placement. RL represents a class of machine learning algorithms in which an agent learns what action to take through the reward that it receives after interacting with the environment. Figure 2 shows a general scenario where at each time step t, the *agent* observes the *environment* state, a set of parameters or features that describe the environment, s_t and takes an *action* a_t. Then, the environment transitions to state s_{t+1} and sends the agent a reward r_t. A reward function defines the goal of the RL problem. The policy is responsible to guide the agent's actions at a given time. It is a mapping from perceived states of the environment to actions to be taken when in those states. The policy is the core of a RL agent in the sense that it alone is sufficient to determine behavior.[17]

RL algorithms faces the explore vs exploit dilemma, that is, the agent must decide if it should follow the action plan or try different, unexplored paths. So if the agent focus on exploitation it will not be able to learn anything valuable. The other case is, if the agent focus on exploration it might not be able to find the optimal sequence of actions which will provide the better utility. The agent only controls its own actions and has not a priori knowledge regarding the consequences of each action, that is, which state the environment would transition to or what the reward may be. This way, the agent must explore the environment so it can gain knowledge and learn the consequences from actions taken while it observes the environment. Through multiple iterations the agent should learn which of the actions led to the specific compensation.

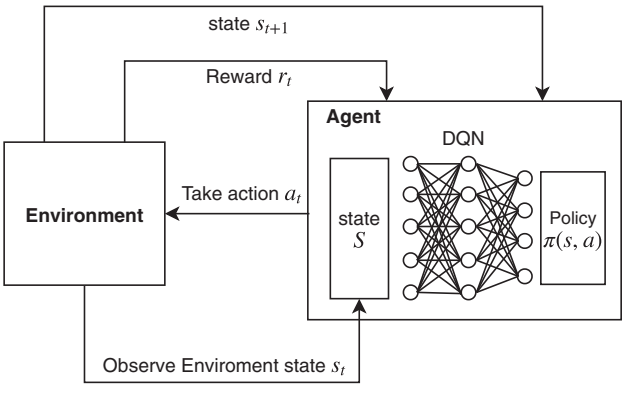


FIGURE 2 Reinforcement Learning using a deep Q-network to model the policy.

3.1 | Q-learning

Q-learning is a model-free RL algorithm. For any finite Markov decision process, Q-learning finds a policy that is optimal, that is, it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. In each time instant t, the agent will be at a state s_t and decide to make an action $a_t = \pi(s_t)$ according to the policy π. Then the agent takes an action and receives a reward based on a function r_t. Q-learning uses a value function $Q(s, a)$ which is the expected value of the sum of future rewards by following policy π.

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[R_t]. \tag{1}$$

The policy π guides the agent what actions to take at any given states. R_t represents the sum of discounted future rewards.

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-t} r_n. \tag{2}$$

The discount factor γ determines the importance of future rewards. A factor of 0 will make the agent short-sighted, $Q^{\pi}(s, a)$ only takes current rewards into consideration, while a factor approaching 1, $Q^{\pi}(s, a)$ will be equal to the sum of the rewards, making the agent strive for a long-term high reward. Once there is a good approximation of the Q function for all pairs state/action (s_t, a_t), it is possible to obtain the optimal policy π^* through the following expression.

$$\pi^*(s) = arg \max_a Q(s, a). \tag{3}$$

With the data on states transition, actions, rewards in the format $< s, a, r, s' >$, it is possible to iteratively approximate the Q function through temporal difference learning. The so-called *Bellman equation* is used to relate the Q functions of consecutive time steps.

$$Q^{\pi^*}(s, a) = r + \gamma \max_{a'} Q(s', a'). \tag{4}$$

3.2 | Deep Q-learning

Q-learning is known to lose performance when facing large state-action space since the amount of time required to explore each state would be unrealistic. To overcome that, it is common to use function approximators.[26] Deep Q-network (DQN) is capable of successfully learning directly from high-dimensional inputs.[20] Deep Q-learning uses a DNN[27] to estimate the Q-values for each state-action pair in a given environment, and in turn, the network will approximate the optimal Q-function. The DQN input is the state of an specified environment and for each given state input, the network outputs estimated Q-values for each action that can be taken from that state. The loss from the network is calculated by comparing the outputted Q-values $Q(s, a)$ to the target Q-values $Q^{\pi^*}(s, a)$ from the right-hand side of the Bellman equation. The objective is, at each time step, to minimize the mean squared error as shown in Equation (5).

$$Loss = \frac{1}{2}[Q^{\pi^*}(s, a) - Q(s, a)]^2. \tag{5}$$

Once the loss has been calculated, the weights within the network are updated using stochastic gradient descent and backpropagation. This process is done repeatably for each state in the environment until the loss has been sufficiently minimized and an approximate optimal Q-function is obtained. As shown in Reference 28 approximation of Q-values using nonlinear functions is not very stable. To avoid forgetting about previous experiences and break the similarity of subsequent training samples, data in the format $< s, a, r, s' >$ is stored in a so-called replay memory. When training the neural network, random minibatches from the replay memory are used instead of the most recent transition.

4 | KVS REPLICATION APPROACH

A cloud object storage is composed of a set of virtual nodes and a set of storage nodes, as shown in Figure 3. It is structured in two distinct mapping layers, the upper layer that exposes stored objects on the interface and maps them to virtual nodes, and the replica placement that allocates virtual nodes into storage nodes. A storage node is a server that has at least one nonvolatile memory device such as a HDD or SSD. In our approach, the partitioning of virtual nodes to storage nodes is based on CHT, an important mechanism that dynamically partitions the entire data over a set of storage nodes.

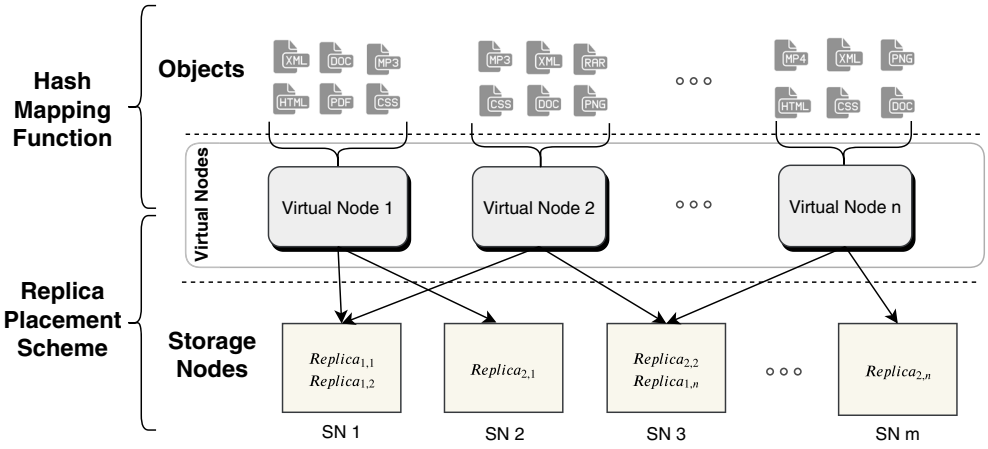


FIGURE 3 Objects, virtual nodes and storage nodes mappings in a traditional distributed key-value store

TABLE 1 Example of replica placement scheme

	v_0	v_1	…	V
s_0	1	1	…	
s_1	1	0	…	
s_2	1	1	…	
s_3	0	1	…	
…	…	…	…	
S				

TABLE 2 Modified replica placement scheme

	v_0	v_1	…	V
s_0	1	0	…	
s_1	1	1	…	
s_2	1	1	…	
s_3	0	1	…	
…	…	…	…	
S				

Our virtual nodes have the same concept of virtual nodes in Dynamo and partitions in OpenStack-Swift which is an abstract layer for managing all system data into smaller parts, that is, a set of objects, as it is shown in the hash mapping layer of Figure 3. Each data object on the system is mapped to a virtual node through the consistent hash function mapping. A hash function applies the identification of a data object to calculate the modulo operation using the total number of virtual nodes, defining then which virtual node the object belongs to. The number of objects in every virtual node is balanced due to the hash function of the hash mapping layer that outputs hashed values uniformly distributed. The hash function responsible for mapping data objects to the virtual node is set up only once and remains the same during the entire system operation. At deployment, before system start-up, the system administrator sets the total number of virtual nodes to a large value and never changes it; otherwise, it would break the property of the CHT technique by creating the side-effect of huge data movements.

A virtual node can be replicated multiple times on different storage nodes, for example, Virtual Node 2 is replicated to Storage Nodes 1 and 3 in Figure 3. A RPS is responsible for defining the mapping of virtual node replicas to storage nodes. It specifies the replication factor, which happens to be 2 in our example meaning that each virtual node has two copies in our storage system and the placement of every virtual node replica as shown in Figure 3. By modifying the RPS, the storage system can dynamically manage data through operations of replica creation, migration, and deletion. In our particular case, we modify this hash function to redefine this placement scheme in order to achieve load balancing of *Get* operations.

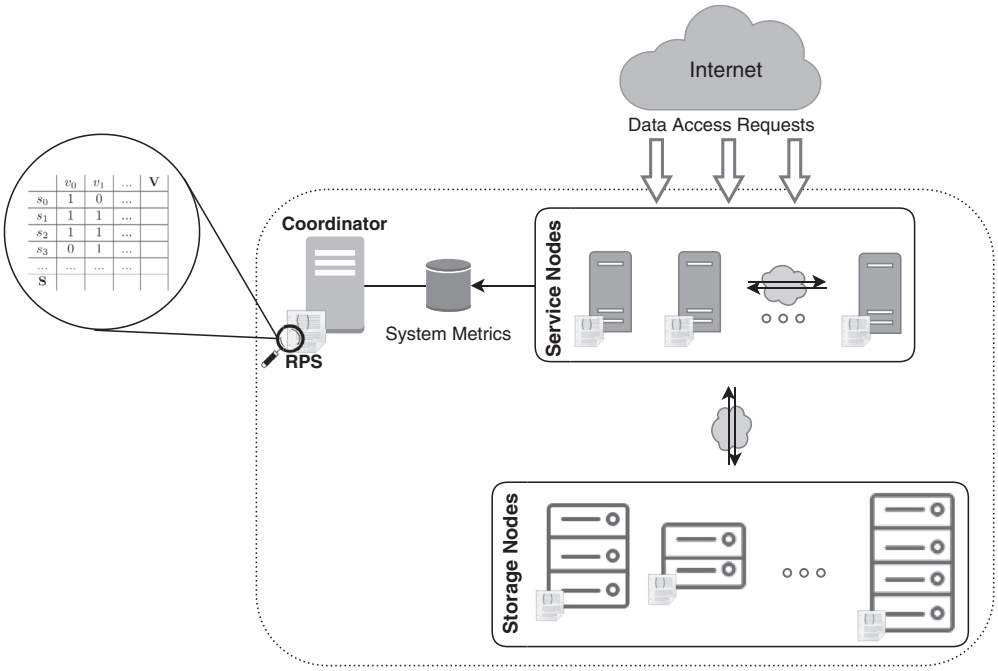


FIGURE 4 Key-value stores system architecture

Each storage node has a task, different from the process to serve data access requests, to asynchronously meet a new data placement. This task considers the new RPS to synchronize all replica units, ensuring consistency between replicas. Every storage node has a copy of the RPS, thus making possible for each storage node to know exactly which replicas it manages.

The RPS is a binary matrix of $\text{RPS}_{s,v}$ cells in which a $\text{RPS}_{s,v}$ cell $\in \{0, 1\}$. A RPS matrix has size $|S||V|$ in which a row represents a storage node $s \in S$ and a column represents a virtual node $v \in V$. A virtual node may be replicated $|R|$ times and each replica $r \in R$ of a virtual node $v \in V$ is replicated into a storage node $s \in S$ if the $\text{RPS}_{s,v}$ cell value is 1, otherwise is 0. The total number of virtual nodes $|V|$ and the total number of replicas of the virtual nodes $|R|$ are both set only once by the system administrator prior to system initialization while the number of storage nodes $|S|$ may be changed after system initialization. Also for this work, $|R|$ is the same for every virtual node $v \in V$, that is, all virtual nodes have the same replication factor. On the other hand, our RPS performs data migration by making incremental changes to the RPS already in-use by a KVS system. For instance, if our strategy decides that the replica r_1 of the virtual node v_1 that is within the storage node s_0 as shown in Table 1, should be migrated from storage node s_0 to s_1, then the RPS is modified as in Table 2. $RPS_{0,1}$ is set to 0 and $RPS_{1,1}$ is set to 1.

The amount of *Get* requests targeted to each virtual node $v \in V$ according to the hash function is defined by $\text{VN}_v get$. This works with the consideration that $\text{VN}_v get$ is spread uniformly among each replica of a virtual node $v \in V$ with replication factor rf. This way, it is possible to calculate the amount of *Get* requests submitted to each $s \in S$ according to the Equation (6).

$$\text{Storage Node Workload (SNW)} = \sum_{s=0}^{S-1} \sum_{v=0}^{V-1} (\text{RPS}_{s,v}) \left(\frac{\text{VN}_v get}{rf} \right). \tag{6}$$

4.1 | System architecture

An overview of our system architecture is shown in Figure 4. It is composed of service nodes, storage nodes, and a coordinator node. The service nodes handle data access requests from clients. They make use of the RPS (eg, Table 1) to locate data and redirect requests to appropriate storage nodes.

The service nodes are responsible for accepting hypertext transfer protocol requests from users. Requests are read and write operations over data objects by supporting *Put* requests for uploading objects and *Get* requests for accessing data objects. The system supports and serves any unstructured data (eg, text files, photos, videos, compressed achieves) and is capable of handling any object size in a write once read many (WORM) manner. The service nodes use a shuffle algorithm for replica selection which distributes *Get* requests uniformly among virtual node replicas.

The coordinator node acts as a centralized controller and is responsible for monitoring the total number of *Get* requests per virtual node served by the service nodes. Also, it maintains a copy of the RPS in-use by the other nodes in the system. The response time of every operation *Get* on an object is logged, allowing the coordinator to monitor the average response time of a virtual node, that is, the average time required to *Get* an object in that virtual node. The Coordinator has an RL module, that is responsible for training and updating a model that will take data replica migration

decisions. The main task of the coordinator node is to use our replica placement strategy to periodically compute and incrementally apply a new RPS in order to achieve load balancing, improve resource utilization and overall system performance (ie, low latency).

4.2 | RL formulation

The RL formulation is the task of defining all the elements and transitions illustrated in Figure 2. But before that, we have to define a few requirements that will guide and justify our formulation. One regarding workload characteristics and another relates to resource utilization. The first requirement is that it has to be robust to different workload patterns. Once the model has been built, it must be able to perform well under unseen workload patterns and different data popularity aspects. It is hard for system administrators to predict the performance of heterogeneous computational resources especially in a resource shared environment and adjust those prediction parameters once storage nodes' hardware has been changed. The second requirement is that our model has to be able adjust itself to the current hardware setup in order to make the best use of those resources. Also, it must be able to dynamically adapt in case of changes in that regard, building up on previous acquired knowledge.

4.2.1 | State space

Let n be the number of storage nodes in the computer cluster. The states WS of the world are defined as follows:

$$\text{WS} = \langle \text{SNW}_1, \text{SNL}_1, \text{SNW}_2, \text{SNL}_2, \dots, \text{SNW}_n, \text{SNL}n, \text{MigrationLoad} \rangle \tag{7}$$

where

- $\text{SNW}_n \in \mathbb{N}$. It is calculated using Equation (6).
- $\text{SNL}_n \in \mathbb{R}$. It is the SNL, that is, the average time it takes storage node n to serve a data request.
- MigrationLoad $\in \mathbb{N}$. It is an integer number that represents the sum of *Get* requests targeted to R, where R is a set of replicas to be migrated.

Our KVS approach captures the number of *Get* requests issued to each replica in the system, and records SNW_n and MigrationLoad shown in Equation (7). We measure the performance by monitoring the latency. Therefore, that important information is also represented in our state. As we can see, the information that goes into our state are generic and easy to be monitored in different KVS systems. The state representation passes through the DQN, which outputs a vector of Q-values for each possible action in the given state. We take the biggest Q-value of this vector to find the action with highest quality, that is, the best one. To ensure adequate exploration of the state space we use an $\epsilon - greedy$ strategy that chooses the best action with probability $1 - \epsilon$ and selects a random action with probability ϵ.

4.2.2 | Action space

The set of actions A is represented by the index $j \in \mathbb{N}$ of the storage node to which the current MigrationLoad is going to be assigned. If the cluster has for example 10 storage nodes, there will be 10 output nodes in the DNN representing the quality of migrating MigrationLoad to each of the 10 storage nodes. The agent may also choose that a specific MigrationLoad does not need to be migrated, that is, the best place for that MigrationLoadis the node that is currently allocating it.

4.2.3 | Rewards

The reward function encourages the agents to pursue the following goal: lower the total system latency and distribute the load according to each storage node performance characteristics. The total system latency is calculated as stated in Equation (8), where N is the number of storage nodes in the system.

$$\text{Total system latency (TSL)} = \sum_{n=1}^{N} \left(\frac{\text{SNL}_n}{N} \right). \tag{8}$$

After a performed action a_t at $step_t$, the agent will receive a reward calculated as following:

$$\text{Reward} = \text{TSL}_{\text{before}} - \text{TSL}_{\text{after}}. \tag{9}$$

A step is the process of migrating data replicas from a storage node to other. The total system latency is calculated immediately before action a_t, that is, before the migration in time step t and calculated again after the action is executed (replicas migration completed). If it improved or worsen the total system latency the agent receives feedback accordingly, the agent will receive rewards that precisely measure how positive or negative that action is at a specific moment. In other words, how good or bad was to migrate those replicas with certain data access characteristics to a specific storage node.

Algorithm 1. Replica placement

```
Input : MS                                                    // Number of migration steps
 1: Initialize replay memory;
 2: Initialize action-value function Q with random weights;
 3: Initialize state ws;                                      // using Equation 7
 4: while !terminated do
 5:   for i=0 to MS do
 6:     Collect data popularity;
 7:     Collect each SNL;
 8:     Copy current ring;
 9:     Select storage node with highest latency;
10:     Calculate system total latency;                      // using Equation 8
11:     Select replicas to migrate;
12:     Select a storage node(destination) following e-greedy policy;
13:     Deploy new ring in KVS system;
14:     Wait a timeout;
15:     Calculate system total latency again;                // using Equation 8
16:     Calculate the reward;                                // using Equation 9
17:     Train Q-Network with experience replay;
18:     ws = ws';
19:   end for
20: end while
```

The Q-learning algorithm has a function that calculates the quality of taking an action in a particular state. However, if the state-action space is very large which is our case, exploring the entire state-action space is unrealistic. Thus, we have to use a function approximator (deep neural network) to approximate the Q-function. The Algorithm 1 describes our overall strategy step by step. After initialization of deep RL elements, data popularity (number of requests per data item), and storage nodes latency are collected. Then, a copy of the current ring (ie, data placement) in the KVS system is made. In line 8, the most overloaded storage node in the system is selected. Replicas from that storage node will be migrated to other storage note in order to lower its workload (SNW Equation (6)). Among the replicas that the overloaded node manages, replicas with most *Get* requests are selected to be migrated. The number of replicas is may be customized. Next, our RL agent following an e-greedy policy, chooses the destination, that is, which storage node the selected replicas will be migrated to. Then, the placement of the selected replicas are changed by modifying the copy of the current ring and deploying the new one. Once the new ring is synchronized, a timeout is set so the new latency measurements reflect the impact that migration step caused. After that, the reward is then calculated to measure the quality of that action. Finally, our Q-network is updated. The algorithm terminates when the error (Equation 5) stabilizes.

5 | EXPERIMENTAL EVALUATION

5.1 | Environment

To generate the workload we extended cloud object storage Benchmark (COSBench) tool version 0.4.2 rc2[16] so it could generate data access skew. 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 our prototype in a private cloud on Openstack environment. All VMs in our experiments were provided by Openstack Nova and storage devices by Openstack Cinder. We used two 4 vCPUs, 8 GB RAM, and 80 GB storage capacity VMs to host the COSBench controller and 2 COSBench drivers. We configured the proxy node with enough resources so it does not

act as bottleneck. The proxy node services were running on a VM instance with 16 vCPUs, 16 GB RAM, and 160 GB storage capacity. We deployed six VMs instances (storage nodes 1 to 6) with 1 vCPU, 512 MB RAM, and 60 GB storage capacity.

We designed each experiment so in the beginning the system is under stress with unbalanced workload among storage nodes. To make the storage nodes heterogeneous, we specify each storage node performance factor (SNPF) by limiting the max number of read input/output operations per seconds (IOPS). Since we had three SATA HDDs as backend of Openstack Cinder with roughly 100 IOPS each, we configured two Cinder disk volumes on each backend HDD. HDD1 had volume 1 (20 IOPS) and volume 6 (70 IOPS), HDD2 had volume 2 (30 IOPS) and volume 5 (60 IOPS) and HDD3 had volume 3 (40 IOPS) and volume 4 (50 IOPS). The coordinator node had an Intel i5 CPU, 8 GB RAM, and 512 GB HDD.

On every evaluation scenario, we used similar values as in Reference 16 we created 100 containers with 100 objects per container resulting a total of 10 000 objects (64 Kb each) in our experiments. We inserted half of the total number of objects in Swift to simulate an object storage system already in production. We used 12 clients/workers on all workload configurations. That number was found empirically as we wanted to send as much workload as we could and still achieve 100% success rate, that is, all requests were served successfully. We fixed the read/write ratio at 30:1, same value showed in analysis done in Reference 15. As explained in Section 4.1, the service nodes (proxy nodes in Swift) are responsible for accepting and serving data requests to clients. They are integrated with Graphite[29] which is an enterprise-ready monitoring tool widely used in cloud applications. The coordinator node has access to data popularity information and response times of every data request through queries to Graphite. Latency values were computed by averaging all latency values within 300 seconds time windows. Before each experiment execution, all VMs were rebooted and Swift services restarted. Every experiment scenario was run multiple times. The results represent the average. We chose to compare it to two other strategies. The policy to select the overloaded storage node and the replica(s) to be migrated was the same across all of them. The thing that separates them apart is the way they pick the destination node in the migration of the replicas. One baseline strategy to select that destination node is to always pick the storage node with the lowest latency (LL). Data replicas in Openstack Swift are uniformly distributed among storage nodes by default. To simulate that, we define another baseline strategy that will uniformly pick a random storage node (RND).

In our experiments, we define 30 migration steps. Each strategy perform 30 migration steps and in the end we evaluate the results obtained by each strategy. After the COSBench starts, we wait 300 seconds to take the first migration step, so we can collect initial metrics (data popularity and latency). After each migration step is done (ie, data have been migrated and system is synchronized), a timeout of 300 seconds is performed. Empirically, it was noticed that 300 seconds was enough time to safely evaluate the impact of a migration step and calculate a reward. In each step, we migrate 1% of the total replicas in the system, as we noticed that migrating 1% of data replicas had a noticeable impact in the system latency in the scenarios we designed. We evaluate the impact of migration steps by using two main metrics. First, we measure the latency of each storage node. Second, we measure the resource utilization by measuring how distant each storage node is from its performance threshold. To calculate those thresholds, we divide the current workload (ie, sum of all storage node request loads given by Equation (6)) proportionally to storage node performance capacities as it is shown in Equation (10). For instance, imagine we have a two storage node setup, a 50 IOPS node and a 100 IOPS node. Given a total workload of 300 *Get* req/s. The 50 IOPS node should be getting roughly 100 *Get* req/s and the 100 IOPS node 200 *Get* req/s. We want to avoid overloaded/underloaded storage nodes in the cluster and Equation (11) measures that aspect by summing the resource utilization of all nodes.

$$\text{Storage Node Threshold (SNT)} = \sum_{s=0}^{S-1} \text{SNW}_s \left(\frac{\text{SNPF}}{\sum_{s=0}^{S-1} \text{SNPF}_s} \right). \tag{10}$$

$$\text{Sum of distances to threshold (SDT)} = \sum_{s=0}^{S-1} |\text{SNRL}_s - \text{SNT}_s|. \tag{11}$$

We implemented our approach using keras-rl.[30] It implements some state-of-the-art deep RL algorithms in Python and seamlessly integrates with the deep learning library Keras which is itself a high-level neural network application programming interface on top of a deep learning backend like Tensorflow, Theano. We chose to use Tensorflow.[31] The implementation of Google DeepMind's DQN agent is used.[20] The optimizer used for training is Adam[32] with a learning rate of 10^{-4}. The network consisted of three densely connected layers with 18 neurons, where the last layer corresponds to the actions, in our case 6. The activation functions are the rectified linear units. The target model update of the DQN agent is set to 10^{-3}, while the batch size takes the value 32. The discount factor in Q-learning is $\gamma = 0.99$. Reference for all parameters is the publication of the DQN agent algorithm[20] and the open source implementation of the DQN agent in keras-rl.[30] During training, it is expected the RL agent to take poor placement decisions since it is exploring. However, Tesauro et al[19] demonstrated that it is possible to overcome the RL poor decisions in the beginning by employing an already known external policy. Thus, the experiments focus on demonstrating the effectiveness and adaptability of our approach once initial training is done. During the initial training we configured the cosbench to generate a workload with zipf 0.1.

In Figure 5, we can observe the loss function 5 during training. Each iteration performs 30 migration steps. We can see that after 25 iterations the error of our model does not change much. After convergence, we saved the model and used it throughout the experiments. During the experiments, after every migration step, the model can be improved using Algorithm 1. It was allocated enough computational resources to the coordinator node so it can do fast model updates.

FIGURE 5 Loss function progression while training

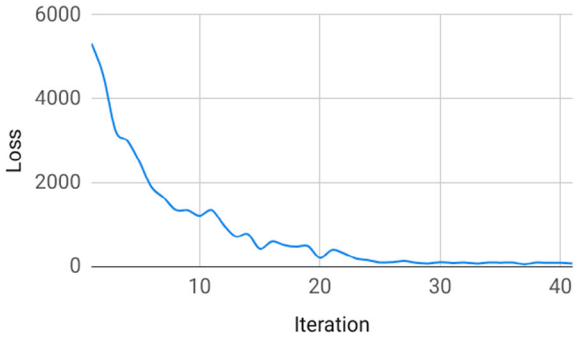
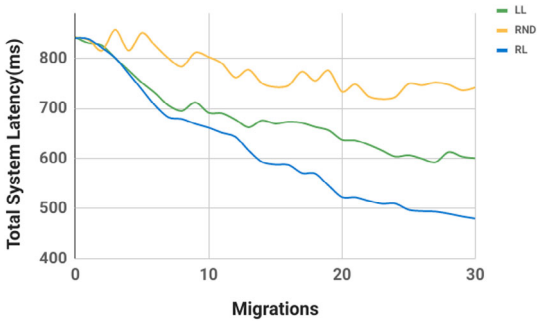


FIGURE 6 System overall latency progression with zipf 0.1



5.2 | Workload changes

In this experiment section, we investigate how our approach performs under unseen workload. To do that, we change the popularity of data from one workload to the other. In this case, we are interested to see how our strategy deals with data access pattern it has never seen. Empirical studies have shown that requests in a P2P system follow a Zipf distribution.[33] We vary the zipf parameter to change data popularity. For this experiment subsection, the system is configured with 1024 virtual nodes. In the first experiment, we set the zipf parameter to 0.1.

In Figure 6, we can see the progression of the total system latency throughout 30 migration steps. In the beginning, every algorithm encounters the system under the same configuration and workload, thus at time step 0, the total system latency is roughly the same for all strategies. As we can see, our strategy RL was the one that reduced the most the total system latency. It was effective in balancing the workload, that is, migrating replicas with high *Get* requests rates from overloaded nodes to less loaded ones. The other strategies also achieved improvements reducing the latency but made some bad decisions along the way.

We can see in Figure 6 that in some steps, the other strategies do not choose right nodes to place data replicas, thus resulting in not reducing the latency and sometimes even increasing it. RND, as it chooses the destination node randomly, it often chooses not the best node.

As expected LL strategy performed better than RND. We notice that, in the beginning, LL and RL usually make similar decisions. LL always chooses the node with the LL, but it does not take into consideration that the storage nodes have different performance capacities. If two storage nodes have similar latency values, it does not necessarily mean they are equally appropriate to allocate specific data replicas, because depending on their performance capacity one might get overloaded while the other might not. Hence, sometimes LL take suboptimal migration actions which causes the latency to increase. Since our strategy learns the impact of placing different replicas (ie, workloads) on different storage nodes, it shows to make the best placement decisions.

As the experiments starts, the load is very unbalanced. For instance, storage node 1 (20 IOPS) is overloaded and the storage node 6 (70 IOPS) is underloaded. In Figure 7, we can see the progression of the resource utilization after 5, 10, 15, 20, 25, and 30 migration steps. The lower, the better. Lower meaning that the load is more well balanced among the storage nodes, that is, the total load of the system is distributed proportionally to the storage nodes, taking into account their heterogeneous performance. As LL and RL, achieved roughly the same load-balancing results after 5 steps but after 10 steps RL sets apart from other strategies. Our strategy achieved the best load balancing among all strategies as it took actions that distributed the load smartly among the storage nodes.

In the second experiment, we increased the zipf parameter to 1.0 which changes data popularity aspects. In this scenario, we have the majority of the system workload concentrated in some virtual nodes replicas. As these replicas have high *Get* request rates, they must be placed carefully as they will drive extra load to the storage node that manages them. Although each step migrates the same number of replicas, the initial steps move replicas of popular data that have higher *Get* req/s rates, thus the algorithm must be careful, that is, making poor decisions in the beginning will have a greater impact in the system latency.

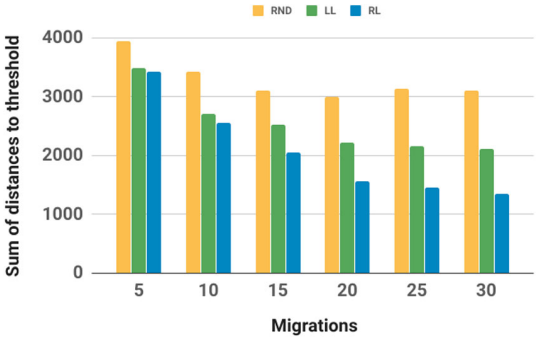


FIGURE 7 Resource utilization progression with zipf 0.1

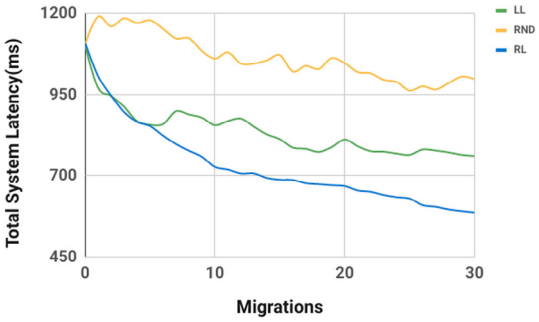


FIGURE 8 System overall latency progression with zipf 1.0

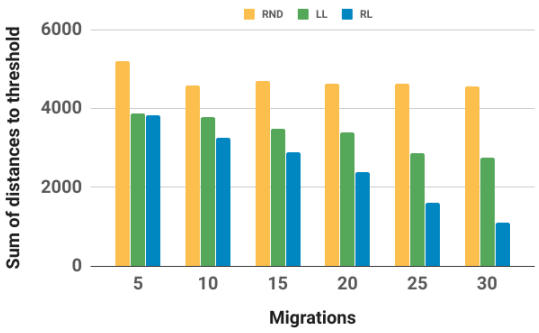


FIGURE 9 Resource utilization progression with zipf 1.0

In Figure 8, we can see the progression of the total system latency along the migration steps. As we can see, even though RND reduced the system overall latency by some fraction, it performed very poorly, especially in the initial steps when it sometimes migrates heavy replicas to already overloaded nodes. LL made good migration decisions in the beginning but due to the fact that it does not know storage nodes performance aspects, various suboptimal placement decisions happened causing the latency improvements to be lessened. Our strategy, in the other hand, had the best results among them all. It continued to find good placements along all migration steps, decreasing the latency after each step taken. Figure 9 shows the impact the placement decisions of each strategy had after 5, 10, 15, 20, 25, and 30 migration steps. We can see that the load imbalance is greater with zipf 1.0. That is due to the fact that some heavy data replicas are initially placed on storage nodes with low performance capacities. Our approach RL was able to achieve the best results and improve resource utilization. Our strategy demonstrates that it can find good placement for data replicas with their different access rates. Thus, minimizing system load imbalance.

Next, we change data popularity again and increase the zipf parameter to 1.8, causing the workload to concentrate even more on fewer data replicas. We want to evaluate how the strategies perform when placing higher popular data replicas. In this scenario, the migration steps made in the beginning are crucial since replicas with very high requests rates are moved. Bad placements decisions have a greater impact on the overall system latency especially in this scenario.

As we can see in Figure 10, RND some poor decisions specially the in the beginning which caused the system overall latency to reach values higher than the initial measurement. Data replicas with high access rate offer smaller room for error, meaning they will usually consume more storage nodes' resources.

In the initial steps LL performs roughly the same as RL by making similar placement decision, but sometimes it fails to find a proper replica placement by assuming that the storage node with lower latency is the most suitable, but that is not always the case, and LL ends up increasing latency at specific steps. The first then steps, the latency is minimized at a faster rate, especially in the case of RL, as the heavy replicas are

FIGURE 10 System overall latency progression with zipf 1.8

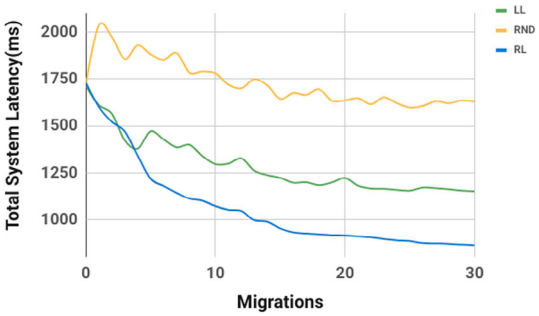


FIGURE 11 Results with zipf 1.8

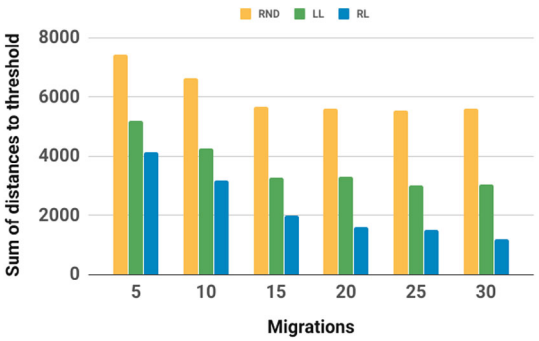


TABLE 3 Summary of system latency minimization results by strategy

	RL (%)	LL (%)	RND (%)
zipf 0.1	42.91	28.78	11.86
zipf 1.0	47.19	30.92	9.79
zipf 1.8	50.19	32.87	5.35

Abbreviations: LL, lowest latency; RL, reinforcement learning; RND, random storage node.

moved first. Toward the second half of the experiment, the data replicas moved do not hold as much popular data, then the latency is reduced at a slower rate.

Our RL strategy showed excellent results in this scenario as well as it was not only the approach that most reduced the latency, but it kept taking consistent good replica placement decisions. Our strategy also had the best results related to load balancing and resource utilization as shown in Figure 11. We can observe that the load imbalance is even higher than the previous scenario. Our RL approach is able to generalize and make good replica placement decisions even when facing different data access patterns. Table 3 summarizes the results concerning the latency minimization by strategy. Compared to the initial state, our strategy was able to reduce the latency by up to 50.19% in the case with higher popular data. As expected, RND performed the worst. Especially, when dealing with popular data.

5.3 | Virtual nodes scaling

The number of virtual nodes is normally set by the system administrator in the system deployment but it can be changed later if needed. In this experiment, we want to observe the efficacy of our strategy when we change the number of virtual nodes in the system. With more virtual nodes, we have fewer files in each virtual node. Therefore, each virtual node have fewer data access. In previous experiments, we run with 1024 virtual nodes. In this experiment, we increase that number to 4096. That means all data files stored in the KVS system are now distributed in 4096 virtual nodes. We set the value of the zipf parameter to 0.1.

Figure 12 shows the sum of latencies in all storage nodes in the system over 30 migration steps. RND minimized the latency by 10.53% but took various bad placement decisions as we can see, increasing latency during some steps. The LL strategy reduced the system overall latency by 27.51%. Our strategy had the best results as it consistently pursued lower latency measurements in every replica placement step. It achieved 41.03% overall latency reduction.

In Figure 13, we can observe the improvements on load balancing achieved by each strategy after 30 migration steps. RND improvements on resource utilization were very limited. The LL algorithm showed somewhat a constant improvement rate. However, our strategy was able to obtain results at an even better rate regarding load balancing and resource utilization.

5.4 | Heterogeneity change

In this experiment, we evaluate another important feature of the our proposed approach. The intention is to test its adaptability aspect. More specifically, the ability to learn new behavior and adjust the migration decision-making when the heterogeneity of the storage nodes change. For instance, the KVS system administrator might upgrade storage nodes hardware components (eg, replacement of HDD with SSD) increasing its capacity to handle demand. To simulate a similar scenario, we increased by 10% the IOPS performance value of the storage nodes 1, 3, and 5. For this experiment subsection, the KVS system is configured with 1024 virtual nodes and the data access rate was chosen using a value of 0.1 as parameter for the zipf distribution. In this experiment, we loaded our model built in the beginning and let it adjust, that is, do additional training until the loss value stabilizes which took generally seven iterations. After that, we deployed our adjusted model again and evaluated it. Figure 14 shows the latency measurements for all strategies obtained along 30 migrations steps.

As LL does not take into consideration heterogeneity aspects, it ends up making inferior migrations decisions which limited the total latency improvement. In the end, LL achieved 28.46% overall latency reduction. RND algorithm, as it take random actions, minimized the latency by only 10.76%. Our strategy made similar placement decisions compared to LL for the first 10 steps on average. However, unlike LL, RL was able to find good placement for data replicas along the entire experiment which helped to reduce latency by 43.76% when compared to the initial state.

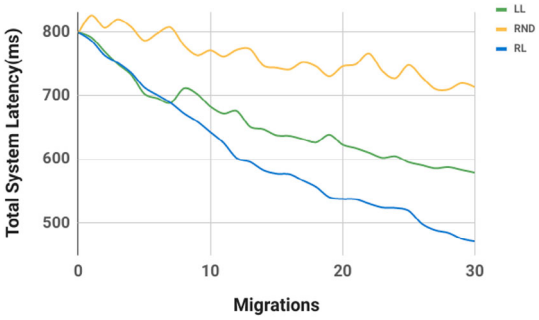


FIGURE 12 Results with 4096 virtual nodes

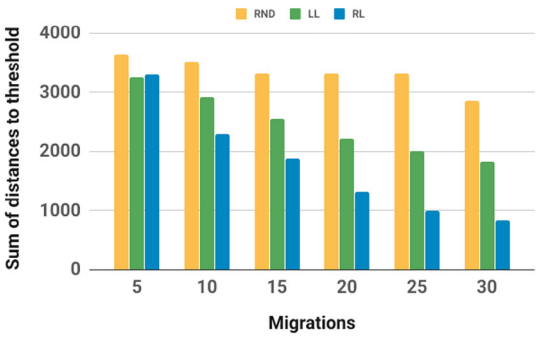


FIGURE 13 Comparison of results on load-balancing aspects

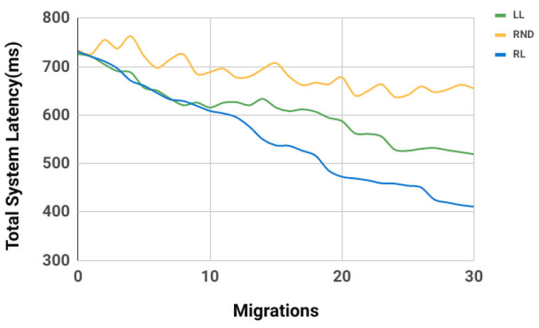


FIGURE 14 System overall latency progression after changing storage nodes performance capacities

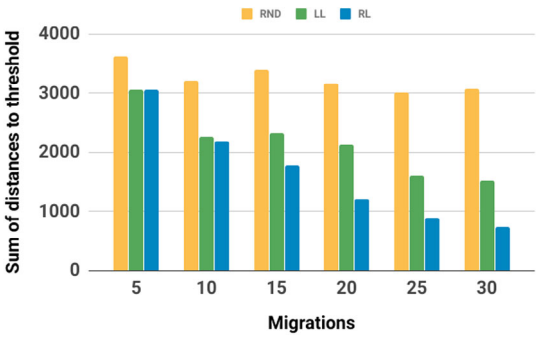


FIGURE 15 Results after improving some storage nodes performance capacities

Concerning load balancing and resource utilization aspects, RND as expected had the worst performance in this aspect as well. LL and our RL strategy achieved close results after five and ten migration steps as seen in Figure 15. However, our strategy achieved greater improvement after 30 migration steps, distributing the load accordingly to the storage nodes performance capacities. Particularly, taking into consideration new acquired knowledge on the improved performance capacities of some nodes.

6 | CONCLUSION

This paper presented a novel approach to perform placement of data replicas in KVS system. It addressed two important issues that concern KVS system based on CHT. One of those is data access skew, in which data stored in KVS system does not have the same access characteristics. Thus, we designed our strategy to generalize and perform well in different workload patterns. Another important aspect we considered is heterogeneity characteristics of storage nodes. Our approach is structured to work and adapt in a KVS architecture where storage nodes have different performance capacities. We considered that those aspects are crucial during replica placement decisions. Hence, we leveraged deep RL in order to build an adaptable model that assists our strategy on deciding the placement of data replicas. Our model learns the impact of migrating data replicas, with different characteristics, to storage nodes with different performance capacities.

In this work, we answered all three questions regarding the migration and placement decision. We compared our approach to others and tested out strategy on different scenarios of workloads and storage nodes configurations. In the experiment results, we analyzed the performance of our approach concerning storage nodes latency minimization, load balancing, and resource utilization aspects. Our approach achieved excellent results, and it was very consistent throughout all different scenarios.

6.1 | Limitations and future work

New research opportunities come out of our results. Considering availability and bandwidth aspects in our replica placement decision is an open issue we want to address. We plan to analyze the impact of different DNN architectures and hyper-parameters. Also, we intend to incorporate replication aspects into our approach, like dynamically defining different replication factor for each replica based on its popularity. Finally, we intend to implement a hybrid model[19] that trains offline using data from a already known good heuristic and then in a second phase updates the model online. That will help to avoid potential bad placement decisions in the beginning of training.

ACKNOWLEDGMENTS

This work was partially supported by Lenovo, as part of its R&D investment under Brazil's Informatics Law, by a grant from Capes/Brazil and also by LSBD/UFC.

ORCID

José S. Costa Filho https://orcid.org/0000-0002-8452-1975

REFERENCES

1. Cavalcante DM, Farias VA, Sousa FRC, Paula MRP, Machado JC, Souza N. PopRing: a popularity-aware replica placement for distributed key-value store. Paper presented at: Proceedings of the 8th International Conference on Cloud Computing and Services Science Vol. 1; CLOSER, INSTICC, SciTePress; 2018:440–447.
2. Mesnier M, Ganger GR, Riedel E. Object-based storage. *IEEE Commun Mag*. 2003 Aug;41(8):84-90.
3. DeCandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Oper Syst Rev*. 2007;41(6):205-220.

4. Chekam TT, Zhai E, Li Z, Cui Y, Ren K. On the synchronization bottleneck of OpenStack Swift-like cloud storage systems. Paper presented at: Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on IEEE; 2016:1–9.
5. He Q, Li Z, Zhang X. Study on cloud storage system based on distributed storage systems. Paper presented at: 2010 International Conference on Computational and Information Sciences; 2010:1332–1335.
6. Karger D, Lehman E, Leighton T, Panigrahy R, Levine M, Lewin D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. Paper presented at: Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing STOC '97; ACM; 1997; New York, NY:654–663. doi:https://doi.org/10.1145/258533.258660.
7. Makris A, Tserpes K, Anagnostopoulos D, Altmann J. Load balancing for minimizing the average response time of get operations in distributed key-value stores. Paper presented at: 2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC); 2017:263–269.
8. Yeo S, Lee H. Using mathematical modeling in provisioning a heterogeneous cloud computing environment. *Computer*. 2011;44(8):55-62.
9. Fan B, Lim H, Andersen DG, Kaminsky M. Small cache, big effect: provable load balancing for randomly partitioned cluster services. Paper presented at: Proceedings of the 2Nd ACM Symposium on Cloud Computing SOCC '11; ACM; 2011; New York, NY:23:1–23:12. doi:https://doi.org/10.1145/2038916.2038939.
10. Bohn C, Lamont G. Load balancing for heterogeneous clusters of PCs. 2002;01(18):389-400.
11. Eager DL, Lazowska ED, Zahorjan J. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans Softw Eng*. 1986;SE-12(5):662-675.
12. Wang Z, Chen H, Fu Y, Liu D, Ban Y. Workload balancing and adaptive resource management for the swift storage system on cloud. *Future Gener Comput Syst*. 2015;51:120-131.
13. Cheng Y, Gupta A, Butt AR. An in-memory object caching framework with adaptive load balancing. Paper presented at: Proceedings of the Tenth European Conference on Computer Systems EuroSys '15; ACM; 2015; New York, NY:4:1–4:16. doi:https://doi.org/10.1145/2741948.2741967.
14. Naiouf MR, De Giusti LC, Chichizola F, De Giusti AE. Dynamic Load Balancing on Non-homogeneous Clusters. In: Min G, Di Martino B, Yang LT, Guo M, Rünger G, eds. *Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops Berlin, Heidelberg*. Berlin, Germany: Springer; 2006:65-73.
15. Atikoglu B, Xu Y, Frachtenberg E, Jiang S, Paleczny M. Workload analysis of a large-scale key-value store. Paper presented at: Proceedings of the 12th ACM SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems SIGMETRICS '12; ACM; 2012; New York, NY:53–64. doi:https://doi.org/10.1145/2254756.2254766.
16. Zheng Q, Chen H, Wang Y, Zhang J, Duan J. COSBench: cloud object storage benchmark. Paper presented at: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering ICPE '13; ACM; 2013; New York, NY:199–210. doi:https://doi.org/10.1145/2479871.2479900.
17. Kubat M. Reinforcement Learning by AG Barto and RS Sutton, MIT Press, Cambridge, MA 1998, ISBN 0-262-19398-1. *Knowl Eng Rev*. 1999 Dec;14(4):383-385. https://doi.org/10.1017/S0269888999003082.
18. Mao H, Alizadeh M, Menache I, Kandula S. Resource management with deep reinforcement learning. Paper presented at: Proceedings of the 15th ACM Workshop on Hot Topics in Networks HotNets '16; ACM; 2016; New York, NY:50–56. doi:https://doi.org/10.1145/3005745.3005750.
19. Tesauro G, Jong NK, Das R, Bennani MN. A hybrid reinforcement learning approach to autonomic resource allocation. Paper presented at: 2006 IEEE International Conference on Autonomic Computing; 2006:65–73.
20. Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning. *Nature*. 2015 Feb;518(7540):529-533. https://doi.org/10.1038/nature14236.
21. Makris A, Tserpes K, Anagnostopoulos D. A novel object placement protocol for minimizing the average response time of get operations in distributed key-value stores. Paper presented at: 2017 IEEE International Conference on Big Data (Big Data); 2017:3196–3205.
22. Long SQ, Zhao YL, Chen W. MORM: a multi-objective optimized replication management strategy for cloud storage cluster. *J Syst Arch*. 2014;60(2):234-244.
23. Paula MRP, Rodrigues E, Farias VAE, Sousa FRC, Machado JC. BACOS: a dynamic load balancing strategy for cloud object storage. Paper presented at: Anais do XXXV SimpÃ³sio Brasileiro de Redes de Computadores e Sistemas DistribuÃ SBC; 2017;dos Porto Alegre, RS, Brasil. http://ojs.sbc.org.br/index.php/sbrc/article/view/2609. Accessed April 25, 2019.
24. Mseddi A, Salahuddin MA, Zhani MF, Elbiaze H, Glitho RH. On optimizing replica migration in distributed cloud storage systems. Paper presented at: 2015 IEEE 4th International Conference on Cloud Networking (CloudNet); 2015:191–197.
25. Sutton RS, Barto AG. *Introduction to Reinforcement Learning*. 1st ed. Cambridge, MA: MIT Press; 1998.
26. Menache I, Mannor S, Shimkin N. Basis function adaptation in temporal difference reinforcement learning. *Ann Oper Res*. 2005 Feb;134(1):215-238. https://doi.org/10.1007/s10479-005-5732-z.
27. Demuth HB, Beale MH, De Jess O, Hagan MT. *Neural Network Design*. 2nd. Notre Dame, Indiana: Martin Hagan; 2014.
28. Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning. *CoRR* 2013:abs/1312.5602. http://arxiv.org/abs/1312.5602.
29. Graphite. https://graphiteapp.org. Accessed April 25, 2019.
30. Plappert M. keras-rl. GitHub. 2016. https://github.com/keras-rl/keras-rl.
31. Tensorflow. https://www.tensorflow.org/. Accessed April 25, 2019.
32. Kingma DP, Ba J. Adam: a method for stochastic optimization. *CoRR* 2014;abs/1412.6980. http://arxiv.org/abs/1412.6980.
33. Gupta A, Dinda PA, Bustamante F. Distributed popularity indices. In: Proceedings of ACM SIGCOMM, Philadelphia, 2005.

How to cite this article: Costa Filho JS, Cavalcante DM, Moreira LO, Machado JC. An adaptive replica placement approach for distributed key-value stores. *Concurrency Computat: Pract Exper*. 2020;32:e5675. https://doi.org/10.1002/cpe.5675