M.Sc. Engg. (CSE) Thesis

A Prediction Based Replica Selection Strategy for Reducing Tail Latency in Distributed Systems

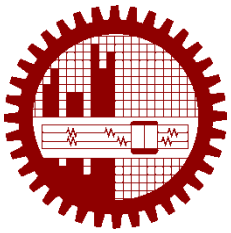
Submitted by

Santa Maria Shithil

0417052112

Supervised by

Dr. Muhammad Abdullah Adnan



Submitted to

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

July 2022

Candidate's Declaration

I, do, hereby, certify that the work presented in this thesis, titled, "A Prediction Based Replica Selection Strategy for Reducing Tail Latency in Distributed Systems", is the outcome of the investigation and research carried out by me under the supervision of Dr. Muhammad Abdullah Adnan, Professor, Department of CSE, BUET.

I also declare that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Santa Maria Shithil

Santa Maria Shithil

0417052112

i

The thesis titled "**A Prediction Based Replica Selection Strategy for Reducing Tail Latency in Distributed Systems**", submitted by Santa Maria Shithil, Student ID 0417052112, Session April 2017, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfilment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents on July 30, 2022.

Board of Examiners

1. _____

Dr. Muhammad Abdullah Adnan — Chairman
Professor — (Supervisor)
Department of CSE, BUET, Dhaka

2. _____

Dr. Mahmuda Naznin — Member
Professor and Head — (Ex-Officio)
Department of CSE, BUET, Dhaka

3. _____

Dr. Mahmuda Naznin — Member
Professor
Department of CSE, BUET, Dhaka

4. _____

Dr. Rifat Shahriyar — Member
Professor
Department of CSE, BUET, Dhaka

5. _____

Dr. Tushar Kanti Saha — Member
Professor — (External)
Department of CSE
Jatiya Kabi Kazi Nazrul Islam University, Mymensingh

ii

Acknowledgement

Foremost, I am thankful to Almighty Allah for his blessings for the successful completion of my thesis. I would like to express my heartiest gratitude, profound indebtedness and deep respect to my supervisor, Dr. Muhammad Abdullah Adnan, Professor, Dept. of CSE, BUET, Dhaka, Bangladesh, for his constant supervision, affectionate guidance and great encouragement and motivation. His keen interest on the topic and valuable advices throughout the study was of great help in completing thesis.

I would also want to thank the members of my thesis committee for their valuable suggestions. I thank Dr. Mahmuda Naznin, Dr. Rifat Shahriyar and specially the external member Dr. Tushar Kanti Saha.

I am especially grateful to Department of Computer Science and Engineering (CSE) of Bangladesh University of Engineering and Technology (BUET) for providing their support during the thesis work. My sincere thanks goes to CSE Office staffs for providing logistic support to me to successfully complete the thesis work.

Finally, I would like to thank my family and all of those who supported me for their appreciable assistance, patience and suggestions during the course of my thesis.

Dhaka
July 30, 2022

Santa Maria Shithil
0417052112

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

The deployment of modern applications in geo-distributed systems results in performance fluctuation which is a consequence of long-tail latency. To deliver high-quality services these applications always strive to adapt to the changing situation and an appropriate replica selection strategy is one efficient way to achieve this. Several replica selection strategies have already been developed but none of them are efficient enough to reduce tail latency and to adapt to the dynamic environment of the geo-distributed systems. Hence for developing a more efficient replica selection strategy first we conduct some extensive experiments to analyze and evaluate two popular state-of-the-art replica selection strategies of key-value store systems: Cassandra dynamic snitch and C3. After analyzing all the experimental results we develop a prediction-based replica selection strategy for a locally distributed system and implement it on Cassandra 3.0. For evaluation, we test Cassandra dynamic snitch, C3, and our proposed strategy on a locally distributed 15 nodes Cassandra cluster. Experimental results show that our proposed algorithm outperforms both C3 and dynamic snitch. Then we deploy a geo-distributed 15 nodes Cassandra cluster on Amazon EC2 and conducted some experiments to analyze how our developed replica selection strategy performs in geo-distributed systems. Though our strategy outperforms both C3 and dynamic snitch in locally distributed systems however its performance is not that much promising in the geo-distributed systems. Analyzing the experimental outcomes, finally, we extend the prediction model of the replica selection strategy that is designed for locally distributed system and designed a prediction-based replica selection strategy for the geo-distributed systems. In this paper, we present the design and implementation of the prediction-based replica selection strategy for reducing tail latency in geo-distributed systems. We have meticulously designed the proposed strategy to adapt to the dynamic behavior of the distributed system. For evaluating its effectiveness in reducing tail latency and improving the overall throughput we perform some extensive experiments in a 15 nodes Cassandra Cluster that is deployed on Amazon EC2. For generating test datasets and workloads we use industry-standard Yahoo Cloud Serving Benchmark (YCSB). Our experimental results show that the proposed strategy not only reduces tail latency but also increases the overall throughput of the geo-distributed systems. Apart from that, our proposed strategy also increases the performance of the request reissue and request duplication strategies.

Chapter 1

Introduction

This thesis presents a prediction-based replica selection strategy for reducing tail latency in the geo-distributed systems. It critically analyzes and evaluates state-of-the-art replica selection strategies and first presents a replica selection strategy for reducing tail latency in the locally distributed systems. After critically analyzing the performance of the first designed strategy this paper then presents a more efficient replica selection strategy for reducing tail latency in the geo-distributed systems and analyzes results of carefully-designed experiments with the industry-standard data sets.

In this introductory chapter, the rationale for this research is explained and an overview of the thesis is provided. The chapter starts off by presenting the motivations and objectives of the research. Then the contributions of the research are briefly outlined. Finally, an overview of the organization of the thesis is provided.

Motivation

The deployment of modern applications are built on a geo-distributed system to achieve their important requirements such as high availability, reliability, and scalability. Although managing applications in a geo-distributed manner has several advantages, applications may suffer from unexpectedly high response time fluctuations which may result in severe degradation on user experiences which in turn affects the revenue of the service providers. There may be several factors but long tail latency is considered as one of the main reasons for response time fluctuations [7], [8], [9], [10]. In a geo-distributed system a client's request is split into several subrequests and spread out through the geographically distributed nodes. In such a scenario the response time of the request depends on the longest subrequest response time and this is defined as tail latency. Long tail latency occurs when one of the nodes where subrequests are spread out is not responding over other nodes [11], [12], [13]. This may happen due to contentions among shared resources or the unbalanced load among the nodes.

A study [14] shows that long-tail latency occurs when the higher percentiles (90th percentile or above) began to have values that are greater than the average. These values can be magnitudes greater than the average whereas a slite increase in latency greatly reduces the revenue of the service providers [15]. Several mechanisms have been developed to reduce tail latency in the geo-distributed systems including request reissuing techniques [16] [17] [18] [19], request duplication techniques [20] [21] [22] [23], and replica selection strategies [24] [25] [26] [27]. Though request reissuing and request duplication techniques can reduce the tail latency however if multiple reissued or duplicated requests select a single service that is already suffering from a long-tail latency problem then the situation becomes scarier [28]. On the other hand, [25] shows that the replica selection strategies have a direct impact on the tail latency of the distributed systems and an efficient replica selection mechanism can sharply reduce the tail latency problem. Moreover, when the request reissuing and request duplication techniques are designed on top a more efficient replica selection strategy then their performance also improves.

Among several services used for building applications in distributed system, storage plays a vital role in the tail latency's of overall services. An efficient replica selection strategy can reduce the long tail latency of the distributed storage systems [25] specifically of the key-value stores that are the dominant class of storage solutions in this context and are the focus of this research. In a key-value store system, each piece of information is replicated throughout several nodes for fault-tolerance. When a client requests a read operation one of the replicas is selected for serving the request and if the slowest replica is selected then the request may suffer from long tail latency problem. Now it is possible to reduce the problem of long-tail latency by selecting the replica more intelligently specifically when a request attempting to access value for a given key, this request can be directed to a replica that is expected to serve the request with the smallest time.

Selecting the best replica is challenging enough as the performance and loads of the nodes fluctuate over time and many of which vary even at sub-second timescales [8]. To adapt the replica selection strategy to the performance fluctuation and load oscillation of the nodes several factors need to consider including service time, queue size, and disk I/O operations rate (IOPS) [29]. Apart from these, in geo-distributed systems, the Round Trip Time (RTT) of the network has also a dominant effect on the tail latency of the systems [30]. However, still, now none of the state-of-the-art replica selection strategies considered all the mentioned factors. The Cassandra dynamic snitch [24] considers only latency; C3 [25] and Heron [26] consider round trip time, service time and queue size; and L2 [27] considers only round trip time for calculating a score based on which the best replica is selected for routing the read request.

None of the replica selection strategies discussed above can meet the challenges of geo-distributed systems. Hence to meet all the challenges we proposed a prediction-based model considering all the important factors related to the replica selection. Dynamic snitch is designed based on latency only and does not take into account the load across the node for calculating score which is important for adapting to time-varying performance of geo-distributed systems. To address

this problem, L2 designed a replica selection strategy in which each node selects the server to which it has the lowest number of outstanding keys which is a measure of load across the cluster. However, this strategy uses only local information that does not necessarily reflect the actual scenario of the cluster. Moreover, in reality, workloads get skewed and access patterns change over time [31]. Hence, strategies that are designed based on only local information are not efficient enough to adapt to the dynamic nature of the geo-distributed systems and any indifferent design decision may result in herd behavior. Maybe one possible solution is to use global load information. However, it is unrealistic to assume that all the nodes of a geo-distributed system have up-to-date information. On one hand, the global information may be updated periodically or the time delay for a replica request to reach the selected node is long enough that the load information is out of data as long as the replica request arrives at the node [32]. On the other hand, obtaining up-to-date global information requires complex system design and increases computational complexity [32]. To reduce the complexity as well as to make the replica selection strategy more sensitive to the dynamic nature of the geo-distributed systems C3 and Heron calculate a concurrency compensation value based on which the final score is calculated that is used for selecting the best replica for routing the read request. However, the calculation of concurrency compensation is based on just an intuition that does not reflect the actual scenario of the cluster and can degrade the overall performance of the systems. For a better understanding of the actual scenario of the system, in this paper first we proposed a prediction-based replica selection strategy that performs better than the other strategies in the locally distributed systems but not efficient enough for the geo-distributed systems. Finally, considering the discussed factors and issues we meticulously designed a new prediction-based replica selection strategy for reducing tail latency in the geo-distributed systems that will be able to adapt to the dynamic nature of the systems. Our proposed replica selection strategy reduces the tail latency of the system as well as increases the overall throughput.

Performance Fluctuations are the Norms in the Distributed Systems

Performance fluctuations are the norm in the geo-distributed systems and a study [13] showed that a major cause of high-performance fluctuation in the geo-distributed systems is latency variation across machines and time. J. Dean and L. A. Barroso in [8] listed several sources of latency variability including shared resources, background activities, global resource sharing, garbage collection, and queuing delay. Apart from these, virtualization has also a greater impact on long-tail latency in the latency distribution that is illustrated in [7].

- Local Resource Sharing: Applications running on a single machine may share local resources such as s CPU cores, processor caches, memory bandwidth, and network

bandwidth. Several requests may contend for accessing those resources which in turn may result in performance fluctuations of the system.

- Global Resource Sharing: In distributed systems, applications run on different machines and they share several global resources such as network switches, shared file systems, etc. Hence contention among the applications for these global resources is very common and this contention leads to performance fluctuations. The situation becomes more severe in geo-distributed systems where machines in a cluster are spread across a wide geographical area.

- Background Activities: Background activities such as data reconstructions in distributed file systems, periodic log compaction in log oriented-storage systems, periodic garbage collection activities in garbage-collector language may significantly increase CPU, disk, and network load. This increased load introduces a latency spike in the distributed systems which in turn results in overall performance fluctuations

- Queuing Delay: In distributed systems, several layers of queuing exist among the servers which have a significant impact on the performance fluctuations of the systems. Queuing delay may be caused by delay at the originating and intermediate switches, buffers at the servers, ready queue for CPU scheduling. These delays amplify latency variability which has a greater impact on performance fluctuations of the distributed systems.

- Virtualization: A study [7] showed that the virtualization technique used in the commercial clouds such as Amazon EC2 has a greater impact on the tail latency of the distributed systems. The 99.9th percentile RTTs of amazon EC2 are upto four times longer than the dedicated data centers.

Performance fluctuations are the norm in the distributed systems hence it is important that the systems must be able to adapt to the changing situation. However, this fluctuations can be mitigated by reducing the tail latency of the systems. Hence, in this thesis, our main purpose is to reduce the tail latency of the distributed systems that in turn will reduce performance fluctuations and increase the revenue of the service providers. We also design the system in such a way that will be able to adapt to the changing environment.

Effects of Tail Latency in Distributed Systems

Generally, communications among machines in a data center take few microseconds but very few may take a few milliseconds once every while. The communications that take few milliseconds usually belong to the 90th percentile or higher latencies. Long-tail latencies occur when these higher percentiles began to have values and can be magnitudes greater than the average. Fig. 1.1

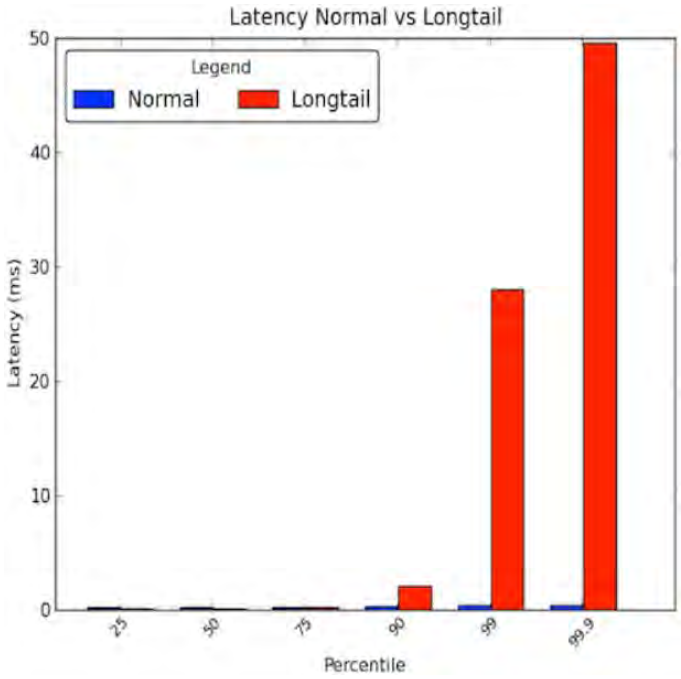


Figure 1.1: Difference between good latency distribution and long tail latency distribution [1]

shows the comparison between doof latency distribution and long tail latency distribution and we can see that $99th$ percentile latency is 30% worse than the median latency whereas the $99.9th$ percentile latency is 50% worse than the mean latency.

A 99th percentile latency of 30ms means 1 of every 100 requests experience 30ms of delay. For high traffic websites, like LinkedIn, for 1 million page views per day about 10000 requests face this delay. Moreover, in distributed systems a single request may generate multiple downstream requests. If multiple downstream requests hit a service suffering from long tail latency problem then the problem becomes scarier. This problem becomes more severe in the geo-distributed systems. Again, the tail latency of a geo-distributed system has direct impact on the applications deployed in it. It increases the applications load time which has adverse effect on the user experience. A recent research [33] reveals that about 40% users abandon a site if it takes more than $3s$ to load. This in turn impacts the revenue of the service providers. At Amazon, $100ms$ increase in latency reduces it's revenue by 1% [34]. Similarly, at Bing, a $2s$ slowdown reduces the revenue per user by 4.3% [34]. On the contrary, when latency spike falls flat, the revenue increases. At Shopzilla, a latency reduction from $7s$ to $2s$ increased page views by 25% and revenue by 7% [34].

The Need for Effective Replica Selection in Geo-Distributed Systems

There are mainly three classes of state-of-the-art strategies are available for reducing tail latency in geo-distributed systems: 1) request duplication strategies 2) request reissuing strategies and 3) replica selection strategies. In request duplication strategies [20] [21] [22] [23], multiple copies of requests are dispatched to multiple servers for every user's read request. This technique is usually applied to systems with low system utilization. The main advantage of such techniques is that all copies of a request have an equal chance to respond before the tail latency deadline as they are sent at the same time [35]. However, its disadvantage is that it's not efficient enough for systems with moderate or higher utilization [35]. Moreover, selecting poorly performing servers for routing the copies of the read request makes the problem more severe [25]. For systems that run at higher system utilization, one alternative approach is to use request reissuing techniques [16] [17] [18] [19] where a second copy of the request is dispatched after a specified delay d. The advantage of such a technique is that we can save the cost of request reissue when the response of the first request is fast anyway [35]. However, if the delay d is too large, then the reissued request may not have enough time to respond before the latency target [35]. However, if multiple downstream requests hit a single service affected with long-tail latencies, then the problem becomes scarier [25]. An alternate solution is to use replica selection strategies where instead of sending multiple requests a single request dispatched to the best server. The selection of the best server is done based on a score that is calculated depending on several factors including service time, queue size, latency, disk I/O operations, etc. One advantage of such techniques is that they vanished the cost of request duplication and reissuing. Moreover, the efficiency of the request duplication or request reissuing techniques can be improved by designing these on top of an efficient replica selection strategy [25]. The replica selection strategies have a direct impact on reducing tail latency of the distributed systems which are specially designed to rely on replication and partitioning for scalability, such as key-value stores [25].

Thesis Objectives

Many researchers have proposed strategies for reducing tail latency in geo-distributed systems by introducing efficient techniques such as request duplication, request reissuing, replica selection, etc. In this thesis, our main objective is to introduce a new prediction based replica selection strategy for reducing tail latency in geo-distributed systems.

One of the important issues in designing a replica selection strategy is to select appropriate factors that are used for calculating the score which in turn is used to select the best replica for routing the read request. One of the objectives of this thesis is to find and analyze all important

factors that have a direct impact on the long-tail latency of the distributed systems and then first design a prediction based replica selection strategy for reducing tail latency in the locally distributed systems.

Today, the latency of a storage system can be lower than the millisecond range when serving local requests under normal load conditions. However, in geo-distributed systems, the inter-continental network latency becomes the predominant factor in determining the final performance of the system. Therefore, our next objective is to extend the prediction model of the replica selection strategy of the locally distributed system and design a new prediction based replica selection strategy for the geo-distributed systems considering the inter-continental network latency.

One of the main challenges of designing an efficient replica selection strategy is to handle the herd behavior problem where a large number of requests are routed to the same node and make it a hot spot. In this research work, another objective is to design the replica selection strategy in such a way that it can avoid herd behavior problems.

The replica selection strategies in the geo-distributed systems, calculate a score to represent the goodness of the replica. The replica with the best score is selected to route the user's read request. However, most of the state-of-the-art replica selection strategies calculate the score in such a way that does not reflect the actual state of the replicas and reduces the overall performance of the system. Our other objective is to design a new prediction based replica selection strategy that can calculate the score more efficiently that reflects the actual state of the replicas.

According to the discussion of [25], the performance of both request duplication and request reissue strategy can be improved by designing on top of an efficient replica selection strategy. Hence another objective of this research is to identify whether an efficient replica selection strategy really improves the performance of the request reissue and request duplication strategy or not.

Contributions

The main contributions of this thesis are as follows:

1. We identify and analyze the challenges and important factors to consider during replica selection.

2. First we propose a prediction based replica selection strategy for reducing tail latency efficiently in the locally distributed systems.

3. We then extend the replica selection strategy of the locally distributed system and design a prediction based replica selection strategy for reducing tail latency more efficiently in the geo-distributed systems.

4. We conduct comprehensive experimentation to evaluate the replica selection strategy for locally distributed systems using a popular open source Big Data system (Cassandra) and demonstrate that it improves performance in the locally distributed systems.

5. We conduct comprehensive experimentation to evaluate the replica selection strategy for geo-distributed systems using Cassandra and demonstrate that it improves performance in the geo-distributed systems.

6. We conduct comprehensive experimentation to identify whether an efficient replica selection strategy really improves the performance of the request reissue and request duplication strategy or not and demonstrate that it really improves the performance of the request reissue and request duplication strategy.

Thesis organization

The organization of the rest of the thesis is as follows.

In **Chapter 2**, we have presented relevant background studies related to reducing tail latency in geo-distributed systems. We discuss the concepts of cloud computing, geo-distributed systems, and replica selection strategies. We also discuss the challenges and essential factors to consider during replica selection. Finally, we discuss the architecture and working procedure of Cassandra.

In **Chapter 3**, we have presented literature review on research works related to our study.

Chapter 4 explains in detail our first proposed strategy to reduce tail latency in the locally distributed systems, along with experimental setup, experimental results and their analyses. It also discusses the implementation detail of our proposed strategy.

Chapter 5 explains in detail our second proposed strategy to reduce tail latency in the geo-distributed systems, along with experimental setup, experimental results and their analyses. Similar to the previous chapter, it also discusses the implementation detail of our proposed strategy.

Chapter 6, explains in detail how our proposed strategy improves the performance of the request reissue and request duplication techniques, along with experimental results and their analysis.

In **Chapter 7**, we have discussed some of our observations that we found throughout this research work.

Finally, **Chapter 8** concludes our thesis. This chapter also includes the outlines of some future works related to this dissertation.

Chapter 2

Background

In this chapter, we discussed the concepts and techniques used in the rest of the paper. We propose a replica selection strategy in the geo-distributed systems. Hence, we begin by introducing the concepts of cloud computing, geo-distributed systems, replica selection strategies. We also discuss the challenges and important factors to consider during replica selection. We finally, evaluate our system in Cassandra. Hence, we also discuss the architecture and working procedure of Cassandra.

Cloud Computing

The construct of cloud computing relies on the idea of hardware virtualization that was introduced in 1960s when IBM introduced a time-sharing virtual machine operating system for the mainframes [36]. The idea of time-sharing system converted to cloud computing in 2000. Cloud computing provides us computing services including servers, storage, databases, networking, software, analytics, and intelligence—over the Internet with pay-as-you-go pricing. It helps individual people and businesses to reduce their cost, increase productivity, increase speed and efficiency, improve performance and security, and allows to scale business as need [37].

Cloud computing service providers maintain a datacenter/datacenters of thousands of computing resources. These resources are logically shared among the users using virtualization techniques. The concept of cloud computing revolutionized the way of modern services and applications operate and generated a broad market of public and private cloud service providers. Some of the most popular public cloud service providers are Amazon Elastic Compute Cloud (EC2) [38] (launched in 2006), Google Cloud [39] (launched in 2009), Microsoft Azure [40] (launched in 2010), and IBM Cloud [41] (launched in 2011).

Now-a-days, modern applications such as email, social networks, audio, and video streaming, financial systems, and many more are built on cloud computing platforms. Millions of users are accessing these applications every day. The core of these applications are distributed

storage systems such as Gmail and Picaso uses Google Spanner [42], Facebook is based on Cassandra [24], LinkedIn uses Voldemort [43], and Amazon Store is based on DynamoDB [11]. Most of the services and applications are latency-critical as discussed in Chapter 1, Section 1.2, and 1.3. The increase in latency reduces the performance of the applications and services which has a negative effect on the user's experience which in turn reduces the popularity as well as the revenue of the service providers. To improve the performance a replication process is applied where multiple copies of the application's data are stored in different physical machines. Replication is also done for data survivability and availability.

The replication process allows the applications and services to survive failure and disasters by having multiple copies of the same data to different machines. If data in a machine is inaccessible then copy from another machine will be available to access. Copying data in a single data center will enable surviving the failure of a machine or a rack. To guarantee data survivability even in the event of a local natural disaster, or a complete datacenter failure, or a failure in the network connectivity to and from the datacenters and end-users or between data centers., data is replicated throughout the geo-distributed data centers.

Replicating data throughout the geo-distributed data centers improves the performance of the applications and services by moving the data closer to the user. By selecting geographically closer data the response time of the user's request is reduced thus improves the performance which in turn improves the revenue of the service providers. However, selecting the closest replica does not always reduce the response time, as response time depends on several important factors including network conditions, load on that specific replica, and the performance of the machine the replica stored. These factors need to understood, measured, and take into account when selecting a replica for routing the request of the users. Because of many contributing factors performing replica selection automatically at run-time is difficult.

Geo-distributed Systems

In geo-distributed systems, the cloud service providers deploy their data centers throughout different geo-graphical region for improving the performance and ensuring survival of the applications and services as discussed in section 2.1. Fig. 2.1 shows an example of a geo-distributed system. The smallest unit in a data center is the rack which comprises of several server machines which are connected by a top-of-rack switch. The communications within racks in a data center are carried out by intra data center link. Racks within a data center are connected by cluster switch. The data centers are spread throughout different geo-graphical region and communication are carried out among these data centers by inter data center link. Cloud users access the data center services through a gateway spread across WAN.

Now-a-days most of the public and private cloud service providers deployed their data centers in geo-distributed manners. For instance, one of the most popular public cloud service provider

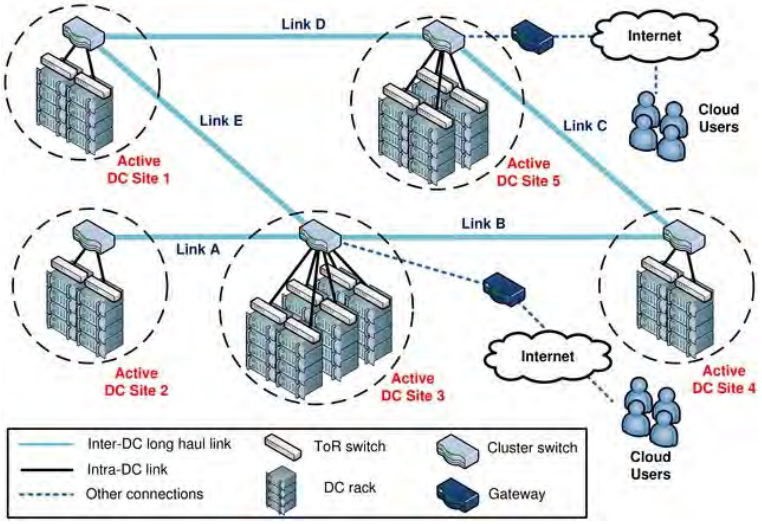


Figure 2.1: Example of a geo-distributed data center composed of interconnected sites [2]

Figure 2.2: Amazon EC2 Global Infrastructure Map [3]

Amazon EC2 has 80 availability zones through 25 geographic regions. Moreover, they are planning to launch 15 more availability zones and 5 more AWS regions in Australia, India, Indonesia, Spain, and Switzerland [3]. Fig. 2.2 depicts the global infrastructure of Amazon EC2's geo-distributed data centers. Similarly, Google has 21 data center locations currently in the Americas, Europe and Asia [44] [45]. IBM has more than 60 data centers located in 10 countries covers every continent except Antarctica [46].

Latency Measurements in the Geo-distributed Systems

Today, the latency of a storage system can be lower than milliseconds range when serving local request under normal load condition. However, in geo-distributed systems, the inter-continental network latency becomes the predominant factor in determining the final performance of the

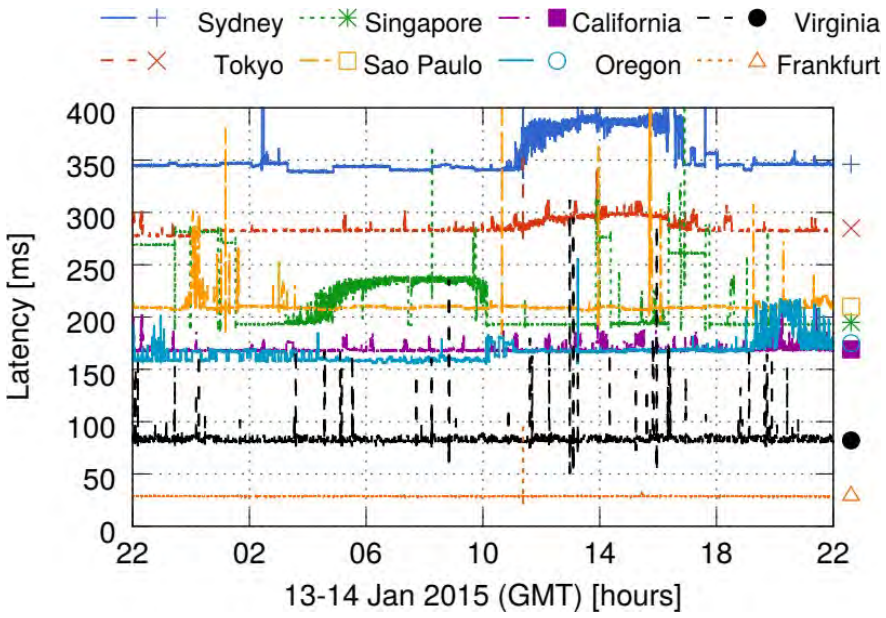


Figure 2.3: RTT measurement from Amazon EC2 Ireland data center to all other 8 regions of Amazon EC2 [4]

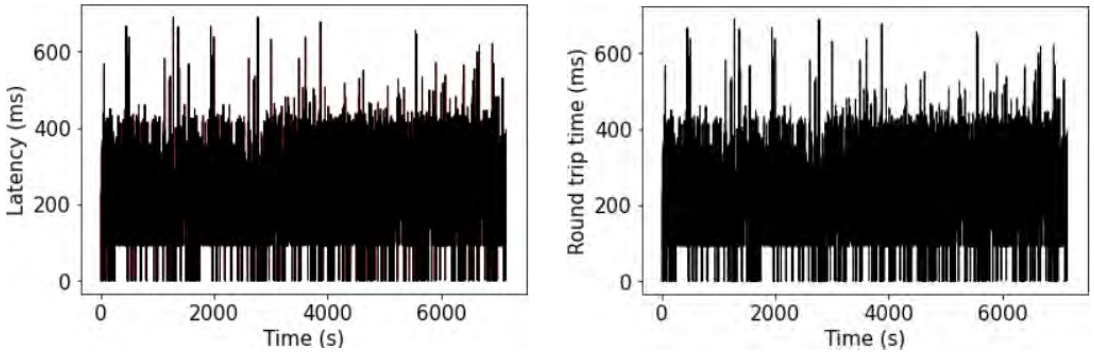
system. Fig. 2.3 shows the RTT measurement from Amazon EC2 Ireland data center to all other 8 regions of amazon EC2 over a period of one day. This figure depicts that the greater the distance of the data center from the Ireland data center the greater the RTT value. That means the inter-continental network latency is a crucial factor in determining the best replica. Hence, in geo-distributed systems, the latency of a user request comprises of RTT in the network and response time in the remote replica as in (2.1).

$$TotalLatency = RTTinthenetwork + ResponseTimeoftheServer \tag{2.1}$$

The response time of a server has direct relationship with the queue size and service time of that specific server as discussed in Section 2.4.

Factors to Consider During Replica Selection

In geo-distributed systems, nodes of a cluster are spread across a large geographical region and several factors have a direct impact on the latency of such a system. These factors should be kept into consideration during designing the replica selection strategy that aimed to reduce the tail latency of the geo-distributed systems. Some of the most important factors are discussed below.



(a) (b)

Figure 2.4: Effects of RTT on latency (a) latency over a specific period of time (b) corresponding RTT over the same period of time

Round Trip Time (RTT)

One of the main reasons for long read latency in geo-distributed systems is the distance between the requested client and the responding server [47]. For example, if a website is hosted in a data center in Trenton, New Jersey, it will respond faster to the requests from users in Farmingdale, NY (100 miles away) than the users from Denver, Colorado (about 1800 miles away). One way to measure the distance between the client and the server is to keep track of the RTT of a read request [47]. The RTT of a replica read request is the amount of time it takes for a request to go from a client to the server and back the response to the client from the server [48] as in (2.2).

$$RTT = T_{cts} + T_{stc} \tag{2.2}$$

Where T_{cts} is the amount of time takes a read request to reach from the client to the server and T_{stc} is the amount of time takes a response to reach from the server to the client. In geo-distributed systems, since the nodes of a cluster are geographically dispersed hence it is very crucial to keep into consideration the RTT during designing a replica selection strategy. To evaluate the effects of RTT on the latency in geo-distributed systems we conduct an experiment on Amazon EC2 with a 15-node Cassandra cluster that spreaded over five geographical regions. The experimental results show that the RTT has a direct and huge impact on latency. We record both the latency and RTT over a specific period of time of each Cassandra node. Fig.2.4 shows, that both latency and RTT have almost the same order of magnitude over the whole period of time. That depicts that in a geo-distributed system RTT is one of the main contributors to latency and hence it is very crucial to keep into consideration the RTT during designing a replica selection strategy.

Queue Size and Service Time

The response time of a read request directly depends on the queuesize and service time of that server. At any specific time, the response time of a read request is equal to the product of the

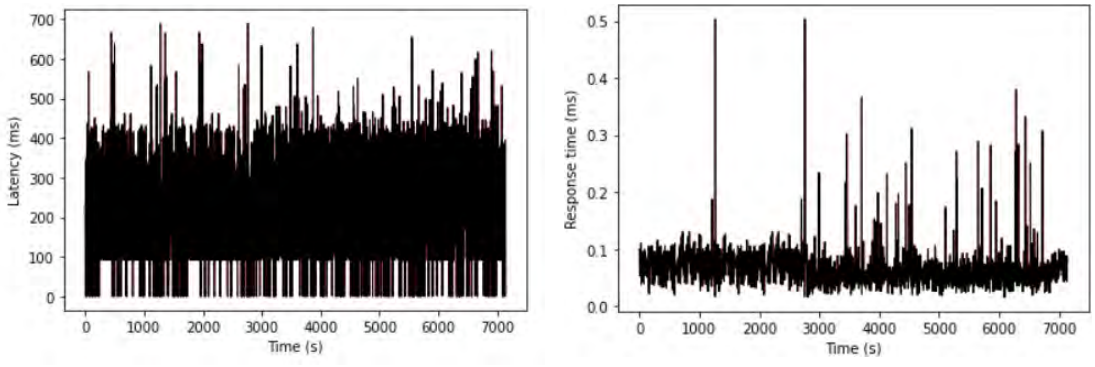
queuesize and service time of that server at that specific time as in (2.3).

$$r_s(t) = q_s(t) \times \frac{1}{\mu_s}(t) \tag{2.3}$$

Where r_s is the response time, q_s is the queuesize and $1/\mu_s$ is the service time of each server s at any specific time t. The response time of a read request has a direct impact on the overall latency of that request and the greater the response time the greater the latency. By reducing the queue size and service time of a server the response time of a read request server can be reduced which in turn will reduce the overall latency. Hence, queue size and service time should be an important factor to consider during replica selection in geo-distributed systems. In the same experiment that is mentioned in section 2.4.1, we also keep track of the corresponding response time of the previously recorded latencies of all the nodes over the same period of time. Fig. 2.5 shows that the magnitude of latency is 1000 times higher than the response time. Hence it seems that the response time has almost no contribution to the overall latency and we can just ignore it. However, the response time depends on the service time and queue size. In our experimental environment, we have only 15 nodes and the load was not very heavy. Hence, the value of queue size was varied among 1-3. To understand the effect of response time on latency we conducted another experiment in a different testbed. We deploy a cluster with a lower system configuration and generate a 10 times higher load to the cluster. We did this so that the queue size can have higher values. This time the queue size has values ranges from 8 to 50 and the value of response time increased significantly as shown in fig 2.6. The figure depicts that the response time increases as the queue size increases and the distribution of response time and queue size are analogous. To determine the effect of response time on latency in such a scenario we also recorded the corresponding latency over the same period of time and their distribution is shown in fig.2.7. This time the magnitude of latency has a greater impact on the response time and hence has a more significant contribution to the overall latency of the system. Therefore, from this observation, we can conclude that in high workload situations the response time has a significant contribution to the overall latency of the system.

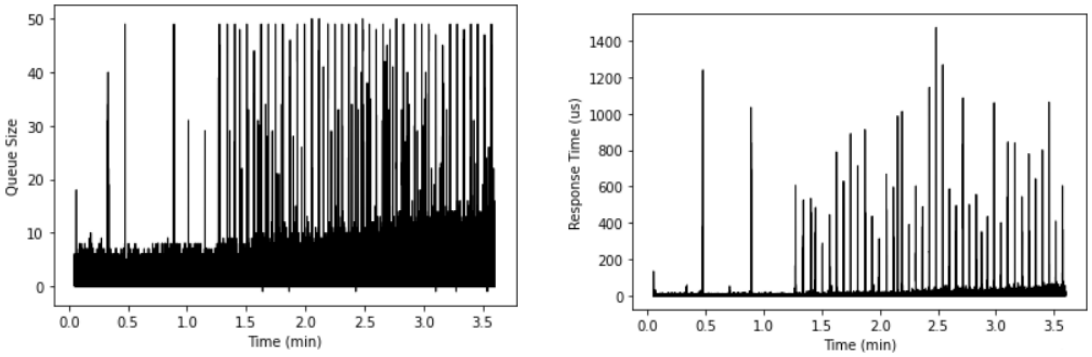
Disk I/O Operations Rate (IOPS)

The disk input/output (I/O) operations of servers penalize the disk read operations which in turn increases the disk read latency that results in the increased latency of the nodes [29]. In a Cassandra [24] cluster, each node performs compaction periodically to merge multiple SSTables [16] [49]. This significantly increases disk input-output operations which in turn increases the overall latency of the node. Hence for designing an efficient replica selection strategy we also need to consider the impact of disk I/O operations of a server as a selection criterion. To evaluate the effect of IOPS on latency we conducted an experiment in a 15 node Cassandra cluster that is deployed on Amazon EC2 and the experimental result in fig. 2.8 shows



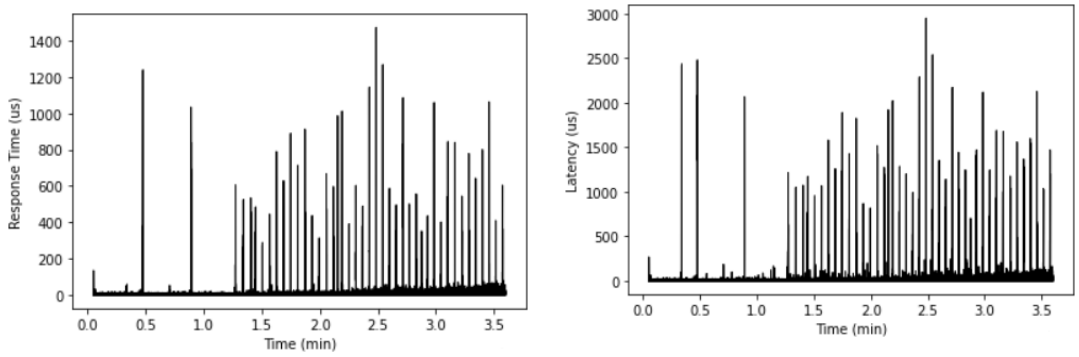
(a) (b)

Figure 2.5: Effects of response time on latency (a) latency over a specific period of time (b) corresponding response time over the same period of time



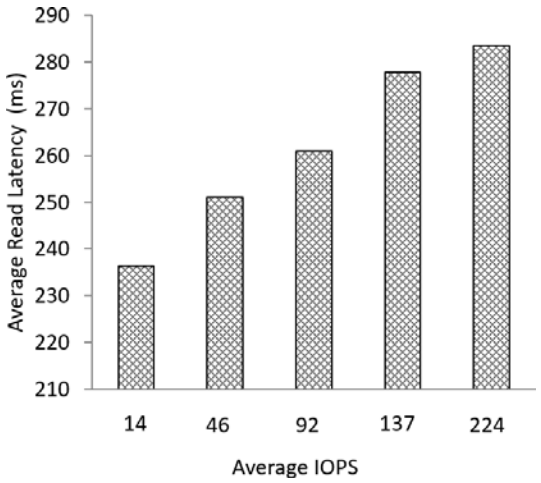
(a) (b)

Figure 2.6: Effects of queue size on response time (a) queue size over a specific period of time (b) corresponding response time over the same period of time



(a) (b)

Figure 2.7: Effects of response time on latency in a heavy laod situation (a) response time over a specific period of time (b) corresponding latency over the same period of time



(a)

Figure 2.8: Effects of IOPS on latency.

as the average number of IOPS increases the average read latency increases. We started our experiment with 25 read workload generators and 25 write workload generators with a total of 1 million read tasks and 1 million write tasks respectively. In this setting, the average IOPS is 14 and the average read latency is around 236 ms. Then we increase the number of workloads to 125 for both read and write. At this point, the average IOPS is 46 and the average read latency is about 251 ms. As we increase the number of workload generators from 125 to 225, 325, and 425; the average IOPS increases to 92, 137, and 224 respectively. The increased average number of IOPS increases the average read latency. Therefore, there is a direct relationship between IOPS and the read latency. Hence for designing an efficient replica selection strategy we need to consider the impact of disk I/O operations of a server as a selection criterion.

Dynamic Replica Selection

In geo-distributed systems, data are replicated throughout the geographical region for improving performance and ensuring survivability of the applications and services. Because of the performance fluctuations of the network and the servers choosing the closest replica does not always ensure better performance. To select the best performing replica a replica selection algorithm is used. However, as the behavior of the network and the server is dynamic in nature hence the replica selection algorithm should also be dynamic. A dynamic replica selection algorithm is designed in such a way that is able to adapt to the changing nature of the network and the servers. To select a replica for routing the users read request a geo-distributed system continuously monitors the network and server state. A dynamic replica selection algorithm processes this data and uses it for ordering the replica. Replicas are ordered to select a subset of replicas that are capable of answering the user's query within the shortest time. Fig. 2.9 shows

the stages of a dynamic replica selection strategy in a cluster with 7 replicas.

Stage1: Measuring the Factors and Calculating the Score

As discussed in Section 2.4, several factors need to keep into consideration during selecting the replica. Hence, the first step of dynamic replica selection is to measure algorithm-specific factors during run time. Then a score is calculated based on the measured factors for identifying the goodness of the replicas. The score can be calculated based on a single factor or a set of factors. The measurements of the factors and the calculation of the score are done periodically over discrete intervals.

Stage2: Replica Filtering

In the distributed database systems, a fixed number of copies of data are stored which is called the replication factor. As the number of replication factors can be smaller than the total replicas in a cluster hence all the replicas do not store the data necessary to perform a query. The replicas that do not store the copy of the requested data are removed from the list. Thus the list of the replicas is filtered based on the presence of the requested data. This filtering is done very frequently, during the execution of each query.

Stage3: Score Based Replica Ordering

In this stage, filtered replicas are ordered based on their score. This ordering is also performed periodically at discrete time intervals. This score is preserved and is utilized by every query until a new ordering is calculated.

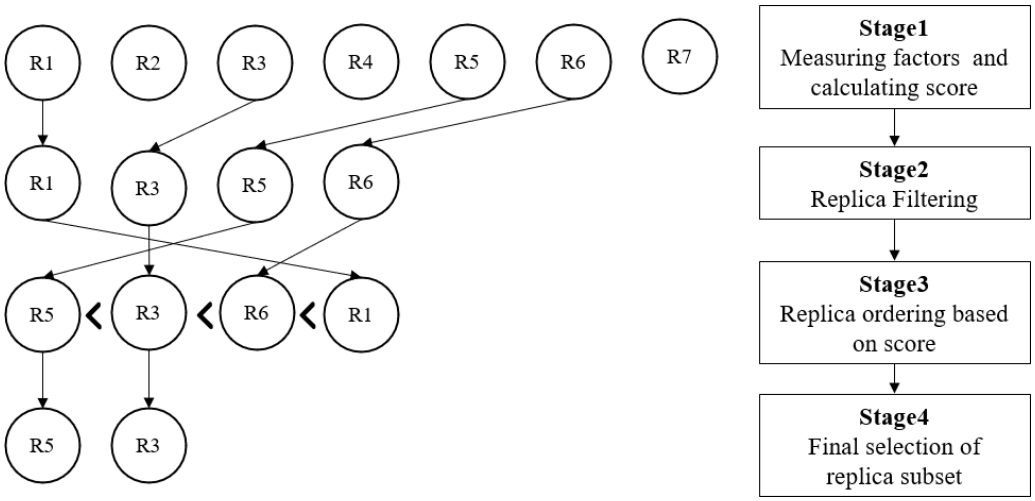


Figure 2.9: Stages of dynamic replica selection.

Stage4: Final Selection of Replica Subset

At this stage, a final subset is selected. To one of the replica of this subset a a complete read request is sent and a digest request is sent to the remaining replicas. This stage is optional and is only applied if the number of replicas that can answer the query is greater than the consistency requirement of the request.

Challenges of Replica Selection

In the following subsections, we discussed some of the challenges related to designing an efficient replica selection mechanism in the geo-distributed systems.

Round Trip Time (RTT) Variability

In geo-distributed systems, the round trip time (RTT) becomes the predominant factor in determining the final system's performance. This RTT is fed to the replica selection algorithm to derive future choices of replicas that will service the client's read request. Unfortunately, because of the dynamic nature of the geo-distributed systems, this RTT varied dramatically over time. In [30], Kirill Bogdanov et al. showed that there is significant variability in RTT over the course of a day, as well as over several weeks. To try to compensate for this variability, replica selection algorithms need to have a mechanism to estimate the RTT when processing the request and this makes replica selection more challenging in the geo-distributed systems.

Replica Reordering

Frequent reordering of the replicas makes it more difficult to design a replica selection strategy for geo-distributed systems. Kirill Bogdanov et al. in [30] showed that replica reordering can change up to ten times a day, from the viewpoint of a data center. Because of this reordering, static replica selection could be suboptimal even over a dedicated infrastructure. Suboptimal replica selection due to the dynamic environment of the geo-distributed system reduces system performance which results in longer latency.

Load Oscillations

Modern applications that are built on geo-distributed systems demonstrate varied load over time [50] [51]. This load oscillation can be due to sudden spikes in content popularity caused by a major event or failure of the node [52] [53]. To adapt to the load oscillations several replica selection strategies use techniques such as round-robin (RR), least outstanding request (LOR),

and IP-hash [54]. These techniques are easy to implement and do not require global information. However, studies show that these techniques are not efficient enough to reduce the tail latency of the geo-distributed systems since the realistic workloads are skewed in practice and may change over time [25]. Methods with perfect and up to date global information will be more suitable to adapt to the load oscillations nature of the geo-distributed systems but obtaining up to date global information requires complex system design and increases computational complexity [32] of the replica selection strategies.

Response Time Variations

In geo-distributed systems, the latency of a replica read request has a direct relationship with the response time which is the total amount of time it takes to respond to a read request for service [55]. On the other hand, the response time is the sum of the service time and wait time. The service time varies a little if the load changes however the wait time varies from zeros when there is no request in the server's queue to a large multiple of the service time as there may be many read requests that are already in queue and have to be serviced first. Since in geo-distributed systems load oscillation is a norm hence the queue size will be varied, as a result, there will be response time variation and the response time variation makes the design of the replica selection strategy more challenging.

Herd Behavior

Designing an appropriate replica selection strategy for the geo-distributed systems is very challenging and any indifferent design decision may result in "herd behavior". Herd behavior caused when several replica read requests are routed to the lowest loaded server simultaneously [56]. As a consequence, the lowest loaded server becomes a hot spot and results in long tail latency which in turn degrade the overall performance of the geo-distributed systems. The problem of herd behavior can be reduced by feeding perfect and updated information to the replica selection strategy [56]. However, the information is inevitably imperfect and stale in geo-distributed systems. Hence, special care should be taken when designing a replica selection strategy to deal with the herd behavior.

Traffic Variations

Applications built on geo-distributed systems are very network intensive [57] and their performance largely depends on the traffic nature of the network. According to a study [58], about 80% of the traffic originated by servers stays within the rack which has a direct impact on the tail latency of the geo-distributed systems. The amount of traffic within the rack increases the round-trip time (RTT) of the replica read request [59] which consequently increases the overall

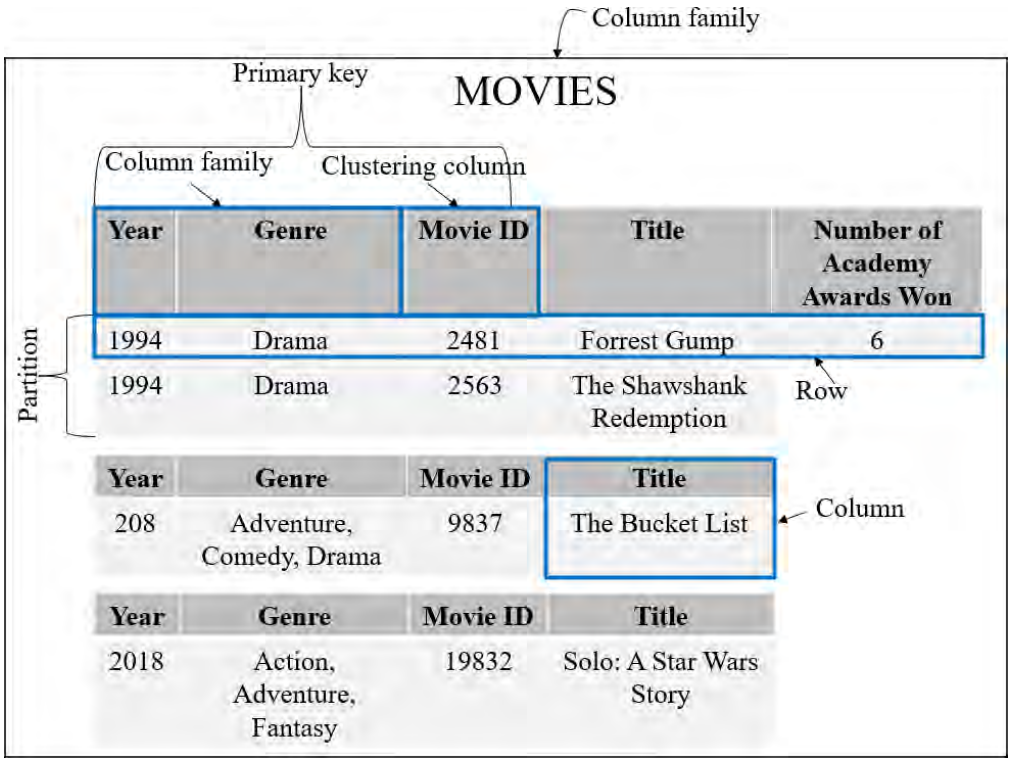


Figure 2.10: Key-value store data model in Cassandra.

latency of the geo-distributed systems. Moreover, the non-uniform placement of a variety of applications across the geo-distributed systems results in traffic variations [58] which makes it more challenging to design an appropriate replica selection strategy to reduce the tail latency in the geo-distributed systems.

Key-Value Store

The key-value store system is a kind of nonrelational database where data are stored using the key-value model. In this type of database, the key serves as the unique identifier. The key and value can be anything ranging from a simple object to a complex compound object. key-value databases are highly partitionable and allow horizontal scaling which is the base of today's geo-distributed systems. Fig. 2.10 shows an example of data stored as key-value pairs in Cassandra. It's a simple Cassandra column family. Here we can see that the rows can have a different number of columns. Every column family has a primary key which can be simple or compound. A simple primary key consists of only the partition key which determines in which node and which partition the data is going to be stored. The compound primary key comprises a partition key and a clustering column. The compound primary key is also used for the same purpose as of simple primary key but it also sorts the data within one partition [60]. Key-value store systems are commonly used in high scale session management, user preference and profile

store, product recommendations, ad servicing and so on. Some popular key-value store systems are Aerospike, Apache Cassandra, Berkeley DB, Couchbase Server, and Redis Riak [61]. To ensure fast and reliable service to the end users high speed computation and data survivability is must. The key-value store system is a popular way to achieve this by providing high speed computation to manipulate data at high accuracy and allowing scaling the systems over a broad geo-graphical region by implementing partitions, replication, and auto-recovery [62].

Cassandra

Apache Cassandra is an open-source non-relational, key-value store database that provides high availability, extreme scalability, and distribution of data across multiple geographical regions. It was originally designed at Facebook combining the Amazon's Dynamo distributed storage and replication techniques with Google's Bigtable data and storage engine model [63]. Cassandra is designed by employing peer-to-peer distributed systems with no single point of failure. In Cassandra data are distributed among homogeneous nodes which spread across different geographical regions. In Cassandra, any authorized user can connect to any nodes in the cluster. The client's read and write requests are sent to any node in the cluster. When a client connects to a node with a request, that node acts as a coordinator for that particular client's request. The coordinator acts as a proxy between the client's application and the node that contains the requested data. The coordinator determines to which nodes in the ring the request is to route [64]. Cassandra is a partitioned row store database, where rows are organized into tables with a required primary key as discussed in section 2.7. The key components of Cassandra Architecture those are related to this research are discussed in the following subsections.

Data Centers and Racks

Cassandra is widely used in geo-distributed systems. It used two levels of grouping to describe the topology of the cluster: data center and rack. A rack is a logical collection of nodes which are in close proximity of each other and in a single rack there may be several nodes. A data center is a logical collection of racks which may located in the same building and are connected via reliable network. A sample topology with multiple data centers and racks is shown in fig. 2.11. Cassandra uses the cluster's topology information to decide where to store the data and to which node the clients request is to route. Cassandra tries to replicate the data in multiple data centers to ensure availability and partition tolerance where prefers to route the client's request to the local data center to improve the performance [65].

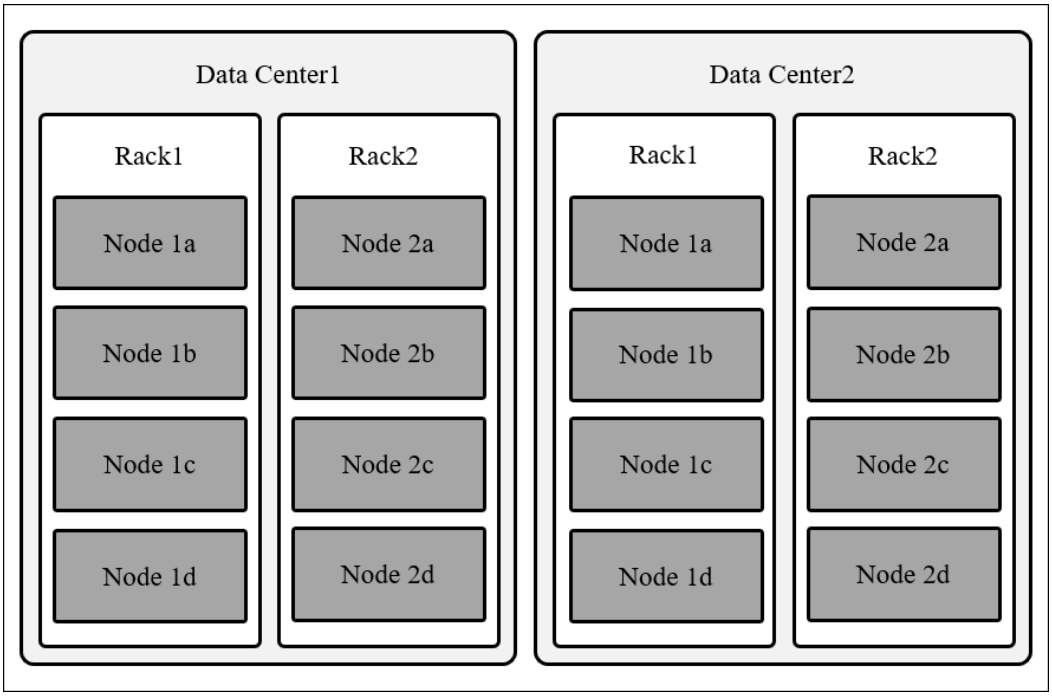


Figure 2.11: Topology of a sample Cassandra cluster with data centers, racks, and nodes.

Rings and Tokens

The data managed by a Cassandra cluster is described as a ring. Every node within the ring contains one or additional ranges of data represented by a token that determines the position of a node within the ring. The Cassandra token is a 64-bit integer ID that is used to identify each partition. Each node contains a range of values that is greater than the token of the previous node and less than or equal to its token. The node with the lowest token contains the range of values that greater than the highest token and is less than or equal to its token. Fig. 2.12 depicts the conceptual layout of token ring which includes the nodes of a single datacenter. For assigning data to a specific node a token for the partition key is calculated using a hash function. This partition key is compared with the token of the nodes to find out the range that means the node that will store the data [65].

Partitioner

Data in the Cassandra are stored in a wide row or partition. Each row has a partition key that is used to identify the partitions individually. A hash function is used to generate a token for a specific partition. In Cassandra, each row of data is distributed across the ring according to the partition key token [65].

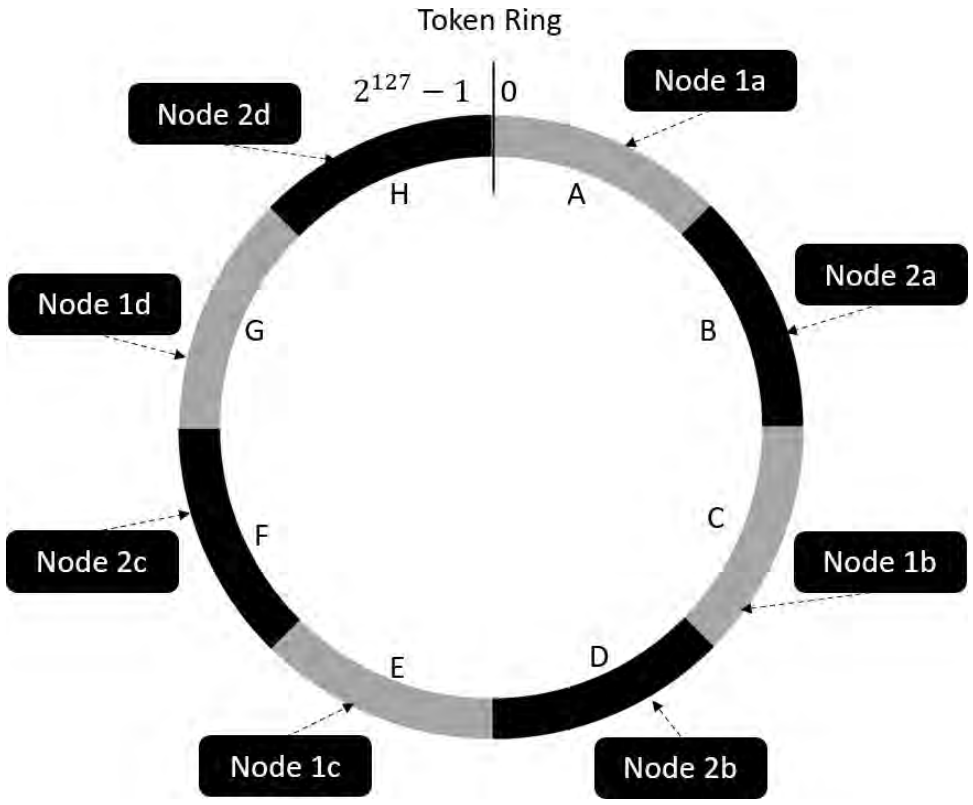


Figure 2.12: Example ring layout including nodes in a datacenter

Replication Strategies

Cassandra stores multiple copies of data which are called replicas for ensuring reliability and fault tolerance. The replication strategy determines the nodes where the replicas are stored. In a Cassandra cluster, the total number of the replica is called the replication factor. All replicas have equal importance, none of these acts as a principal or secondary replica. The default replication factor in Cassandra is 3. The replication factor should not exceed the number of nodes in the cluster. There is two replication strategy in Cassandra [66].

1. SimpleStrategy: It is used only for a single datacenter and a single rack. It places the first replica to the nodes selected by the partitioner. The other replicas are placed to the next nodes clockwise in the ring without considering the cluster topology.

2. NetworkTopologyStrategy: It is the recommended strategy and specifically uses for placing data in multiple datacenter and multiple racks. Using this strategy the users can specify how many replicas they want in each datacenter. The NetworkTopologyStrategy places the data in the same datacenter by walking the ring clockwise until the first node is found in a different rack.

Snitches

In Cassandra, snitches are used to determining the relative proximity of each node in a cluster, which is used to determine which nodes to read and write from. Snitches collect the information about the network topology of a cluster. This information is used for efficient routing of the read and write request [65]. Cassandra comes with different snitches for different cloud environments as listed below.

- SimpleSnitch: It is the default snitch of Cassandra and good for the development environment. It is unaware of racks and data centers and hence unusable for the multi-datacenter environment [67].
- GossipingPropertyFileSnitch: In Cassandra, this is a very important file snitch and recommended for production use. This snitch look for the Cassandra-topologies.properties file to get the cluster information such as which data center and rack the node belongs to and propagates this information to other nodes via gossip [67].
- PropertyFileSnitch: This snitch determines the proximity of the nodes based on the rack and data center to which they belong. It uses the network topology information from the cassandra-topology.properties file [67].
- Ec2Snitch: This strategy is appropriate for Amazon EC2 deployment where all nodes are in a single region. It uses the Region and Availability Zone information from EC2 API where the Region is treated as datacenter and the Availability Zone as the rack [67].
- Ec2MultiRegionSnitch: This snitch is specifically used for a cluster that spans multiple regions and deployed in Amazon EC2 [67].
- RackInferringSnitch: This snitch is specifically useful for writing custom snitch classes. It finds out the proximity of the nodes by the rack and datacenter. In this snitch, the 3rd and 4th octets of the IP address of the nodes refer to the rack and datacenter respectively [67].
- DynamicEndpointSnitch: This is a special snitch that helps Cassandra to optimize the read and write requests over time. The selected Cassandra snitch is wrapped with this snitch. The dynamic snitch gets a basic understanding of the topology of the cluster from its underlying snitch. It monitors the performance of the request to the other nodes and also keeps track of this information. This information is then used to select the best replica for the query. This enables Cassandra to avoid routing read requests to the poorly performing nodes [65].

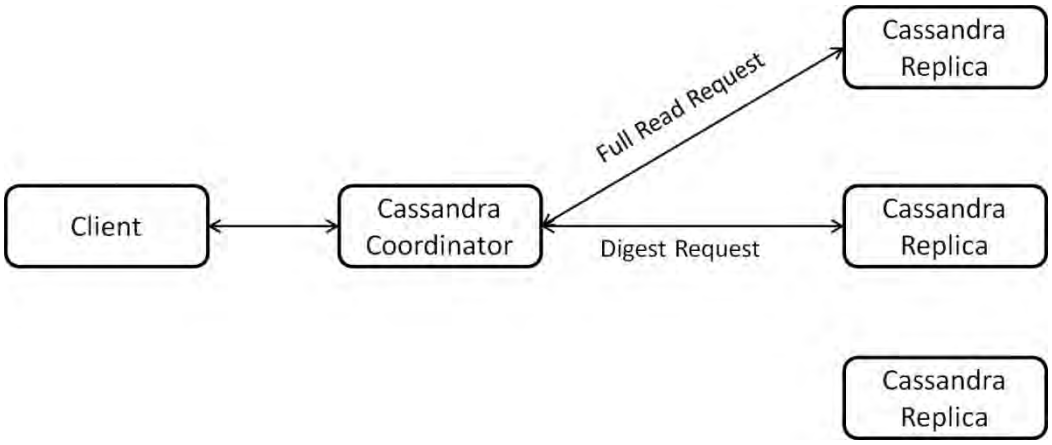


Figure 2.13: No retry policy (replication factor 3)

Speculative Retry

Cassandra uses the *speculative_retry* property to configure rapid read protection. In a normal read, Cassandra sends read requests to enough replicas to satisfy the consistency level. In rapid read protection, Cassandra pre-emptively sends a second read request to another replica, before the first replica has replied or errored out [68]. If that second replica replies faster, Cassandra sends the response back to the client and cancels the read request to the first replica. Or if the first replica reply just after the second read request was sent, Cassandra sends the response of the first replica back to the client and cancels the second read request. In other words, whichever replica replies faster ”wins” and completes the client read request. Cassandra supports four speculative retry policies as discussed below.

- No Retry Policy: In this policy, Cassandra sends only one full read request to one replica and additional digest requests to other replicas based on the consistency level as shown in fig. 2.13. We can enable this policy by setting the *speculative_retry* property to *NONE*.
- Always Retry Policy: When use this policy, Cassandra sends two full read requests to two replicas and additional digest request to other replication based on the consistency level as shown in fig 2.14. To enable this policy we have to set the *speculative_retry* property to *ALWAYS*.
- Custom Retry Policy: In custom retry policy, Cassandra sends one full read requests and one or more digest requests to the replicas depending on the consistency level. If the replica to which a full request has been sent does not response within a specific time interval then the coordinator sends another read request to another replica as shown in fig. 2.15. The time interval that the coordinator waits for reissuing the read request can be specified by setting the *speculative_retry* property to a a fixed amount of time.

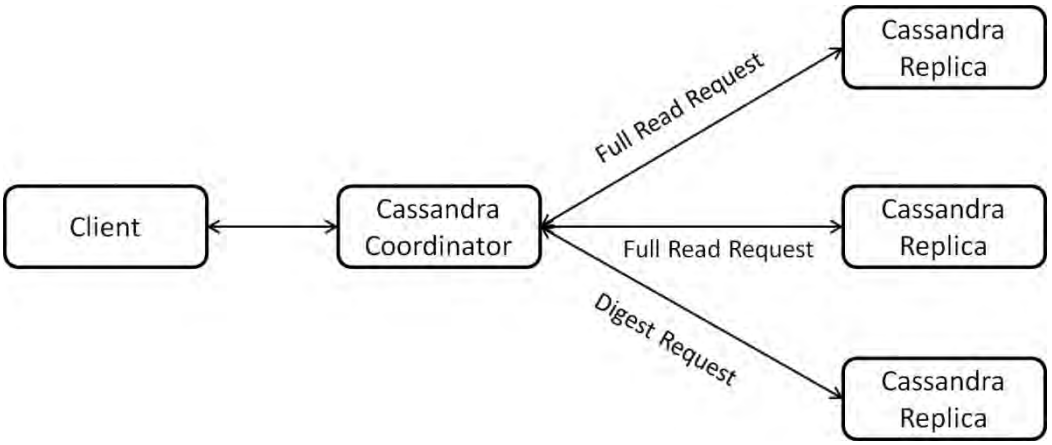


Figure 2.14: Always retry (replication factor 3)

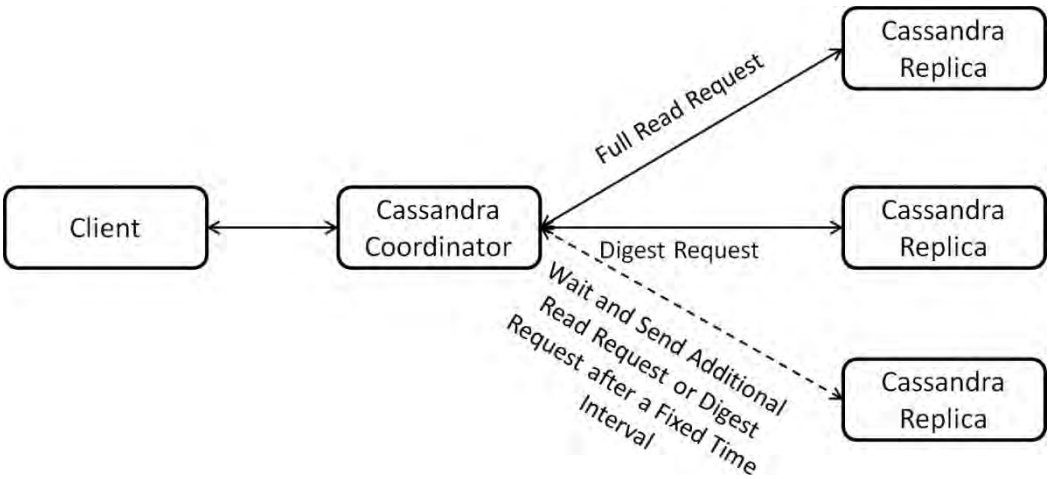


Figure 2.15: Custom retry (replication factor 3)

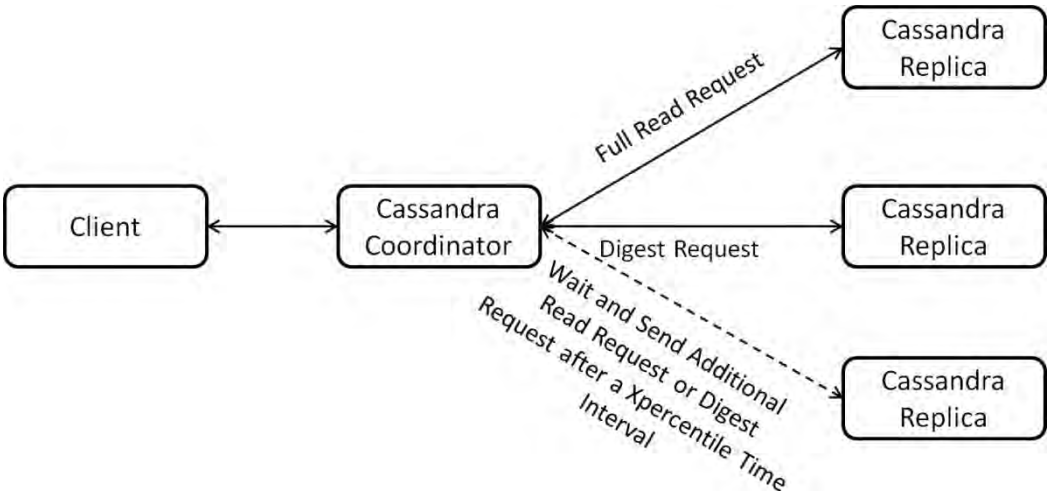


Figure 2.16: Percentile retry (replication factor 3)

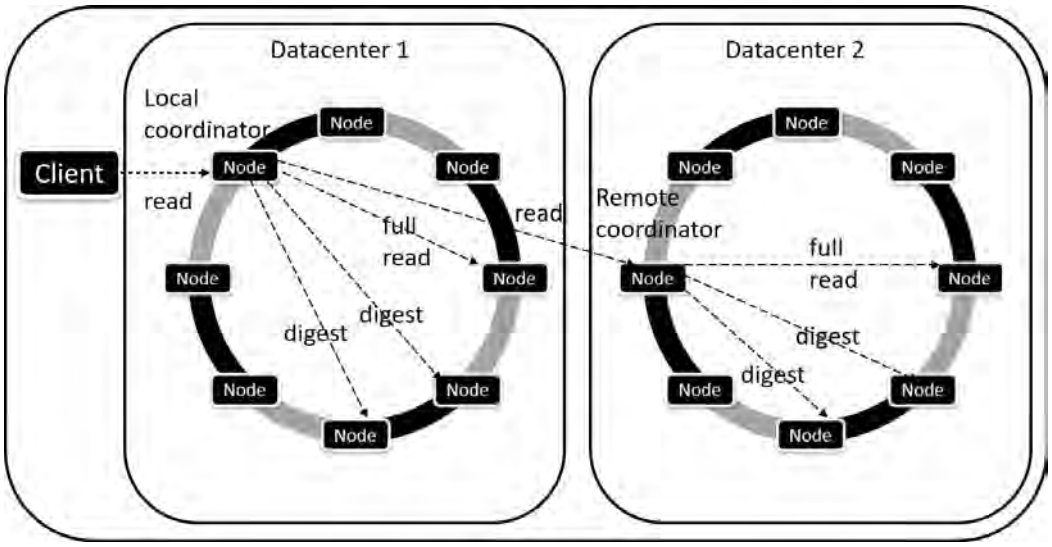


Figure 2.17: Interaction among nodes in the data read path during a read query.

- Percentile Retry Policy: Like the custom retry policy, in this policy Cassandra also sends one full read requests and one or more digest requests to the replicas depending on the consistency level. If the replica to which a full request has been sent does not response within a specific time interval then the coordinator sends another read request to another replica as shown in fig. 2.16. The time interval that the coordinator waits for reissuing the read request can be specified by setting the *speculative_retry* property to a specific percentile.

Reading Data in Cassandra

In Cassandra, clients can send read requests to any of the nodes of the cluster without even knowing whether that node contains the requested data or not. If the node does not contain

the data then it acts as a coordinator for routing that read request to a node that does have the data. The coordinator node identifies the target node for routing read requests by using token ranges. Fig. 2.17 shows the interaction among nodes on the read path during read query in Cassandra. The read path initiates when the clients send the read query to the coordinator node. The coordinator determines the replicas by using a partitioner and checks if there are enough replicas up for satisfying the requested consistency level. Individual remote coordinators are selected for each datacenter if the read query involves multiple datacenters. If the coordinator does not contain the data itself, then it sends a full read request to the fastest replica that is identified by the dynamic snitch. The coordinator also sends digest requests to other replicas. The digest request is just like the full read request except for that replica in full request returns the requested data whereas the replicas in digest request return only the digest or hash of the data. The coordinator calculates a digest hash from the data returned from the fastest replica and compares it with the digests that are returned from other replicas. If the digests are matched and the requested consistency level has met then the data from the fasted node can send to the client. Otherwise, the coordinator must perform a read repair.

Summary

We started this chapter by discussing cloud computing and geo-distributed systems in brief. After that, we have shown how latency is measured in geo-distributed systems. We also listed and discussed important factors and challenges that we should consider during designing a replica selection strategy. We then described the dynamic replica selection strategy step by step. Finally, we concluded this chapter by discussing the architecture and working procedure of Cassandra.

Chapter 3

Literature Review

Several techniques have already been developed for reducing tail latency in geo-distributed systems. This chapter provides an overview of the research works related to reducing tail latency.

Ways to Reduce Tail Latency

There are mainly three classes of state-of-the-art strategies are available for reducing tail latency in geo-distributed systems: 1) request duplication strategies 2) request reissuing strategies and 3) replica selection strategies. In request duplication strategies [20] [21] [22] [23] [68], multiple copies of requests are dispatched to multiple servers for every user's read request. This technique is usually applied to systems with low system utilization. For systems that run at higher system utilization, one alternative approach is to use request reissuing techniques [16] [17] [18] [19] [68] where a second copy of the request is dispatched after a specified delay d. Another solution is to use replica selection strategies [24–27] where instead of sending multiple requests a single request dispatched to the best server. The selection of the best server is done based on a score that is calculated depending on several factors including service time, queue size, latency, disk I/O operations, etc.

Request Duplication Strategies

This section discusses several state-of-the-art request duplication strategies for reducing tail latency in the geo-distributed systems.

In [20], Flach et. al. presents the design of a novel loss recovery mechanism for reducing web latency. They proposed three qualitatively different mechanisms 1) Proactive- recover from loses proactively 2) Reactive- recover from loses as soon as possible and 3) Creative- reconstruct the erroneous message. Among these the proactive mechanism is a request reissuing technique. This mechanism proactively transmits multiple copies of a single request. Sending multiple requests

increase network traffic which consequently decreases the overall throughput and increases response time. Hence proactive mechanism can only be applied to latency sensitive services on networks while these services occupied a very small percentage of the whole network traffic.

In [21], Stewart et. al. proposed a new key-value store system named Zoolander. It masks slow storage access via replication for the predictability. This system supports scale-out by copying the same data across multiple nodes where each node calls the duplicate, sends each read/write request to every duplicate, and receives only the fastest response. Though sending each read/write request to each duplicate can reduce the tail latency, however, it will increase the traffic of the network and as a result will reduce the overall performance of the system.

Vulimiri et. al. in [22] also uses the concept of duplicate requests. They demonstrate that by introducing redundant requests to multiple resources and using the first result that is complete, the long tail latency can be reduced.

Repflow [23] is a datacenter transport scheme for reducing latency for short TCP flows both on average and the 99th percentile. It replicates each short TCP flow by creating another TCP connection to the receiver. The applications use the fastest flow that finishes the transfer.

Cassandra supports request duplication strategy for reducing tail latency. One can enable Cassandra's request duplication strategy by setting the *speculative_retry* property to *ALWAYS*. When enabled, Cassandra sends two full read requests to two replicas and sends additional digest requests to other replicas depending on the consistency level [68].

Request Reissue Strategies

This section discusses several state-of-the-art request reissue strategies for reducing tail latency in the geo-distributed systems.

Kaler et. Al. in [18] introduces a new family of strategy that reissue a request after a delay d and a probability q. While other reissue strategies reissue a request after a certain delay, this strategy employs randomness to limit the rate of reissue but allows the request to be reissued as soon as possible so that it has enough time to respond. It provides an algorithm for calculating optimal reissue delay and probability from response time.

CostLO [19] also presents the efficiency of redundant requests for coping with latency variability. It reduces the latency variability associated with cloud storage services by reissuing requests so that the fastest response can be considered. It combines the forms of multiple redundancies by applying different configurations that are viable in practice.

The Google Big Table [16] also uses reissue techniques for reducing tail latency and for fast recovery from failure. If it Big Table routes a read request to one cluster and that cluster is excessively slow to reply or returns an error then it tries to reissue the request to another cluster.

Cassandra also supports request reissue strategy for reducing tail latency. When enabled this

strategy, Cassandra sends one full read requests and one or more digest requests to the replicas depending on the consistency level. If the replica to which a full request has been sent does not response within a specific time interval then the coordinator sends another read request to another replica. The time interval that the coordinator waits for reissuing the read request can be specified by setting the *speculative_retry* property to a specific percentile or a fixed amount of time [68].

Replica Selection Strategies

This section discusses several state-of-the-art replica selection strategies for reducing tail latency in the geo-distributed systems.

Cassandra dynamic snitch [24] calculates a score for each node based on the latency. The system receives this information from the nodes as a feedback and selects the fastest node based on the calculated score. It comprises of two operations: one is receiving the update information that is cheap and another is calculating the score that is expensive. If it spends too much time in calculating the score then that would become counterproductive because it can reduce the throughput of the read. For mitigating this problem Cassandra calculates the score after every 100 milliseconds [69]. However, this fixed interval-based scheme arises two problems. First, the system cannot react to the performance fluctuations that occur within the time that is less than the fixed interval. Second, the client's requests show herd behavior that is, more and more read requests are routed to the same node which is best based on its score and this score remains the same until the next computation after the completion of the fixed interval. Moreover, this herd behavior results in performance fluctuations that again exacerbate the first problem. It may seem that these problems can be solved by reducing the interval of score re-computation but it results in low read throughput as discussed above. Hence, this fixed interval score calculation based dynamic snitch can lead to a poor replica selection strategy that degrades the overall performance of the system [25].

In [25], L. Suresh et al. presents C3 a replica selection mechanism to reduce tail latency. In this system each client calculates a score of individual servers to select the fastest server for each read request. This scoring is calculated based on the queue size and service time of each server that is feed-backed to the client at each read response. The client uses (3.1) for calculating the score.

$$\Phi_s = R_s - \frac{1}{\bar{\mu}_s} + \frac{(\hat{q}_s)^3}{\bar{\mu}_s} \tag{3.1}$$

Where, R_s, $\frac{1}{\mu_s}$ and $\hat{q}_s$ are exponentially moving averages(EMA) of the response time, service time, and queue-size estimation respectively. Again, clients calculate the queue-size estimate $\hat{q}_s$

of each server according to (3.2).

$$\hat{q_s} = 1 + os_s \cdot w + \bar{q_s} \tag{3.2}$$

Where, os_s is the number of outstanding requests from the client to server s, w is the number of clients in the system, and $\bar{q}_s$ is the estimated weighted moving average (EWMA) of the queue size. In (3.2), the term $os_s.w$ is called the concurrency compensation which is calculated based on an intuition. During calculating concurrency compensation each client considers that all other clients in the system have the same value of os_s as itself. However, this estimation does not reflect the real scenario at all. Moreover, the contribution of the term $R_s - 1/\bar{\mu}_s$ of (3.1) diminishes quickly when the client has a non-zero queue size estimate. Therefore, the score based on which the client selects the server, largely depends on $\hat{q}_s$ which is based on just an intuition that does not reflect the actual scenario of the cluster and can degrade the overall performance of the system.

Heron [26] addressed the problem of tail latency for heterogeneous workloads. First, it classifies the read request as small and large using a bloom filter [70]. Then calculates a score for each of the nodes of the cluster using the scoring mechanism of C3. Finally, it selects the best replica for routing the read request based on the score and the class information of the read request. As this replica selection mechanism uses the same scoring mechanism as C3 hence it also faces the same problem as C3 that is discussed above and as a consequence, it will not be able to react to the dynamic behavior of the geo-distributed systems.

In [27], W. Jiang et al. proposed a replica selection strategy that works in two steps. First chooses the replica servers corresponding to the least number of outstanding keys, and then sorts this subset of replica servers according to their response times. This way it selects the replica for a routing read request which has a lower number of outstanding keys as well as lower response time. This strategy is prone to herd behavior as it is unaware of the existence of other clients in the system who may also choose the same replica for routing the read request. As a consequence, more and more read requests are routed to a server with a lower outstanding keys and lower response time which in turn makes the server a hot spot and increase the tail latency of the system.

Jun Wu in [5], proposed a replica selection strategy that adopted the score calculation function of C3. Like C3, it considered network latency, service time, and queue size for calculating the score. Moreover, they added the waiting time in the queue to the score functions to avoid selecting a replica with a long waiting time. However, this strategy recalculates the score at every 100 ms like dynamic snitch and hence may suffer from the same problem as dynamic due to the fixed time interval.

Table 3.1 shows the comparison of the state-of-the-art replica selection strategies that are discussed above. None of the strategies considered all important factors that should be considered during replica selection. C3 and Heron use 3 factors, [5] uses 4 factors, however they calculated

Replica Selection Strategies vs Characteristics	Used Factors	Adaption to Dynamic Environ-ment	Use of Intuition	Prone to Herd Behavior
Dynamic Snitch [24]	Latency	Partially	No	Yes
C3 [25]	Queue size, Service time, Response time	Partially	Yes	Partially
Heron [26]	Queue time, Service time, Response time	Partially	Yes	Partially
L2 [27]	Number of Outstanding Keys, Response Time	Partially	No	Yes
[5]	Queue size, Service time, Response time, Waiting time in Queue	Partially	Yes	Partially

Table 3.1: Comparison of the state-of-the-art replica selection strategies.

the score of each node based on an intuition that does not reflect the actual scenario of the distributed systems. Dynamic snitch uses only the latency information while L2 uses a number of outstanding keys and response time as factors for calculating the score. All of the replica selection strategies are partially adaptable to the dynamic behavior of the distributed systems. Both dynamic snitch and L2 are unaware of the existence of the other clients in the system hence are prone to herd behavior. On the other hand, C3,Heron, and [5] are partially prone to herd behavior.

Areas of Performance Optimization of the Replica Selection Strategies

The request duplication and request reissue strategies relay on redundant request whereas the replica selection strategies does not have such dependency. Moreover, designing a request duplication or request reissue strategies atop an efficient replication selection strategy may improve their performance. According to our discussion in section 3.4, it is seen that none of the state-of-the-art replica selection strategies are efficient enough to reduce tail latency and adapt to the dynamic behavior of the geo-distributed and there are several areas of optimization. Firstly, none of the replica selection strategies used all the important factors that should be

considered during replica selection. Hence, effective use of all important factors can improve the performance of the replica selection strategies. Secondly, none of the replica selection strategies are fully adaptable to the dynamic nature of the geo-distributed systems and this is another important area of optimization. Thirdly, Some of the replica selection strategies used intuition for calculating the score of each of the nodes which does not reflect the actual scenario of the geo-distributed systems and hence can reduce its performance. Hence, we should avoid intuition during calculating the score of the nodes. Finally, none of the replica selection strategies can fully avoid the herd behavior problem which has a greater impact on the tail latency of the geo-distributed systems. Hence, designing a replica selection strategy that can avoid herd behavior can improve the performance of the replica selection strategies.

Summary

In this chapter, first, we discussed some related state-of-the-art replica selection strategies that reduce tail latency in the distributed systems. Then we discussed some contemporary and related replication selection strategies. We also present a comparison among the replica selection strategies. Based on our discussion, finally, we identified some areas of performance optimization of the replica selection strategies.

Chapter 4

Reducing Tail Latency in Locally Distributed Systems

The main objective of this research is to develop an efficient replica selection strategy for reducing tail latency in geo-distributed systems. Keeping this in mind, first, we develop a replica selection strategy for the locally distributed systems. We did so for developing our prediction model first. In this chapter, we discussed the methodology of our replica selection strategy for reducing tail latency in locally distributed systems. We also present our implementation detail and experimentation results.

Proposed Method

We proposed a prediction based replica selection mechanism that predicts the latency of individual server s of the cluster. It then calculates a score for each server based on this latency. This score is calculated and used by the clients to select the best server in the system to route the read request.

Feedback from the Servers

During each read response the server sends queuesize q_s and service time $\frac{1}{\mu_s}$ (inverse of service rate μ_s) as the feedback information to the clients. These are used as dependent variables in (3.2) for predicting the latency l_{ps} using multiple linear regression. Servers also provide the latency l_s as feedback to the clients that is used for calculating the coefficients of the equation of the regression.

Finding Equation of Regression for Predicting Latency

The latency of a read request directly depends on the queuesize and service time of that server. At any specific time, the latency of a read request is equal to the product of the queuesize and service time of that server at that specific time as in (4.1).

$$l_s(t) = q_s(t) \times \frac{1}{\mu_s}(t) \tag{4.1}$$

Where l_s is the latency, q_s is the queuesize and $1/\mu_s$ is the service time of each server s at any specific time t. Based on this relationship we developed an equation of multiple linear regression in (4.2) that predicts the latency of each server s.

$$l_{ps} = a + b \cdot q_s + \frac{c}{\mu_s} \tag{4.2}$$

Where l_{ps} is the predicted latency of server s and a, b, c are the coefficients those are calculated in (4.3), (4.4) and (4.5) respectively. You can find the derivation of the coefficients a,b, and c in B.

$$a = \bar{l_s} - b \cdot \bar{q_s} - \frac{c}{\bar{\mu_s}} \tag{4.3}$$

where $\bar{l_s}$, $\bar{q_s}$ and $\frac{1}{\bar{\mu_s}}$ are the average latency, queuesize and service time of each server s respectively.

$$b = \frac{(\sum(\frac{1}{\mu_s})^2)\ (\sum q_s l_s)\ (\sum \frac{q_s}{\mu_s})\ (\sum \frac{l_s}{\mu_s})}{(\sum q_s^2) \cdot (\sum(\frac{1}{\mu_s})^2) - (\sum \frac{q_s}{\mu_s})^2} \tag{4.4}$$

$$c = \frac{(\sum q_s^2)\ (\sum \frac{l_s}{\mu_s})\ (\sum \frac{q_s}{\mu_s})\ (\sum q_s . l_s)}{(\sum q_s^2) \cdot (\sum(\frac{1}{\mu_s})^2) - (\sum \frac{q_s}{\mu_s})^2} \tag{4.5}$$

Smoothing the Predicted Latency

We derived (4.2) for predicting latency where we use q_s and $\frac{1}{\mu_s}$ directly as parameters. However, to make the prediction more real we use EWMA of both the parameters instead of using these directly. We represent these by $\hat{q_s}$ and $\frac{1}{\hat{\mu_s}}$ respectively and get a new smooth latency $\hat{l}_{ps}$ in (4.6).

$$\hat{l}_{ps} = a + b \cdot \hat{q}_s + \frac{c}{\hat{\mu}_s} \tag{4.6}$$

Considering the Effect of disk I/O Operations of Servers

Considering only the predicted latency of the servers is not sufficient enough for choosing the best server for routing read requests. The disk input/output (I/O) operations of servers also

penalizes the disk read operations and increases the disk read latency [29]. This extended disk read latency increases the overall read latency in a distributed system. For example, compaction is such a process in Cassandra which takes as input two or more SSTables and output a new SSTable [71]. The compaction process may need to do a lot of disk I/O operations frequently which can increases the overall read latency [69]. Hence for more efficient replica selection we also consider the impact of disk I/O operations of a server as a selection criteria. With every read response, the servers feedback the current disk I/O operations per second (IOPS) of that server and we represent this IOPS by r_s. Like the q_s and $\frac{1}{\mu_s}$, clients maintain an EWMA of r_s for smoothing the value and is represented by $\hat{r}_s$.

Replica Selection Strategy Using the Score

Finally, clients calculate the score for each server s using (4.7).

$$\varphi_s = \hat{l}_{ps} + \hat{r}_s \tag{4.7}$$

The server with the lowest score is considered to be the best node for read request routing. The workflow of our proposed replica selection strategy is depicted in Fig. 4.1.

Replica Selector

Whenever a read request comes, the replica selector sorts those servers that contain the requested replica, selects the best server based on the score and sends a read request to it as Algorithm 1. The replica selector also invokes the latency tracker (Algorithm 1, line 6) to keep account the latency l_s of the selected server s.

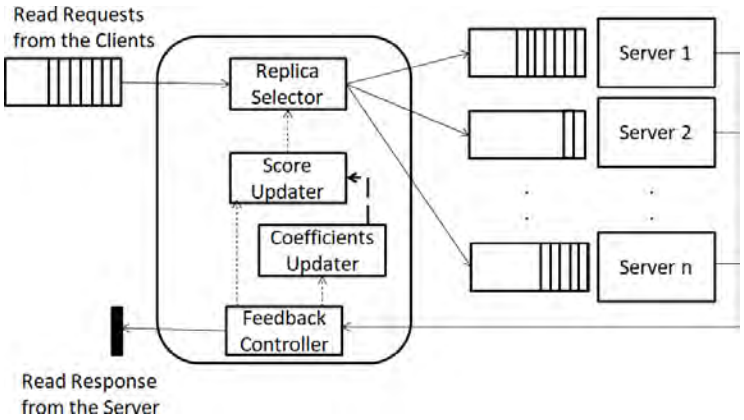


Figure 4.1: Proposed replica selection strategy

Feedback Controller

The feedback controller collects q_s, r_s and $\frac{1}{\mu_s}$directly from the feedback of the servers during each read response and sends them to the score updater (Algorithm 2, line 4). Another task of the feedback controller is to collect the l_s of each server s from the latency tracker on every read response (Algorithm 2, line 2). It then sends q_s,$\frac{1}{\mu_s}$and l_s to the coefficients updater (Algorithm 2, line 5).

Coefficients Updater

Initially, all the coefficients of (4) are zero which makes the predicted latency of all the servers zero. In this situation, only the $\hat{r}_s$ value contribute to the score of the servers. Over time the coefficient updater recorded the q_s, $\frac{1}{\mu_s}$, and l_s and update $\hat{q}_s$, $\frac{1}{\hat{\mu}_s}$, and $\hat{l}_s$ (Algorithm 3, line 2) as a background task. The coefficient updater receives q_s,$\frac{1}{\mu_s}$ and l_s as feedback from the feedback controller. After a specific time interval, the coefficients of (4) are recalculated (Algorithm 3, line 4) and updated (Algorithm 3, line 5). This recalculation is done for coping with the changing situation of the system. The time interval of coefficients recalculation is specified by the users of the system.

Score Updater

The score updater collects q_s, r_s and$\frac{1}{\mu_s}$from the feedback controller and update EWMA of each of these ($\hat{q}_s$, $\hat{r}_s$ and $\frac{1}{\hat{\mu}_s}$) (Algorithm 4, line 2). It then recalculates and updates the score for each server s (Algorithm 4, line 3,4 and 5) that is used for choosing the best replica for routing read requests.

Implementation

We implemented our proposed replica selection mechanism in Cassandra 3.0 version.We implemented the replica selection mechanism at each of the nodes of the cluster. In Cassandra, the client can send a read request to any of the nodes in the cluster. After receiving the read request a node acts as a coordinator for that specific read request [72]. The coordinator selects the best server for routing read requests by running our implemented replicaSelector() method. The coordinator tracks the queuesize, service time, IOPS rate, and the latency. For calculating the EWMA of queuesize and service time, we set the smoothing factor alpha to 0.8 and we did so to assign more weight to the recent values than the values from the long past. In Cassandra, for calculating IOPS rate we use the severity information of each server that is shared among the nodes in the cluster via gossip. The severity of a node in Cassandra is a numerical value that represents the amount of compaction events occurring in that node [73].

We use this severity value without calculating EWMA because the nodes scale this relative to its load and the amount of task required for compaction as mentioned in line no. 306 of DynamicEndpointSnitch.java file of Cassandra 3.0 version [74]. For updating score we modified the code of the updateScores() method of DynamicEndpointSnitch.java file at line no. 274 [74]. We implemented coefficientUpdater() method for updating and calculating the coefficients and our implemented method consists of 222 lines of code. In our implementation, we calculate the coefficients for once this is because we conducted the experiments for a short time.

Experimentation

Experiment Setup

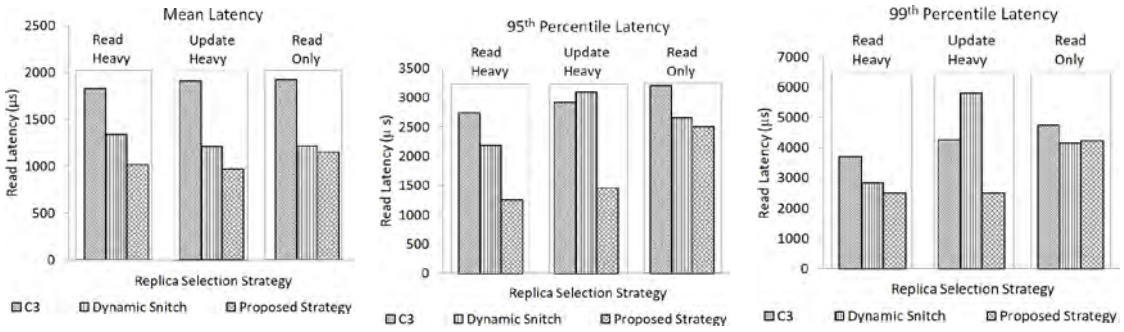
For evaluation, we deployed a 15 node Cassandra cluster with 5 racks. Each of the nodes comprises of 2 GB RAM, core i3 processor, and a HDD hard disk of 264 GB capacity. We used Ubuntu 16.04 operating system for deploying Cassandra cluster. Our used snitch was *GossipingPropertyFileSnitch* and the replication factor was set to 3. We set *Num_token* to 256 for even distribution of the tokens to the Cassandra nodes. As we do not have real workloads hence we used the YCSB [75] for generating data sets and running workloads. We inserted 20 million records each with 1 KB of length which are generated from the YCSB. We used an instance of YCSB running in a separate machine which was generating the workloads to the Cassandra cluster. The YCSB instance was consists of 10 request generator with 1 million read operation. We used the Zipfian distribution as workload distribution with the default value of the Zipf parameter. We used three different types of workloads for evaluating our system: workload A: Update heavy workload (50/50 reads/writes), Workload B: Read mostly workload (95/5 reads/writes), Workload C: Read only (100/0 reads/writes).

Experimental Results

For evaluation, we conducted 4 different experiments and each experiment includes 1 million operations. We repeated each experiment 5 times. We conducted the experiments in Cassandra and evaluate the dynamic snitch, C3, and our proposed system.

Comparison of latency at different workloads

Fig. 4.2 depicts the latency characteristics of Cassandra when using C3, dynamic snitch, and our proposed strategy. Our proposed strategy has the lowest mean, 95^{th} percentile, and 99^{th} percentile latency almost at all workloads except the 99th percentile latency at read-only workload. With update-heavy workload, the difference between mean and 99th percentile latency is $2344\ \mu s$



(a) (b) (c)

Figure 4.2: Latency of Cassandra when using C3, dynamic snitch and our proposed strategy (a) mean latency (b) 95^{th} percentile latency (c) 99^{th} percentile latency.

for C3, 4584.8 μs for dynamic snitch and 1518.6 μs for our proposed strategy. C3 has $1.5x$ and dynamic snitch has $3.0x$ more increase in latency compared to our proposed strategy. Our proposed strategy has also the lowest increase in latency with read-heavy workload. The only exception has seen at read-only workload where our proposed strategy's latency increase is slightly more than the two others. It is also seen dynamic snitch has lower latency compared to C3 almost in all scenarios except the 95^{th} and 99^{th} percentile latency with update-heavy workload.

Comparison of throughput at different workloads

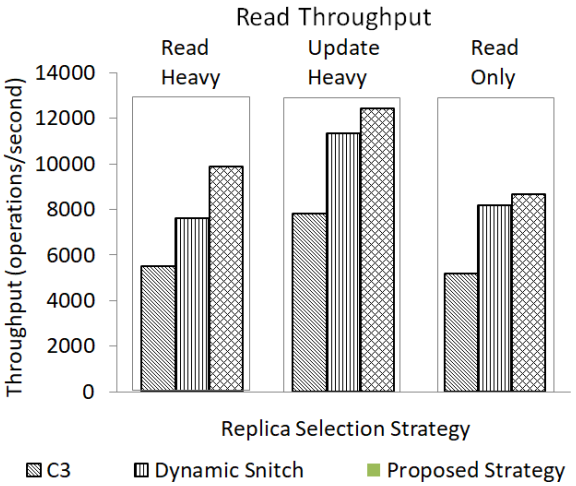


Figure 4.3: Comparison of read throughput at different workloads. Our proposed strategy has higher throughput at all the scenarios.

Fig. 4.3 depicts the comparison of throughput of the Cassandra when using C3, dynamic snitch, and our proposed strategy. With all the scenarios Cassandra has higher throughput when using our proposed system other than using C3 or dynamic snitch. With read-heavy workload, our

proposed strategy has 79% more throughput than C3 and 30% more throughput than dynamic snitch. With update-heavy workload, our system has 59% and 9% more throughput and with read-only workload, our system has 67% and 6% more throughput than the C3 and dynamic snitch respectively. We also observed that Cassandra has higher throughput in all scenarios when using dynamic snitch than C3.

Performance at higher workload

We conducted this experiment for observing the performance of C3, dynamic snitch, and our proposed strategy at a higher workload. We increase the number of workload generators from 10 to 30 and we conducted this experiment only with read-heavy workload. Fig 4.4 compared the performance of the replica selection strategies at a higher workload. This figure also depicts how the latency of different strategies changes when the number of workload generators increases

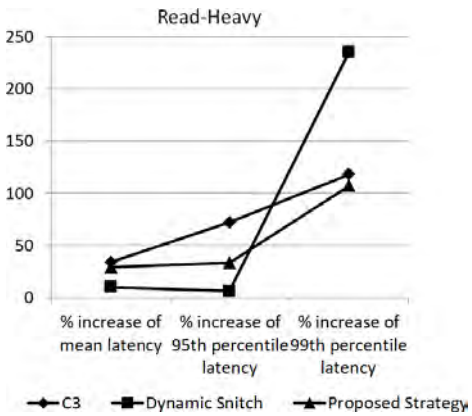
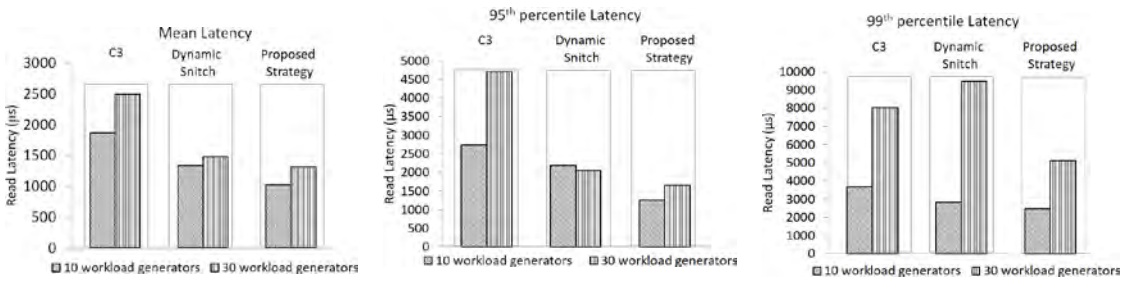


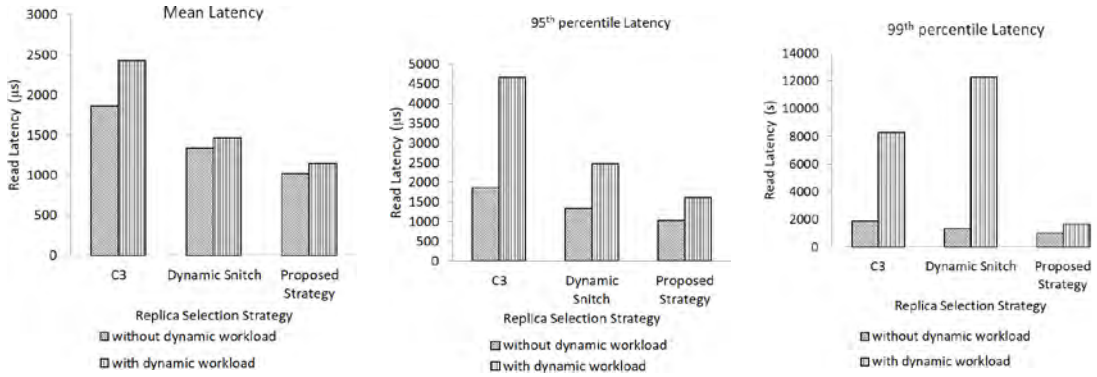
Figure 4.5: Percentage increase of mean, 95^{th} percentile and 99^{th} percentile latency of the C3, dynamic snitch and our proposed strategy with higher workloads

from 10 to 30. As the number of workload generators increases by 200%, the 99^{th} percentile latency for our proposed strategy increases 106% whereas the increase is 118% for C3 and



(a) (b) (c)

Figure 4.4: Latency of Cassandra when using C3, dynamic snitch and our proposed strategy with higher workloads (a) mean latency (b) 95^{th} percentile latency (c) 99^{th} percentile latency.



(a) (b) (c)

Figure 4.6: Latency of Cassandra when using C3, dynamic snitch and our proposed strategy at dynamic workloads (a) mean latency (b) 95^{th} percentile latency (c) 99^{th} percentile latency.

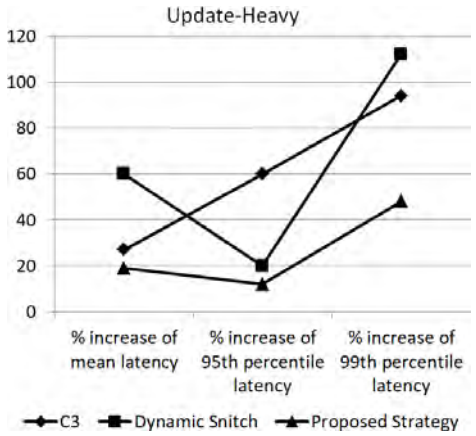


Figure 4.7: Percentage increase of mean, 95^{th} percentile and 99^{th} percentile latency of the C3, dynamic snitch and our proposed strategy as the workload changes dynamically.

235% for dynamic snitch. Fig. 4.5 shows that though the percentage increase of mean and 95^{th} percentile latency of dynamic snitch is lower than both our proposed system and C3 however the 99^{th} percentile latency of dynamic snitch increases extremely sharply. This represents the performance fluctuation behavior of Cassandra dynamic snitch. We also observed that the percentage increase of latency of all the metrics of our proposed strategy is always less than that of the C3.

Performance at dynamic workload fluctuation

We conducted this experiment for understanding the behavior of C3, dynamic snitch, and our proposed strategy in a dynamically changing workload situations. For this purpose, we initially started 20 workload generators with read-heavy workload and after 154 sec we started another 10 workload generators with update-heavy workload. Fig. 4.6 shows the latency variation of the replica selection strategies only at the update-heavy workloads. The increase of the mean, 95^{th}

percentile, and 99^{th} percentile of our proposed strategy is the lowest among the replica selection strategies. Fig. 4.7 depicts the percentage increase of the mean, 95^{th} percentile, and 99^{th} percentile latency of the C3, dynamic snitch, and our proposed strategy. The replica selection strategies behaved the same way as they do at a higher workload situation as in Fig. 4.5. Our proposed strategy has the lowest percentage increase at all the metrics than the C3 and dynamic snitch. Dynamic snitch again shows the performance fluctuation behavior and its 99^{th} percentile latency increased rapidly.

Summary

In this chapter, we extensively studied two state-of-the-art replica selection strategies: Cassandra dynamic snitch and C3. Based on our findings we meticulously designed a prediction based replica selection strategy for reducing the tail latency in the distributed system. In Fig. 4.5 and 4.7 though the percentage increase of all the metrics of our strategy is less than the C3 however the percentage increase of C3 is smoother than our proposed strategy. One of the reasons for the wavy percentage increase of latency of our strategy is the uncontrolled herd behavior of the read requests where more and more read requests are routed to a server with a lower score which in turn makes the server a hot spot and increase the tail latency of the system. In the future, we will add a rate limiter in our system which will limit the number of read requests routed to a specific server. Despite all these issues still, our proposed strategy decreases the overall latency as well as increases the overall throughput of the system compared to C3 and dynamic snitch.

Chapter 5

Reducing Tail Latency in Geo-Distributed Systems

None of the state of the art replica selection strategies can meet the challenges of geo-distributed systems. In this chapter, we describe the design and implementation of an efficient replica selection strategy that can meet all the challenges of the geo-distributed systems. We proposed a prediction-based strategy that consider all the important factors related to the replica selection. Dynamic snitch is designed based on latency only and does not take into account the load across the node for calculating score which is important for adapting to time-varying performance of geo-distributed systems. To address this problem, L2 designed a replica selection strategy in which each node selects the server to which it has the lowest number of outstanding keys which is a measure of load across the cluster. However, this strategy uses only local information that does not necessarily reflect the actual scenario of the cluster. Moreover, in reality, workloads get skewed and access patterns change over time [51]. Hence, strategies that are designed based on only local information are not efficient enough to adapt to the dynamic nature of the geo-distributed systems and any indifferent design decision may result in herd behavior. Maybe one possible solution is to use global load information. However, it is unrealistic to assume that all the nodes of a geo-distributed system have up-to-date information. On one hand, the global information may be updated periodically or the time delay for a replica request to reach the selected node is long enough that the load information is out of data as long as the replica request arrives at the node [59]. On the other hand, obtaining up-to-date global information requires complex system design and increases computational complexity [59]. To reduce the complexity as well as to make the replica selection strategy more sensitive to the dynamic nature of the geo-distributed systems C3, [5], and Heron calculate a concurrency compensation value based on which the final score is calculated that is used for selecting the best replica for routing the read request. However, the calculation of concurrency compensation is based on just an intuition that does not reflect the actual scenario of the cluster and can degrade the overall performance of the systems. For a better understanding of the actual scenario of the

system, in chapter 4 we proposed a prediction-based replica selection strategy that performs better than the other strategies in the locally distributed systems but not efficient enough for the geo-distributed systems. Hence considering the discussed issues we meticulously designed a new prediction-based replica selection strategy for reducing tail latency in the geo-distributed systems that will be able to adapt to the dynamic nature of the systems. Our proposed replica selection strategy reduces the tail latency of the system as well as increases the overall throughput.

Proposed Method

We proposed a prediction based replica selection mechanism that predicts the latency of individual servers of a geo-distributed cluster. It then calculates a score for each server based on this latency. This score is calculated and used by the clients to select the best server in the system to route the read request.

Finding Equation of Regression for Predicting Latency

As discussed in section 4, in the geo-distributed system the latency of a read request is linearly dependent on both the RTT and the response time of the server. The time it takes to process individual read requests in a geo-distributed system is determined as (5.1).

$$L = RTT_s + R_s \tag{5.1}$$

Where, L is the total latency, RTT_s is the round trip time of the request, and R_s is the response time of the server s. Therefore, if we know the RTT and the response time of a server then we can predict the latency of a read request by using multiple linear regression. Multiple linear regression is a statistical technique that uses several independent variables to predict the value of a dependent variable [76]. The general equation of multiple linear regression for p independent variables is shown in (5.2).

$$Y = \beta_0 + \beta_1 \cdot X_1 + \beta_2 \cdot X_2 + ... + \beta_p \cdot X_p \tag{5.2}$$

where Y is the dependent variable whose value the regression model predicts, X_1,X_2,. ,X_p are the independent variables, β_0 is the y-intercept and β_1, β_2,. ,β_p are slope the coefficients for each independent variable.

In our case, we have two independent variables RTT and response time and the dependent variable is latency that we are going to predict. Therefore, based on the relationship of (5.1) we develop an equation of multiple linear regression in (5.3) that predicts the latency of each server s.

$$L_{ps} = a + b \cdot RTT_s + c \cdot R_s \tag{5.3}$$

Where L_{ps} is the predicted latency of server s and a is the y-intercept, b measures the unit change in the latency as the RTT changes, c measures the unit change in the latency when the response time changes. The values of a,b, and c are calculated according to (5.4), (5.5) and (5.6) respectively. You can find the detail derivation of the coefficients a,b, and c in B .

$$a = \bar{L_s} - b \cdot \bar{RTT_s} - c \cdot \bar{R_s} \tag{5.4}$$

where $\bar{L_s}$, $\bar{RTT_s}$ and $\bar{R_s}$ are the average latency, round trip time and response time of each server s respectively.

$$b = \frac{\sum_s R^2 \cdot \sum (RTT_s \cdot L_s) - \sum (RTT_s \cdot R_s) \cdot \sum (L_s \cdot R_s)}{\sum RTT^{s\,2} \cdot \sum R^2 - (\sum (RTT^s \cdot R^s))^2} \tag{5.5}$$

$$c = \frac{\sum RTT_s^{\,2} \cdot \sum (L_s \cdot R_s) - \sum (RTT_s \cdot R_s) \cdot \sum (RTT_s \cdot L_s)}{\sum RTT^{s\,2} \cdot \sum R^2 - (\sum (RTT^s \cdot R^s))^2} \tag{5.6}$$

During each read response the server sends queue size qs and service time $\frac{1}{\mu_s}$ as the feedback information to the clients wheres the value of RTT_s and L_s are keep tracked by the clients.

Smoothing the Predicted Latency

We derived (5.4) for predicting latency where we use RTT_s and R_s directly as parameters. However, to make the prediction more real we use Exponentially Weighted Moving Average (EWMA) of both the parameters instead of using these directly. We represent these by $\hat{RTT_s}$ and $\hat{R_s}$ respectively and get a new smooth latency $\hat{L_{ps}}$ in (5.7).

$$\hat{L}_{ps} = a + b \cdot \hat{RTT_s} + c \cdot \hat{R_s} \tag{5.7}$$

Accuracy Analysis of the Prediction Model

To analyze the accuracy of our prediction model we calculated the percentage error rate using the equation (5.8).

$$percentagepredictionerror = \frac{|actualvalue - predictedvalue|}{actualvalue} \times 100 \tag{5.8}$$

To measure the percentage error we recorded the predicted latency and actual latency for a whole day. According to our finding, our prediction model has an average percentage error of ±8.51% with 8.70% average positive error and 7.329% average negative error. We plot the predicted vs.

actual latency in fig.5.1 and we get almost a straight line with a slope of 1. That means that the value of the predicted latency and the actual latency is very close and it ensures that our prediction model fits the plane of the regression quite accurately.

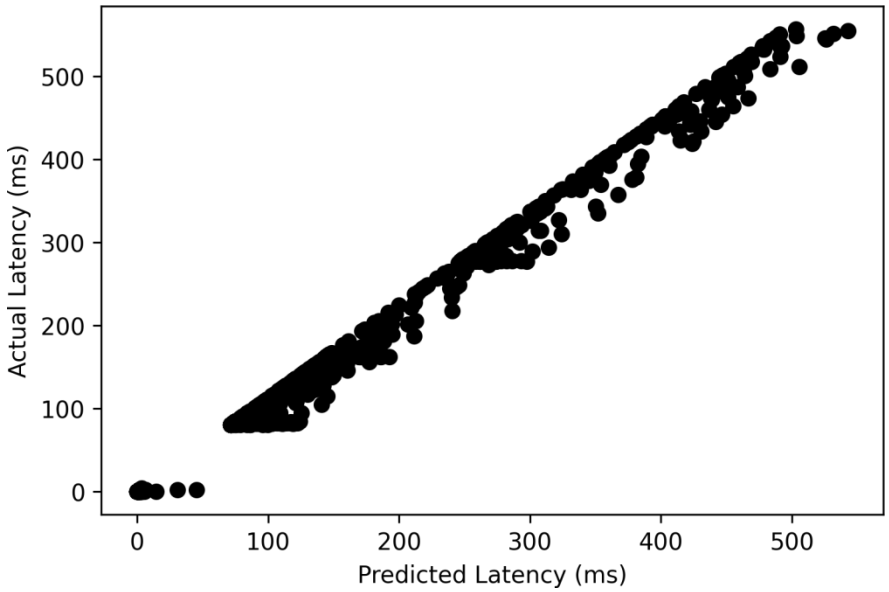


Figure 5.1: Predicted latency vs. actual latency

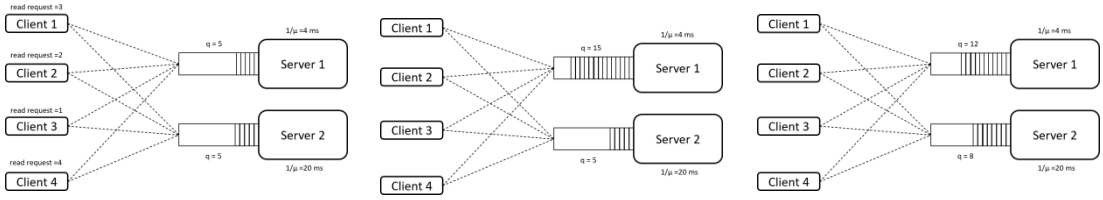
Considering the Effect of disk I/O Operations of Servers

Considering only the predicted latency of the servers is not sufficient enough for choosing the best server for routing read requests. The disk input/output (I/O) operations of servers also penalizes the disk read operations and increases the disk read latency [29]. This extended disk read latency increases the overall read latency in a distributed system. Hence for more efficient replica selection we also consider the impact of disk I/O operations of a server as a selection criteria. With every read response, the servers feedback the current disk I/O operations per second (IOPS) of that server and we represent this IOPS by R_{ios}. Like the RTT_s and R_s, clients maintain an EWMA of R_{ios} for smoothing the value and is represented by $\hat{R}_{ios}$. Finally, clients calculate the score for each server s using (5.9).

$$\varphi_s = \hat{L}_{ps} + \hat{R}_{ios} \tag{5.9}$$

Handling the Problem of Herd Behavior

The response time of a server can be determined from the queue size and the service time of that server as in (5.10).



(a) (b) (c)

Figure 5.2: Routing read request in the distributed systems (a) clients with new replica read request where the distributed system has nodes with heterogeneous service time(b) more and more read requests are routed to the faster server and results in herd behavior (c) read requests are routed in more balanced way and reduced the problem of herd behavior

$$R_s = q_s \times \frac{1}{\mu_s} \tag{5.10}$$

where R_s is the response time, q_s is the queue size and $\frac{1}{\mu_s}$ is the service time of the server s. Using the above estimation, now a score for each server can be calculated by calculating the value of R_s. However, considering the fact that there are also other clients in the systems who can send read requests and also there is service time variations in the systems this scoring mechanism is not efficient enough for ranking the replica. Consider the scenario of fig. 5.2(a), there are two servers with service time $4ms$ and $20ms$ and each with queue size 5. Now, if four clients come with a total of 10 read requests and nodes are selected based on the calculated score as in (5.10) then more and more read request will be routed to the faster server and results in herd behavior as shown in fig 5.2(b). Now, if the service time of the faster server increases due to unpredictable performance fluctuations then the requests in it's queue will have to wait for a longer time. By penalizing the server with longer queue this waiting time can be reduced. This can be achieved by raising the power of q_s in (5.10) by a higher degree. We choose to increase the degree by 3 as shown in (5.11) which provides a better trade-off between choosing a faster server and service time fluctuations as discussed in [25].

$$R_s = q_s^3 \times \frac{1}{\mu_s} \tag{5.11}$$

After using (5.2) as scoring function, the distribution of read requests will be more balanced and the problem of herd behavior can be avoided as shown in fig. 5.2(c).

Replica Selection Strategy Using the Score

In our proposed strategy, clients calculate the score for each server s using (5.9) and the server with the lowest score is considered to be the best node for read request routing. The workflow of our proposed replica selection strategy is depicted in Fig. 5.3.

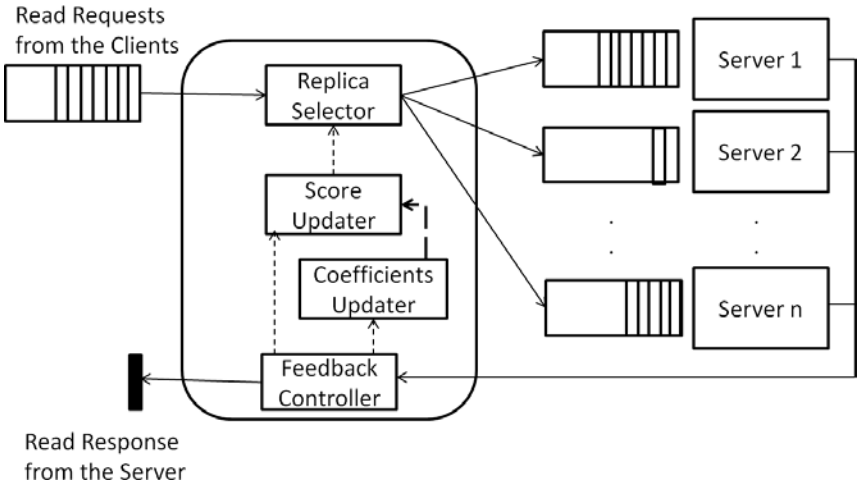


Figure 5.3: Proposed replica selection strategy

Replica Selector

Whenever a read request comes, the replica selector sorts those servers that contain the requested replica, selects the best server based on the score and sends a read request to it as Algorithm 5. The replica selector also invokes the latency tracker (Algorithm 5, line 6) and round trip time tracker (Algorithm 5, line 7) to keep account the latency L_s and RTT_s of the selected server s.

Feedback Controller

The feedback controller collects q_s, R_{ios} and $\frac{1}{\mu_s}$ directly from the feedback of the servers during each read response and sends them to the score updater (Algorithm 6, line 6). Another task of the feedback controller is to collect the L_s (Algorithm 6, line 2) and RTT_s (Algorithm 6, line 3) of each server s from the latency tracker on every read response. It then sends RTT_s, q_s, $\frac{1}{\mu_s}$ and L_s to the coefficients updater (Algorithm 6, line 7).

Coefficients Updater

Initially, all the coefficients of (5.7) are zero which makes the predicted latency of all the servers zero. In this situation, only the $\hat{R_{ios}}$ value contribute to the score of the servers. Over time the coefficient updater records the RTT_s, q_s, $\frac{1}{\mu_s}$, and L_s and updates $\hat{RTT}_s$ $\hat{q}_s$, $\frac{1}{\hat{\mu}_s}$, and $\hat{L}_s$ (Algorithm 7, line 2) as a background task. The coefficient updater receives RTT_s, q_s, $\frac{1}{\mu_s}$ and L_s as feedback from the feedback controller. After a specific time interval, the coefficients of (5.7) are recalculated (Algorithm 7, line 4) and updated (Algorithm 7, line 5). This recalculation is done for coping with the changing situation of the system. The time interval of coefficients recalculation is specified by the users of the system.

Score Updater

The score updater collects q_s, R_{ios}, $\frac{1}{\mu_s}$ and RTT_s from the feedback controller and update EWMA of each of these ($\hat{q}_s$, $\hat{r}_s$, $\hat{R}_{ios}$ and $\frac{1}{\hat{\mu}_s}$) (Algorithm 8, line 2). It then recalculates and updates the score for each server s (Algorithm 8, line 3,4 and 5) that is used for choosing the best replica for routing read requests.

Implementation

We implemented our proposed replica selection mechanism in Cassandra 3.0 version [77]. For calculating the EWMA of all the considered factor, we set the smoothing factor alpha to 0.8 and we did so to assign more weight to the recent values than the values from the long past. We ensured the stability of our derived multiple linear regression model using the recursive least square test. In Cassandra, for calculating IOPS rate we use the severity information of each server that is shared among the nodes in the cluster via gossip [73]. The severity of a node in Cassandra is a numerical value that represents the amount of compaction events occurring in that node. We use this severity value without calculating EWMA because the nodes scale this relative to its load and the amount of task required for compaction as mentioned in line no. 306 of DynamicEndpointSnitch.java file of Cassandra 3.0 version. For updating score we modified the code of theupdateScores() method of DynamicEndpointSnitch.java file at line no. 272 [74]. We implemented coefficientUpdater() method for updating and calculating the coefficients and our implemented method consists of 223 lines of code. In our implementation, we calculated the coefficients for once because we conducted the experiments for a short time.

Experimentation

Experiment Setup

For evaluation, we deployed a Cassandra cluster in Amazon EC2 with 15 c5d.large instances. There were 5 racks in the cluster which were geographically distributed across Singapore, US East (N. Virginia), Europe (Paris), Canada (Central), and Australia (Sydney). Each of the nodes comprises of 4GB RAM and a hard disk of 264 GB capacity. We used Ubuntu 16.04 operating system for deploying Cassandra cluster. Our used snitch was GossipingPropertyFileSnitch and the replication factor was set to 3. We set Numtoken to 256 for even distribution of the tokens to the Cassandra nodes. As we do not have real workloads hence we used the YCSB [75] for generating data sets and running workloads. We inserted 100 million records each with 1 KB of length which are generated from the YCSB. We used an instance of YCSB running in a separate c5d.large instance which was generating the workloads to the Cassandra cluster. The YCSB

instance was consists of 30 request generator with 10 million read operation. We used the Zipfian distribution as workload distribution with YCSB prescribed Zipf parameter p=0.99. Such a value of Zipfian distribution is specifically used to distribute the key in the keyspace in such a scattered way that is analogous to the actual web applications. For the Zipfian distribution, when choosing a record, some records have more chance to be chosen and generates more unexpected situations as the real world scenario of the geo-distributed systems. We used three different types of workloads for evaluating our system: workloadA: Update heavy workload (50/50 reads/writes), WorkloadB: Read heavy workload (95/5 reads/writes), Workload C: Read only (100/0 reads/writes).

Experimental Results

For evaluating the performance of our proposed strategy we conducted 4 different experiments each with 10 million operations generated by the YCSB. We repeated each experiment 5 times for better accuracy of the experimental outcome. We conducted all the experiments in Cassandra and compare the performance of our proposed strategy with [5], c3, dynamic snitch, and [6].

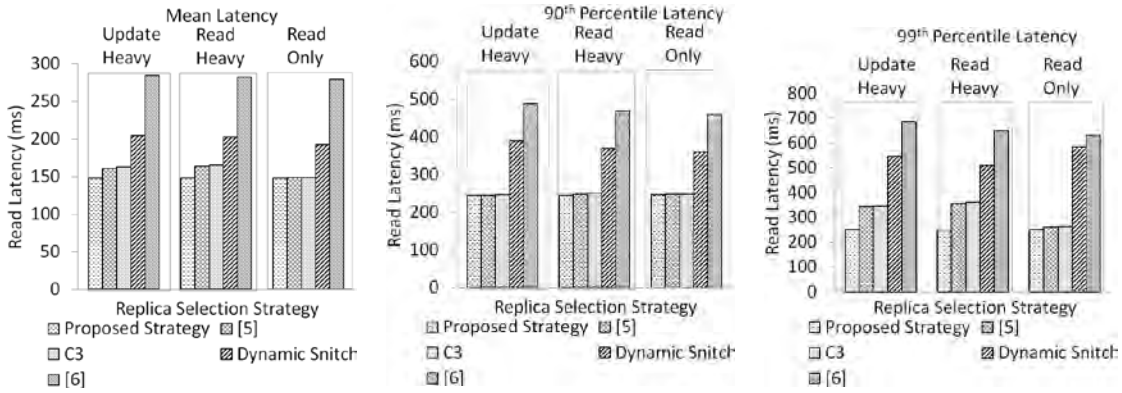
Comparison of Latency at Different workloads

Fig. 5.4 depicts the latency characteristics of Cassandra when using C3, [5], dynamic snitch, [6] and our proposed strategy. Our proposed strategy has the lowest mean, 90^{th} percentile, and 99^{th} percentile latency almost at all workloads. The only exception has been seen with read-only workload where the mean, 90^{th} percentile, and 99^{th} percentile latency of C3, [5], and our proposed strategy is almost equal. The main reason for such an outcome is that with a read-only workload no SSTable compaction occurs. SSTable compaction causes a temporary spike in disk space usage and disk IOPS rate [78]. In our strategy, we have used the IOPS rate in calculating the score that has no impact in such a situation. Hence, the performance of our proposed strategy, [5], and C3 remains same. With read-heavy workload, the difference between mean and 99th percentile latency is $365.266\ \mu s$ for [6], $306.716\ \mu s$ for dynamic snitch, $196.964\ \mu s$ for C3, $192.87\ \mu s$ for [5], and only $99.751\ \mu s$ for our proposed strategy. Dynamic snitch has

times, [6] has 3.66 times, c3 has 1.97, and [5] has 1.92 times increase in latency compared to our proposed strategy.

Comparison of Throughput at Different Workloads

Fig.5.5 depicts the comparison of throughput of the Cassandra when using [5], C3, dynamic snitch, [6] and our proposed strategy. By reducing latency across the replicas, our strategy makes better use of the system capacity and thereby increases the system throughput. In all the scenarios, except with read-only workload, Cassandra has higher throughput when using our



(a) (b) (c)

Figure 5.4: Latency of Cassandra when using C3, [5], dynamic snitch, [6] and our proposed strategy (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

proposed replica selection strategy. With update-heavy workload, our strategy has 11.98% more throughput than [5], 12% more throughput than C3, 89% more throughput than [6] and 17% more throughput than dynamic snitch. The experimental result is quite similar to read-heavy workload. However, with read-only workload, the throughput of both our proposed strategy and C3 is almost equal. That is because as we have seen in the previous experiment discussed previously that with the read-only workload the latency of our proposed strategy and C3 is almost the same because with read-only workload no SSTable compaction occurs.

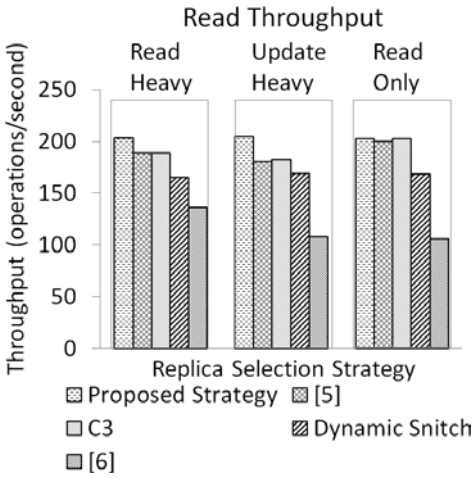
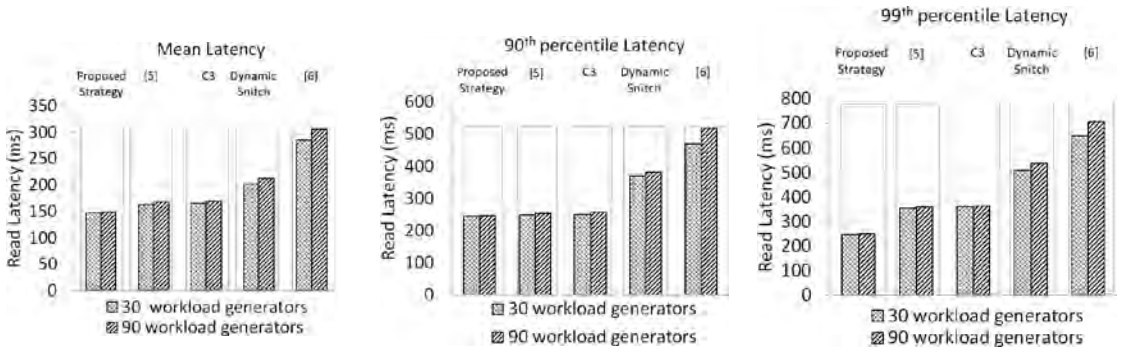


Figure 5.5: Comparison of read throughput at different workloads. Our proposed strategy has higher throughput at all the scenarios.

Performance at Higher Workloads

We conduct this experiment for analyzing the performance of C3, [5], dynamic snitch, [6], and our proposed strategy at a higher workload. We increase the number of workload generators from 30 to 90 and we conduct this experiment only with read-heavy workload. Fig. 5.6 depicts how the

latency of different strategies increase when the number of workload generators increases from 30 to 90. Still at higher workload the latency of our proposed strategy is lower than [5], C3, [6] and dynamic snitch. Fig. 5.7 shows that as the number of workload generators increased by 200%, the 90^{th} percentile latency of our proposed strategy increases 0.61% whereas the increase is 2.23% for [5], 2.73% for C3, 10.25% for [6] and 3.75% for dynamic snitch. Similarly, the percentage increase of mean and 99th percentile latency of our proposed strategy is lower than [5], C3, [6] and dynamic snitch. Moreover, the variation of percentage increase of latency is very low for our proposed strategy whereas the variation is comparatively higher for other strategies. The reason for such an improvement is that as we increased the workloads the queue size and disk IOPS also increased. The dynamic snitch does not consider queue size and IOPS, [6] does not consider RTT and IOPS, C3 estimates the queue size based on just an intuition that does not reflect the actual scenario of the servers and also it does not consider IOPS, and [5] does not consider IOPS. Only our proposed strategy considered all the metrics and it can predict the latency more accurately and thereby can select the replica in a better way compared to other replica selection strategies.



(a) (b) (c)

Figure 5.6: Latency of Cassandra when using [5], C3, dynamic snitch, [6] and our proposed strategy at a higher workload (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

To evaluate the performance in a higher workload situation, we also keep track of the throughput of our proposed strategy, [5], C3, dynamic snitch, and [6] as shown in fig. 5.8. As the number of workload generators increased from 30 to 90 the throughput also increases for all the strategies. However, our proposed system has the highest increase in throughput compared to all the other strategies. When the number of workload generators increased to 90, the throughput of our proposed strategy is 610.58 ops/sec (operations per second) whereas the throughput is 580.55 ops/sec for [5], 571.1 ops/sec for C3, 414.08 ops/sec for the dynamic snitch, and 301.9 ops/sec for [6].

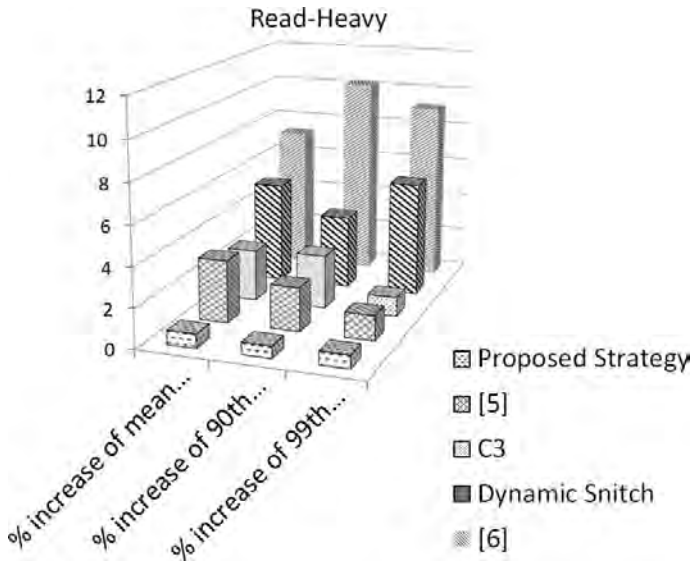


Figure 5.7: Percentage increase of mean, 95^{th} percentile, and 99^{th} percentile latency of the [5], C3, dynamic snitch, [6], and our proposed strategy with higher workloads

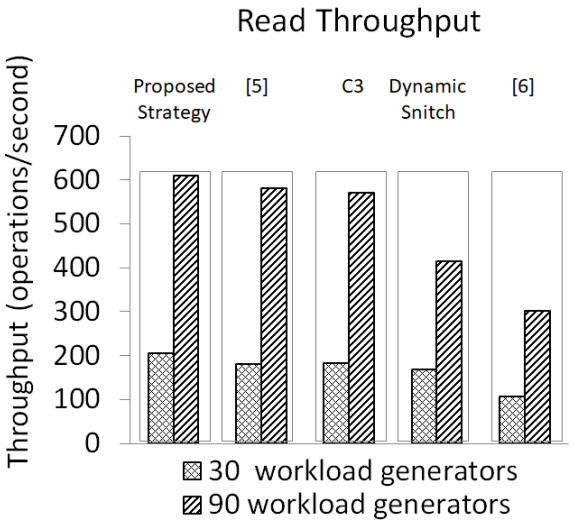
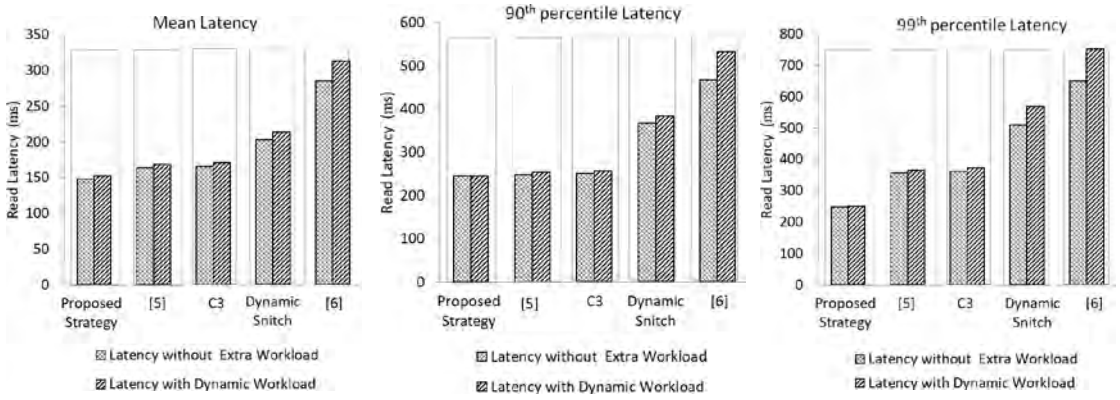


Figure 5.8: Throughput of Cassandra when using [5], C3, dynamic snitch, [6], and our proposed strategy at higher workloads



(a) (b) (c)

Figure 5.9: Latency of Cassandra when using [5], C3, dynamic snitch, [6] and our proposed strategy at a dynamic workload (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

Performance at Dynamic Workloads

We conduct this experiment for evaluating the behavior of C3, dynamic snitch, [6], [5], and our proposed strategy in a dynamically changing workload situations. For this purpose, we initially start 30 workload generators with read-heavy workload and after 330 sec we start another 30 workload generators with update-heavy workload. The replica selection strategies behaved quite the same way as they do at a higher workload situation and is shown in Fig. 5.9. Fig. 5.10 shows the percentage increase of mean, 90th and 99th percentile latency of our proposed strategy, C3, [6], [5], and dynamic snitch at dynamic workload. The percentage increase of our proposed strategy is lower than C3, [6], [5], and dynamic snitch. After entering into dynamic workload situation, the increase of 90th percentile latency for our proposed strategy is 0.3%, whereas the increase is 1.97% for [5],2.38% for C3, 13.67% for [6], and 4.14% for dynamic snitch. Our proposed strategy performed better at dynamic workload situations because of the same reason as it performed better at higher workload situations.

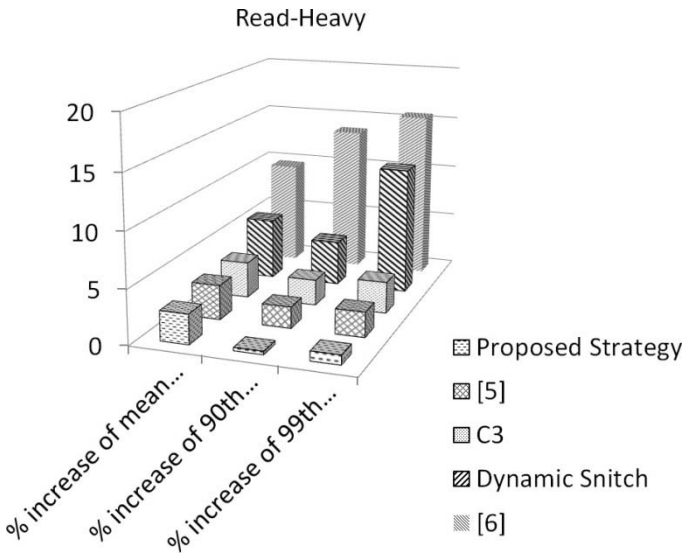


Figure 5.10: Percentage increase of mean, 90^{th} percentile and 99^{th} percentile latency of the C3, dynamic snitch, [6] and our proposed strategy with dynamic workloads

We also keep track of the throughput of our proposed strategy, [5], C3, dynamic snitch, and [6] to evaluate the performance in a dynamic workload situation. Fig. 5.11 shows as the system enters into the dynamic workload situation the throughput of our proposed strategy, [5], and C3 increases whereas the throughput of dynamic snitch and [6] decreases. At dynamic workload situation, the throughput of our proposed strategy is 212.66 ops/sec whereas the throughput is 202.4 ops/sec for [5], 199.58 ops/sec for C3, 164.14 ops/sec for the dynamic snitch, and 99.18 ops/sec for [6].

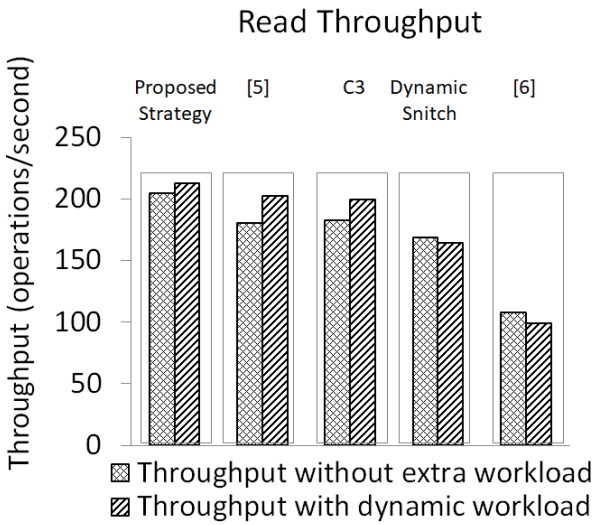
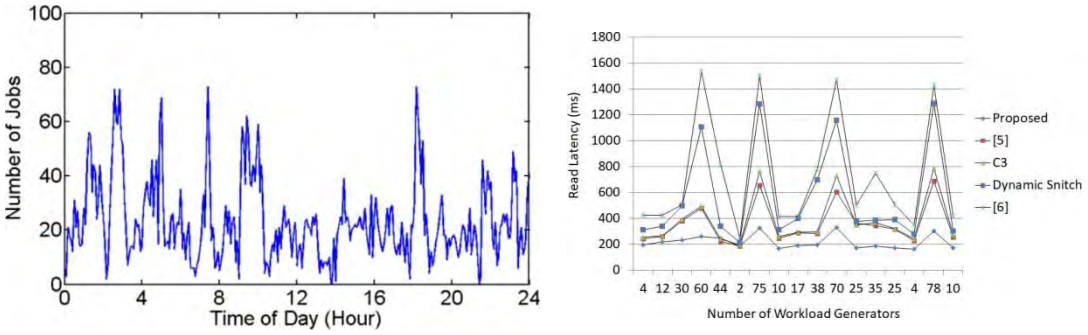


Figure 5.11: Throughput of Cassandra when using [5], C3, dynamic snitch, [6], and our proposed strategy at dynamic workload

To evaluate the performance of the replica selection strategies at real world dynamic workload situations we conducted another experiment based on a publicly available MapReduce trace as



(a) (b)

Figure 5.12: Performance at dynamic workload situation (a) Illustration of the traces for dynamic workload used in the experiment. (b) Latency distribution of the [5], C3, dynamic snitch, [6], and our proposed strategy for different number of workload generators for the first 8 hours of the MapReduce workload trace.

an example of dynamic workload. Chen et al. in [79] release the MapReduce trace which is produced from the real users of Facebook for one day (24 hours) from a cluster of 600 machines. We count the number of different types of job submissions over a time slot of 30 minutes for the first 8 hours and use that as a dynamic workload (fig. 5.12 (a)) for the experiment. The MapReduce trace for two workloads is available (WorkloadA and WorkloadB) and we used the trace of the workloadB.

Fig. 5.12 (b) shows that the latency of our proposed strategy is consistently lower than all the other strategies throughout the 8 hours. Moreover, as the number of workload increases dynamically our proposed strategy has the lowest increase in the latency compared to all the other strategies. For example, as the number of workload generators increased from 2 to 75, the percentage increase in latency of our proposed strategy is 76%, whereas the increase is 234% for [5], 272% for C3, 504% for [6] and 503% for dynamic snitch. Our proposed strategy performed better at dynamic workload situations because of the same reason as it performed better at higher workload situations.

Evaluation of the impact of different factors in latency

To evaluate the impact of different factors in latency we conduct an experiment by using our proposed strategy and ignoring a single factor at a time and the experimental result is shown in fig. 5.13. 5.13(a) depicts that our original strategy that considered the RTT, response time, and IOPS as the scoring factor has the lowest mean, 90th percentile and 99th percentile latency. The strategy that ignores the RTT has the highest latency in all the scenarios. That is because, in geo-distributed systems, the RTT has the greatest impact on the latency of the system as discussed in section 3. As the IOPS has also a direct impact on the latency of the geo-distributed systems, hence the strategy that ignores IOPS has a higher mean, 90th percentile, and 99th percentile latency than the strategy that considered all the factors. As we discussed in section 3.2

that in our experimental environment response time has very little significance to overall latency as the queue size has very low values hence to illustrate the impact of response time we conduct an experiment in a different testbed with low system configuration as mentioned in section 3.2. The experimental result is shown in 5.13(b). In this experimental environment, the response time has a significant impact on the overall latency of the system hence the strategy that ignores response time has a higher latency in all the scenarios compared to our proposed strategy which considered response time.

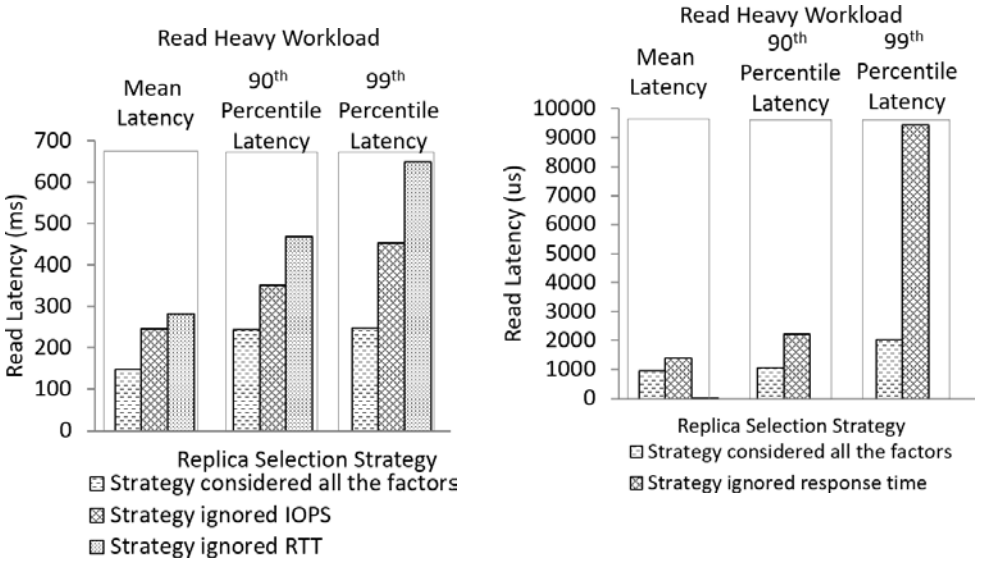


Figure 5.13: Impact of different factors in latency (a) Impact of RTT and IOPS in latency. (b) Impact of response time in latency.

Scalability of the Proposed Strategy

To evaluate the scalability of our proposed strategy we conducted an experiment with different numbers of geo-distributed nodes in the data center. Initially, we started with 5 nodes distributed over the 5 geographical regions and then gradually increases the number of nodes to 10, 15, 20, 25, and 30. We conducted the experiment with read-heavy workload and we used 90 workload generators with 10 million read operations. The experimental result is shown in fig. 5.14 from which we can see that as the number of geo-distributed nodes increases the mean, 90th percentile, and 99th percentile latency decreases. This is because as the number of nodes increases, the loads get distributed over the nodes, and as a result, they respond faster. Our proposed strategy also plays a role here by selecting the fastest node among the faster nodes. Our strategy is not only scalable in terms of latency but also increases the overall throughput as the number of geo-distributed nodes increases.

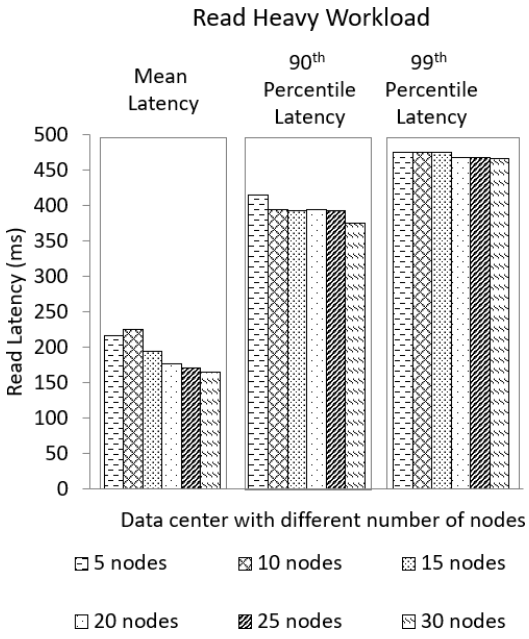


Figure 5.14: Latency at the different numbers of nodes in the data center.

Distribution of Tasks over the Latency Range

To evaluate how replica selection strategies affect the latency of individual tasks and help to reduce the latency and tail latency we conduct an experiment with read-heavy workload and collect individual latency of about 3 thousand different tasks. Fig. 5.15 shows the distribution of 3 thousand tasks over the latency range for different replica selection strategies. Among 3 thousand tasks, the highest number of tasks fall within the latency range 76-100 ms for our proposed strategy, 101-125 ms for [5] and C3, 201-225 ms for the dynamic snitch and [6]. Such a distribution explains why our proposed strategy has the lowest mean latency. Moreover, the highest recorded latency is within the range 626-650 ms, 901-925 ms, 901-925 ms, 1001-1025 ms, and 1326-1350 ms for our proposed strategy, [5], C3, dynamic snitch, and [6] respectively. This observation explains why our proposed strategy has the lowest 90th and 99th percentile latency.

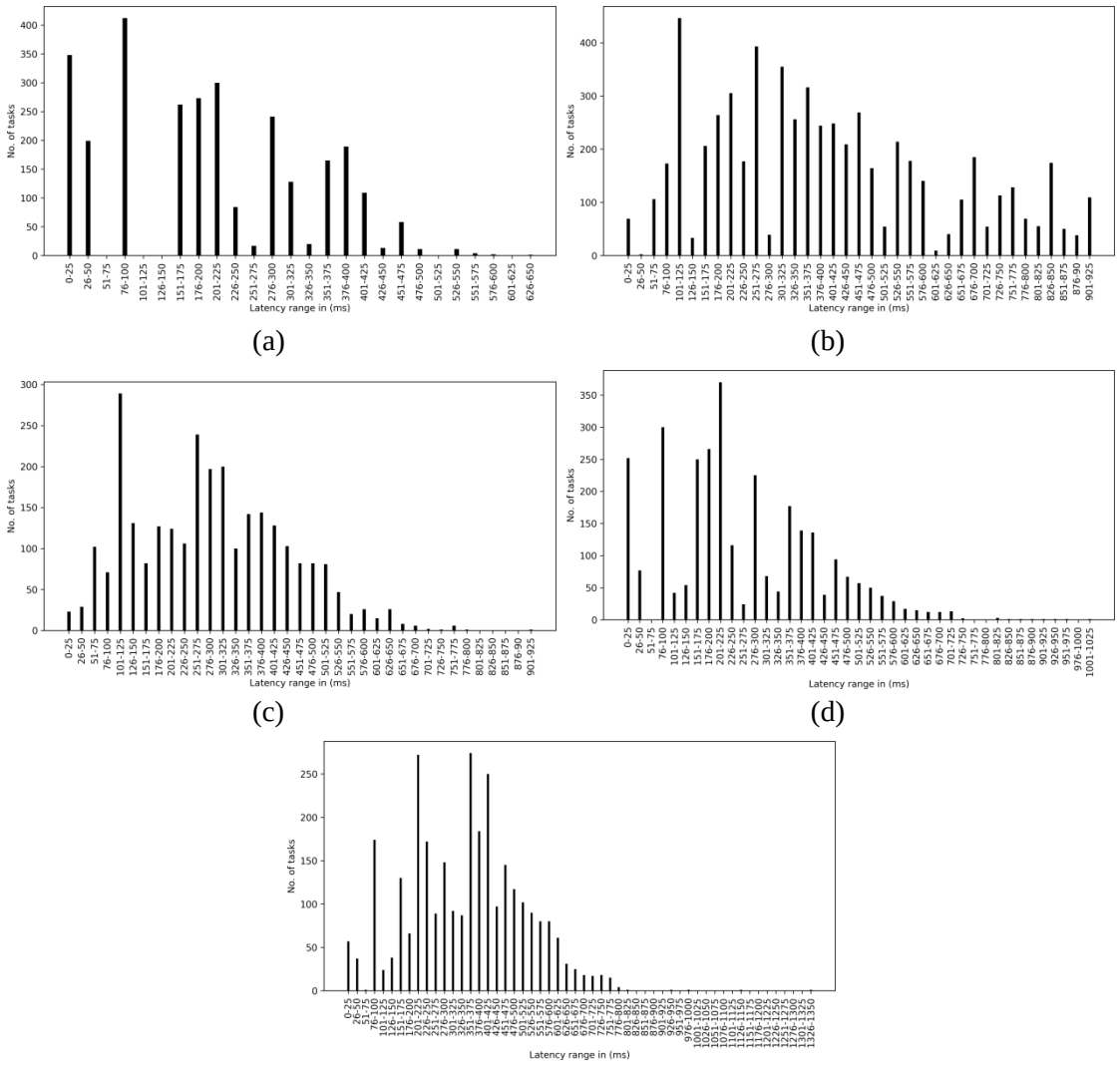
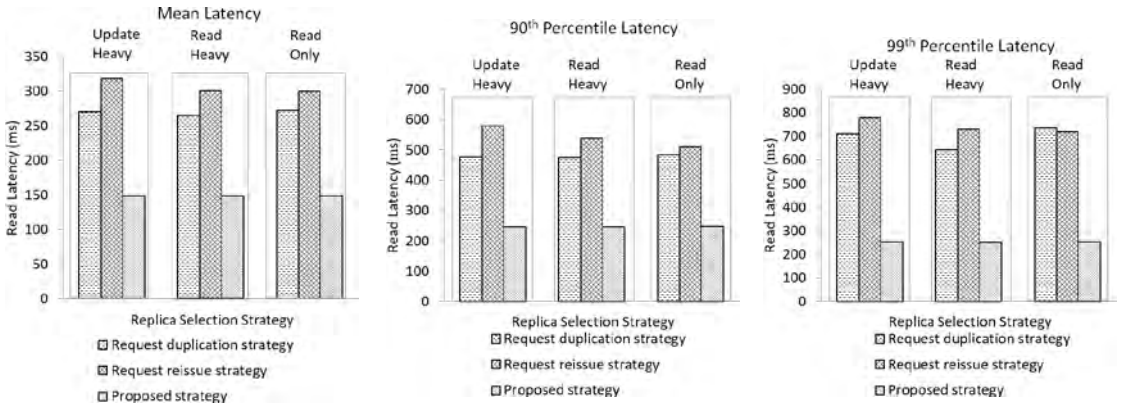


Figure 5.15: Distribution of Tasks over the Latency Range (a) proposed strategy (b) [5] (c) C3 (d) dynamic snitch (e) [6]

Comparison of Proposed Strategy with Request Duplication and Request Reissue Strategy

Fig. 5.16 shows the latency of Cassandra when using request reissue, request duplication, and our proposed strategy in three different workloads. In all workload situations, our proposed strategy has the lowest mean, 90th percentile, and 99th percentile latency. When the system uses request reissue and request duplication techniques, the coordinator frequently generates too many redundant requests in the presence of high latency fluctuation. This in turn increases the load on both the network and the disk, resulting in higher latency. However, when we use our proposed strategy there is less load on both network and disk and hence has the lowest latency in all the scenarios. From fig. 5.16, we can also find that the request reissue strategy has higher latency compared to request duplication strategy in all the scenarios.



(a) (b) (c)

Figure 5.16: Latency of Cassandra when using request reissue, request duplication, and proposed strategy (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

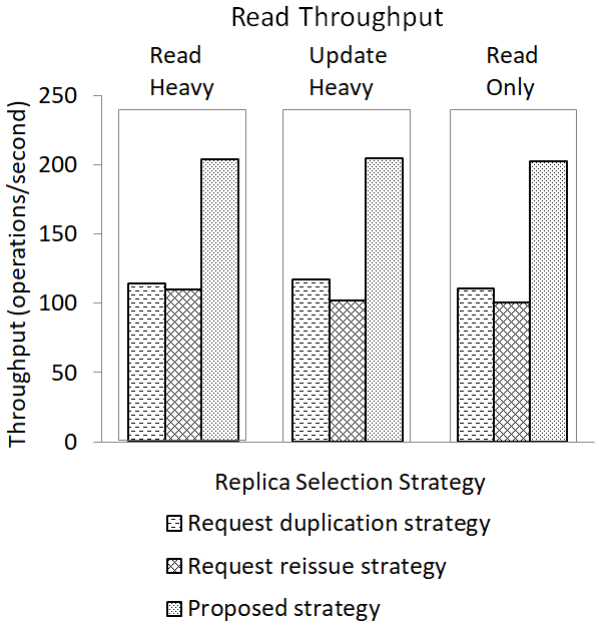
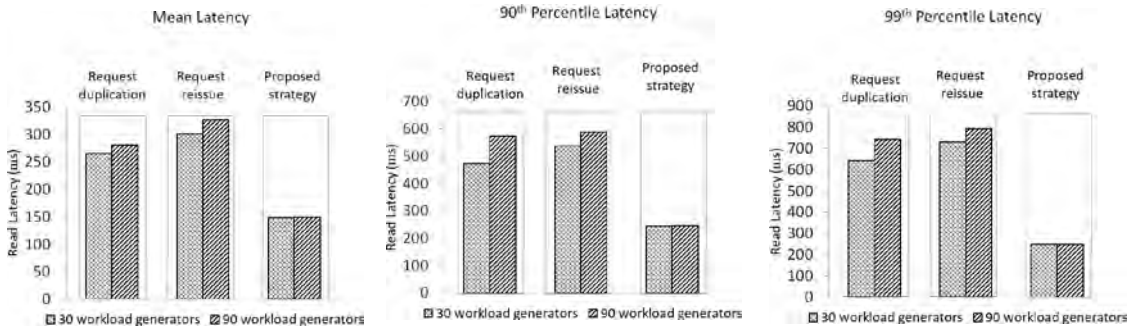


Figure 5.17: Comparison of read throughput of our proposed strategy with request duplication and request reissue strategy at different workloads.

We also compared the throughput of our proposed strategy with request reissue and request duplication strategy as shown in fig 5.17. In all the scenarios, our strategy has the highest throughput compared to the others. The reason for such an improvement is the reduced load on both the network and the disk when using our proposed strategy. The redundant requests that are generated when using request duplication and request reissue strategy increase load on both the network and the disk which in turn reduces the overall throughput of the systems.

To compare the performance of our proposed strategy at a higher workload with request duplication and request reissue strategy we conduct another experiment where we increased the number of workload generators from 30 to 90. We use read heavy workload to conduct this



(a) (b) (c)

Figure 5.18: Latency of Cassandra when using request reissue, request duplication, and proposed strategy at a higher workload (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

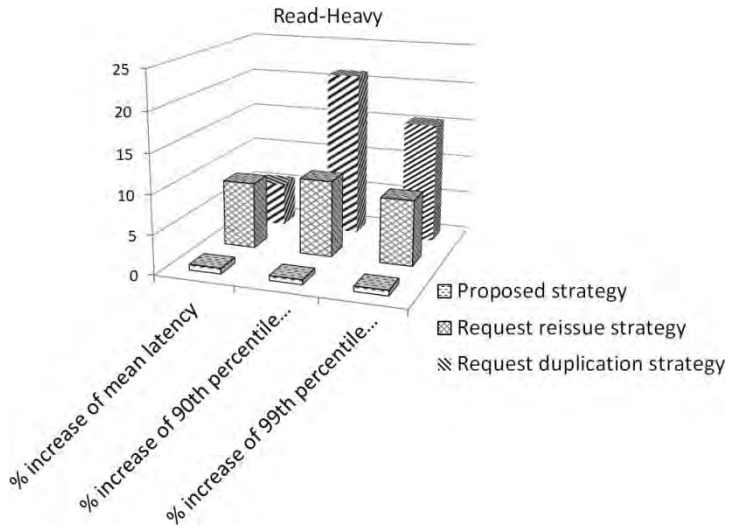


Figure 5.19: Percentile increase of mean, 90^{th} percentile, and 99^{th} percentile latency of request duplication, request reissue, and our proposed strategy at higher workload

experiment and the experimental result is shown in fig 5.18. Still at higher workload the latency of our proposed strategy is lower than both request duplication and request reissue strategy. From fig. 5.19 it is seen, as the number of workload generators increased by 200 %, the 90th percentile latency of our proposed strategy increases by 0.67% whereas the increase is 9.79% for request reissue strategy and 21.22% for request duplication strategy. Similarly, the percentage increase of mean and 99th percentile latency of our proposed strategy is lower than both request reisssue and request duplication strategy. The reason for such an improvement is that as the workload increases the queue size and disk IOPS also increase. However, our proposed strategy keeps into consideration this increase in queue size and disk IOPS and still at a higher workload can select the best replica for routing the read request. On the other hand, in the case of request reissue and request duplication strategy, the extra load of redundant requests increases the load on the network and the disk twice or thrice the original load and thereby increases the overall latency.

To compare the performance in a higher workload, we also keep track of the throughput of request duplication, request reissue, and our proposed strategy. Fig. 5.19 shows as the number of workload generators increased from 30 to 90 the throughput also increased for all the strategies. However, our proposed strategy has the highest increase in throughput compared to the other two strategies. The throughput of our proposed strategy increased by 198.54% whereas the increase is 193.81% for the request duplication strategy and 174.99% for the request reissue strategy.

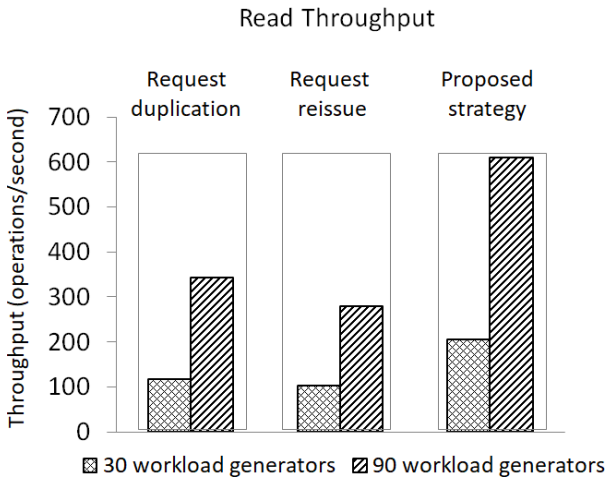
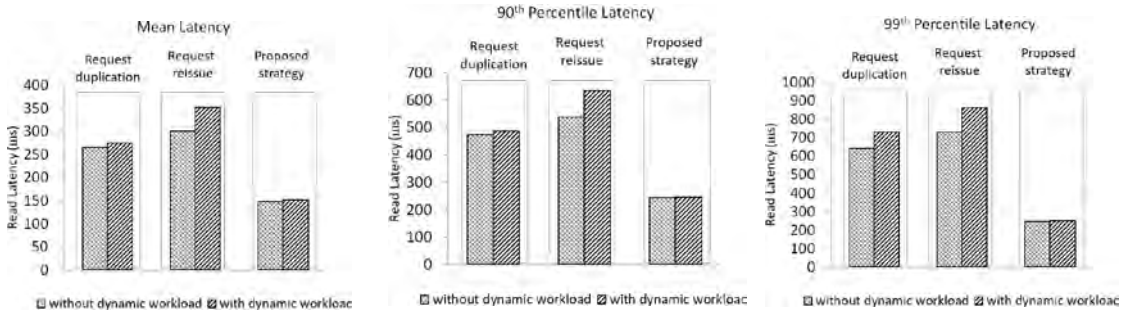


Figure 5.20: Throughput of Cassandra when using request duplication, request reissue, and our proposed strategy at higher workloads.

To compare the performance of our proposed strategy with request duplication and request reissue strategy at dynamic workload we conduct another experiment where we initially start 30 workload generators with the read-heavy workload and after 330 sec we start another 30 workload generators with the update-heavy workload. Also in a dynamic workload situation, our proposed strategy has the lowest latency compared to the request duplication and request reissue strategy as shown in fig. 5.21. Fig 5.22 shows the percentage increase of mean, 90th percentile, and 99th percentile latency of our proposed strategy, request duplication, and request reissue strategy at the dynamic workload. The percentage increase of our proposed strategy is lower than both strategies. After entering into dynamic workload the increase in 99th percentile latency of our proposed strategy is 0.87% whereas the increase is 18.00% for request reissue strategy and 13.63% for request duplication strategy. To compare the performance of our proposed strategy with request duplication and request reissue strategy at a dynamic workload we also keep track of throughput. Fig. 5.23 shows as the system enters into the dynamic workload situation the throughput of our proposed strategy increases whereas the throughput of the other two strategies decreases.



(a) (b) (c)

Figure 5.21: Latency of Cassandra when using request reissue, request duplication, and proposed strategy at dynamic workload (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

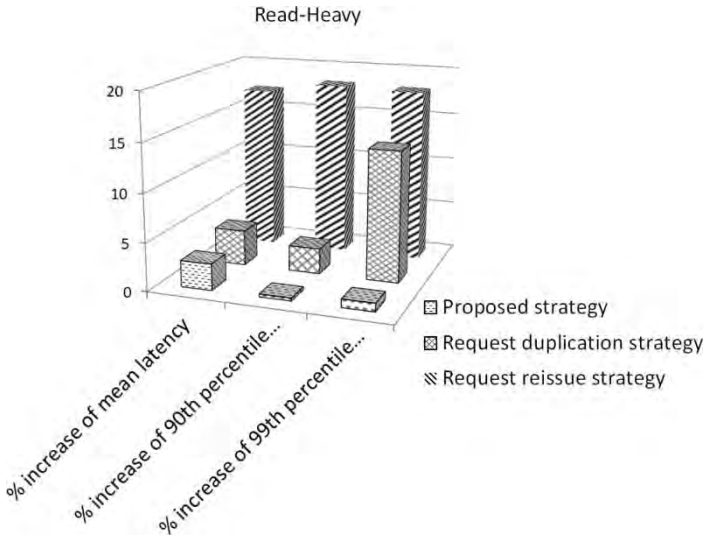


Figure 5.22: Percentile increase of mean, 90^{th} percentile, and 99^{th} percentile latency of request duplication, request reissue, and our proposed strategy at dynamic workload

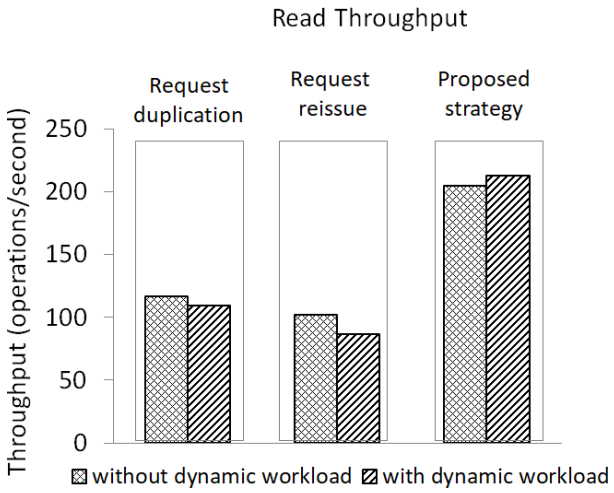


Figure 5.23: Throughput of Cassandra when using request duplication, request reissue, and our proposed strategy at dynamic workload

5.4 Summary

In this chapter, we proposed a prediction-based replica selection mechanism for reducing tail latency in the geo-distributed systems. For predicting the latency we derive an equation of multiple linear regression and finally calculated a score for each of the nodes of the cluster based on which the best replica is chosen for routing the read request. We considered almost all the important performance metrics for calculating the score. We also solved the problem of herd behavior. For evaluating the performance of our proposed strategy we conducted some extensive experiments and our experimental results show that our proposed strategy decreases the overall latency as well as increases the overall throughput of the system.

Chapter 6

Performance Improvement of Redundant Request by Designing a Top of an Efficient Replica Selection Strategy

The request duplication and request reissue techniques send the redundant read request to a replica for reducing tail latency in the distributed systems. Though request duplication and request reissue techniques can reduce the tail latency however if multiple reissued or duplicated requests select a replica that is already suffering from a long-tail latency problem then the situation becomes scarier. According to the experimental results discussed in section 5.3.2 of chapter 5, the performance of request duplication and request reissue strategy in terms of both latency and throughput is quite lower than our proposed strategy. According to the discussion of [25]the performance of both strategies can be improved by designing it a top of an efficient replica selection strategy. In this chapter, we implemented the request reissue and request duplication strategy of Cassandra a top of our proposed replica selection strategy as discussed in chapter 5. For evaluating the effectiveness of our designed system we deployed a 15 nodes Cassandra cluster in Amazon EC2 that spread over 5 geographical regions. For generating test datasets and workloads we used industry-standard Yahoo Cloud Serving Benchmark (YCSB). We use the same experimental setup as discussed in section 5.3.1 of chapter 5. Experimental results show that the use of our proposed replica selection strategy improves the performance of both request duplication and request reissue strategy and enables them to be more efficient in reducing tail latency in the distributed systems.

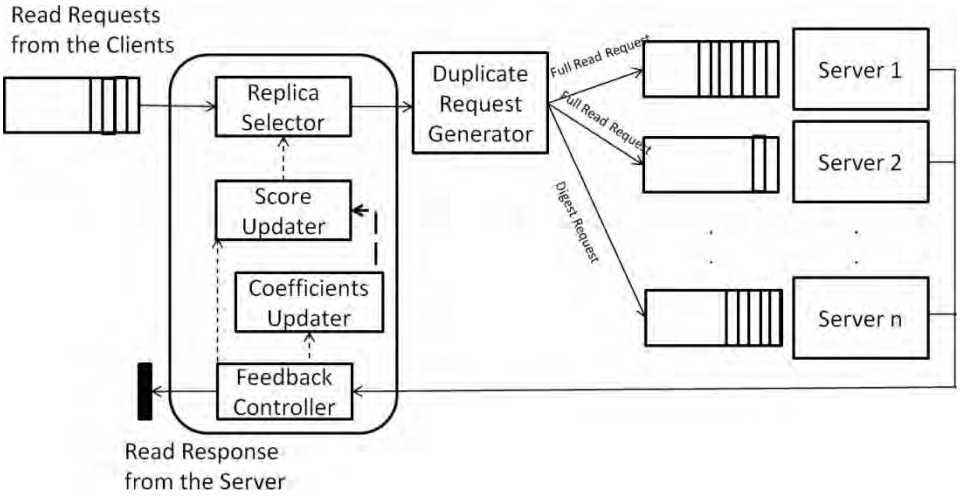


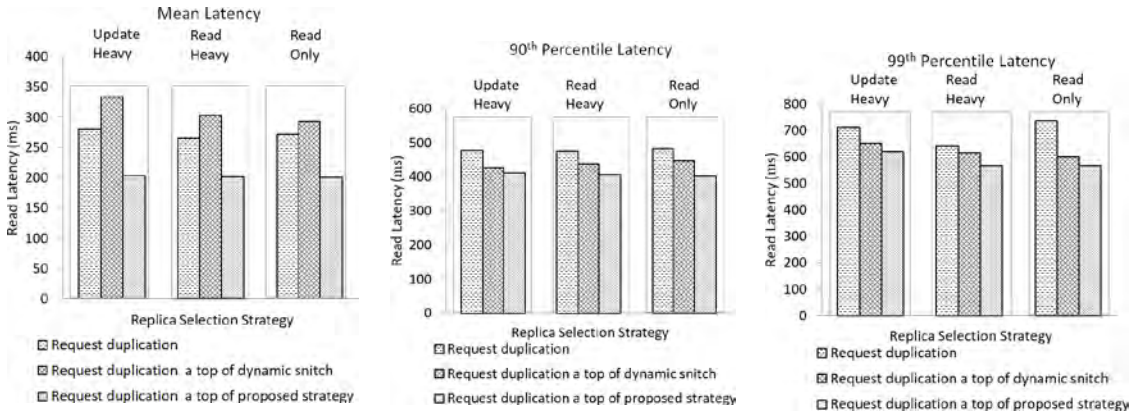
Figure 6.1: Request duplication strategy a top of our proposed replica selection strategy.

Performance Improvement of Request Duplication Strategy

To improve the performance of request duplication strategy we implemented it a top of our proposed strategy as shown in fig. 6.1. The replica selector first selects a subset of the replicas based on their score and feed this subset to the duplicate request generator. This duplicate request generator sends two full read requests to two replicas and additional digest requests based the consistency level. Cassandra has the implementation of both request duplication and request reissue technique and we can use these techniques by configuring the *speculative_ retry* property that allows the coordinator to send redundant read requests to another replica after a configurable timeout value [80]. To enable request duplication strategy we set the *speculative_ retry* property to *ALWAY S*. We also implement request duplication strategy a top of dynamic snitch to determine how efficiently our proposed replica selection strategy can improve the performance of replica duplication strategy compared to dynamic snitch.

Comparison of Latency at Different Workload

We conduct this experiment to determine how dynamic snitch and our proposed strategy improve the performance of the request duplication strategy and thereby reduce the latency. Fig. 6.2 shows that the implementation of the request duplication strategy a top of our proposed strategy and dynamic snitch reduces tail latency in almost all the scenarios. The only exception is seen at mean latency, whereas the request duplication strategy shows lower latency than its implementation a top of the dynamic snitch. The request duplication strategy shows the lowest mean, 90th percentile, and 99th percentile latency at all workloads when implemented a top of our proposed strategy. With read heavy workload the 99th percentile latency of the Cassandra is 733.56*ms* when using only request duplication strategy, 600.16*ms* when using request duplication a top



(a) (b) (c)

Figure 6.2: Latency of Cassandra when using request duplication, request duplication a top of dynamic snitch, and request duplication a top of our proposed strategy (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

of dynamic snitch, and only $566.35ms$ when using request duplication a top of our proposed strategy. The reason for such an improvement is that when we implement the request duplication strategy a top of a replica selection strategy, it sends the duplicate request to a sub-optimal subset of the replicas. According to our experimental results as discussed in chapter 5, as our proposed strategy selects better replicas compared to dynamic snitch hence the implementation of the request duplication strategy a top of our strategy shows the lowest latency.

Comparison of Throughput at Different Workload

Fig.6.3 depicts the comparison of throughput of the Cassandra when using only request duplication strategy, request duplication a top of dynamic snitch, and request duplication a top of our proposed strategy. By reducing latency across the replicas, the implementation of request duplication strategy a top of our strategy makes better use of the system capacity and thereby increases the system throughput. In all the scenarios, Cassandra has higher throughput when using request duplication strategy a top of our proposed replica selection strategy. With read-only workload, our strategy has 34.81% more throughput than request duplication strategy and 51.70% more throughput than request duplication designed a top of dynamic snitch strategy. The experimental result is quite similar for read-heavy and update-heavy workload. We can also observe from fig. 6.3 that instead of increasing the throughput the implementation of request duplication strategy a top of dynamic snitch reduces the throughput of Cassandra. From this observation, we can conclude that designing a request duplication strategy a top of a replica selection strategy does not necessarily improve the performance of the system unless the replica selection strategy is efficient enough.

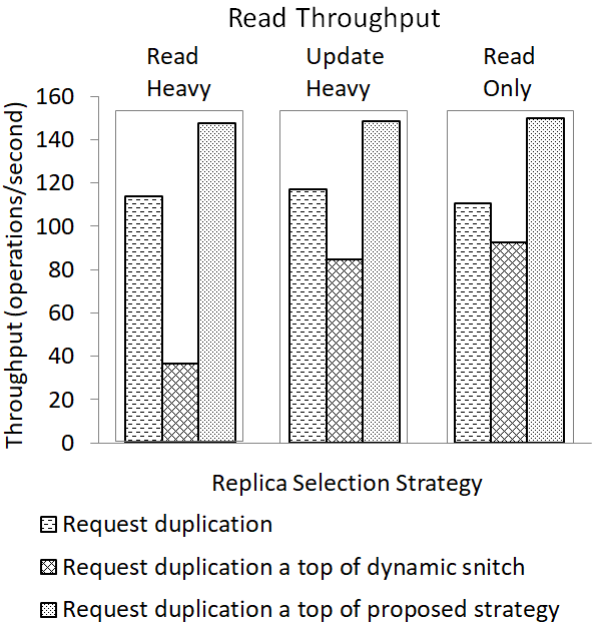
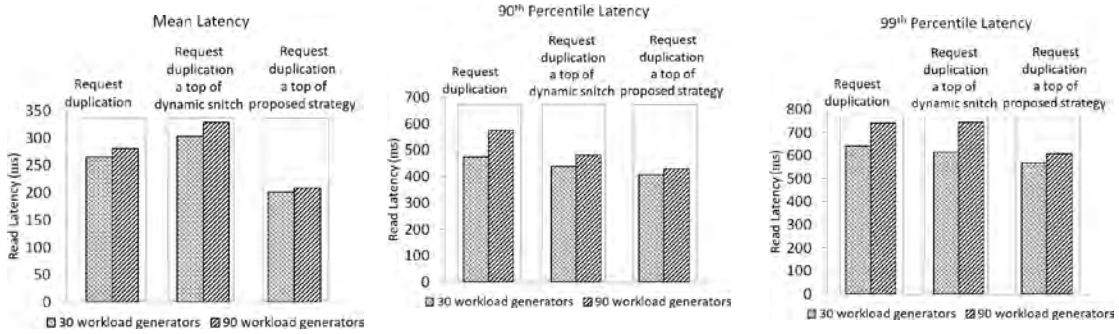


Figure 6.3: Comparison of read throughput at different workloads. Implementation of request duplication strategy a top of our proposed strategy shows higher throughput at all the scenarios.

Performance at Higher Workload

We conduct this experiment to evaluate the performance of the request duplication strategy, request duplication a top of dynamic snitch, and request duplication a top of our proposed strategy at a higher workload. To do so we increase the number of workload generators from 30 to 90 and we only conduct this experiment with read-heavy workload. Fig. 6.4 shows how the latency of the strategies affects as the workload increases. Still at a higher workload situation the implementation of the request duplication strategy a top of our proposed strategy has the lowest mean, 90th percentile, and 99th percentile latency. Moreover, as the workload increases the percentage increase in latency of the request duplication designed a top of our proposed strategy is lower compared to the only request duplication strategy and request duplication a top of dynamic snitch as shown in fig. 6.5. As the workload increases by 200% the percentage increase in 99^th percentile latency for request duplication strategy a top of our proposed strategy is only 7.17% whereas the increase is 15.47% for only request duplication and 20.91% for request duplication a top of dynamic snitch.

To evaluate the performance improvement of the request duplication strategy when designed a top of our proposed strategy we also keep track of the throughput during this experiment. From fig. 6.6 we can see also at higher workload situations the throughput of the request duplication technique is the highest when implemented a top of our proposed strategy. As the number of workload generators increases from 30 to 90, the request duplication a top of our proposed strategy shows the highest throughput which is about 431.62*ops/sec* whereas the throughput of only request duplication is 343.12*ops/sec*, and request duplication a top of the dynamic snitch is



(a) (b) (c)

Figure 6.4: Latency of Cassandra when using request duplication, request duplication a top of dynamic snitch, and request duplication a top of our proposed strategy at a higher workload (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

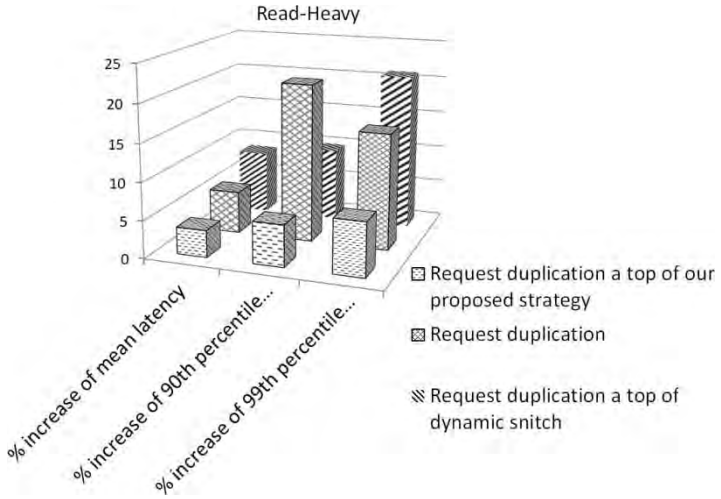


Figure 6.5: Percentage increase of mean, 90^{th} percentile, and 99^{th} percentile latency of the request duplication strategy, request duplication a top of dynamic snitch, and request duplication a top of our proposed strategy at higher workloads

233.55*ops/sec*.

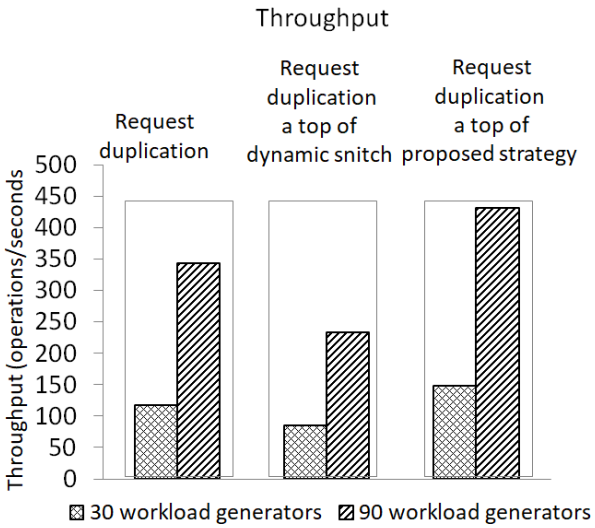
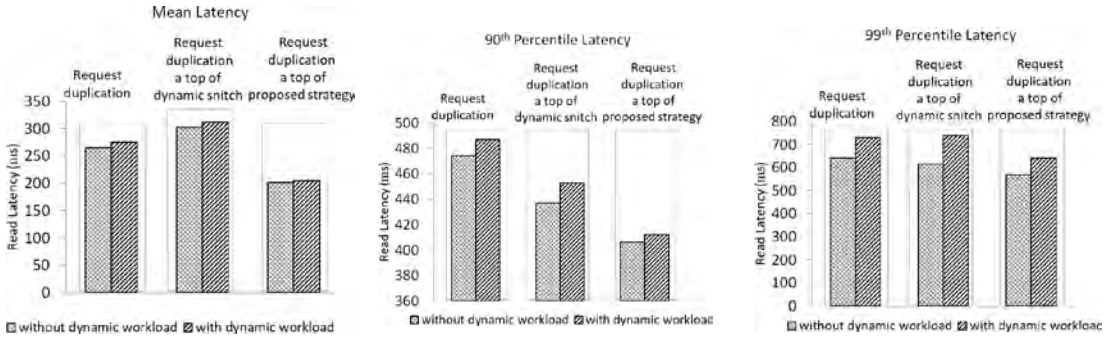


Figure 6.6: Throughput of Cassandra when using request duplication strategy, request duplication a top of dynamic snitch, and request duplication a top of our proposed strategy at higher workloads

Performance at Dynamic Workload

We conduct this experiment to evaluate how our proposed strategy improves the performance of the request duplication strategy at dynamic workload situations. For this purpose, we implement Cassandra's default request duplication strategy a top of our proposed strategy. For comparison, we also implement request duplication strategy a top of the dynamic snitch. To compare the performance of the strategies in a dynamic workload situation, we initially start 30 workload generators with read-heavy workload and after 330 sec we start another 30 workload generators with update-heavy workload. Fig. 6.7 depicts that the performance of the request duplication strategy, request duplication a top of the dynamic snitch, and request duplication a top of our proposed strategy is quite identical to the performance in higher workload situations. Fig. 6.8 shows the percentage increase in latency of all the strategies as the system enters into the dynamic workload situation. Still, in a dynamic workload situation, the request duplication implemented a top of our proposed strategy shows the lowest increase in mean, 90^th percentile, and 99^th percentile latency compared to the only request duplication, and request duplication a top of the dynamic snitch. The increase in 99^th latency of the request duplication a top of our proposed strategy is 12.77% whereas the increase is 13.63% for only request duplication strategy, and 20.29% for request duplication a top of the dynamic snitch. We also compare the throughput of all the strategies in dynamic workload situation as shown in fig. 6.9. Unlike the higher workload situation where the throughput of all the strategies increases as the workload increases, in the dynamic workload situation the throughput of all the strategies decreases. However, the percentage decrease of the request duplication a top of our proposed strategy is 4.01% whereas the decrease is 6.24% for only request duplication strategy, and 7.39% for request duplication a

top of the dynamic snitch.



(a) (b) (c)

Figure 6.7: Latency of Cassandra when using request duplication, request duplication a top of dynamic snitch, and request duplication a top of our proposed strategy at a dynamic workload (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

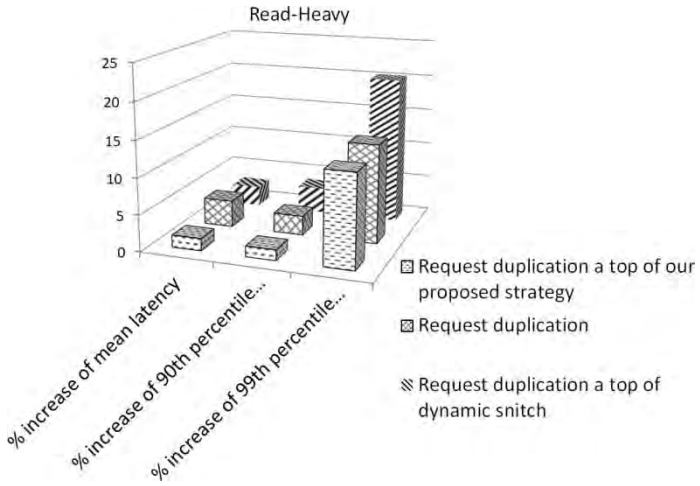


Figure 6.8: Percentage increase of mean, 90^{th} percentile, and 99^{th} percentile latency of the request duplication strategy, request duplication a top of dynamic snitch, and request duplication a top of our proposed strategy at dynamic workloads

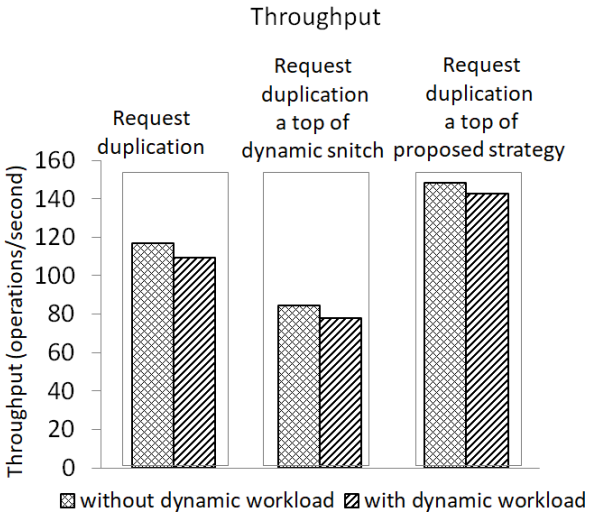


Figure 6.9: Throughput of Cassandra when using request duplication strategy, request duplication a top of dynamic snitch, and request duplication a top of our proposed strategy at dynamic workloads

Performance Improvement of Request Reissue Strategy

To improve the performance of the request reissue strategy we implement it a top of our proposed replica selection strategy as shown in fig. 6.10.The replica selector selects a sub-optimal subset of the replicas based on their score and sends this subset to the reissue request generator. Then the reissue request generator sends one full read request and one or more digest requests based on the consistency level. If the replica to which a full request has been sent does not respond within a specific/ Xpercentile time interval then the coordinator sends another read request to another replica.To evaluate the performance of request reissue we configure the *speculative_retry* to 99*percentile*. It is the default setting of Cassandra and provides a good balance between not performing a lot of extra requests while still dealing with the worst problems [81]. We also implement the request reissue technique a top of the dynamic snitch to determine how efficient our proposed strategy is compared to the dynamic snitch to improve the performance of the request reissue strategy.

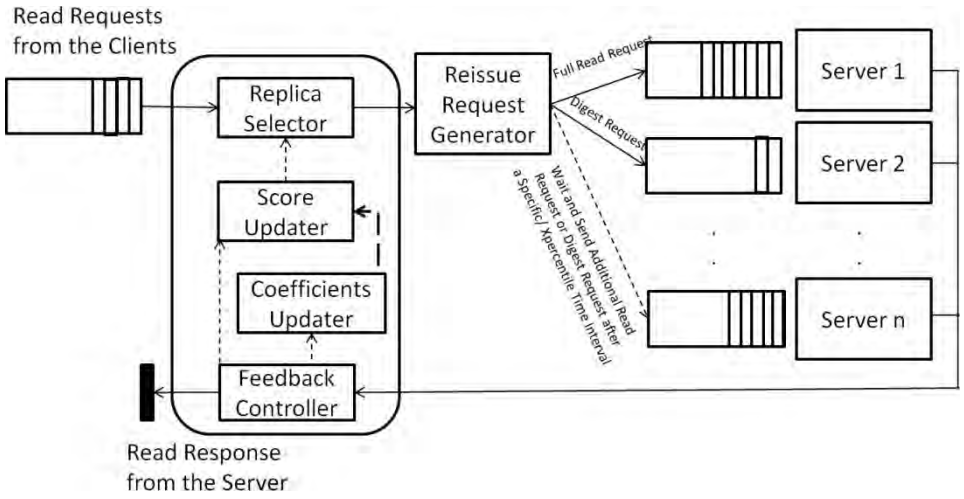


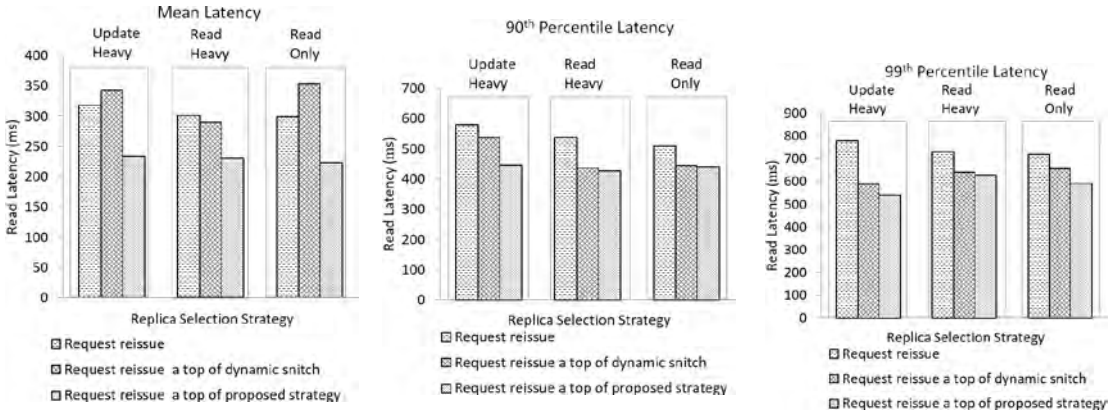
Figure 6.10: Request reissue strategy a top of our proposed replica selection strategy.

Comparison of Latency at Different Workload

We conduct this experiment to determine how effective our proposed replica selection strategy and the dynamic snitch are in improving the performance of the request reissue strategy by reducing tail latency. The experimental result of fig. 6.11 shows as we implement the request reissue strategy a top of our proposed replica selection strategy and dynamic snitch the latency reduces in almost all the scenarios except the mean latency in update-heavy and read-only workloads whereas the latency of only request reissue strategy is lower than it's implementation a top of the dynamic snitch. However, Cassandra shows the lowest latency in all the scenarios when the request reissue strategy has implemented a top of our proposed replica selection strategy. In an update-heavy workload, the 99^th percentile latency of Cassandra when implemented a top of our proposed strategy is $590.53\mu s$ whereas the latency is $657.4\mu s$ for request reissue a top of the dynamic snitch, and $717.51\mu s$ for only request reissue strategy. When we use only request reissue strategy Cassandra performs worst as this strategy sends the redundant requests to the random replicas which may already suffer from long-tail latency problem. Between our proposed replica selection strategy and dynamic snitch our proposed strategy is more effective to improve the performance of the request reissue strategy as it can select the replicas more efficiently than the dynamic snitch.

Comparison of Throughput at Different Workload

Fig. 6.12 shows the comparison of throughput of Cassandra when it uses the request reissue strategy, request reissue a top of the dynamic snitch, and request reissue a top of our proposed strategy. In all the scenarios the implementation of request reissue a top of our proposed replica selection strategy has the highest throughput. The reason for such an improvement is the reduction in latency as discussed in section 6.2. By reducing latency the implementation of



(a) (b) (c)

Figure 6.11: Latency of Cassandra when using request reissue, request reissue a top of dynamic snitch, and request reissue a top of our proposed strategy (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

request reissue a top of our proposed replica selection strategy enables Cassandra to better use the system capacity and thereby increase the overall throughput. Though our proposed strategy increases the throughput of the system, the dynamic snitch shows a very different outcome. The throughput of Cassandra decreases in all the workload situations. Moreover, the decrease is quite sharp in the read-heavy workload. We also observed the same outcome in the case of the request duplication strategy. We are not confirmed what is the main reason behind this and may require some extensive experiments to identify the reason for such an outcome.

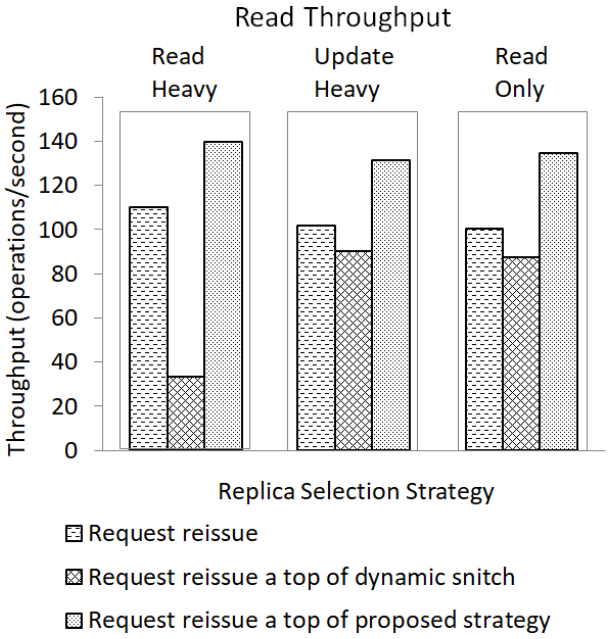
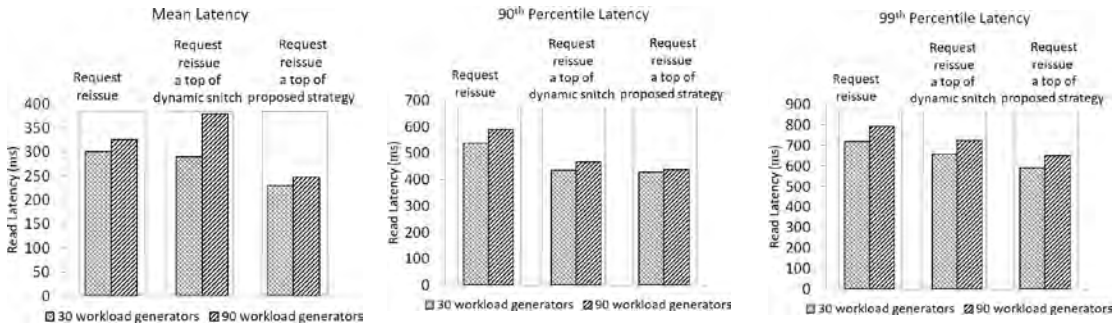


Figure 6.12: Comparison of read throughput at different workloads. Implementation of request reissue strategy a top of our proposed strategy shows higher throughput at all the scenarios.

Performance at Higher Workload

We conduct this experiment to identify how our proposed replica selection strategy improves the performance of the request reissue strategy. To determine the latency characteristics of Cassandra when it uses only the request reissue strategy, request reissue a top of the dynamic snitch, and request reissue a top of our proposed replica selection strategy we increase the number of workload generators from 30 to 90 and we conduct this experiment only with the read-heavy workload. Fig. 6.13 depicts how the latency of all the strategies increases as the workload increases. In all the scenarios, the request reissue strategy has the lowest increase in latency when implemented a top of our proposed replica selection strategy. Fig. 6.14 shows the percentage increase in mean, 90^th percentile, and 99^th percentile latency of the request reissue strategy, request reissue a top of our proposed replica selection strategy, and request reissue a top of the dynamic snitch. Also at higher workload situations, the implementation of the request reissue strategy a top of our proposed replica selection strategy has the lowest percentage increase in latency in all the scenarios. From the figure we can also see that though the percentage increase in latency for all the strategies is quite close, the implementation of request reissue a top of dynamic snitch shows an exceptionally sharp increase in mean latency. The percentage increase in mean latency is 7.5% when the system used request reissue a top of our proposed replica selection strategy, 8.51% when the system used only request reissue strategy, and 30.97% when the system used request reissue a top of dynamic snitch.



(a) (b) (c)

Figure 6.13: Latency of Cassandra when using request reissue, request reissue a top of dynamic snitch, and request reissue a top of our proposed strategy at a higher workload (a) mean latency (b) 90^{th} percentile latency (c) 99^{th} percentile latency.

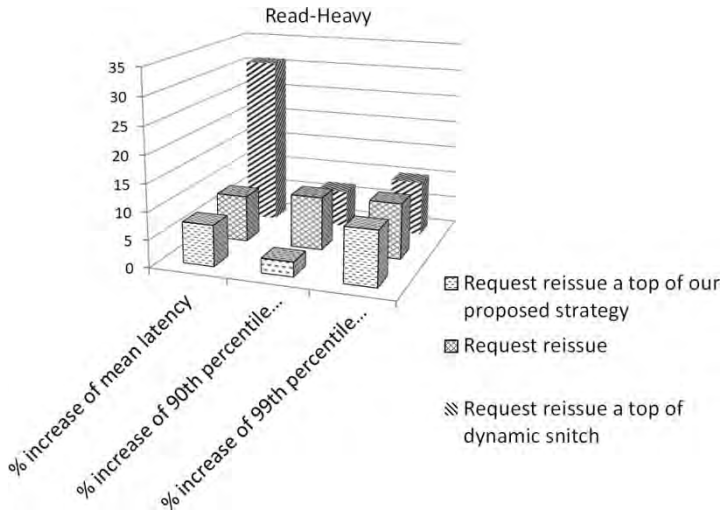


Figure 6.14: Percentage increase of mean, 90^{th} percentile, and 99^{th} percentile latency of the request reissue strategy, request reissue a top of dynamic snitch, and request reissue a top of our proposed strategy at higher workloads

To determine how effective our proposed replica selection strategy is to increase the throughput of the request reissue strategy in higher workload situations we also keep track of the throughput and the experimental result is shown in fig. 6.15. Still, in a higher workload situation the implementation of the request reissue a top of our proposed replica selection strategy has the highest throughput compared to only the request reissue strategy, and request reissue a top of the dynamic snitch. Moreover, as the workload increases by 200%, the percentage increase in throughput of request reissue a top of our proposed strategy is 179.65%, 176.56% for only request reissue, and 139.73% for request reissue a top of the dynamic snitch.

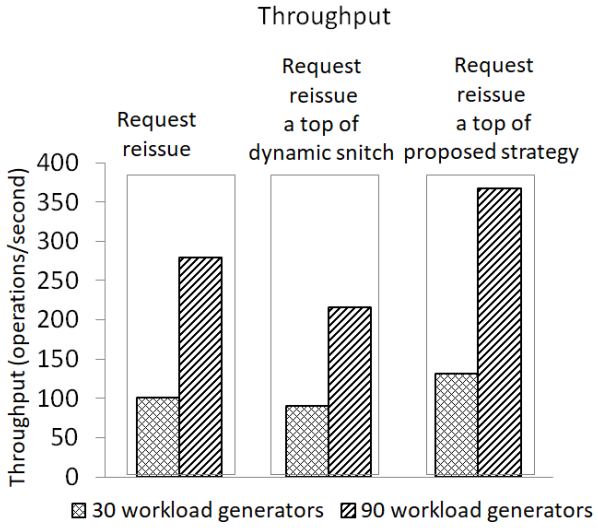
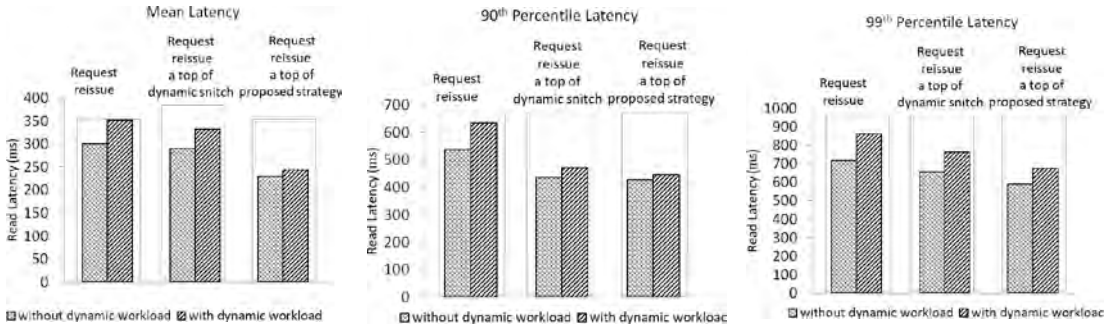


Figure 6.15: Throughput of Cassandra when using request reissue strategy, request reissue a top of dynamic snitch, and request reissue a top of our proposed strategy at higher workloads.

Performance at Dynamic Workload

We conduct this experiment to determine how our designed replica selection strategy improves the performance of the request reissue strategy. Fig. 6.16 shows the comparison of latency at dynamic workload situation of request reissue, request reissue a top of our proposed strategy, and request reissue a top of the dynamic snitch. To conduct this experiment, we initially start 30 workload generators with a read-heavy workload and after 330 sec we start another 30 workload generators with an update-heavy workload. From fig. 6.16 we can see, that as the system enters into the dynamic workload situation the latency increases. However, the increase in latency is the lowest when request reissue is implemented a top of our proposed strategy. Fig. 6.17 shows the percentage increase in mean, 90^th, and 99^th percentile latency of different strategies as the system enters into a dynamic workload situation. In all the scenarios, the implementation of request reissue a top of our proposed replica selection strategy has the lowest percentage increase in latency. The percentage increase in 99^th percentile latency is 14.06% when request reissue is implemented a top of our proposed strategy whereas the increase is 15.84% for the implementation of request reissue a top of the dynamic snitch and 19.99% for only the request reissue strategy.



(a) (b) (c)

Figure 6.16: Latency of Cassandra when using request reissue, request reissue a top of dynamic snitch, and request reissue a top of our proposed strategy at a dynamic workload (a) mean latency (b) 90^th percentile latency (c) 99^th percentile latency.

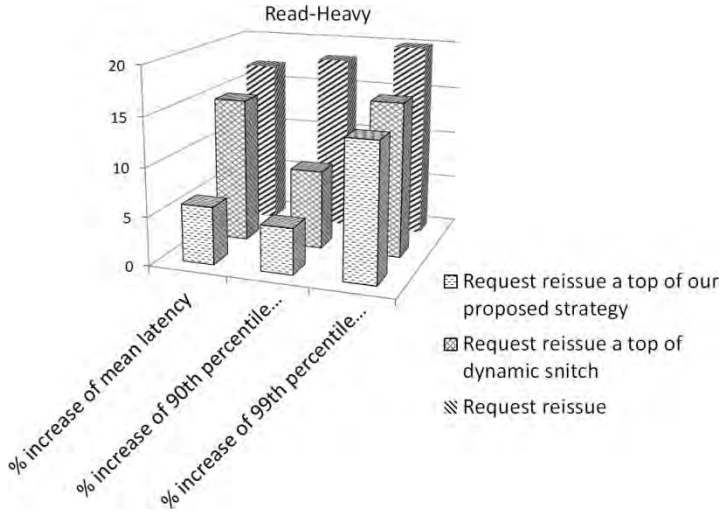


Figure 6.17: Percentage increase of mean, 90^{th} percentile, and 99^{th} percentile latency of the request reissue strategy, request reissue a top of dynamic snitch, and request reissue a top of our proposed strategy at dynamic workloads

To analyze how our proposed replica selection strategy affects the request reissue strategy in a dynamic workload situation we also keep track of throughput and the experimental result is shown in fig. 6.18. The result is quite identical to the request duplication strategy. As the system enters into the dynamic workload situation the throughput of all the strategies decreases. However, the amount of decrease is the lowest when the request reissue strategy is implemented a top of our proposed strategy. Though the throughput decreases still at a dynamic workload situation the request reissue strategy has the highest throughput when implemented a top of our proposed strategy.

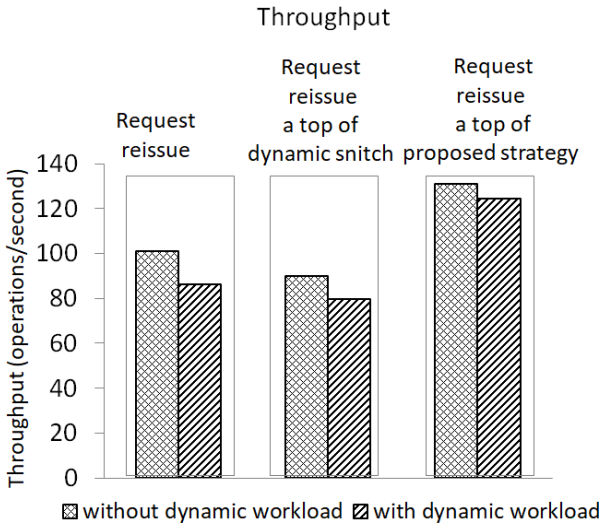


Figure 6.18: Throughput of Cassandra when using request reissue strategy, request reissue a top of dynamic snitch, and request reissue a top of our proposed strategy at dynamic workloads

Summary

The request duplication and request reissue strategy are effective ways to reduce tail latency in the geo-distributed systems. However, these strategies do not always guarantee to reduce tail latency and in some cases may increase the latency instead. For example, if multiple reissued or duplicated requests select a replica that is already suffering from a long-tail latency problem then the situation becomes scarier. Therefore, instead of randomly routing the reissued or duplicated requests if we route these requests in a more intelligent way then the performance of these strategies can be improved. According to the discussion of [25], the performance of both request duplication and request reissue strategy can be improved by designing a top of an efficient replica selection strategy. Our experimental results also support this claim. Though our proposed replica selection strategy improves the performance of both the strategies in all the scenarios, the effect of dynamic snitch on both the strategies is quite unstable. In some cases, it increased the performance of both request duplication and request reissue strategy while in other cases it decreased.

Chapter 7

Discussion

In this chapter, we discuss some of our observations that we found throughout this research work.

Observation1

During one of our experiments, we found that some of the nodes were going down due to space unavailability. In such a scenario, when we were trying to load data to the cluster, the system was trying, again and again, to load data to the nodes that were already down. Though there were other available normal nodes, the system was hitting, again and again, the down nodes, and finally was showing the error that unable to load the data to the cluster. Fig. 7.1 shows the Cassandra cluster status of such a scenario and we used the *nodetool status* command to display the status. From the figure we can see, that the node with the IP address 192.168.34.27 and 192.168.34.29 are down due to space unavailability. However, when we were trying to load data, the system was, again and again, trying to insert data to these two nodes. As these nodes did not have enough space, the operating system of these nodes showed the error in fig. 7.2 again and again. After trying several times the YCSB workload generators stopped loading showing the error message. As we can see from fig. 7.1, 11 other nodes were up at that time and also had enough space. However, Cassandra had never chosen those nodes to load the data.

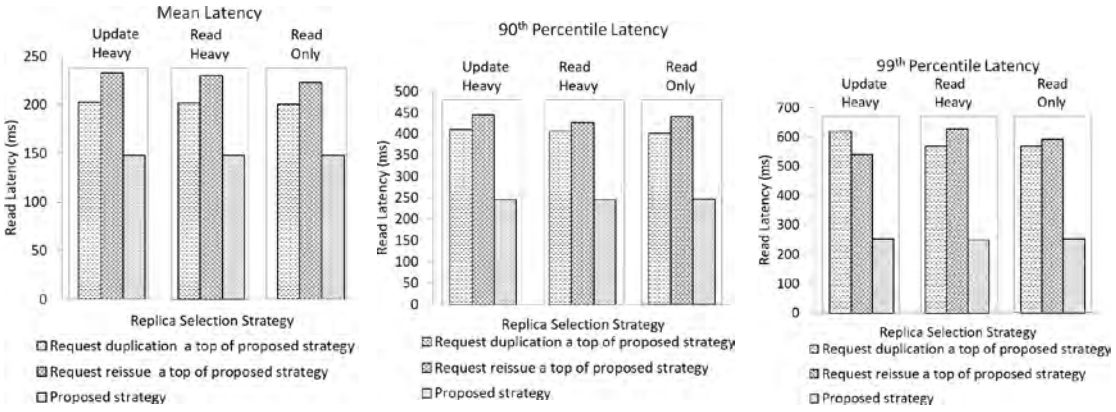
Observation2

Though our proposed replica selection strategy improves the performance of the request reissue and request duplication strategy, still their performance is lower than our proposed strategy. Fig.7.3 depicts that in all the scenarios the mean, 90^th, and 99^th percentile latency of our proposed replica selection strategy is lower than both the request duplication a top of our proposed strategy, and request reissue a top of our proposed strategy. Our proposed strategy not

```
Datacenter: dc1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens       Owns (effective)  Host ID                               Rack
UN  192.168.34.8  5.55 GiB   256          20.6%             5ff76f63-9a10-433d-9c08-d9c962b4333e  rack1
UN  192.168.34.11 6.02 GiB   256          21.6%             630d5037-f6df-4260-b6ab-396cd1662537  rack4
UN  192.168.34.12 3.82 GiB   256          21.2%             57881080-a276-4de6-9833-179b9f82c3c4  rack4
UN  192.168.34.34 4.64 GiB   256          21.8%             717f8282-797a-48d0-83a4-cb21ffba9b64  rack3
UN  192.168.34.5  7.9 GiB    256          21.3%             e1c98e6a-e714-44a3-b7cb-4eb70845bf94  rack3
UN  192.168.34.6  10.26 GiB  256          21.5%             23539906-4061-48a1-a956-c42c4d4eb41a  rack4
UN  192.168.34.25 2.05 GiB   256          20.1%             5a363d78-2d68-424b-b0d5-7000822442fc  rack2
UN  192.168.34.26 5.79 GiB   256          22.1%             2372f060-fc16-42ee-a5db-bdaafa7a80bd  rack2
DN  192.168.34.27 11.57 GiB  256          21.6%             17058f0f-5240-4531-a7b3-879d812df79e  rack2
DN  192.168.34.29 9.33 GiB   256          22.0%             75884e2c-ee72-4e07-a6ee-476d18119712  rack5
UN  192.168.34.31 5.69 GiB   256          22.0%             99861d0f-18fe-4083-9ba7-d3367a2b6d07  rack3
UN  192.168.34.17 5.27 GiB   256          21.6%             515eac4d-b47c-4593-b13c-885aac462a11  rack5
DN  192.168.34.18 ?          256          20.8%             29897cfd-f5d1-47e7-a1bc-764f9c3e1c0d  rack1
UN  192.168.34.22 2.85 GiB   256          21.9%             95e15779-a32c-44f9-a495-c3aceba72601
rack5
```

Figure 7.1: Cassandra cluster status.

Figure 7.2: Error message shown in the nodes with the IP address 192.168.34.27 and 192.168.34.29.



(a) (b) (c)

Figure 7.3: Latency of Cassandra when using request duplication a top of our proposed strategy, request reissue a top of our proposed strategy, and our proposed strategy (a) mean latency (b) 95^{th} percentile latency (c) 99^{th} percentile latency.

only has lower latency but also has higher throughput compared to request duplication a top of our proposed strategy, and request reissue a top of our proposed strategy as shown in fig. 7.4.

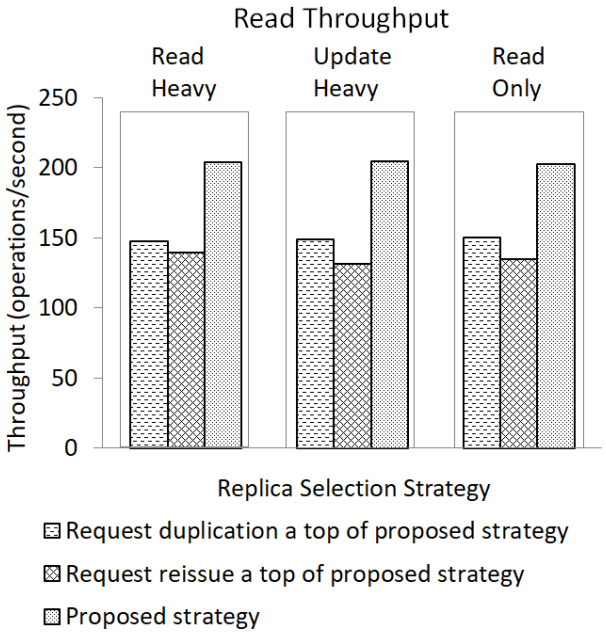
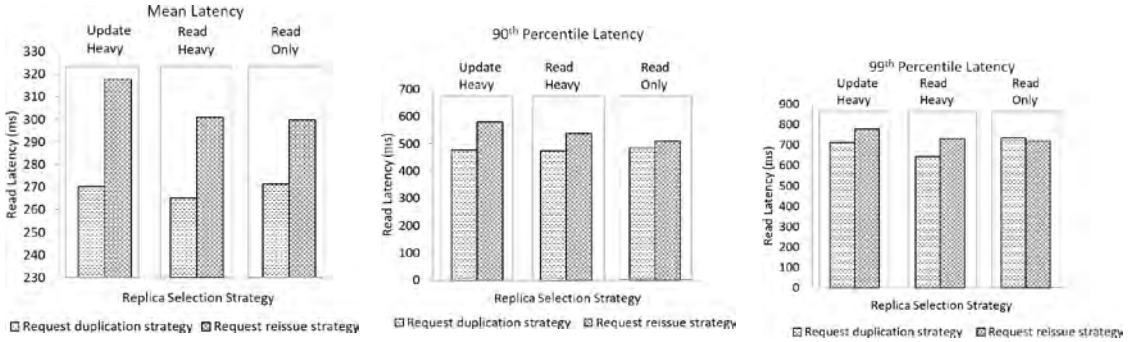


Figure 7.4: Comparison of read throughput at different workloads. Our proposed strategy has higher throughput at all the scenarios.

Observation3

According to our experimental results, we came to the conclusion that designing a request reissue or request duplication strategy a top of a replica selection strategy does not necessarily improve the performance of the system unless the replica selection strategy is efficient enough. In the



(a) (b) (c)

Figure 7.5: Latency of Cassandra when using request duplication and request reissue strategy (a) mean latency (b) 95^{th} percentile latency (c) 99^{th} percentile latency.

experiments discussed in chapter 6, we implemented request duplication and request reissue a top of two replica selection strategies 1) our proposed replica selection strategy and 2)dynamic snitch. Among these two, our proposed replica selection strategy improves the performance of both request reissue and request duplication in all the scenarios. On the other hand, the dynamic snitch improves the performance of the request reissue, and request duplication strategy in some cases but reduces in other cases.

Observation4

According to our experimental results, the request duplication strategy performs better than the request reissue strategy. Fig. 7.5 depicts that in all the workload situations the mean, 90^th, and 99^th percentile latency of the request duplication strategy is lower than the request reissue strategy. The request duplication strategy does not only reduce latency more efficiently than request duplication but also increases the throughput of the system in all the scenarios as shown in fig. 7.6. In all the workload situations the request duplication strategy has higher throughput than the request reissue strategy.

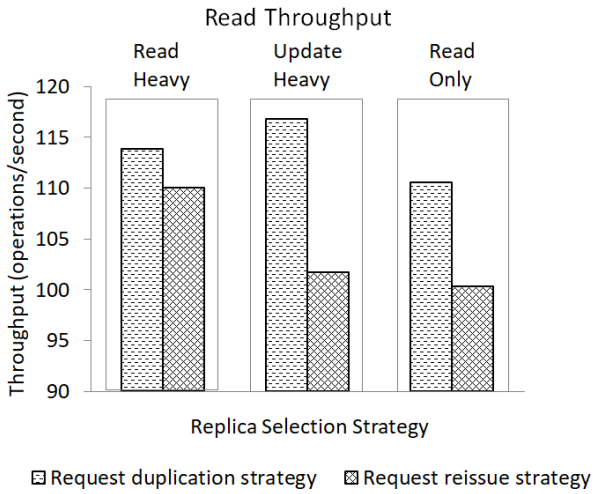


Figure 7.6: Comparison of read throughput at different workloads. Request duplication strategy has higher throughput at all the scenarios.

Chapter 8

Conclusion

Reducing tail latency is one of the most challenging issues in geo-distributed systems. Several strategies have already been developed including replica selection strategy, request duplication strategy, and request reissue strategy to reduce tail latency in the geo-distributed systems. Among these, the replica selection strategy is considered as the most efficient technique and is the main concern of this research. Several replica selection strategies have also already been developed but none of the strategies are efficient enough to adjust to the dynamic environment of the geo-distributed systems.

This research proposed a prediction-based replica selection strategy that can reduce the tail latency of the geo-distributed system and is able to adjust to the dynamic environment of the geo-distributed systems. Instead of directly designing the prediction-based strategy for the geo-distributed system, first, we designed the prediction model using multiple linear regression for the locally distributed system. After extensive experiments, we find out that this prediction-based strategy outperforms both C3 and dynamic snitch (which are two popular state-of-the-art replica selection strategies) in the locally distributed systems. We then extend this model and finally design the prediction-based replica selection strategy for reducing tail latency in the geo-distributed systems. We consider almost all the important factors for designing the replica selection strategy. For evaluating the performance of the proposed strategy we conducted some extensive experiments and compare these with four state-of-the-art replica selection strategies including C3 and dynamic snitch. Our proposed strategy outperforms all the other strategies in all the experiments in terms of both latency and throughput. Experimental results also show that our designed strategy also improves the performance of the request duplication and request reissue strategy.

8.1 Future Work

In the future, we plan to extend our work achieving the followings:

- In our proposed strategy, the coefficients are recalculated and updated after a user-specified time interval and we do this for coping with the changing situation of the system. An extensive study is required to determine the standard time interval and in the future, we will explore and evaluate some standard time intervals for different system settings.
- In our equation, we use the EWMA of all the metrics. In the future, we will explore and evaluate the effect of other MA's such as SMMA, and LWMA.
- For conducting the experiments of the request reissue strategy we set the $speculative_{retry}$ to $99^{percentile}$. In the future, we will explore and evaluate the effect of some other percentiles.
- When we implement request reissue and request duplication strategy a top of dynamic snitch the throughput falls exceptionally with a read-heavy workload. In the future, we will investigate and identify the reason for such an outcome.

References

[1] R. Hsu, “Who moved my 99th percentile latency?,” 2015. Last accessed on April 24, 2021. [Online]. Available: https://engineering.linkedin.com/performance/who-moved-my-99th-percentile-latency.

[2] R. S. Couto, S. Secci, M. E. M. Campista, and L. H. M. K. Costa, “Geo-distributed data centers: Distance and robustness trade-offs,” in *2014 Brazilian Symposium on Computer Networks and Distributed Systems*, p. 402–411, ACM, 2014.

[3] “Global infrastructure,” 2021. Last accessed on May 7, 2021. [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/.

[4] H. S. Adiche, *Reducing Long Tail Latencies in Geo-Distributed Systems.* PhD thesis, KTH, School of Information and Communication Technology (ICT), 2016.

[5] J. Wu, *Reducing the Tail Latency of a Distributed NoSQL Database*. PhD thesis, University of Nebraska - Lincoln, 2018.

[6] S. M. Shithil and M. A. Adnan, “A prediction based replica selection strategy for reducing tail latency in distributed systems,” in *IEEE International Conference on Cloud Computing, CLOUD*, pp. 522–526, IEEE, 2020.

[7] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, “Bobtail: Avoiding long tails in the cloud,” in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 329–341, USENIX Association, 2013.

[8] J. Dean and L. A. Barroso, “The tail at scale,” *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[9] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, “Measuring interference between live datacenter applications,” in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE, 2012.

[10] J. Li, D. R. K. Sharma, Naveen Kr.and Ports, and S. D. Gribble, “Tales of the tail: Hardware, os, and application-level sources of tail latency,” in *ACM Symposium on Cloud Computing (SOCC)*, pp. 1–12, ACM, 2014.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Symposium on Operating Systems Principles (SOSP)*, pp. 205–220, ACM, 2007.

[12] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 63–74, 2010.

[13] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 219–230, 2013.

[14] "Who moved my 99th percentile latency?." Last accessed on January 19, 2021. [Online]. Available: https://engineering.linkedin.com/performance/who-moved-my-99th-percentile-latency.

[15] "Storage: How 'tail latency' impacts customer-facing applications." Last accessed on January 19, 2021. [Online]. Available: https://www.computerweekly.com/opinion/Storage-How-tail-latency-impacts-customer-facing-applications#: :text=Tail

[16] F. Chang, J. Dean, S. Ghemawat, W. C. . Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, 2008.

[17] "How are read requests accomplished?," 2021. Last accessed on May 5, 2021. [Online]. Available: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlClientRequestsRead.html.

[18] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up distributed request-response workflows," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 219–230, 2013.

[19] Z. Wu, C. Yu, and H. V. Madhyastha, "Costlo: Cost-effective redundancy for lower latency variance on cloud storage services," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, p. 543–557, USENIX, 2015.

[20] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 159–170, 2013.

[21] C. Stewart and C. xChakrabart, "Zoolander: Efficiently meeting very strict, low-latency slos," in *10th International Conference on Autonomic Computing*, pp. 265–277, USENIX Association, 2013.

[22] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker, "More is less: Reducing latency via redundancy," in *11th ACM Workshop on Hot Topics in Networks*, pp. 13–18, ACM, 2012.

[23] H. Xu and B. Li, "Repflow: Minimizing flow completion times with replicated flows in data centers," in *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, p. 1581–1589, IEEE, 2014.

[24] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[25] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *USENIX Conference on Networked Systems Design and Implementation(NSDI)*, USENIX, 2015.

[26] V. Vikas Jaiman, S. B. Mokhtar, V. Quema, L. Y. Chen, and E. Riviere, "Heron: Taming tail latencies in key-value stores under heterogeneous workloads," in *Symposium on Reliable Distributed Systems (SRDS)*, IEEE, 2018.

[27] W. Jiang, H. Xie, X. Zhou, L. Fang, and J. Wang, "Performance analysis and improvement of replica selection algorithms for key-value stores," in *IEEE International Conference on Cloud Computing, CLOUD*, p. 786–789, IEEE, 2017.

[28] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. P. Ratnasamy, and S. J. Shenker, "Low latency via redundancy," in *ACM conference on Emerging networking experiments and technologies (CoNEXT)*, p. 283–294, ACM, 2013.

[29] S. Larsen and B. Lee, "Survey on system i/o hardware transactions and impact on latency, throughput, and other factors," *Advances in Computers*, vol. 92, pp. 67–104, 2014.

[30] K. Bogdanov, *Reducing Long Tail Latencies in Geo-Distributed Systems*. PhD thesis, School of Information and Communication Technology, KTH Royal Institute of Technology, Stockholm, Sweden, 2016.

[31] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. H. Paleczny, "Workload analysis of a large-scale key-value store," *ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS)*, vol. 40, no. 1, p. 53–64, 2012.

[32] M. Mitzenmacher, "How useful is old information?," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 1, pp. 6–20, 2000.

[33] S. Simic, "7 ways to reduce server response time," 2019. Last accessed on April 24, 2021. [Online]. Available: https://phoenixnap.com/kb/reduce-server-response-time.

[34] H. He, "Storage: How 'tail latency' impacts customer-facing applications," 2019. Last accessed on April 24, 2021. [Online]. Available: https://www.computerweekly.com/opinion/Storage-How-tail-latency-impacts-customer-facing-applications—#: :text=Tail%20latency%20is%20the%20small,bulk%20of%20its%20response%20times.—&text=That%201%25%20is%20the%20tail%20latency%20problem%20for%20the%20shepherd.

[35] T. Kaler, Y. He, and S. Elnikety, "Optimal reissue policies for reducing tail latency," in *29th ACM Symposium on Parallelism in Algorithms and Architectures*, p. 195–206, ACM, 2017.

[36] W. (Boston) and MIT, "1963 timesharing: A solution to computer bottlenecks," 2010. Last accessed on May 6, 2021. [Online]. Available: https://www.youtube.com/watch?v=Q07PhW5sCEk.

[37] "What is cloud computing?," 2021. Last accessed on May 6, 2021. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/.

[38] "Amazon ec2: Secure and resizable compute capacity to support virtually any workload," 2021. Last accessed on May 6, 2021. [Online]. Available: http://aws.amazon.com/ec2/.

[39] "Google cloud," 2021. Last accessed on May 6, 2021. [Online]. Available: https://cloud.google.com/.

[40] "Microsoft azure," 2021. Last accessed on May 6, 2021. [Online]. Available: https://azure.microsoft.com/en-gb/.

[41] "Ibm cloud," 2021. Last accessed on May 6, 2021. [Online]. Available: https://www.ibm.com/cloud.

[42] Z. Wum, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: cost-effective geo-replicated storage spanning multiple cloud services," in *Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 292–308, ACM, 2013.

[43] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *10th USENIX conference on File and Storage Technologies*, pp. 18–18, USENIX Association, 2012.

[44] "Google data centers," 2021. Last accessed on May 7, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Google_data_centers.

[45] N. Sattairaju, "The secret cost of google's data centers: Billions of gallons of water to cool servers," 2020. Last accessed on May 7, 2021. [Online]. Available: https://time.com/5814276/google-data-centers-water/.

[46] I. C. Education, “Data centers,” 2020. Last accessed on May 7, 2021. [Online]. Available: https://www.ibm.com/cloud/learn/data-centers.

[47] “Network latency - common causes and best solutions — ir,” 2021. Last accessed on February 10, 2021. [Online]. Available: https://www.ir.com/guides/what-is-network-latency.

[48] “What is round-trip time? — rtt definition,” 2021. Last accessed on February 10, 2021. [Online]. Available: https://www.cloudflare.com/en-gb/learning/cdn/glossary/round-trip-time-rtt/.

[49] “Apache cassandra,” 2021. Last accessed on January 21, 2021. [Online]. Available: http://cassandra.apache.org/.

[50] R. Buyya, R. Ranjan, and R. N. Calheiros, “Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services,” in *International Conference on Algorithms and Architectures for Parallel Processing*, p. 13–31, Springer, 2010.

[51] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, “Workload analysis and demand prediction of enterprise data center applications, in workload characterization,” in *Symposium on, IEEE*, p. 171–180, IEEE, 2017.

[52] J. Jung, B. Krishnamurthy, and M. I. Rabinovich, “Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites,” in *11th international conference on World Wide Web (WWW)*, p. 293–304, ACM, 2002.

[53] T. V. Eicken, “Animoto’s facebook scale-up,” 2008. Last accessed on January 27, 2021. [Online]. Available: http://goo.gl/UDNXS9.

[54] “Using nginx as http load balancer,” 2021. Last accessed on February 9, 2021. [Online]. Available: http://nginx.org/en/docs/http/load_balancing.html.

[55] “Response time (technology),” 2021. Last accessed on February 24, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Response_time_(technology)
#: :text=Response%20time%20is%20the%20total,to%20a%20request%20
for%20service.&text=The%20service%20time%20is%20the,takes%20
X%20amount%20of%20time.

[56] H. Yang, Q. Zheng, M. Lim, and Y. Sun, “How to avoid herd behavior: A stochastic multi-choice scheduling algorithm and parameters analysis in grid scheduling,” *International Journal of Information Technology & Decision Making*, vol. 14, no. 2, pp. 287–315, 2015.

[57] "Overcoming cassandra write performance problems," 2021. Last accessed on February 9, 2021. [Online]. Available: https://www.senticore.com/overcoming-cassandra-write-performance-problems/.

[58] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *10th ACM SIGCOMM conference on Internet measurement*, p. 267–280, ACM, 2010.

[59] "What is round-trip time (rtt)?," 2021. Last accessed on February 9, 2021. [Online]. Available: https://searchnetworking.techtarget.com/definition/round-trip-time.

[60] A. Bekker, "Dynamodb vs. cassandra: from "no idea" to "it's a no-brainer"," 2018. Last accessed on May 26, 2021. [Online]. Available: https://www.kdnuggets.com/2018/08/dynamodb-vs-cassandra.html.

[61] "What is a key-value store?," 2021. Last accessed on May 26, 2021. [Online]. Available: https://aerospike.com/what-is-a-key-value-store/.

[62] "What is key-value store and why is it important?," 2019. Last accessed on May 26, 2021. [Online]. Available: https://www.linkedin.com/pulse/what-key-value-store-why-important-ken-grohe/.

[63] "Cassandra: Overview," 2019. Last accessed on May 26, 2021. [Online]. Available: https://cassandra.apache.org/doc/latest/architecture/overview.html.

[64] "Apache cassandra 3.0: Architecture in brief," 2019. Last accessed on May 26, 2021. [Online]. Available: https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archIntro.html.

[65] J. Carpenter and E. Hewitt, *Cassandra: The Definitive Guide: Distributed Data at Web Scale 2nd Edition*. O'Reilly Media, Inc, 2016.

[66] "Data replication," 2016. Last accessed on June 26, 2021. [Online]. Available: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/architecture/archDataDistributeReplication.html.

[67] "Snitch," 2016. Last accessed on June 17, 2021. [Online]. Available: https://cassandra.apache.org/doc/latest/operating/snitch.html.

[68] "Create table," 2022.

[69] "Dynamic snitching in cassandra: past, present, and future — datastax," 2020. Last accessed on May 16, 2020. [Online]. Available: https://www.datastax.com/blog/2012/08/dynamic-snitching-cassandra-past-present-and-future.

[70] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, 1997.

[71] "Documentation," 2020. Last accessed on June 1, 2020. [Online]. Available: https://cassandra.apache.org/doc/latest/operating/compaction/index.html#: :text=The

[72] J. Carpenter and E. Hewit, *Cassandra: the definitive guide*. O'Reilly Media, Inc, 2020.

[73] "The cassandra utility," 2020. Last accessed on June 1, 2020. [Online]. Available: https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/tools/toolsCUtility.html.

[74] "apache/cassandra," 2020. Last accessed on June 1, 2020. [Online]. Available: https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/locator/DynamicEndpointSnitch.java.

[75] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud servings ystems with ycsb," in *ACM symposiumon Cloud computing*, ACM, 2010.

[76] Investopedia, "Multiple linear regression (mlr) definition," 2021. Last accessed on May 5, 2021. [Online]. Available: https://www.investopedia.com/terms/m/mlr.asp.

[77] "Santa-maria-shithil /prediction-based-replica-selection-strategy-for-geo-distributed-cluster," 2020. Last accessed on Feb 19, 2021. [Online]. Available: Santa-Maria-Shithil /prediction-based-replica-selection-strategy-for-geo-distributed-cluster.

[78] D. Documentation, "How is data maintained?," 2021. Last accessed on Nov 7, 2021. [Online]. Available: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlHowDataMaintain.html.

[79] Y. Chenm, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, IEEE, 2011.

[80] D. Documentation, "How are read requests accomplished?," 2021. Last accessed on Jun 5, 2022. [Online]. Available: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlClientRequestsRead.html.

[81] T. D. Blog, "Rapid read protection in cassandra 2.0.2," 2013. Last accessed on Jun 15, 2022. [Online]. Available: https://www.datastax.com/blog/rapid-read-protection-cassandra-202.

Appendix A

Algorithms

Algorithm for Replica Selector in the Locally Distributed Systems

In algorithm 1 we show how the replica selector selects the replica for routing the read request in the locally distributed systems.

Algorithm 1 Replica Selector for Locally Distributed Systems

```
1: procedure ReplicaSelector(req, R) {Request req, Replicas R }
2:  R ←sort(R) {sort the servers based on the score }
3: for server s in R do
4:     state ← send(req, s)
5:     if state = successful then
6:         latency_tracker.start(s)
7:         return
8:     end if
9: end for
10: end procedure
```

Algorithm for Feedback Controller in Locally Distributed Systems

In algorithm 2 we show how the feedback controller collects feedback from the servers and sends these to the score updater in the locally distributed systems.

Algorithm 2 Feedback Controller for Locally Distributed Systems

1: **procedure** FeedbackController(q_s, r_s, $\frac{1}{\mu_s}$, s){this procedure executes on each read response from the server}
2: $l_s \leftarrow latency\_tracker.getLatency(s)$
3: $latency\_tracker.stop(s)$
4: $scoreUpdater(q_s, r_s, \frac{1}{\mu_s})$
5: $coefficientsUpdater(q_s, \frac{1}{\mu_s}, l_s)$
6: **end procedure**

Algorithm for Coefficients Updater in the Locally Distributed Systems

In algorithm 3 we show how the coefficients updater updates the coefficients in the locally distributed systems.

Algorithm 3 Coefficients Updater for Locally Distributed Systems

1: **procedure** CoefficientsUpdater(q_s, $\frac{1}{\mu_s}$, l_s, s)
2: $updateAverage(q_s, \frac{1}{\mu_s}, l_s)$ {updates $\bar{q}_s$, $\frac{1}{\bar{\mu}_s}$ and $\bar{l}_s ofservers$}
3: **if** `interval time elasped` **then**
4: $calculateCoefficients(\bar{q}_s, \frac{1}{\mu_s}, \bar{l}_s, s)$ {calculates the coefficients a, b, and c of server s}
5: $updateCoefficients(a, b, c, s)$ {updates coefficients of server s}
6: **end if**
7: **end procedure**

Algorithm for Score Updater in the Locally Distributed Systems

This algorithm shows how the system updates the score in the locally distributed systems.

Algorithm 4 Score Updater for Locally Distributed Systems

1: **procedure** ScoreUpdater(q_s, r_s, $\frac{1}{\mu_s}$, s)
2: $updateEWMA(q_s, r_s, \frac{1}{\mu_s})$ {this updates $\hat{q}_s$, $\hat{r}_s$ and $\frac{1}{\mu_s}$}
3: $\hat{l}_{ps} \leftarrow a + b \cdot \hat{q}_s + \frac{c}{\hat{\mu}_s}$
4: $\varphi_s \leftarrow \hat{l}_{ps} + \hat{r}_s$
5: $updateScore(\varphi_s, s)$
6: **end procedure**

Algorithm for Replica Selector in the Geo-Distributed Systems

In algorithm 5 we show how the replica selector selects the replica for routing the read request in the geo-distributed systems.

Algorithm 5 Replica Selector for Geo-Distributed Systems

```
1: procedure ReplicaSelector(req, R) {Request req, Replicas R }
2:  R ← sort(R) {sort the servers based on the score }
3: for server s in R do
4:     state ← send(req, s)
5:     if state = successful then
6:         latency_tracker.start(s)
7:         round_trip_time_tracker.start(s)
8:         return
9:     end if
10: end for
11: end procedure
```

Algorithm for Feedback Controller in Geo-Distributed Systems

In algorithm 6 we show how the feedback controller collects feedback from the servers and sends these to the score updater in the geo-distributed systems.

Algorithm 6 Feedback Controller for Geo-Distributed Systems

1: **procedure** FeedbackController(q_s, R_{ios}, $\frac{1}{\mu_s}$, s){this procedure executes on each read response from the server}
2: $L_s \leftarrow latency\_tracker.getLatency(s)$
3: $RTT_s \leftarrow round\_trip\_time\_tracker.getRTT(s)$
4: $latency\_tracker.stop(s)$
5: $round\_trip\_time\_tracker.stop(s)$
6: $scoreUpdater(RTT_s, q_s, R_{ios}, \frac{1}{\mu_s}, s)$
7: $coefficientsUpdater(RTT_s, q_s, \frac{1}{\mu_s}, L_s, s)$
8: **end procedure**

Algorithm for Coefficients Updater in the Geo-Distributed Systems

In algorithm 7 we show how the coefficients updater updates the coefficients in the geo-distributed systems.

Algorithm 7 Coefficients Updater for Geo-Distributed Systems

1: **procedure** CoefficientsUpdater(RTT_s, q_s, $\frac{1}{\mu_s}$, L_s, s)
2: *updateAverage*(RTT_s, q_s, $\frac{1}{\mu_s}$, L_s)**{**updates $\bar{RTT}_s$, $\bar{q}_s$, $\frac{1}{\mu_s}$, $\bar{L}_s ofservers$**}**
3: **if** `interval time elasped` **then**
4: *calculateCoefficients*($\bar{RTT}_s$, $\bar{q}_s$, $\frac{1}{\mu_s}$, $\bar{L}_s$, s) **{**calculates the coefficients a, b, and c of server s**}**
5: *updateCoefficients*(a, b, c, s) **{**updates coefficients of server s**}**
6: **end if**
7: **end procedure**

Algorithm for Score Updater in the Geo-Distributed Systems

This algorithm shows how the system updates the score in the geo-distributed systems.

Algorithm 8 Score Updater for Geo-Distributed Systems

1: **procedure** ScoreUpdater(RTT_s, q_s, R_{ios}, $\frac{1}{\mu_s}$, s)
2: *updateEWMA*(RTT_s, q_s, R_{ios}, $\frac{1}{\mu_s}$)**{**this updates $\hat{q}_s$, $\hat{r}_s$, $\hat{R}_{ios}$ and $\frac{1}{\hat{\mu}_s}$**}**
3: $\hat{L}_{ps} \leftarrow a + b \cdot \hat{RTT}_s + c \cdot \hat{R}_s$
4: $\varphi_s \leftarrow \hat{L}_{ps} + \hat{R}_{ios}$
5: *updateScore*(φ_s, s)
6: **end procedure**

Appendix B

Equations

B.1 Derivation of the Coefficients of Multiple Linear Regression

The general equation of multiple linear regression for k independent variables is shown in (B.1).

$$\hat{y} = b_0 + b_1 \cdot x_1 + b_2 \cdot x_2 + ... + b_k \cdot x_k \tag{B.1}$$

where y is the dependent variable whose value the regression model predicts. x_1,x_2,. ... ,x_k are the independent variables, b_0 is the y-intercept and b_1, b_2,. .. ,b_k are slope the coefficients for each independent variable. We can express the regression equation of (B.1) in the matrix form by using three matrices: Y, b, and X. Where

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix} \tag{B.2}$$

$$b = \begin{bmatrix} b_0 \\ b_1 \\ . \\ . \\ b_k \end{bmatrix} \tag{B.3}$$

$$X = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & . & . & x_{1,k} \\ 1 & x_{2,1} & x_{2,2} & . & . & x_{2,k} \\ . & & & & & \\ . & & & & & \\ 1 & x_{n,1} & x_{n,2} & . & . & x_{n,k} \end{bmatrix} \tag{B.4}$$

Given these matrices, the multiple regression equation can be expressed concisely as:

$$Y = Xb \tag{B.5}$$

And the normal equation is:

$$X^T Y = X^T X b \tag{B.6}$$

And so the solution for the coefficients is:

$$b = (X^T X)^{-1} X^T Y \tag{B.7}$$

Now if we solve the equation (B.7) for two independent variables x_1 and x_2 then we will find:

$$b_0 = \bar{y} - b_1 \cdot \bar{x_1} - b_2 \cdot \bar{x_2} \tag{B.8}$$

$$b_1 = \frac{\sum x_2^2 \cdot \sum (x_1 \cdot y) - \sum (x_1 \cdot x_2) \cdot \sum (x_2 \cdot y)}{\sum x_1^2 \cdot \sum x_2^2 - (\sum (x_1 \cdot x_2))^2} \tag{B.9}$$

$$b_2 = \frac{\sum x_1^2 \cdot \sum (x_2 \cdot y) - \sum (x_1 \cdot x_2) \cdot \sum (x_1 \cdot y)}{\sum x_1^2 \cdot \sum x_2^2 - (\sum (x_1 \cdot x_2))^2} \tag{B.10}$$

Generated using Postgraduate Thesis LaTeX Template, Version 1.03. Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

This thesis was generated on Monday 8th August, 2022 at 6:22am.