UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JOSÉ SERAFIM DA COSTA FILHO

AN ADAPTIVE REPLICA PLACEMENT APPROACH FOR DISTRIBUTED KEY-VALUE STORES

FORTALEZA
2019

JOSÉ SERAFIM DA COSTA FILHO

AN ADAPTIVE REPLICA PLACEMENT APPROACH FOR DISTRIBUTED KEY-VALUE STORES

Dissertação apresentada ao Curso de Mestrado Acadêmico em Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Javam de Castro Machado

Coorientador: Prof. Dr. Leonardo Oliveira Moreira

FORTALEZA

2019

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C873a Costa Filho, José Serafim da.
An Adaptive Replica Placement Approach for Distributed Key-Value Stores / José Serafim da Costa Filho. – 2019.
68 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2019.
Orientação: Prof. Dr. Javam de Castro Machado.
Coorientação: Prof. Dr. Leonardo Oliveira Moreira.

1. Distributed Key-value Store. 2. Replica Placement. 3. Load Balancing. I. Título.

CDD 005

JOSÉ SERAFIM DA COSTA FILHO

AN ADAPTIVE REPLICA PLACEMENT APPROACH FOR DISTRIBUTED KEY-VALUE STORES

Dissertação apresentada ao Curso de Mestrado Acadêmico em Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Universidade Federal do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Aprovada em: 12 de Março 2019

À minha família, por sempre ter acreditado em mim.

AGRADECIMENTOS

Aos meus pais, José Serafim da Costa e Maria do Socorro de Andrade Mota Costa, pelo suporte e motivação para seguir em frente nos momentos mais importantes da minha vida.

Aos meus orientadores e mentores, Javam Machado e Leonardo Moreira, pelos conselhos, orientações e oportunidades concedidas que foram fundamentais para o sucesso desta dissertação.

Aos professores João Paulo Pordeus Gomes e Marcial Porto Fernandes pela disponibilidade de participar da minha banca de mestrado.

Ao Laboratório de Sistemas e Banco de Dados (LSBD) por ter fornecido toda a estrutura necessária para o desenvolvimento da pesquisa científica contida na minha dissertação.

Aos meus amigos Denis Cavalcante, Sergio Marinho, Rui Pessoa, Flávio Sousa pelo apoio contínuo e a paciência durante as discussões envolvendo computação e nuvem e balanceamento de carga no LSBD.

A todos os meus amigos colaboradores do LSBD pela ajuda em diversas situações que de uma forma direta ou indireta contribuíram para construção da minha pesquisa científica.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

"Anyone who has never made a mistake has never tried anything new."

(Albert Einstein)

RESUMO

O uso de KVS (armazenamento de valores-chave distribuídos) sofreu adoção rápida por vários tipos de aplicativos nos últimos anos devido às várias vantagens oferecidas, como APIs RESTful baseadas em HTTP, alta disponibilidade e elasticidade. Devido às excelentes características de escalabilidade, os sistemas KVS geralmente usam hashing consistente como mecanismo de alocação de dados. Embora os sistemas KVS ofereçam muitas vantagens, eles não foram projetados para se adaptar dinamicamente a cargas de trabalho que geralmente incluem enviesamento no acesso aos dados. Além disso, os nós de armazenamento físico subjacentes podem ser heterogêneos e não expor seus recursos e métricas de desempenho a camada acima responsável pela alocação dos dados. Nesta dissertação, essas questões são abordadas e propõe-se um passo essencial para uma solução autônoma dinâmica, alavancando a aprendizagem por reforço profundo. Uma abordagem de autoaprendizagem é projetada para alterar de forma incremental o posicionamento de réplicas de dados, melhorando o balanceamento de carga entre os nós de armazenamento. A abordagem proposta é dinâmica no sentido de que é capaz de evitar a concentração de dados populares evitando que nós de armazenamento se tornem sobrecarregados. Além disso, a solução desenvolvida pensada para ser conectável. Ela não pressupõe nenhum conhecimento prévio dos recursos dos nós de armazenamento, portanto diferentes implantações de KVS podem utilizá-la. Os experimentos mostram que a estratégia proposta funciona bem diante de mudanças de diferentes cargas de trabalho, que podem incluir enviesamento no acesso aos dados. Além disso, uma avaliação da abordagem proposta é feita em cenários em que a heterogeneidade dos nós de armazenamento é alterada. Os resultados demonstram que a abordagem proposta pode se adaptar, construindo sobre o conhecimento do desempenho dos nós de armazenamento adquirido previamente.

Palavras-chave: Armazenamento Chave-valor Distribuído. Alocação de Réplica. Balanceamento de Carga.

ABSTRACT

The use of distributed key-value stores (KVS) has experienced fast adoption by various types of applications in recent years due to key advantages such as HTTP-based RESTful APIs, 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 dissertation, those issues are addressed and it is proposed an essential step towards a dynamic autonomous solution by leveraging deep reinforcement learning. A self-learning approach is designed which incrementally changes the placement of data replicas, improving the load balancing among storage nodes. The proposed approach is dynamic in the sense that is capable of avoiding hot spots, i.e. overloaded storage nodes when facing different workloads including uneven data popularity situations. Also, the solution developed is intended to be pluggable. It assumes no previous knowledge of the storage nodes capabilities, thus different KVS deployments may make use of it. The experiments show that the proposed strategy performs well on changing workloads including data access skew aspects. In addition, an evaluation of the proposed approach is done on scenarios when storage nodes heterogeneity changes. The results demonstrate that the proposed approach can adapt, building up on the knowledge about the storage node's performance it has already acquired.

Keywords: Distributed Key-value Store. Replica Placement. Load Balancing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CHT	Consistent hashing table
DBMS	Database Management System
DHT	Distributed hash table
DQN	Deep Q-Network
EBS	Elastic Block Store
GFS	Google File System
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
ID	Inique Identifier
IDC	International Data Corporation
IOPS	Input/Output Operations Per Second
iSCSI	Internet Small Computer System Interface
IT	Information Technology
KVS	Distributed key-value stores
LL	Lowest Latency
MDP	Markov Decision Process
NAS	Network-Attached Storage
NFS	Network File System
NIST	National Institute of Standards and Technology
OSD	Object-based Storage Device
P2P	Peer-to-peer
REST	Representational State Transfer
RL	Reinforcement Learning
RND	Random
RPS	Replica Placement Scheme
SAN	Storage Area Network
SATA	Serial ATA
SSD	Solid State Drive
VM	Virtual Machine

WORM	Write once read many

SUMÁRIO

1 INTRODUCTION

In this chapter, the aim is to introduce the problem this dissertation focus on solving. The problem is motivated by showing the relevancy in its context and applicability. Research questions and hypothesis are defined as well as the objectives of this dissertation. Moreover, it is presented what are the contributions of this work.

1.1 Motivation

Distributed key-value stores (KVS) are a well-established approach for cloud data-intensive applications (CAVALCANTE *et al.*, 2018) 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 (e.g., photo, image and video) (MESNIER *et al.*, 2003). Object Store such as Dynamo (**??**) and OpenStack-Swift (CHEKAM *et al.*, 2016) have become widely accepted due to its scalability, high capacity, cost-effective storage and reliable Representational State Transfer (REST) programming interface. These systems take advantage of peer-to-peer architecture (HE *et al.*, 2010) 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) (KARGER *et al.*, 1997a) with virtual nodes. 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 (**??**). 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 (MAKRIS *et al.*, 2017). Second, it assumes that storage nodes are homogeneous, having the same performance capabilities (e.g., I/O throughput, bandwidth, processing power) which is not always the case. Although a cloud infra-structure can start with near-homogeneous resources, it will likely grow more heterogeneous over time due to upgrades and replacement (Yeo; Lee, 2011). With heterogeneous machines, the system designers have the option of quickly adding newer hardware that is more powerful than the existing hardware (FAN *et al.*, 2011). When this is done, the assignment of equal load among nodes results in suboptimal performance (BOHN; LAMONT, 2002). 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 (Eager *et al.*, 1986). 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 work, those two issues are addressed and it is proposed 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 (WANG *et al.*, 2015) (CHENG *et al.*, 2015). Thus it is necessary to adjust the load dynamically while the application is being executed (NAIOUF *et al.*, 2006). In this work, it is employed 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 (ATIKOGLU *et al.*, 2012), 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, a KVS system's load is defined as a function of the number of *Get* requests per time frame. More specifically, it is monitored the number of *Get* requests for each data replica in the system per time frame. By changing the replica placement mapping scheme, data replica migrations can be triggered, 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 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 (FAN *et al.*, 2011). As shown in (ZHENG *et al.*, 2013), 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.

Research Questions. 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 destiny 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(s) might cause in each storage node. For instance, a storage node equipped with a solid state drive may be less affected by the additional load that replica will draw than a storage node equipped with a hard disk drive. It is crucial to alleviate hotspots without overloading other nodes.

To tackle this issue, it is proposed the use of reinforcement learning (RL) (KUBAT, 1999). It offers potential benefits in autonomic computing but 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 (TESAURO *et al.*, 2006). 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 (MNIH *et al.*, 2015). In this work, deep reinforcement technique allows us to find the most suitable storage node, i.e. 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.

Hypothesis. My hypothesis is that by leveraging deep reinforcement learning technique it is possible to calculate the impact that migrating data replicas would cause to each storage node in the system, thus it is possible to choose the storage node that will contribute the most to reduce system overall latency and improve resource utilization.

1.2 Objectives

1.2.1 General Objective

The main objective of this work is to develop an adaptive strategy of replica placement on distributed key-value store that takes into account heterogeneous environments and different workloads with data skew access aspects.

1.2.2 Specific Objectives

To guide the development of the general objective, the following specific are defined:

- Research and review the state of the art on replica placement techniques in distributed key-value stores, analysing its advantages and limitations.
- Propose an adaptive replica placement strategy that is capable of migrating data replicas in response to dynamic workloads on heterogeneous environments.
- Implement and evaluate through experiments the effectiveness of the proposed strategy in a real world deployment of a distributed key-value store.

1.3 Contributions

The major contributions of this dissertation are the following:

- The proposal of a new machine learning-based approach for replica placement. 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).
- The modeling of this approach is designed to be robust to different workloads that may include data access skew. Also, it is built to be adaptable when the storage node backend performance aspects change.
- An evaluation of the performance of the proposed approach through experiments. The approach was deployed in a real world distributed key-value store system and the results are analysed in terms of system overall latency reduction and resource utilization.

1.4 Scientific Papers

Tables 1 and 2 include the work done throughout the past two years. The work submitted to JNCA represents the main contribution which is detailed in this dissertation. The other four papers are not related to this dissertation. However, the knowledge on load balancing and data migration obtained from those works helped to explore some ideas that are shown in this dissertation.

Tabela 1 – Scientific Papers in Conferences

Conference	Year	Title	Status
SBBD	2017	A Predictive Load Balancing Service for Cloud-Replicated Databases	Published
CLOSER	2018	HIOBS: A Block Storage Scheduling Approach to Reduce Performance Fragmentation in Heterogeneous Cloud Environments	Published

Tabela 2 – Scientific Papers in Journals

Journal	Year	Title	Status
CONCURRENCY AND COMPUTATION	2018	Predictive elastic replication for multi-tenant databases in the cloud	Published
JIDM	2019	LABAREDA: A Predictive and Elastic Load Balancing Service for Cloud-Replicated Databases	Published
JNCA	2019	An Adaptive Replica Placement Approach for Distributed Key-Value Stores	Review

1.5 Dissertation Structure

The remainder of this dissertation is organized as follows. Chapter 2 provides important background knowledge about distributed key-value stores and deep reinforcement learning. Chapter **??** introduces the related work, emphasizing their advantages and limitations. In Chapter 3, it is presented the approach for replica placement in detail by defining the system architecture, the RL formulation and the overall workflow of the system with the deployed solution. Then, in Chapter 4, the experiments results comparing the proposed strategy to the baselines are discussed. Finally, Chapter 5 concludes the dissertation and indicate the future work.

2 BACKGROUND

This chapter aims to prepare the reader with background information for understanding our problem context and formulation as well as our solution. First, it is given an introduction to data storage systems in the cloud. A description of its types, features and data allocation mechanisms is presented. Then, it is discussed about DHTs and Consistent hashing table (CHT)s, core technologies behind distributed key-value stores, highlighting how they work and their properties. Next, it is presented the concept of "load", associated terms and load balancing approaches for DHT/CHT systems. Also, it is given some background information on Reinforcement Learning (RL), explaining its core elements. Then, it is introduced a particular RL algorithm called Q-Learning and its variant known as Deep Q-Learning.

2.1 Cloud Computing

Cloud computing has been consolidated over the years. As applications have become much more attractive as services, manufacturers have changed the way hardware components are made and the way service providers optimize the use of hardware components. The computational cloud is an abstraction referring to the applications that are delivered as services through the *internet* and the entire hardware infrastructure and software system in *data center* capable of delivering such services. Also, how applications can be accessed on demand and anywhere (ARMBRUST *et al.*, 2010).

Works as (VAQUERO *et al.*, 2008) seek to follow and understand the evolution of Information Technology (IT), along the development of computing technologies in *cluster* and *grid* systems, with the goal of defining cloud computing. However, the definition of the National Institute of Standards and Technology (NIST) is relevant given the fact that the institute responsible for defining the IT standards for the American industry (MELL; GRANCE, 2011). In addition, the definition of NIST is closer to the objectives of this work. According to NIST, the cloud computing is an evolving paradigm and is defined as follows: *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.* The NIST definition encompasses essential features as a means of comparing different service models and deployment strategies in data centers.

2.1.1 Key features

Cloud computing is formed for the next edition. As the main indicators of cloud computing are:

- **Self-service** A standard client can provide computational resources, such as processing time and storage over the network, when necessary, without the need for human intervention with the service providers.
- **Wide network access**. Features are available for the network and can be accessed through standardized mechanisms that support the use of heterogeneous clients. Access can be done either by customers or with high processing capacity of customers for customers with low processing capacity. For example, *smartphones*, *tablets*, *notebooks* and *laptops* (TOLIA *et al.*, 2006).
- **Resource pool**. The computational resources of a provider are grouped to serve multiple clients following a multi-tenant model. Physical and virtual enterprises are dynamically assigned and reassigned according to a customer demand. In general, they are not able to control and locate a physical drive with the same capacity of the resources, but may contain the possibility of specifying a high abstraction location (for example, country, state, or data center). Storage, processing, memory, and network bandwidth are examples of features that abstraction and physical location concept.
- **Fast Elasticity**. The computational resources can be elastic, that is, they can be provisioned and released. What is a way to increase the automatic capacity to scale resources is a measure that demand increases, and liberals as demand decreases. From the customers' point of view, the computing resources available for provisioning usually appear to be unlimited and can be purchased in any quantity and all the time.
- **Service Measurement**. Cloud computing systems automatically control and optimize resource utilization through the ability to measure the system to the level of abstraction for the type of service. Storage, processing, bandwidth, and number of active clients are examples of abstractions. Resource utilization can be monitored, controlled, and delivered transparently to providers and customers. This type of function is useful for service models that the client only pays (*pay-per-use*) (GONG *et al.*, 2010).

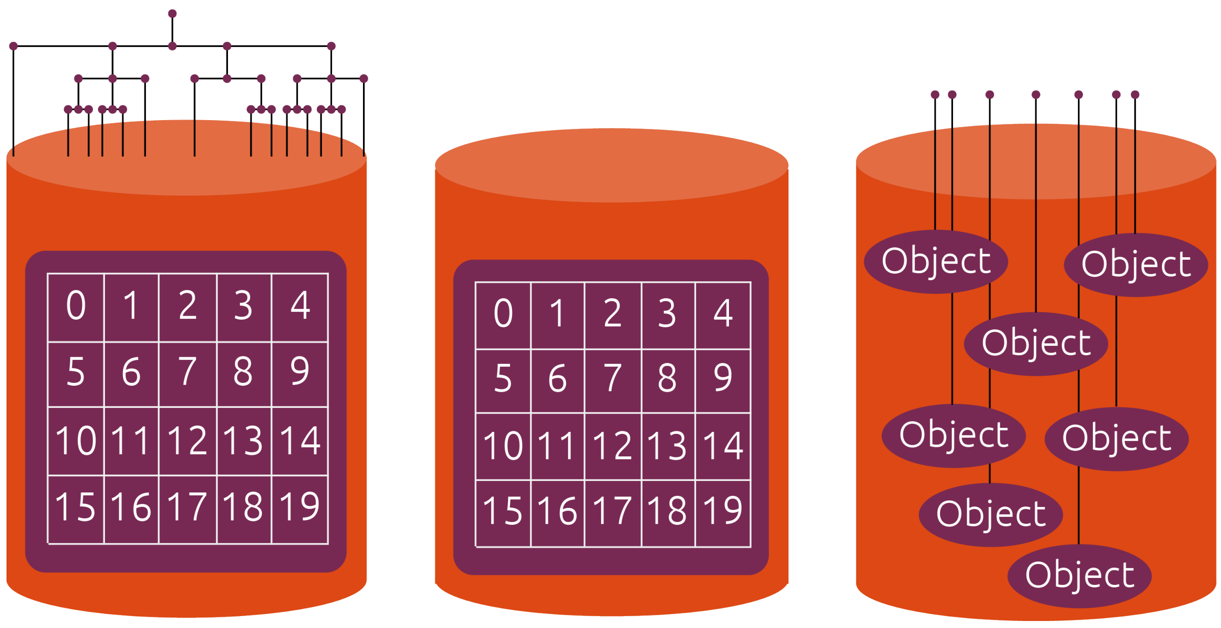
2.2 Cloud storage systems

A cloud storage system (Cloud Storage) is a concept that came with the cloud computing model as a low-cost solution to meet the increasing need for applications through mass storage and high performance. Traditionally, Database Management System (DBMS) has been used by most enterprise applications as a model for data storage and retrieval. Although DBMSs are query-rich, the complexity of their access model coupled with the inflexibility of scaling the infrastructure, form a bottleneck for applications that need a simple and fast storage solution capable of supporting large volumes of data (DEWAN; HANSDAH, 2011). Currently, relational database and other forms of storing structured data are still used in business. However, the growth of unstructured data has been occurring rapidly over the years. According to a study by International Data Corporation (IDC) at 2014, the digital universe in the period from 2013 to 2020 will grow from 4.4 trillion gigabytes to 44 trillion, which is more than double every two years (ZWOLENSKI *et al.*, 2014).

A cloud storage system is understood as a service capable of providing storage for users and clients through the network, usually *internet*. Users can use the storage service in different ways, pay only for usage time, storage space, performance or a combination of these forms of payment (WU *et al.*, 2010). A cloud storage system can still be understood simply as the storage part of a computational cloud. Applications consume cloud storage services based on the *pool* of storage resources. A pool of storage resources can be virtualized to serve applications from within or outside the computational cloud, but not all of the physical storage can be decoupled from the computational cloud (JU *et al.*, 2011).

Cloud storage systems are typically used to store data coming directly from users or computing results from cloud applications. A cloud storage has scalable storage infrastructure and data is stored by multiple storage servers instead of a single dedicated server used in traditional storage solutions. The user sees the storage system as a remote virtual server and not as a manageable physical component. That is, the user has a reference to the stored data and storage space, but has no knowledge of the physical location or control over the infrastructure components. The current location of user data may change frequently as a result of the dynamic management of server performance and available cloud storage space (WU *et al.*, 2010).

In addition to scalable cloud storage systems, they tend to be low-cost financial solutions that provide high durability and data availability. Cloud storage systems can utilize virtualization of storage device resources to support larger numbers of customers in parallel and



(a) File storage (b) Block storage (c) Object storage

Figura 1 – Types of data storage (Source: (CANONICAL, 2015))

lower maintenance costs compared to proprietary solutions with dedicated physical resources accessible through the network. Depending on the cloud storage system, you can still ensure data durability and availability by replicating information between other physical storage servers. Even if one or more servers are unavailable, the system can function normally and provide the data to users without them being aware of the storage infrastructure problems.

2.2.1 Types of cloud storage services

There are several types of cloud storage services offered to meet the specific needs of data-centric applications. Cloud storage service providers can provide storage services focused on managing and manipulating data as an abstraction layer closer to or distant from the persistence level of storage devices (GROSSMAN *et al.*, 2009). Basically, storage technologies can handle storage units in three types of categories: file, block, or object. Figure 1 illustrates the types of data storage in three broad categories. Next, the abstraction levels of data that storage service providers offer are discussed in more detail.

A file-based storage system is a classic approach to data storage. The data is accessed with the file abstraction, where a file is identified by a name and organized hierarchically through directories and subdirectories as shown in Figure 1a. Any file can be found describing the directory path to the file name. Some attributes on a file such as creator name, creation date, and size are stored as static metadata. This convention facilitates secure network sharing across technologies such as Network-Attached Storage (NAS) and communication protocols such as the Network File System (NFS) (GIBSON; METER, 2000) (SHEPLER *et al.*, 2003). File-based systems are performing well and are suitable for local storage, but are not interesting in a large-scale cloud environment because it is difficult to scale. The Distributed File System (GFS) (GHEMAWAT *et al.*, 2003) and the Hadoop Distributed File System (HDFS) (SHVACHKO *et al.*, 2010) are examples of distributed file system services used in environment.

Block-based cloud storage services is a straightforward way to manipulate storage units with a granule smaller than a file. A volume can be thought of as a virtual storage device, independent and with fixed total storage size, composed of small blocks of storage units of the same size. Figure 1b illustrates a set of storage blocks of a certain volume. Therefore, a file or database can be stored in parts using block-based storage. Applications can retrieve blocks over the network using their addresses through interfaces such as the Internet Small Computer System Interface (iSCSI) and Fiber channel, commonly used in Storage Area Network (SAN) architectures (KHATTAR *et al.*, 1999), where high performance can be achieved by consolidating storage devices. In block-based storage, there is no sense in the existence of block-associated metadata, since isolated data parts only have some meaning when grouped and managed by applications such as operating systems and file systems. The OpenStack Cinder and Amazon Elastic Block Store (EBS) (AMAZON, 2017) are examples of cloud services that provide storage in blocks.

This dissertation focuses on object-based services or *object storages* as they are regularly called. *Object storages* are emerging technologies that use object abstraction to facilitate manipulation of data. An object is an abstraction for a conceptually unlimited storage unit, that is, there is no maximum size limit for an object because the object can be stored in parts between several physical storage devices. The Figure 1c illustrates the storage of objects on a storage device. An object can be used to store any type of data such as files, database records, images, or multimedia files. In addition, an object is dynamic and self-contained, that is, the metadata associated with a given data is contained in the object itself (**??**). Users can change the data and metadata of an object without size constraints. Objects can be isolated in containers as a means of organizing objects into semantic sets from the point of view of applications. A container is a logical set that serves to group objects, analogous to a folder in a file system. An object is located by a Inique Identifier (ID) that can be generated from the object content (data and metadata) and its logical path. The ID of an object is useful for retrieving an object quickly through a long-range network such as *internet*. Because storage components in a cloud can be geographically separate, the system can create and retrieve an object using its ID through well-defined application interfaces over the Hypertext Transfer Protocol (HTTP) protocol. Examples of *object storages* include *OpenStack Swift* (SWIFT, 2017), *Ceph* (WEIL *et al.*, 2006) and *Dynamo* (DECANDIA *et al.*, 2007).

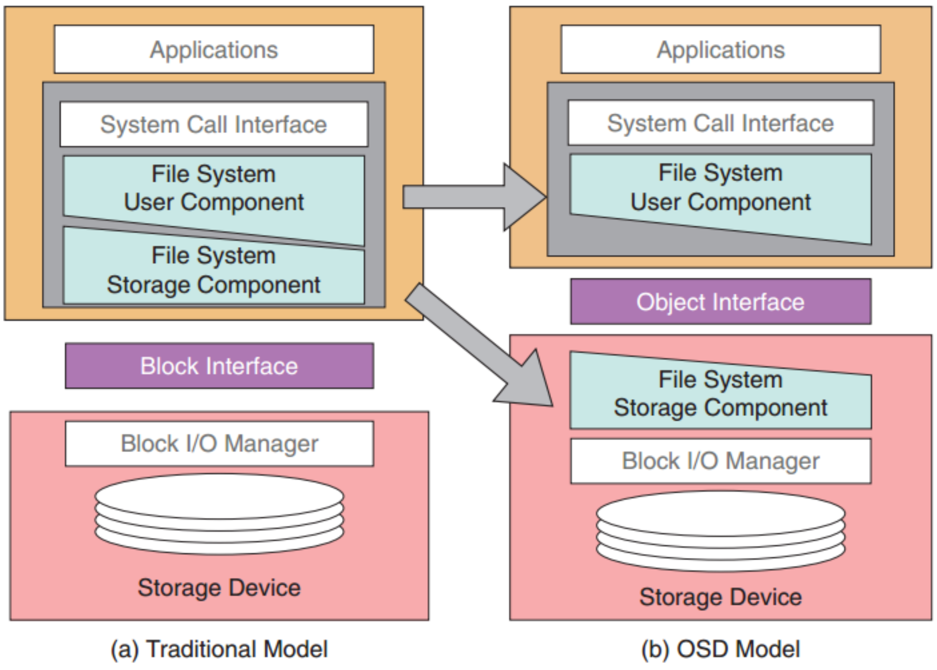


Figura 2 – Transferring responsibility of a file system to manage the storage of an OSD (Source: (Mesnier *et al.*, 2005)).

Object storages manage storage devices called object-based storage devices (OSDs) capable of storing and retrieving data in object format. An OSD can be a simple storage device such as an independent Hard Disk Drive (HDD) or Solid State Drive (SSD), as well as a custom storage device or a cluster of low cost storage devices. An OSD is not limited to random or sequential access. The main difference between an OSD and a block storage device is the interface and not the physical media type (Mesnier *et al.*, 2005). The immediate effect of object-based storage is the transfer of storage space management responsibility for specialized applications, such as the local file system and database. Figure 2 illustrates the transfer of responsibility from a traditional file system to manage storage on an OSD.

The traditional model of a block-based file system consists of two interfaces: user and storage components. The user component is responsible for representing logical data structures such as files and directories, while the storage component maps the data structures to the storage device through a block storage interface. In this model, the storage component is located at the application layer as shown in Figure 2a.

The OSD-based model can take advantage of these interfaces because the OSD adds a lightweight object interface that can reduce the complexity of data management by delegating responsibilities to the file system located on the OSD. The user component remains unchanged, however the storage component is transferred to the storage device so that the device access interface is changed from block to object. The addition of the object interface and the transfer of responsibilities to the storage component of the OSD-based model is shown in Figure 2b.

Object-based storage systems are storage systems with high-level abstraction interfaces, similar to file systems. However, unlike file-based or block-based systems, *object storages* present themselves as highly scalable storage solutions due to object flexibility, standardized communication interfaces, ease of OSD management, and access to data and metadata. The architecture of object storage systems, used by cloud service providers, provides services for applications with different benefits (FACTOR *et al.*, 2005) (AZAGURY *et al.*, 2003). For example, *object storages* bring benefits to service providers and applications such as:

- They facilitate storage management, but with slightly lower performance compared to block-based storage and file system.
- Objects are data abstractions of theoretically infinite size that can be manipulated by applications.
- Storage infrastructure can be built with low-cost and easy-to-scale devices.
- Mass storage of unstructured data such as texts, videos, audios and images.
- Multi-platform data sharing between applications in a simple and fast way.

2.3 Data Allocation

Cloud storage services implement particular strategies and solutions for how to manipulate and organize data between storage devices in order to provide the best possible service to your users. They face the challenge of optimizing the distribution of data between storage devices known as data allocation. Data allocation is related to how to assign data units (objects, files, or blocks) to storage nodes (servers and storage devices) in a distributed system to optimize one or more performance criteria, such as reducing congestion network, use less storage space, improve load balancing, among others (PAIVA; RODRIGUES, 2015). There is a cost-benefit (*trade-off*) bound to each type of data allocation solution criteria to optimize system performance.

Distributed directory-based data allocation strategies are solutions that have the flexibility to define some relationship between the data and the storage node in a (PAIVA; RODRIGUES, 2015) data structure. Distributed file systems such as HDFS and Google File System (GFS) use a custom storage table to manage the location of data stored in the system. When a file is accessed, the system searches a large table to return the location of the storage node that stores the information it queries, analogous to a remote directory. This type of strategy has the advantage of being customizable, but can present problems such as the degradation of query performance. On a large scale, because of the insertion and updating of the location of the files in the data structure, the size of the structure can grow rapidly to become a performance bottleneck and storage in main memory. Because a subset of nodes is responsible for keeping the data structure up-to-date, storage nodes can also be a bottleneck since they need to query the management nodes to locate the data.

Distributed Hash Tables (DHTs) are distributed systems that use *hash* functions (*hash* functions) to distribute data through key mapping to storage nodes. This mapping consists of a key space with the possible *hash* values of the data identifiers. In this way, the *hash* of the identifier of a data is used as a key to access the data stored in a given node. A DHT has the characteristic of being a decentralized system, scalable and easy to implement. The DHTs allow the location of the data very quickly since the key generated by the hash of the data identifier is mapped directly to the storage node where the data is (FELBER *et al.*, 2014). In this way, a DHT can be used as an efficient method of distribution and data access. For more details on the classical approaches to systems and protocols based on DHTs such as Pastry, Chord, Kademlia and etc., it is recommended to read the comparative analysis of these approaches carried out in (LUA *et al.*, 2005) and (FELBER *et al.*, 2014).

Tabela 3 – Example of mapping of objects in a storage system that implements DHT with a conventional *hash*

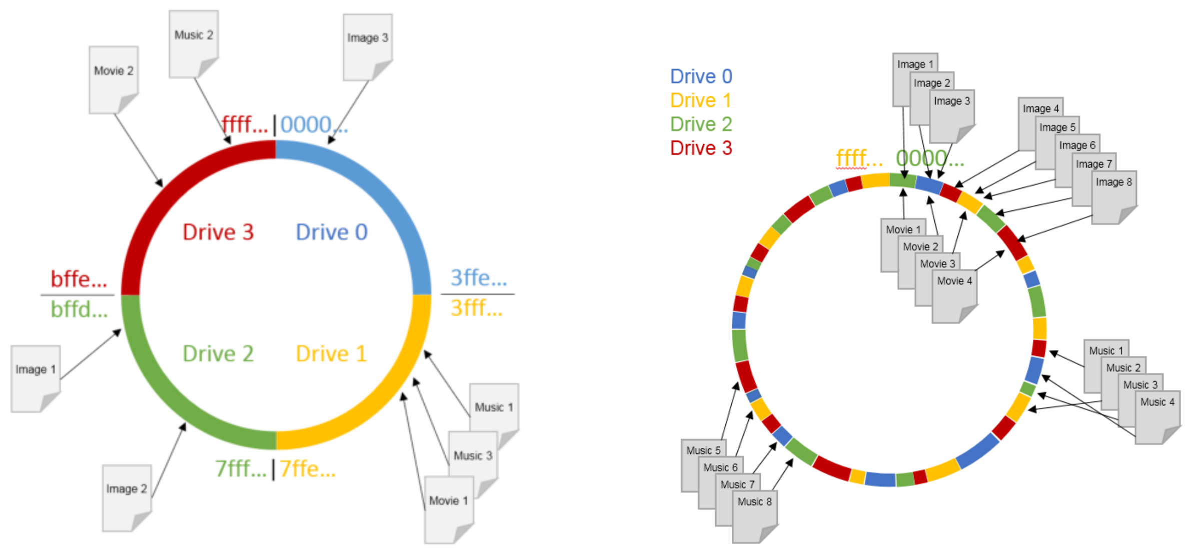
Total number of *drives*	**h(x,n)**	**Mapped to**
5	1	drive 1
6	3	drive 3
7	6	drive 6
8	7	drive 7

One of the major challenges assigned to DHTs in storage systems is related to the redistribution of keys between storage nodes due to the addition and removal of them. The key space managed by a storage node can be changed, that is, a key can point to a new storage node. Thus, the remapping of DHT keys implies that a data migration process will be necessary to reallocate the previously stored data in order to fulfill the new key space mapping. Some hash-based approaches can not cope well with the dynamism of key space and can cause overhead on the network because of excessive data movement between the storage nodes.

To clarify this problem, consider a storage system made up of a set of 5 storage devices that are abbreviated to *drives*. Each object entered in this system receives an ID that is incremented by 1 from the integer value 0. Also consider that the storage system implements a DHT with a hash function *hash* $h : \mathbb{Z}+ \rightarrow \mathbb{Z}+$ which receives as input x and n, where x is the ID of an object and n is the total number of *drives* in the storage system. The h function outputs a $h(x,n)$ key which is the remainder of the division of the object ID by the total number of *drives*. So, a certain object with the ID equal to 111 is mapped to the *drive* number 1, because the remainder of the 111 by 5 division is equal to 1. The problem with this function is that the calculation depends on n. Whenever a *drive* is added or removed from the storage system, the h function tends to generate a new output and map the object to different *drives* from the same object ID. To prove this fact, the table 3 displays for which *drive* the object with ID equal to 111 must be mapped when new *drives* are added. Note that the object must be moved to a different *drive* whenever a *drive* is added or removed. In addition, this recalculation must be done for each object in the storage system making it necessary for each object to be migrated to meet the new remapping of the key space. This remapping process results in a large data movement between different drives after a change in the total number of drives.

Thus, a DHT class named *Hash* consistent (KARGER *et al.*, 1997b) was designed to distribute data between storage nodes evenly and avoid redistributing much of the stored data when added or removed nodes of a storage system. The key space (set containing all hashed values) in a simple consistent hash is represented in the form of a ring so that each *drive* is responsible for managing a continuous range of keys with an average size equal to $1/N$, where N is the total number of keys. Each individual *drive* uses the same hash function to locate remote data and its own data.

The Figure 3a illustrates a simple consistent hash used by a storage system, where each physical drive executes a continuous range of keys. Note that the total key space is divided by 4 *drives*. 0 handles the key range from 0000... to $3ffe$..., *drive* 1 manages the key range of $3fff$... to $7ffe$... and so on. Still in the Figure 3a, the mapping of an object to a *drive* is indicated by an arrow pointing to the key range that it belongs to. Regardless of the hash function



(a) Simple consistent *Hash* with physical *drives* (b) Consistent *Hash* with virtual *drives*

Figura 3 – Examples of consistent *hashs* (Source: (HONG, 2017)).

used to calculate the hash value of an object (eg, image, movie, or music file), a key is generated (an MD5 function can generate a hash value) of the name of an object in the format *9793b...*) used to find in which range of keys it belongs and thus know which specific physical *drive* object should be stored or retrieved. In case a new physical *drive* is inserted into DHT, the location of the vast majority of objects stored in the physical drives of a storage system is not changed. Only a small amount of objects are moved between the physical drives after the physical key strokes are restructured in the DHT because the keys of some objects previously stored in the system can be part of the key range controlled by the new physical *drive*.

A more sophisticated version of the simple consistent hashes is hash consistent with virtual *drives*. Instead of using a large continuous range of keys by physical *drive*, a physical *drive* groups a set of virtual drives representing the partitioning of the key range of a physical *drive* different intervals. In other words, virtual *drives* can be seen as partitions of the key space of a given physical *drive*. The Figure 3b illustrates a consistent DHT with virtual *drives*, see that the physical 2 *green color* uses several virtual *drives* to partition its key space. While there may be many virtual *drives* in the *ring*, the size of the span of a virtual *drive* is small relative to the total number of *ring* keys. Therefore, the addition or removal of a physical *ring* drive involves moving a small number of objects. Consistent DHT with virtual *drives* provides that some physical *drives* manage more objects than others because of the distinct key space size of a virtual *driver*. Therefore, data distribution may become as fair as possible to the performance and/or capacity criteria considered by storage solutions that use consistent DHTs with virtual *drives*.

Consistent hash-based data allocation algorithms are highly employed in storage systems with homogeneous components by virtue of the uniform balancing of data provided by the hash functions. Popular cloud storage systems using consistent hashs include Ceph, Dynamo, OpenStack Swift, and Cassandra (LAKSHMAN; MALIK, 2010). Data allocation algorithms such as the CRUSH (WEIL *et al.*, 2006) and the *ring* DHT of the *OpenStack Swift* provide certain flexibilities for manipulating the data through weights for physical *drives*. The set of weights defines the degree of importance of the devices in the storage system, and are used by system administrators or third-party strategies to meet user performance and storage criteria.

The allocation algorithm of OpenStack Swift uses consistent DHT with virtual drives, support the replication objects and controlling the amount of partial keys for a particular physical drive. Unlike Figure 3b, the value *hash* of an object does not identify directly to which object device is assigned as the DHT Consistent OpenStack Swift considers a dimension more per account of a redundancy policy. The default redundancy policy of *OpenStack Swift* replicates a particular object in the context of *drives* virtual, that is, all objects have the same number of replicas and each *drive* virtual tries to keep only one replica of an object given its value *hash*. The consistent OpenStack Swift is commonly called *ring* and its construction requires some parameters such as:

- The replication factor, which defines the number of replicas of the objects.
- Partition power, which reflects the total number of virtual drives divided between the storage devices
- A set of physical drives and weights associated with them.
- Logical administrative units such as regions and zones to consider possible fault domains.

The ring is a consistent DHT adapted for replication with self-contained information so that the mapping of all replicas of an object is done directly to the physical textures of OpenStack Swift. For this reason, each server maintains a copy of *ring*, as it provides the full localization mechanism for the replicas of an object. *ring* finds the replicas of objects between storage devices through a table containing keys (the hashed values of the objects) and object replica identifiers for storage devices (*drives* physicists). The table 4 is a representation of *ring* from the perspective of a data structure that maps keys and replicas to virtual drives, assuming that each object is configured with three replicas (default configuration used by *OpenStack swift*) in this representation. Note that an object whose key is $29a8$ has its replicas allocated to *drives*

0, 2, and 1.

Tabela 4 – Data structure representation of the DHT with support to replication use in the *OpenStack swift*

		Keys					
		29a8	*bffe*	*ec25*	*3a03*	*b670*	*5g70*
Replicas	1	drive 0	drive 4	drive 3	drive 0	drive 2	drive 0
	2	drive 2	drive 5	drive 1	drive 2	drive 0	drive 1
	3	drive 1	drive 0	drive 0	drive 4	drive 5	drive 3

2.4 Distributed Hash Table

Peer-to-peer (P2P) systems represent a radical shift from the classical client-server paradigm, in which a centralized server processes requests from all clients (**??**). DHT is a type of structured P2P systems which have the property of associating a unique identify (key) to a node of the system, thus making a DHT mapping of keys and nodes.

The space of keys of a DHT is partitioned among the nodes, thus decentralizing the responsibility of data placement look-up. As a result of the DHT management, an overlay network is built on top of the physical network on which the nodes are linked. These links may be based on IP-based P2P systems, wherein a peer may communicate directly with every other peer. While IP-based P2P systems are researched in this work, mobile ad-hoc networks are out of scope. To support this overlay, the DHT component must support two functions:

- Put (key, data): Store data into the node associated with a key;
- Get (key, data): Retrieve data from the node associated with a key.

2.4.1 Object and Request Load

Our understanding of object, request load and resources are similar to the concepts defined by the authors (**??**):

- Object: a piece of information stored in the system. An object has at least its identification and size as meta-data. The popularity of an object is the frequency at which it is accessed.
- Request load: a number of object requests per time unit in which each object request consumes storage space, processing time, bandwidth and disk throughput resources of the system during the request and after the request is completed.

- Resources: system resources have limited capacity in terms of available storage space, processing time, bandwidth and disk throughput. This work refers Get requests for retrieving data and Put requests for inserting data to the system through a web Application Programming Interface (API) component.

2.4.2 Load balancing in DHT systems

The decentralized structure of DHT overlays requires careful design of load balancing algorithms to fairly distribute the load among all participating nodes. For systems based on DHT, name-space, request rate, routing at overlay and underlay are important aspects to understand how DHT designs are affected by load balancing issues and approaches as described in Table 5. Sections 2.4.2.1 and **??** discuss in detail each aspect of our work regarding the literature of load balancing approaches for DHT systems according to the classification proposed by (**??**):

Tabela 5 – Important aspects for DHT load balancing.

Aspect	Meaning
Name-space	It refers to how objects are mapped to keys
Request rate	It refers to how request load is distributed over the keys
Overlay routing	It refers to how keys are mapped to the nodes of the system.
Underlay routing	It refers to how the proximity distance of the underlay nodes matches with the proximity distance of overlay nodes.

2.4.2.1 Load balancing issues

- Name-space balancing: When nodes and keys are not uniformly distributed over the identifier space, objects are expected to be shared between nodes in a skewed manner, thus causing heavy load on some nodes. One classical approach to achieving this property is the name-space balancing (e.g., hashing the IP address of a node or an object name). The DHT adopted in this work uses consistent hashing for name-space balancing.
- Request Rate: The overlays are generally designed to be well balanced under a uniform flow of requests, i.e., with objects having equal popularity, all nodes receive a similar amount of requests. In practice, many workloads have not equal popularity as mentioned

in Chapter 1. By this means, skewed request rate is the main issue which our work aims to solve.

- Overlay routing: each node maintains in its "routing table"the list of outgoing links leading to its immediate neighbors. Nodes with a huge number of incoming links are thus expected to receive on more requests than other nodes, thus causing load imbalance as well. In order to fairly share the traffic load in the overlay, the routing tables should be organized in such a way that the number of incoming links per node is balanced. In our context, replicas means multiple links to the same key, thus the request load on each replica is fairly distributed.
- Underlay routing: the underlying topology is also an important aspect for building the routing tables (e.g., immediate neighbors in the overlay routing which are placed in distant regions in the underlay may cause an unexpected delay for those that are not aware of this discrepancy. Underlay routing is out of scope of our work.

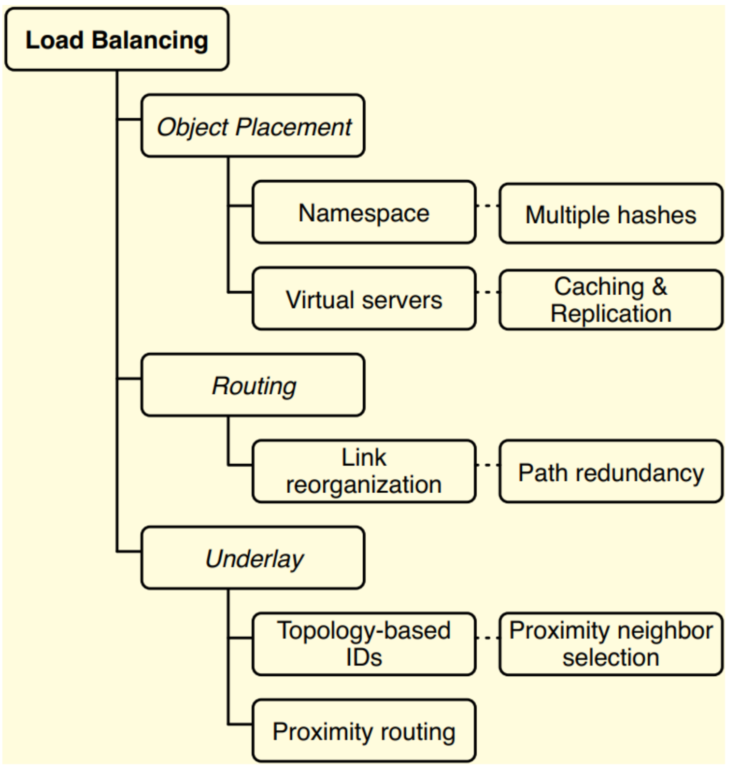


Figura 4 – Load balance solutions for DHT. Source: (**??**)

2.4.3 Consistent hashing

It is a method of name-space balancing presented by (KARGER *et al.*, 1997c). It introduces an appropriate hashing function (e.g., SHA-1) uniformly distributing identifiers and keys over the identifier space. Every node independently uses this function to choose its own identifier. The basic idea of consistent hashing is relatively simple, since each node or object gets its ID by means of the predefined hashing function, e.g., SHA-1. Unfortunately, the usage of only consistent hashing may have the side-effect of having a long interval of keys being managed by a single node. To overcome that, the literature has proposed the usage of consistent hashing with virtual servers.

2.4.4 Virtual servers

Virtual servers or virtual nodes were first introduced by (STOICA *et al.*, 2001) to uniformly distribute identifiers over the addressing space, because the probability of a node to be responsible for a long interval of keys decreases. Beyond that virtual servers give the capacity of activating an arbitrary number of virtual servers per physical node, which is proportional to the peer capacity.

2.4.5 Object Hash Collision

As mentioned before, the consistent hashing function maps objects to virtual nodes through hash operation. The hash operation produces hash collision of a set of different objects mapped to the same virtual node key. To proper handle those hash collisions, each storage node is based on traditional file-systems and block storage layers. This way, different objects with the same key are mapped to the same virtual node and stored into storage nodes without issues.

2.4.6 Consistency Model

Ideally, a distributed system should be an improved version of a centralized system in which fault-tolerance and scalability are proper handled. In fact, this goal is obtained because data may be replicated all over the system, but within the limitations set by the CAP Theorem (GILBERT; LYNCH, 2002). Briefly, CAP Theorem stated that in a distributed system, only two of three following aspects can be met simultaneously: consistency, availability, and partition tolerance. In the case of the type of systems targeted in this work, non-transactional distributed key-value store systems, high availability and partition tolerance are prioritized.

The design decision of prioritizing high availability and partition tolerance may raise a question: how much consistency is sacrificed? According to the literature, there are at least the following levels of consistency:

- Strong Consistency: The gold standard and the central consistency model for non-transactional systems is linearizability, defined by (HERLIHY; WING, 1990). Roughly speaking, linearizability is a correctness condition that establishes that each operation shall appear to be applied instantaneously at a certain point in time between its invocation and its response (VIOTTI; VUKOLIć, 2016).
- Weak Consistency: It is the contrary of strong consistency, which does not guarantee that reads return the most recent value written. Last, but no least important, it does not provide ordering guarantees hence, no synchronization protocol is actually required. For example: relaxed caching policies that can be applied across various tiers of a web application, or even the cache implemented in web browsers (VIOTTI; VUKOLIć, 2016).
- Eventual Consistency: replicas converge toward identical copies in the absence of further updates. In other words, if no new write operations are performed on the object, all read operations will eventually return the same value (TERRY *et al.*, 1994) (VOGELS, 2008) (VIOTTI; VUKOLIć, 2016).

2.4.6.1 Eventual Consistency Implications

As mentioned before, in order to offer high data availability and durability, Distributed key-value stores (KVS)-like systems such as OpenStack-Swift and Amazon DynamoDB typically replicate each data object across multiple storage nodes, thus leading to the need of maintaining consistency among the replicas.

The eventual consistency is embodied by leveraging an object synchronization protocol to check different replica versions of each object. Consequently, this synchronization process introduces network overhead for already existent replicas as well as replica movement throughout storage nodes due to new reconfiguration of replica placement schemes. After a new scheme is deployed, virtual nodes may be remapped to different storage nodes, thus requiring the synchronization of virtual nodes.

2.5 Reinforcement learning

2.5.1 Introduction

Reinforcement Learning (SUTTON; BARTO, 1998) is an area of machine learning concerned with how artificial agents ought to take actions in a certain environment with with goal of maximizing some notion of cumulative reward. The problem, due to its generality, is studied in many other disciplines(e.g. game theory, control theory, etc). In the case of machine learning, the environment is typically formulated as a Markov Decision Process (MDP), as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between reinforcement learning and the classical dynamic programming methods is that in RL do not assume knowledge of an exact mathematical model of the MDP. An advantage of RL algorithms is that they can target large MDPs where exact methods become impracticable.

Reinforcement Learning is considered as one of three machine learning paradigms, the other two being supervised learning and unsupervised learning. Supervised learning is the task of inferring a classification or regression from labeled training data. RL differs from supervised learning as it focus is on performance, which involves finding a balance between exploration (e.g. visit unseen states) and exploitation (of the knowledge it has already acquired) (KAELBLING *et al.*, 1996). The agent must decide if it should follow the action plan or try different, unexplored paths. So if the agent focus on exploration it will not be able to learn anything valuable. The other case is, if the agent focus on exploitation it might not be able to find the optimal sequence of actions which will provide the better utility.

When a model of the environment is known, but an analytic solution is not available is one kind of situation. Another one is when only a simulation model of the environment is given. A third situation is when the only way to collect information about the environment is to interact with it. The first two of these situations could be considered as planning problems since some form of model is available. The latter one, however, could be considered to be a genuine learning problem. Either case, reinforcement learning converts those problems to machine learning problems. The use of samples to optimize performance and the use of function approximation to deal with large environments are the elements that make reinforcement learning powerful.

Rules are often stochastic. The observation typically involves the scalar, immediate reward associated with the last transition. In many works, the agent is assumed to observe the current environmental state (full observability). If not, the agent has partial observability. Sometimes the set of actions available to the agent is restricted (e.g. an agent in a maze game cannot go through a wall). The agent only controls its own actions and has not a priori knowledge regarding the consequences of each action i.e. 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.

Basic reinforcement is modeled as a Markov decision process:

- a set of environment and agent states, S;
- a set of actions, A, of the agent;
- $P_a(s,s') = Pr(s_{t+1} = s'|s_t = s, a_t = a)$ is the probability of transition from state s to state s' under action a.
- $R_a(s,s')$ is the immediate reward after transition from s to s' with action a.

A reinforcement learning agent interacts with the environment in discrete time steps. At each time step t, the agent finds itself in state (s_t where it has to choose an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and a *reinforcement signal* so called reward r_t associated with the transition (s_t, a_t, s_{t+1}) is determined. The goal of a reinforcement learning agent is to collect as much reward as possible.

The agent's action selection is modeled as a map called policy:

$\pi : S \times A \to [0,1]$

$\pi(a|s) = P(a_t = a|s_t = s)$

The policy map gives the probability of taking action a a when in state s. There are also non-probabilistic policies. The value function $V_\pi(s)$ is defined as the expected return starting with state s, i.e. $s_0 = s$, and successively following policy π. Hence, the value function estimates how good it is to be in a given state.

$$V_\pi(s) = E[R] = E[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s],$$

where the random variable R denotes the return, and is defined as the sum of future discounted rewards. $R = \sum_{t=0}^{\infty} \gamma^t r_t$ where r_t is the reward at step t, $\gamma \in [0,1]$ is the discount-rate.

2.5.2 Q-Learning

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations. For any finite Markov decision process (FMDP), Q-learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q"names the function that returns the reward used to provide the reinforcement and can be said to stand for the quality of an action taken in a given state.

Before learning begins, the qualities Q is initialized to a possibly arbitrary fixed value. Then, at each time t the agent selects an action a_t, observes a reward r_t, enters a new state s_{t+1} , and Q is updated. The core of the algorithm is a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}} \tag{2.1}$$

where r_t is the reward received when moving from the state s_t to the state s_{t+1}, and α is the learning rate $0 < \alpha \leq 1$. When γ is equal to 1, the Q function will be equal to the sum of the rewards and when γ equals to zero, Q function only takes current reward into consideration. An episode of the algorithm ends when state s_{t+1} is a final or terminal state. However, Q-learning can also learn in non-episodic tasks.

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. RL uses value function $Q(s,a)$ which is the expected value of the sum of future rewards by following policy π.

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[R_t] \tag{2.2}$$

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.

$$\pi^*(s) = arg \max_a Q(s,a) \tag{2.3}$$

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.

$$Q^{\pi^*}(s,a) = r + \gamma \max_{a'} Q(s', a') \tag{2.4}$$

2.5.3 Deep Q-Learning

The basic version of Q-learning keeps a lookup table of values $Q(s,a)$ with one entry for every state-action pair. However, the number of possible system states can be very large, thus maintain a table in this case is impractical. There is a very large number of state-action pair and learning can be very difficult. Hence, it is impossible to store the policy in tabular form and it is common to use function approximators (MENACHE *et al.*, 2005). Due to the curse of dimensionality in solving large-scale Markov Decision Process (MDP) problems, it is difficult for traditional approaches to handle practical real-world problems (DULAC-ARNOLD *et al.*, 2015).

Fortunately, Deep Q-Network (DQN) is capable of successfully learning directly from high-dimensional inputs (MNIH *et al.*, 2015). Now, instead of a table to represent the policy, we have a deep neural network as shown in Figure 5.

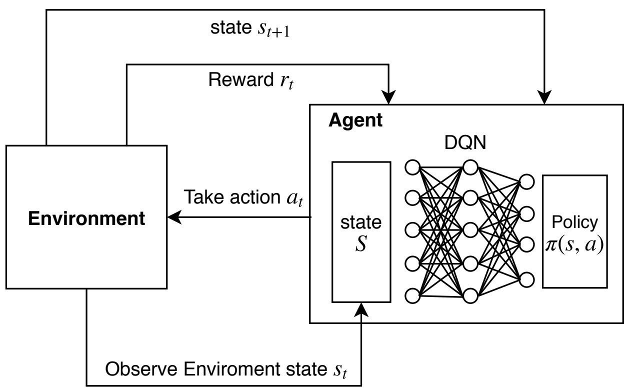


Figura 5 – Reinforcement Learning using a DQN to model the policy.

In Deep Q-learning, at each time step the goal is to minimize the mean squared error (MSE) of the prediction $Q(s,a)$ and the target $Q^{\pi^*}(s,a)$. A deep neural network (DEMUTH *et al.*, 2014) is used to approximate the Q function and gradient descent will minimize the objective function *L*.

$$L = \frac{1}{2}[Q^{\pi^*}(s,a) - Q(s,a)]^2 \tag{2.5}$$

Given a transition $< s,a,r,s' >$, the Q-table update rule in the previous algorithm must be replaced with the following:

1. Do a feedforward pass for the current state s to get predicted Q-values for all actions.
2. Do a feedforward pass for the next state s' and calculate maximum over all network outputs $\max_{a'} Q(s',a')$.
3. Set Q-value target for action a to $r + \gamma \max_{a'} Q(s',a')$ (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Update the weights using backpropagation.

As shown in (MNIH *et al.*, 2013) approximation of Q-values using non-linear 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.

3 AN ADAPTIVE REPLICA PLACEMENT APPROACH FOR DISTRIBUTED KEY-VALUE STORES

This Chapter presents the proposed adaptive approach in detail including the system architecture, replica management, RL formulation and step-by-step placement algorithm. To design this solution, it is addressed the issues of data access skew and load balancing on heterogeneous environments in KVS systems presented in Chapter 1. Also, it is taken into consideration the opportunities and limitations identified in the related work in Chapter **??**. This approach is not to create technology from scratch, but to take advantage of the current DHT properties while aggregating new capacities able to handle its limitations.

3.1 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 6. 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 non-volatile memory device such as a HDD or SSD. In our approach, the partitioning of virtual nodes to storage nodes is based on consistent hashing, an important mechanism that dynamically partitions the entire data over a set of storage 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".

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, i.e., a set of objects, as it is shown in the hash mapping layer of Figure 6. 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

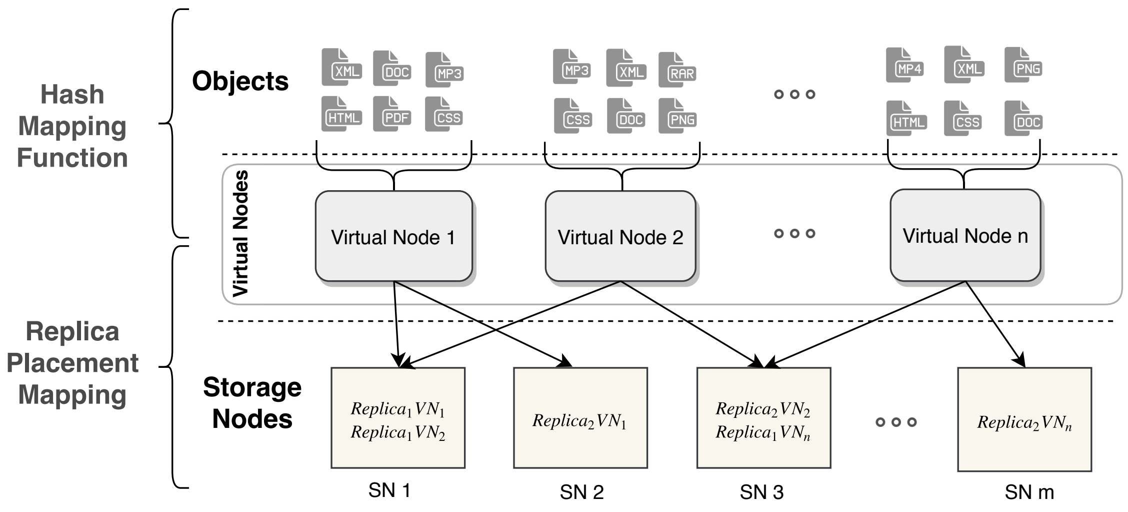


Figura 6 – Objects, virtual nodes and storage nodes mappings

the consistent hashing 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 6. A Replica Placement Scheme (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 6. By modifying the replica placement scheme, the storage system can dynamically manage data through operations of replica creation, migration and deletion. In this work, the RPS is modified in order to achieve load balancing of *Get* operations.

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 replica placement scheme to synchronize all replica units, ensuring consistency between replicas. Every storage node has a copy of the replica placement scheme, thus making possible for each storage node to know exactly which replicas it manages.

The replica placement scheme RPS is a binary matrix of RPS_{sv} cells in which a RPS_{sv} 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 RPS_{sv} 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 $v \in V$, i.e., all virtual nodes have the same replication factor. On the other hand, our replica placement scheme performs data migration by making incremental changes to the replica placement scheme already in-use by a KVS system.

Tabela 6 – 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				

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 6, should be migrated from storage node s_0 to s_1, then the replica placement scheme is modified as in Table 7. RPS_{01} is set to 0 and RPS_{11} is set to 1.

Tabela 7 – 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				

The amount of *Get* requests targeted to each virtual node $v \in V$ according to the hash function is defined by $virt\_node_v\_get$. This works with the consideration that $virt\_node_v\_get$ is spread uniformly among each replica $r \in R$ of a virtual node $v \in V$. It is possible to calculate the amount of *Get* requests submitted to each $s \in S$ according to the Algorithm 1.

Algoritmo 1: Calculate storage node load

```
Input  : Replica Placement Scheme: RPS
Input  : Storage Node id: $id$
Output : Storage Node Load: $s_{id}\_get$
Function SNLoad($id, RPS$):
    $s_{id}\_get = 0$;
    for $v$=0 to $|V|$-1 do
        if $RPS_{id,v}$ == 1 then
            $s_{id}\_get \leftarrow s_{id}\_get + virt\_node_v\_get$;
        end
    end
return $s_{id}\_get$;
```

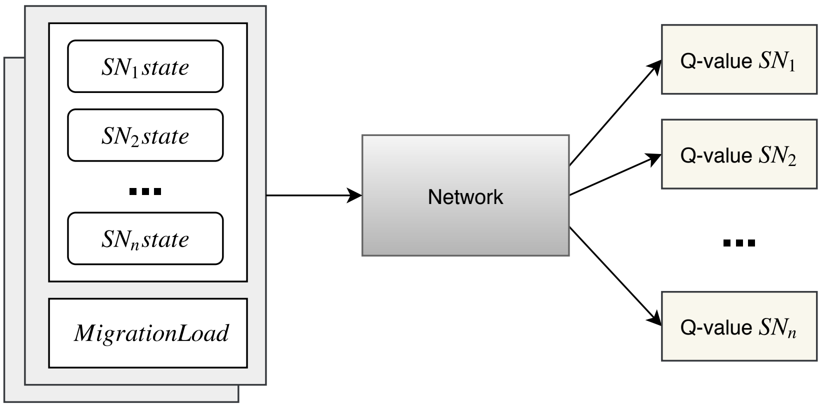


Figura 7 – RL formulation for replica placement in key-value store

3.2 RL Formulation

The RL formulation is the task of defining all the elements and transitions illustrated in Figure 5. But before that, a few requirements that will guide our approach have to defined. The RL formulation has to take a few essential elements into consideration. 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. Then, another requirement is that our model has to adjust itself to the current hardware setup. Also, it must be able to dynamically adapt in case of changes in that regard, building up on previous acquired knowledge.

3.2.1 State Space

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

$$
\begin{aligned}
WS = \langle & SN_1get, SN_1latency, SN_2get, SN_2latency... \\
& SN_nget, SN_nlatency, MigrationLoad\rangle
\end{aligned}
\tag{3.1}
$$

where:

- SN_nget is calculated using Algorithm 1.
- $SN_nlatency$ is the average time it takes storage node n to serve a data request.
- $MigrationLoad$ 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 $MigrationLoad$ and SN_nget shown in equation 3.1. The performance measured 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 pass through the DQN, which outputs a vector of Q-values for each possible action in the given state (see Figure 7). The biggest Q-value of this vector is taken to find the action with highest quality i.e. the best one.

3.2.2 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 deep neural network 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, i.e. the most suitable place for that $MigrationLoad$ is the node that is currently allocating it. To ensure adequate exploration of the state space it is used an $\varepsilon - greedy$ strategy that chooses the best action with probability $1 - \varepsilon$ and selects a random action with probability ε.

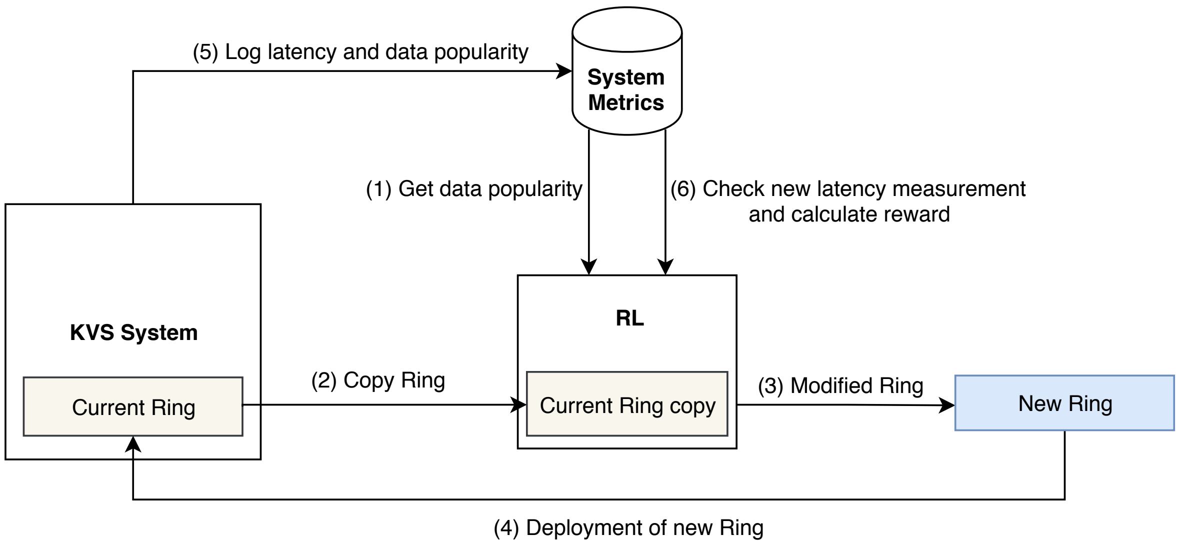


Figura 8 – Workflow of our strategy deployed in the KVS system

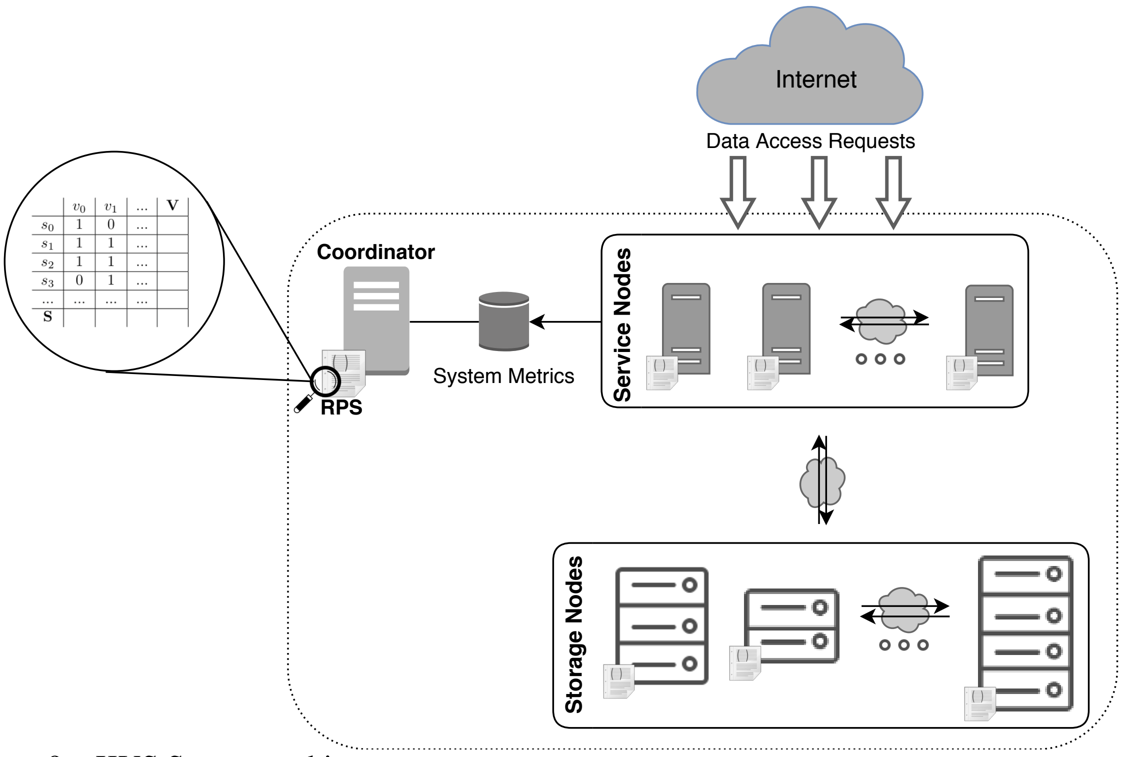
3.2.3 Rewards

The reward function encourages the agents to pursue lower latency across the system and distribute the load according to each storage node performance characteristics. After a performed action a_t at $step_t$, the agent will receive the following reward calculated using:

$$reward_t = latency\_before - latency\_after \tag{3.2}$$

A step is the process of migrating data replica(s) from a storage node to other. The system latency is calculated immediately before action a_t i.e. before the migration in time step t and calculated again after the action is executed (replica(s) migration completed). If it improved or worsen the system overall 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 replica(s) with certain data access characteristics to a specific storage node.

The Figure 8 describes the general workflow of our strategy. First, information on data popularity and on each storage node latency is queried. That information will be formatted into our State representation 3.1. Next, the current ring in the KVS system is copied. The agent then takes an action and modifies the ring. The new ring is then deployed in the KVS system, triggering the migration of data replica(s). After the migration process is completed, the agent receives queries for new latency measurements to evaluate if the action it took reflected on reducing or not the overall latency and by how much.



	v_0	v_1	...	**V**
s_0	1	0	...	
s_1	1	1	...	
s_2	1	1	...	
s_3	0	1	...	
...	...	...	...	
S				

Figura 9 – KVS System architecture

3.3 System Architecture

An overview of our system architecture is shown in Figure 9. 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 Replica Placement Scheme(RPS) (e.g. Table 6) to locate data and redirect requests to appropriate storage nodes.

The service nodes are responsible for accepting HTTP 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, e.g., text files, photos, videos, compressed achieves and son on. The system 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 replica placement scheme 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 i.e. 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 replica placement scheme in order to achieve load-balancing, improve resource utilization and overall system performance(i.e. low latency).

Algoritmo 2: Replica Placement

```
Input :Number of storage nodes: $sn\_count$
Input :Replica count: $r\_count$
 1: Initialize replay memory D;
 2: Initialize action-value function Q with random weights;
 3: while !terminated do
 4:    map$< SN_{id}, SN_{get} >$ p_map = {∅};
 5:    map$< SN_{id}, SN_{latency} >$ l_map = {∅};
 6:    current_ring = get_current_kvs_ring();
 7:    for i=0 to sn_count do
 8:       p_map[i] = SNLoad($i, current\_ring$);
 9:       l_map[i] = get_latency($SN_i latency$);
10:    end for
11:    o_sn = select_overloaded_node(l_map);
12:    latency_before = get_system_latency();
13:    replica_set = select_pop_replica(s)(o_sn, r_count);
14:    s = format_state(p_map[i], l_map[i], replica_set);
15:    sn_destiny = with probability $\varepsilon$ select a random action otherwise select
       $a = arg\max_{a'} Q(s, a')$;
16:    for each $r_{id} \in replica\_set$ do
17:       $current\_ring_{o\_sn, r_{id}} = 0$;
18:       $current\_ring_{sn\_destiny, r_{id}} = 1$;
19:    end for
20:    deploy_ring(current_ring);
21:    wait(timeout);
22:    latency_after = get_system_latency();
23:    step_reward = latency_before - latency_after;
24:    $s'$ = format_state(p_map[i], l_map[i], replica);
25:    store experience $< s, a, r, s' >$ in replay memory D;
26:    sample random transitions $< ss, aa, rr, ss' >$ from replay memory D;
27:    calculate target for each minibatch transition;
28:    if $ss'$ is terminal state then
29:       $tt = rr$;
30:    else
31:       $tt = rr + \gamma \max_{a'} Q(ss', aa')$;
32:    end if
33:    train the Q network using $(tt - Q(ss, aa))^2$ as loss;
34:    s = s';
35: end while
```

The Algorithm 2 describes one step of our strategy. First, the number of *Get* requests directed to each storage nodes is calculated. Then, a copy of the current ring in the KVS system is made. In line 11, the most overloaded storage node *o_sn* in the system (node with the highest value of latency) is selected. Among the replicas that the current *o_sn* manages, *r_count* number of replica(s) with most *Get* requests are selected to be migrated. Next, our RL agent chooses the storage node with the highest quality i.e. the node that will receive data replica(s) and will give the best impact on reducing system overall latency. Then, the current ring copy is modified and deployed. 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 RL agent is updated.

In this chapter, it was explained how the replica management works and how it is possible to migrate data replicas by modifying the replica placement scheme. It was presented in detail the system architecture, RL formulation and step-by-step placement algorithm of the proposed approach. The novel replica placement algorithm proposal is designed to overcome some limitations identified in current implementation of distributed key-value stores.

4 EXPERIMENTAL EVALUATION

In this Chapter, the proposed solution is evaluated. First, the baselines algorithms are presented. Then, it is detailed the experimentation environment and setup used. Finally, it is presented and discussed the results obtained in different scenarios varying workload characteristics.

4.1 Baselines

As the works described in Chapter **??** would take too long to implement in a real KVS deployment and/or there were some missing their algorithm In this dissertation, the proposed approach is compared 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 destiny node in the migration of the replicas. One strategy to select that destiny node is to always pick the storage node with the Lowest Latency (LL) as described in Algorithm 3. Another strategy is to always uniformly pick a Random (RND) storage node as described in Algorithm 4.

Algoritmo 3: LL Replica Placement

```
Input :Replica count: $r\_count$
Input :Total steps: $total\_steps$
1:  steps_count = 0;
2:  while steps_count < total_steps do
3:      current_ring = get_current_kvs_ring();
4:      o_sn = select_overloaded_node(l_map);
5:      latency_before = get_system_latency();
6:      replica_set = select_pop_replica(s)(o_sn, r_count);
7:      sn_destiny = select_sn_with_lowest_latency();
8:      for each $r_{id} \in replica\_set$ do
9:          $current\_ring_{o\_sn,r_{id}} = 0$;
10:         $current\_ring_{sn\_destiny,r_{id}} = 1$;
11:     end for
12:     deploy_ring(current_ring);
13:     wait(timeout);
14:     latency_after = get_system_latency();
15:     log_latency(latency_after - latency_before)
16:     steps_count++;
17: end while
```

Algoritmo 4: RND Replica Placement

```
Input  : Replica count: $r\_count$
Input  : Total steps: $total\_steps$
 1: steps_count = 0;
 2: while steps_count < total_steps do
 3:     current_ring = get_current_kvs_ring();
 4:     o_sn = select_overloaded_node(l_map);
 5:     latency_before = get_system_latency();
 6:     replica_set = select_pop_replica(s)(o_sn, r_count);
 7:     sn_destiny = select_sn_randomly();
 8:     for each $r_{id} \in replica\_set$ do
 9:         $current\_ring_{o\_sn,r_{id}} = 0;$
10:         $current\_ring_{sn\_destiny,r_{id}} = 1;$
11:     end for
12:     deploy_ring(current_ring);
13:     wait(timeout);
14:     latency_after = get_system_latency();
15:     log_latency(latency_after - latency_before)
16:     steps_count++;
17: end while
```

4.2 Environment setup and Workload configuration

To generate the workload it is used Cloud Object Storage Benchmark (COSBench) tool version 0.4.2 rc2 (ZHENG *et al.*, 2013). COSBench has supported many cloud object storage solutions on the market like Swift, Amazon S3, and Ceph thus making easier any future comparison among those cloud object storage solutions. The proposed approach prototype is deployed and tested in a private cloud on Openstack environment. All virtual machines (VM) in the experiments were provided by Openstack Nova and storage devices by Openstack Cinder. It is used two 4 vCPUs, 8 GB RAM and 80 GB storage capacity virtual machines to host the COSBench controller and 2 COSBench drivers. The proxy node is configured with enough resources so it does not act as bottleneck. The proxy node services were running on a Virtual Machine (VM) instance with 16 vCPUs, 16 GB RAM and 160 GB storage capacity. It is deployed 6 VMs instances(storage nodes 1 to 6) with 1 vCPU, 512 MB RAM and 60 GB storage capacity. Each experiment is designed so in the beginning the system is under stress with unbalanced workload among storage nodes. To make the storage nodes heterogeneous, it is configured a limited maximum number of read Input/Output Operations Per Second (IOPS). Since there was three Serial ATA (SATA) hard disk drives(HDD) as backend of Openstack Cinder with roughly 100 IOPS each, it was configured two Cinder disk volumes on each backend HDD. HDD1 had volume1 (20 IOPS) and volume6 (70 IOPS), HDD2 had volume2 (30 IOPS) and volume5 (60 IOPS) and HDD3 had volume3 (40 IOPS) and volume4 (50 IOPS). The coordinator node had an intel i5 CPU, 8 GB RAM and 512 GB HDD.

On every evaluation scenario, it is used similar values as in (ZHENG *et al.*, 2013), we created 100 containers with 100 objects per container resulting a total of 10000 objects(64KB each) in the experiments. Half of the total number of objects in Swift are inserted to simulate an object storage system already in production. A total of 12 clients/workers was configured on all workload configurations. That number was found empirically as the intention is to send as much workload as possible and still achieve 100% success rate i.e. all requests were served successfully. The read/write ratio is fixed at 30:1, same value showed in analysis done in (ATIKOGLU *et al.*, 2012). 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.

In the experiments, it is defined 30 migration steps. Each strategy perform 30 migration steps and in the end the results obtained by each strategy are evaluated. After the COSBench starts, a timeout of 300 seconds is taken before triggering the first migration step, so it is possible to can collect initial metrics(data popularity and latency). After each migration step is done(i.e. data has been migrated and system is synchronized), a timeout of 300 seconds is performed. In each step, it is migrated 1% of the total replicas in the system, as it was noticed that migrating 1% of data replicas had a noticeable impact in the system latency in the scenarios designed. The impact of migration steps are evaluated by using two main metrics. First, it is measured the latency of each storage node. The proposed strategy will pursue the minimization of the sum of all latencies. Second, it is calculated the resource utilization metric by measuring how distant each storage node is from its performance threshold. To calculate those thresholds, the current workload(*Get* req/s) is divided proportionally to storage node performance capacities. For instance, imagine there is 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. The intention is to avoid overloaded/underloaded storage nodes in the cluster.

The proposed approach is implemented using keras-rl (PLAPPERT, 2016). It implements some state-of-the-art deep reinforcement learning algorithms in Python and seamlessly integrates with the deep learning library Keras which is itself a high-level neural network API on top of a deep learning backend like Tensorflow, Theano. Tensorflow (ABADI *et al.*, 2015) was chosen. The implementation of Google DeepMind's DQN agent is used (MNIH *et al.*, 2015). The optimizer used for training is Adam (KINGMA; BA, 2014) 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 (ReLU). 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 (MNIH *et al.*, 2015) and the open source implementation of the DQN agent in keras-rl (PLAPPERT, 2016).

In Figure 10, we can observe the loss function over 40 iterations. Each iteration performs 30 migration steps. We can see that after 25 iterations the error of our model does not change much. Thus, training at that point is stopped. It converges relatively quick. The model is built once and used throughout the experiment section, comparing it to the other strategies.

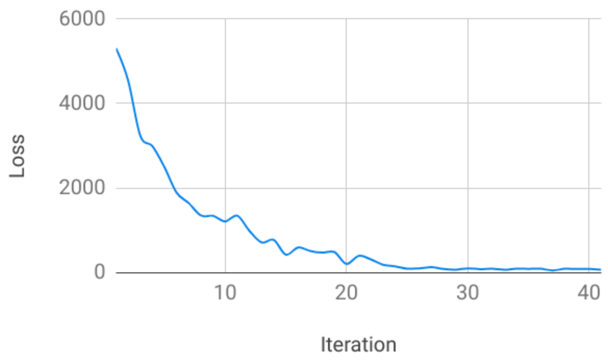


Figura 10 – Loss function progression while training

4.3 Workload Changes

In this experiment section, it is investigated how the proposed approach performs under unseen workload. To do that, it is changed the popularity of data from one workload to the other. In this case, we are interested to see how the proposed strategy deals with data access pattern it has never seen. Empirical studies have shown that requests in a P2P system follow a Zipf distribution (GUPTA *et al.*, 2005). It is varied the zipf parameter to change data popularity. For this experiment subsection, the system is configured with 1024 virtual nodes.

In the first experiment, the zipf parameter is set to 0.1. In Figure 11, 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, i.e. 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.

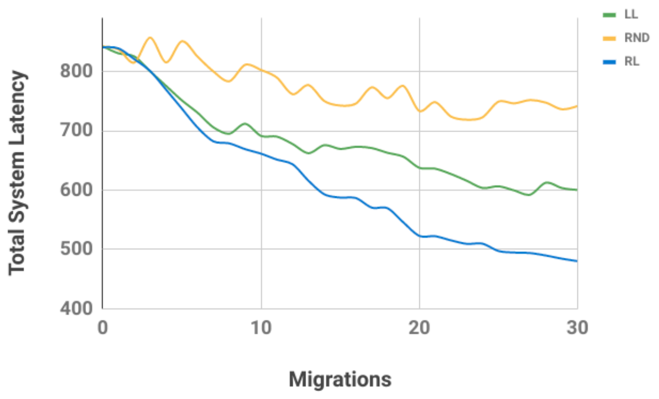


Figura 11 – System overall latency progression with zipf 0.1

We can see in Figure 11 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 destiny node randomly, it often chooses not the best node. As expected LL strategy performed better than RND. It was noticed that, in the beginning, LL and RL usually make similar decisions. LL always chooses the node with the lowest latency, 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 sub-optimal migration actions which causes the latency to increase. Since our strategy learns the impact of placing different replicas(i.e. 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 12, 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, i.e. 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.

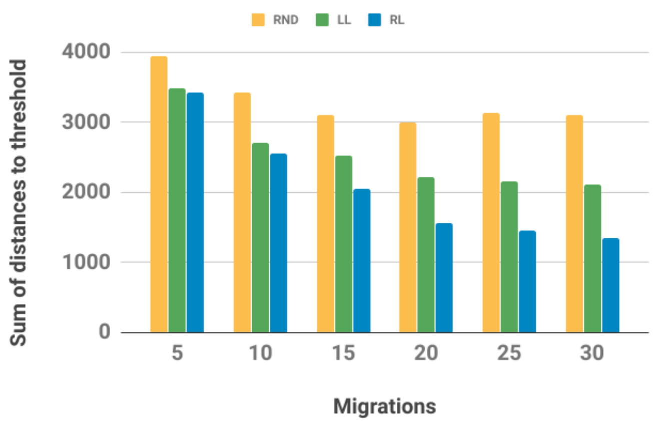


Figura 12 – Resource utilization progression with zipf 0.1

In a second experiment, the zipf parameter is increased to 1.0 which changes data popularity aspects. In this scenario, the majority of the system workload is 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 i.e., making poor decisions in the beginning will have a greater impact in the system latency. In Figure 13, 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 sub-optimal 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.

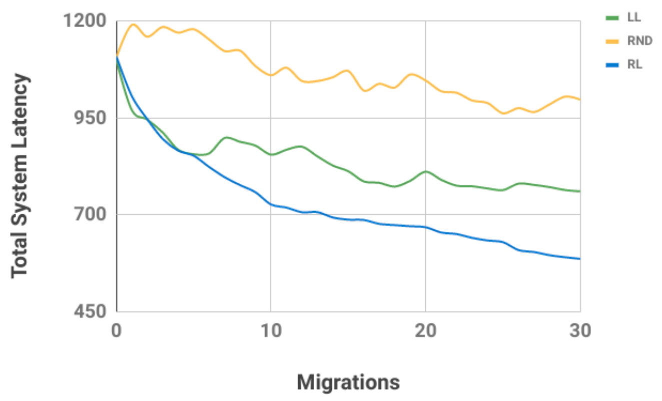


Figura 13 – System overall latency progression with zipf 1.0

Figure 14 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.

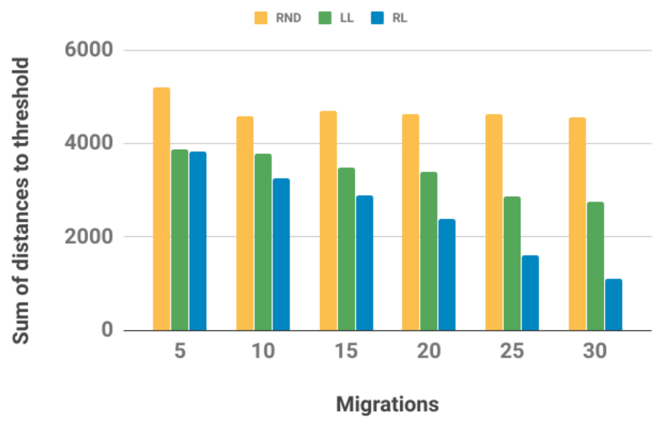


Figura 14 – Resource utilization progression with zipf 1.0

Next, it is changed data popularity again and the zipf parameter is increased to 1.8, causing the workload to concentrate even more on fewer data replicas. The intention is 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 15, 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 moved first. Towards 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.

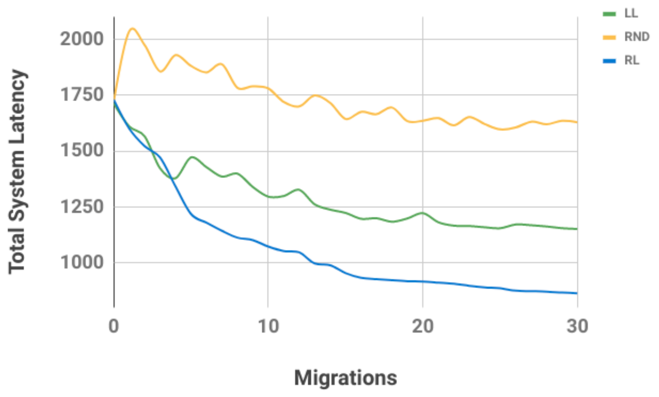


Figura 15 – System overall latency progression with zipf 1.8

Our strategy also had the best results related to load balancing and resource utilization as shown in Figure 16. 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.

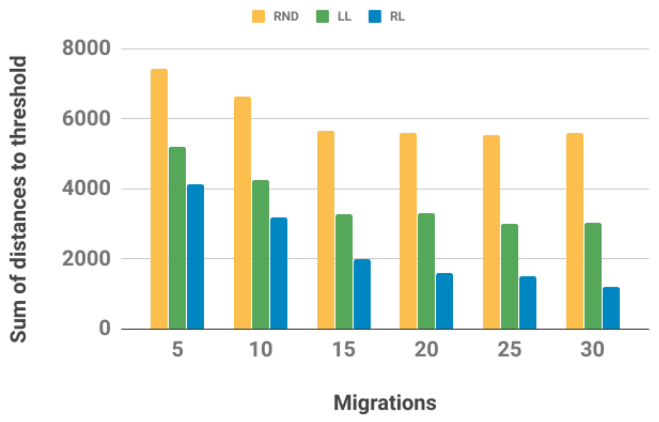


Figura 16 – Results with zipf 1.8

Table 8 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.

	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%

Tabela 8 – Summary of system latency minimization results by strategy

4.4 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, the intention is to observe the efficacy of the proposed strategy when the number of virtual nodes in the system is changed. With more virtual nodes, there are 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, that number is increased to 4096. That means all data files stored in the KVS system are now distributed in 4096 virtual nodes. The zipf parameter is set to 0.1. Figure 17 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.

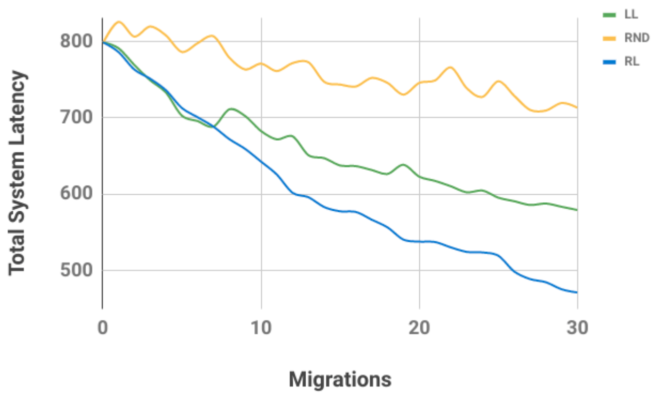


Figura 17 – Results with 4096 virtual nodes

In Figure 18, we can observe the improvements on load balancing achieved by each strategy after 30 migration steps. RND improvements on resource utilization were very limited. LL 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.

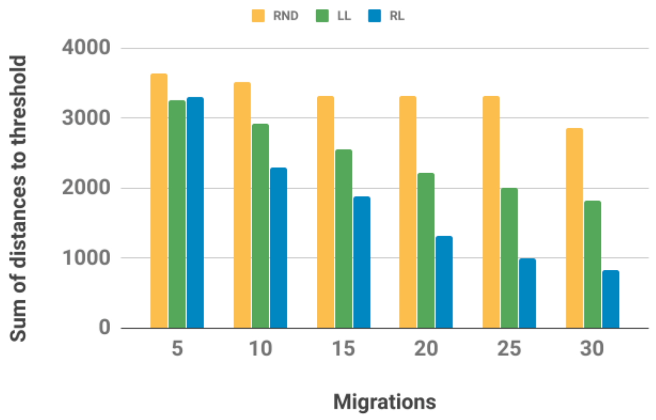


Figura 18 – Comparison of results on load balancing aspects

4.5 Heterogeneity Change

In this experiment, the intention is to test the adaptability aspect of our strategy when the heterogeneity of the storage nodes change. For instance, the system administrator might upgrade storage nodes hardware components increasing its performance capacity. To simulate that scenario, it is increased by 10% the IOPS performance of the storage nodes 1, 3, 5. For this experiment subsection, the system is configured with 1024 virtual nodes and their access rate was chosen using zipf 0.1. In this experiment, the model built in the beginning is loaded and it is given some time so it adjust i.e. do additional training until the loss value stabilizes which took generally seven iterations. After that, the adjusted model was deployed again and evaluated it. In Figure 19 shows the latency measurements obtained along 30 migrations for all strategies. As LL does not take into consideration heterogeneity aspects(see Algorithm 3), it ends up making inferior migrations decisions which limited the total latency improvement. In the end, LL achieved 28.46% overall latency reduction. RND minimized the latency by only 10.76% as it take random actions. Our strategy made similar placement decisions compared to LL for the first ten 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.

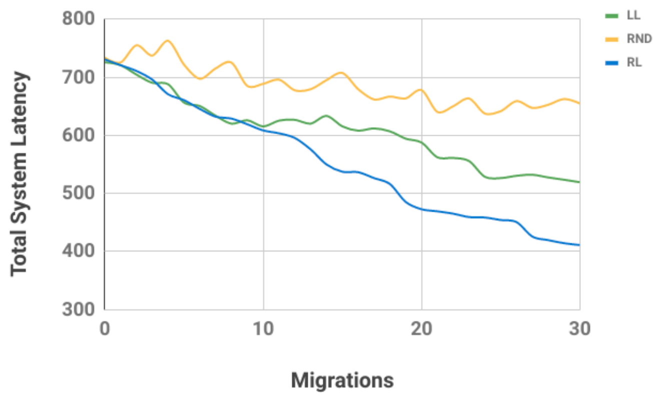


Figura 19 – System overall latency progression after changing storage nodes performance capacities

Concerning load balancing and resource utilization aspects, LL and our RL strategy achieved close results after five and ten migration steps as seen in Figure 20. However, our strategy achieved greater improvement after 30 migration steps, distributing the load accordingly to the newer storage nodes performance capacities. RND as expected had the worst performance in this aspect.

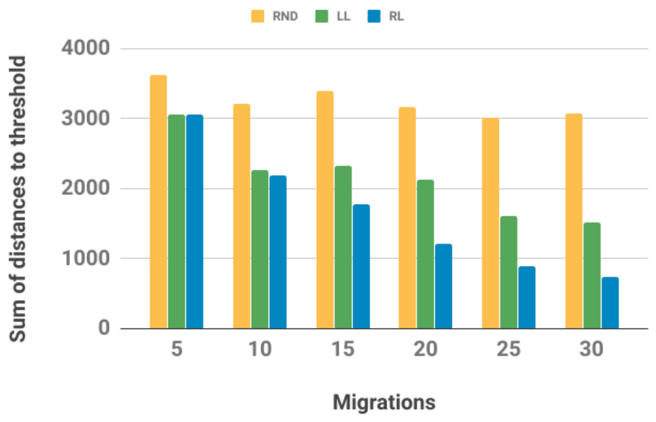


Figura 20 – Results after improving some storage nodes performance capacities

In this chapter, the efficacy of the proposed approach was verified under different scenarios of data access skew. Also, the adaptability aspect of the strategy was tested and the results shown that it can indeed make good replica placement decisions when storage backed performance characteristics change. More importantly, it was shown the potential of using deep reinforcement learning technique to manage the location of data replicas in a distributed key-value store.

5 CONCLUSION AND FUTURE WORK

In this chapter, the conclusion presents what are the most important ideas concluded from this work and the future work indicates what are the new challenges when extending the research.

5.1 Conclusion

This dissertation 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, the proposed strategy is designed to generalize and perform well in different workload patterns. Another important aspect considered in this work is the heterogeneity characteristics of storage nodes. The proposed replica placement approach is structured to work and adapt in a KVS architecture where storage nodes have different performance capacities. Those aspects are crucial during replica placement decisions. Hence, it is leveraged deep reinforcement learning in order to build an adaptable model that assists our strategy on deciding the placement of data replicas. The agent model learns the impact of migrating data replicas, with different characteristics, to storage nodes with different performance capacities.

In this work, it is answered all three questions regarding the migration and placement decision. Also, the hypothesis stated in section 1 is verified. A comparison between the proposed approach and the baselines is done. The experiments tested the efficacy in different scenarios of workloads and storage nodes configurations. In the experiment results, it was analyzed the performance of the proposed approach concerning overall system latency minimization, load balancing, and resource utilization aspects. The adaptive replica placement developed achieved excellent results, and it was very consistent throughout all different scenarios.

5.2 Future Work

New research opportunities come out of results obtained in this work. Considering availability and bandwidth aspects in the replica placement decision is an open issue we want to address. We plan to analyze the impact of different deep neural network 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 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.

REFERÊNCIAS

ABADI, M.; AGARWAL, A.; BARHAM, P.; BREVDO, E.; CHEN, Z.; CITRO, C.; CORRADO, G. S.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; GOODFELLOW, I.; HARP, A.; IRVING, G.; ISARD, M.; JIA, Y.; JOZEFOWICZ, R.; KAISER, L.; KUDLUR, M.; LEVENBERG, J.; MANé, D.; MONGA, R.; MOORE, S.; MURRAY, D.; OLAH, C.; SCHUSTER, M.; SHLENS, J.; STEINER, B.; SUTSKEVER, I.; TALWAR, K.; TUCKER, P.; VANHOUCKE, V.; VASUDEVAN, V.; VIéGAS, F.; VINYALS, O.; WARDEN, P.; WATTENBERG, M.; WICKE, M.; YU, Y.; ZHENG, X. **TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems**. 2015. Disponível em: https://www.tensorflow.org/. Acesso em: 2018-08-12.

AMAZON. **Amazon Elastic Block Store**. 2017. Disponível em: https://aws.amazon.com/ebs. Acesso em: 2017-04-03.

ARMBRUST, M.; FOX, A.; GRIFFITH, R.; JOSEPH, A. D.; KATZ, R.; KONWINSKI, A.; LEE, G.; PATTERSON, D.; RABKIN, A.; STOICA, I. *et al.* A view of cloud computing. **Communications of the ACM**, ACM, New York, NY, USA, v. 53, n. 4, p. 50–58, 2010.

ATIKOGLU, B.; XU, Y.; FRACHTENBERG, E.; JIANG, S.; PALECZNY, M. Workload analysis of a large-scale key-value store. *In*: **Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems**. New York, NY, USA: ACM, 2012. (SIGMETRICS '12), p. 53–64. ISBN 978-1-4503-1097-0.

AZAGURY, A.; DREIZIN, V.; FACTOR, M.; HENIS, E.; NAOR, D.; RINETZKY, N.; RODEH, O.; SATRAN, J.; TAVORY, A.; YERUSHALMI, L. Towards an object store. *In*: IEEE. **Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference**. [*S.l.*], 2003. p. 165–176.

BOHN, C. A.; LAMONT, G. B. Load balancing for heterogeneous clusters of pcs. **Future Generation Computer Systems**, [*S.l.*], v. 18, n. 3, p. 389 – 400, 2002. ISSN 0167-739X. Cluster Computing.

CANONICAL. **Canonical**. 2015. Disponível em: https://insights.ubuntu.com/2015/05/18/what-are-the-different-types-of-storage-block-object-and-file/. Acesso e m: 2017-04-03.

CAVALCANTE, D. M.; FARIAS, V. A.; SOUSA, F. R. C.; PAULA, M. R. P.; MACHADO, J. C.; SOUZA, N. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. *In*: INSTICC. **Proceedings of the 8th International Conference on Cloud Computing and Services Science**. [*S.l.*], 2018. p. 440–447. ISBN 978-989-758-295-0.

CHEKAM, T. T.; ZHAI, E.; LI, Z.; CUI, Y.; REN, K. On the synchronization bottleneck of openstack swift-like cloud storage systems. *In*: IEEE. **Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference**. [*S.l.*], 2016. p. 1–9.

CHENG, Y.; GUPTA, A.; BUTT, A. R. An in-memory object caching framework with adaptive load balancing. *In*: **Proceedings of the Tenth European Conference on Computer Systems**. New York, NY, USA: ACM, 2015. (EuroSys '15), p. 4:1–4:16. ISBN 978-1-4503-3238-5.

DECANDIA, G.; HASTORUN, D.; JAMPANI, M.; KAKULAPATI, G.; LAKSHMAN, A.; PILCHIN, A.; SIVASUBRAMANIAN, S.; VOSSHALL, P.; VOGELS, W. Dynamo: Amazon's highly available key-value store.*In* : **Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2007. (SOSP '07), p. 205–220. ISBN 978-1-59593-591-5.

DEMUTH, H. B.; BEALE, M. H.; JESS, O. D.; HAGAN, M. T. **Neural Network Design**. 2nd. ed. USA: Martin Hagan, 2014. ISBN 0971732116, 9780971732117.

DEWAN, H.; HANSDAH, R. A survey of cloud storage facilities.*In* : IEEE. **Services (SERVICES), 2011 IEEE World Congress**. [*S.l.*], 2011. p. 224–231.

DULAC-ARNOLD, G.; EVANS, R.; SUNEHAG, P.; COPPIN, B. Reinforcement learning in large discrete action spaces. **CoRR**. [*S.l.*], abs/1512.07679, 2015.

Eager, D. L.; Lazowska, E. D.; Zahorjan, J. Adaptive load sharing in homogeneous distributed systems. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, SE-12, n. 5, p. 662–675, May 1986. ISSN 0098-5589.

FACTOR, M.; METH, K.; NAOR, D.; RODEH, O.; SATRAN, J. Object storage: The future building block for storage systems.*In* : IEEE. **Local to Global Data Interoperability-Challenges and Technologies, 2005**. [*S.l.*], 2005. p. 119–123.

FAN, B.; LIM, H.; ANDERSEN, D. G.; KAMINSKY, M. Small cache, big effect: Provable load balancing for randomly partitioned cluster services.*In* : **Proceedings of the 2Nd ACM Symposium on Cloud Computing**. New York, NY, USA: ACM, 2011. (SOCC '11), p. 23:1–23:12. ISBN 978-1-4503-0976-9.

FELBER, P.; KROPF, P.; SCHILLER, E.; SERBU, S. Survey on load balancing in peer-to-peer distributed hash tables. **IEEE Communications Surveys & Tutorials**, IEEE, [*S.l.*], v. 16, n. 1, p. 473–492, 2014.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The google file system.*In* : ACM. **ACM SIGOPS operating systems review**. [*S.l.*], 2003. v. 37, n. 5, p. 29–43.

GIBSON, G. A.; METER, R. V. Network attached storage architecture. **Commun. ACM**, ACM, New York, NY, USA, v. 43, n. 11, p. 37–45, nov. 2000. ISSN 0001-0782.

GILBERT, S.; LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. **Acm Sigact News**, ACM, New York, NY, USA, v. 33, n. 2, p. 51–59, 2002.

GONG, C.; LIU, J.; ZHANG, Q.; CHEN, H.; GONG, Z. The characteristics of cloud computing. *In* : **2010 39th International Conference on Parallel Processing Workshops**. [*S. l.*], 2010. p. 275–279. ISSN 0190-3918.

GROSSMAN, R. L.; GU, Y.; SABALA, M.; ZHANG, W. Compute and storage clouds using wide area high performance networks. **Future Generation Computer Systems**, [*S.l.*], v. 25, n. 2, p. 179 – 183, 2009. ISSN 0167-739X. Disponível em: http://www.sciencedirect.com/science/article/pii/S0167739X08001155. Acesso em: 2018-07-08.

GUPTA, A.; DINDA, P. A.; BUSTAMANTE, F. Distributed popularity indices. *In*: **Proceedings of ACM SIGCOMM**. [*S.l.*], 2005.

HE, Q.; LI, Z.; ZHANG, X. Study on cloud storage system based on distributed storage systems. *In*: **Proceedings of the 2010 International Conference on Computational and Information Sciences**. Washington, DC, USA: IEEE Computer Society, 2010. (ICCIS '10), p. 1332–1335. ISBN 978-0-7695-4270-6.

HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, New York, NY, USA, v. 12, n. 3, p. 463–492, 1990.

HONG, I. **Consistent Hashing Algorithm**. 2017. Disponível em: https://ihong5.wordpress.com/2014/08/19/consistent-hashing-algorithm/. Acesso em: 2017-04-03.

JU, J.; WU, J.; FU, J.; LIN, Z.; ZHANG, J. A survey on cloud storage. **JCP**. [*S.l.*], v. 6, n. 8, p. 1764–1771, 2011.

KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. P. Reinforcement learning: A survey. **Journal of Artificial Intelligence Research**. [*S.l.*], v. 4, p. 237–285, 1996. Disponível em: http://people.csail.mit.edu/lpk/papers/rl-survey.ps. Acesso em: 2018-10-11.

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. *In*: **Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing**. New York, NY, USA: ACM, 1997. (STOC '97), p. 654–663. ISBN 0-89791-888-6. Acesso em: 2018-10-11.

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. *In*: ACM. **Proceedings of the twenty-ninth annual ACM symposium on Theory of computing**. [*S.l.*], 1997. p. 654–663.

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. *In*: ACM. **Proceedings of the twenty-ninth annual ACM symposium on Theory of computing**. [*S.l.*], 1997. p. 654–663.

KHATTAR, R. K.; MURPHY, M. S.; TARELLA, G. J.; NYSTROM, K. E. **Introduction to Storage Area Network, SAN**. [*S.l.*]: IBM Corporation, International Technical Support Organization, 1999.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. **CoRR**. [*S.l.*], abs/1412.6980, 2014.

KUBAT, M. Reinforcement learning by ag barto and rs sutton, mit press, cambridge, ma 1998, isbn&puncsp; 0-262-19398-1. **Knowl. Eng. Rev.**, Cambridge University Press, New York, NY, USA, v. 14, n. 4, p. 383–385, dez. 1999. ISSN 0269-8889.

LAKSHMAN, A.; MALIK, P. Cassandra: A decentralized structured storage system. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 44, n. 2, p. 35–40, abr. 2010. ISSN 0163-5980.

LUA, E. K.; CROWCROFT, J.; PIAS, M.; SHARMA, R.; LIM, S. A survey and comparison of peer-to-peer overlay network schemes. **Commun. Surveys Tuts.**, IEEE Press, Piscataway, NJ, USA, v. 7, n. 2, p. 72–93, abr. 2005. ISSN 1553-877X.

MAKRIS, A.; TSERPES, K.; ANAGNOSTOPOULOS, D.; ALTMANN, J. Load balancing for minimizing the average response time of get operations in distributed key-value stores. *In*: IEEE **Networking, Sensing and Control (ICNSC), 2017 IEEE 14th International Conference on**. [*S.l.*], 2017. p. 263–269.

MELL, P. M.; GRANCE, T. **SP 800-145. The NIST Definition o f C l o u d Computing**. Gaithersburg, MD, United States, 2011.

MENACHE, I.; MANNOR, S.; SHIMKIN, N. Basis function adaptation in temporal difference reinforcement learning. **Annals of Operations Research**, [*S.l*], v. 134, n. 1, p. 215–238, Feb 2005. ISSN 1572-9338. Disponível em: https://doi.org/10.1007/s10479-005-5732-z. Acesso em: 2018/08/08.

Mesnier, M.; Ganger, G.; Riedel, E. Object-based storage: pushing more functionality into storage. **IEEE Potentials**, [*S.l.*], v. 24, n. 2, p. 31–34, April 2005.

MESNIER, M.; GANGER, G. R.; RIEDEL, E. Object-based storage. **Comm. Mag.**, IEEE Press, Piscataway, NJ, USA, v. 41, n. 8, p. 84–90, ago. 2003. ISSN 0163-6804.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; GRAVES, A.; ANTONOGLOU, I.; WIERSTRA, D.; RIEDMILLER, M. A. Playing atari with deep reinforcement learning. **CoRR**, [*S.l.*], abs/1312.5602, 2013. Disponível em: http://arxiv.org/abs/1312.5602. Acesso em: 2018/09/08.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; RUSU, A. A.; VENESS, J.; BELLEMARE, M. G.; GRAVES, A.; RIEDMILLER, M.; FIDJELAND, A. K.; OSTROVSKI, G.; PETERSEN, S.; BEATTIE, C.; SADIK, A.; ANTONOGLOU, I.; KING, H.; KUMARAN, D.; WIERSTRA, D.; LEGG, S.; HASSABIS, D. Human-level control through deep reinforcement learning. **Nature**, Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved., [*S.l.*], v. 518, n. 7540, p. 529–533, fev. 2015. ISSN 00280836. Disponível em: http://dx.doi.org/10.1038/nature14236. Acesso em: 2018/09/08.

NAIOUF, M. R.; GIUSTI, L. C. D.; CHICHIZOLA, F.; GIUSTI, A. E. D. Dynamic load balancing on non-homogeneous clusters. *In*: MIN, G.; MARTINO, B. D.; YANG, L. T.; GUO, M.; RÜNGER, G. (Ed.). **Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 65–73. ISBN 978-3-540-49862-9.

PAIVA, J. a.; RODRIGUES, L. On data placement in distributed systems. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 49, n. 1, p. 126–130, jan. 2015. ISSN 0163-5980.

PLAPPERT, M. **keras-rl**. GitHub, 2016. Disponível em: https://github.com/keras-rl/keras-rl. Acesso em: 2018-10-12.

SHEPLER, S.; CALLAGHAN, B.; ROBINSON, D.; THURLOW, R.; BEAME, C.; EISLER, M.; NOVECK, D. **Network File System (NFS) Version 4 Protocol**. United States: RFC Editor, 2003.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The hadoop distributed file system. *In*: **Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)**. Washington, DC, USA: IEEE Computer Society, 2010. (MSST '10), p. 1–10. ISBN 978-1-4244-7152-2.

STOICA, I.; MORRIS, R.; KARGER, D.; KAASHOEK, M. F.; BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. **ACM SIGCOMM Computer Communication Review**, ACM, [*S.l.*], v. 31, n. 4, p. 149–160, 2001.

SUTTON, R. S.; BARTO, A. G. **Introduction to Reinforcement Learning**. 1st. ed. Cambridge, MA, USA: MIT Press, 1998. ISBN 0262193981.

SWIFT. **OpenStack Swift**. 2017. Disponível em: https://docs.openstack.org/developer/swift/. Acesso em: 2017-04-03.

TERRY, D. B.; DEMERS, A. J.; PETERSEN, K.; SPREITZER, M. J.; THEIMER, M. M.; WELCH, B. B. Session guarantees for weakly consistent replicated data. *In*: IEEE. **Proceedings of 3rd International Conference on Parallel and Distributed Information Systems**. [*S.l.*], 1994. p. 140–149.

TESAURO, G.; JONG, N. K.; DAS, R.; BENNANI, M. N. A hybrid reinforcement learning approach to autonomic resource allocation. *In*: **Proceedings of the 2006 IEEE International Conference on Autonomic Computing**. Washington, DC, USA: IEEE Computer Society, 2006. (ICAC '06), p. 65–73. ISBN 1-4244-0175-5.

TOLIA, N.; ANDERSEN, D. G.; SATYANARAYANAN, M. Quantifying interactive user experience on thin clients. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 39, n. 3, p. 46–52, 2006.

VAQUERO, L. M.; RODERO-MERINO, L.; CACERES, J.; LINDNER, M. A break in the clouds: Towards a cloud definition. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 39, n. 1, p. 50–55, dez. 2008. ISSN 0146-4833.

VIOTTI, P.; VUKOLIĆ, M. Consistency in non-transactional distributed storage systems. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 49, n. 1, p. 19:1–19:34, jun. 2016. ISSN 0360-0300.

VOGELS, W. Eventually consistent. **Queue**, ACM, New York, NY, USA, v. 6, n. 6, p. 14–19, 2008.

WANG, Z.; CHEN, H.; FU, Y.; LIU, D.; BAN, Y. Workload balancing and adaptive resource management for the swift storage system on cloud. **Future Generation Computer Systems**, Amsterdam, The Netherlands, v. 51, p. 120 – 131, 2015. ISSN 0167-739X. Special Section: A Note on New Trends in Data-Aware Scheduling and Resource Provisioning in Modern HPC Systems.

WEIL, S. A.; BRANDT, S. A.; MILLER, E. L.; LONG, D. D. E.; MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. *In*: **Proceedings of the 7th Symposium on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2006. (OSDI '06), p. 307–320. ISBN 1-931971-47-1.

WU, J.; PING, L.; GE, X.; WANG, Y.; FU, J. Cloud storage as the infrastructure of cloud computing. *In*: IEEE. **Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference**. [*S.l.*], 2010. p. 380–383.

Yeo, S.; Lee, H. Using mathematical modeling in provisioning a heterogeneous cloud computing environment. **Computer**, [*S.l.*], v. 44, n. 8, p. 55–62, Aug 2011. ISSN 0018-9162.

ZHENG, Q.; CHEN, H.; WANG, Y.; ZHANG, J.; DUAN, J. Cosbench: Cloud object storage benchmark. *In*: **Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering**. New York, NY, USA: ACM, 2013. (ICPE '13), p. 199–210. ISBN 978-1-4503-1636-1.

ZWOLENSKI, M.; WEATHERILL, L. *et al.* The digital universe: Rich data and the increasing value of the internet of things. **Australian Journal of Telecommunications and the Digital Economy**, Telecommunications Association, [*S.l.*], v. 2, n. 3, p. 47, 2014.