Profit Maximization in Reliability Aware and Deadline Guaranteed Cloud Storage Systems

B. Tech Project Report Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Bachelor of Technology

by

Kartik Verma and Vineet Agarwal

(190101044, 190101099)

under the guidance of

Aryabartta Sahu

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, GUWAHATI

GUWAHATI - 781039, ASSAM

Abstract

Recent years have seen a boom in businesses using cloud solutions. However, this increase in demand has led to issues such as unpredictable latency and low reliability. These issues are critical when dealing with data-intensive applications like cloud storage services. In this work, we develop an approach to meet an SLO which restricts the request of a tenant failing to meet its required deadline below a given threshold while considering server reliability and crash recovery. Five heuristics are proposed in our work, Request Allotment which allocates an incoming request to a set of servers, Replica Creation which creates new data replicas as and when the need arises, Server pairing for better crash recovery; Workload Consolidation for better energy and cost efficiency and Server Wake-Up. We also formulate a model to calculate the upper bound on the service rate at a server so that it meets the deadline requirement of the scheduled requests. Evaluations done by simulation using synthetic and publicly available data show the superior performance of our approach as compared to the state-of-the-art approach.

Contents

Chapter 1

Introduction

Cloud Computing has revolutionized how we think about storage services. More and more companies are shifting their storage to the cloud to reduce capital expenditure. The pay-as-you-go model of the cloud provides them with a cheap solution that can easily be scaled when the need arises, which is highly favorable for storage operations. However, despite these advantages, the widespread adoption of cloud storage systems is still incomplete, as cloud systems face multiple challenges, such as ensuring reliability and security while minimizing energy consumption and cost.

With this rise in demand, one of the critical problems for data storage cloud services is the issue of unpredictable performance and high data latency, the leading cause of which is server overload which can be managed by using better request scheduling approaches.

Fig.1 shows an example of the system model with multiple tenants. Each tenant has an expected deadline and reliability followed by its requests. The profit earned by the service provider after satisfying a job also depends on this deadline and reliability requirements. The higher the reliability and lower the deadline requirement, the more profit the provider earns. A request by tenant may request multiple data partitions and can be scheduled to multiple storage servers.

A server's reliability is dependent on a vast number of factors. Broadly they can be classed into network and service faults. In our paper,each server has an associated **network and service fault rate** using which the server reliability is calculated. Each request passes through a front-end server which runs the request allotment algorithm to allocate the request to the servers. The front-end server also runs other algorithms like server pairing and workload consolidation for performance enhancement.
In this paper, we propose a scheduling approach that meets each tenant's Service Level Objective (SLO), which restricts the request of a tenant failing to meet its required deadline below a given threshold while maximizing the profit earned by the storage service provider.

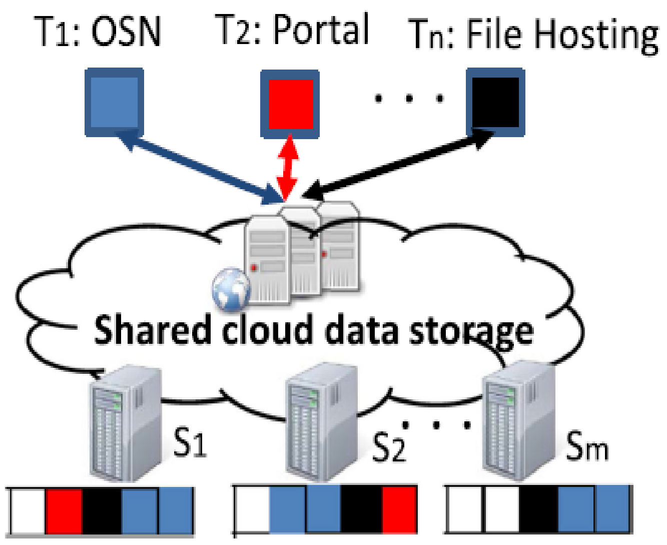

Fig. 1.1: System overview with n tenants and m storage servers.

The solution approach proposed in our work can be divided into five key steps:-

- **Request Allotment** :- This process is executed for every incoming request. It allocates a set of servers containing the requested data partitions to the tenant's request.

- **Replica Creation**:- In case no server containing the requested data partition can serve the request, a new replica of the data partition is created on a server that previously didn't have the particular data partition. To find the most suitable server, the replica creation process is executed.

- **Server Pairing**:- Each server has a list of the most favorable servers to redirect requests to in case the server crashes or overloads. Using the server pairing process, this list is calculated for each server depending on the proximity and service capacities of the servers.
- **Workload Consolidation**:-This process is called at regular intervals to maximize the system's energy efficiency by moving the requests from the most underloaded servers to other servers and switching the underloaded servers off.
- **Server Wake-up**:-It finds the most suitable server to wake up in case a new server has to be created before creating the new replica.

Following are the contributions made in this project:-

1. We came up with an approach for allocating the requests to servers while maximizing the profit of the front-end server under the constraints of reliability and deadline.
2. A workload consolidation algorithm that increases the energy efficiency of the system.
3. A server pairing and server wake-up algorithm for better server crash recovery.

The report is organized into six chapters. Chapter 2 briefs regarding the related works in this domain and work toward solving a similar problem. The third chapter formulates the problem statement and gives an overview of the system model, while Chapter 4 has solution approaches. Chapter 5 contains the performance evaluation and results. The conclusion and possible future work is briefed in Chapter 6. References are contained in Chapter 7.

Chapter 2

Review of Prior Works

The popularity of cloud storage systems has led to numerous studies focusing on enhancing data availability and access efficiency. In one such study [1], Liu *et al.* presented a storage service called DGCloud, which guarantees meeting the service level objective (SLO) of tenants by fulfilling a given percentage of requests within a specified deadline. While the study proposes a mathematical model for deadline-guaranteed service rate, it does not consider server reliability. Another study by Singh *et al.* [3] proposed the use of caching to improve server performance by creating a buffer between front-end servers and data servers. In [4], the authors discuss various fault tolerance techniques and their respective success rates and payment methods.

Cloud computing and storage services have also emphasized meeting Quality of Service (QoS) requirements, including energy consumption, response time, cost, and reliability. Stavrinides and Karatza [5] proposed an energy-efficient, cost-effective scheduling strategy that takes into account QoS requirements. In [6], Tang *et al.* presented a reliability-aware and cost-effective job scheduling algorithm that employs a two-dimensional long-short term memory (TDLSTM) neural network to predict server reliability. Sahoo *et al.* [7] designed a Link-based Virtual Resource Management (LVRM) algorithm to allocate virtual machines to physical machines based on available and requested resources. Long *et al.* proposed the Best Response Algorithm (BRA) [8] to optimize total cost on collaborative edge and cloud systems while meeting response time QoS constraints. Zhu *et al.* extended the traditional PB technique in [9] based on cloud characteristics and built a real-time workflow fault-tolerant model. Additionally, reliability analysis of computing systems, particularly physical resources, is a classical research topic [10], [11], [12]. Dogan and Ozguner [13] performed a formal analysis of the reliability of heterogeneous computing systems and proposed a reliability-aware dynamic level scheduling algorithm.

Chapter 3

System Model and Problem Statement

3.1 System Model

We have a heterogeneous cloud storage system consisting of M data servers with servers having different reliability, system capacity, and cost. There are N tenants in the system, each with a deadline d_k and service reliability ϵ_k. Every request created by a tenant is characterized by its deadline d_k and service reliability ϵ_k, i.e., for every tenant k, a maximum of ϵ_k percentage of requests can exceed the deadline d_k.

Every data request goes through a front-end server that runs the request allotment algorithm to assign the request to a set of servers S_i. A tenant, in its request, can request for multiple data partitions and be scheduled by the front-end server on multiple servers. The front-end server has an array h, which stores the base cost to access data partition p under no constraints of reliability and deadline. Moreover, each server has a service cost per unit denoted by SC_j and a maintenance cost per request denoted by MC_j. Each server has reliability calculated from its software and hardware fault rates. A request can only be scheduled on a server if its reliability exceeds the reliability required by the tenant ϵ_k.

An upper bound on the service rate to satisfy the deadline is calculated for each server.

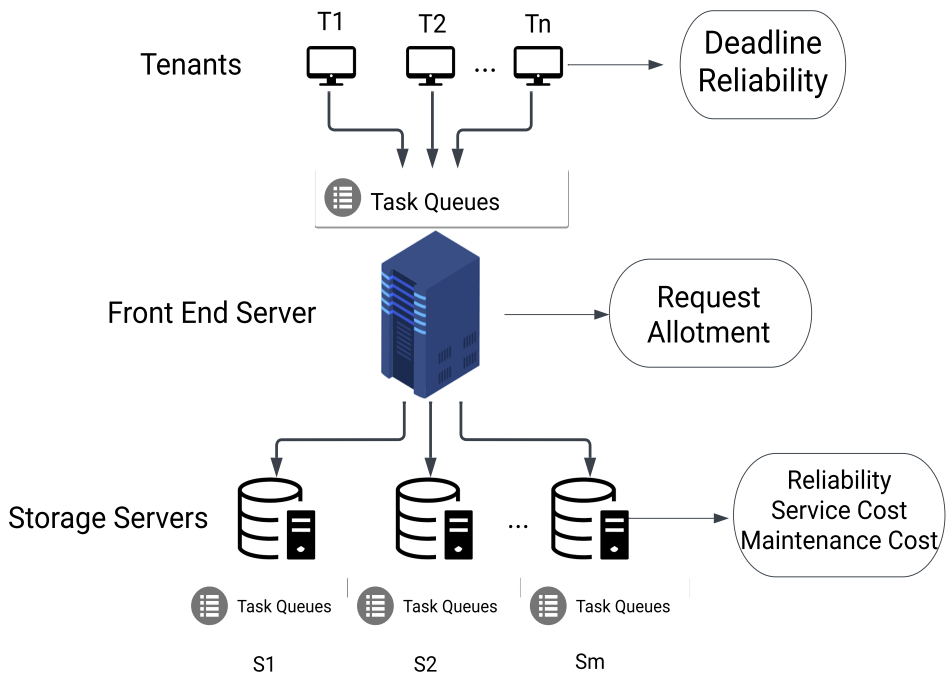

Fig. 3.1: In-depth view of the system

This deadline-guaranteed service rate is unique for every server and depends on the tenants whose request is currently being served by the server. It is denoted by λ'_j. A request can only be allocated to a server if its available service capacity a_j after assuming that the request is scheduled on the server is greater than 0. An important point to note is that before scheduling a request, we check whether the tenant is currently being served by the server and if not, then the deadline guaranteed service rate is calculated assuming that the tenant is allocated the server.

3.2 Fault Prediction

Each server has two parameters, network fault rate f_j^{net} and storage fault rate f_j^{stor}, both of which follow a Poisson Distribution. The exponential probability function for network failure can thus be formulated as $f_{net} = f_j^{nf} e^{-f_j^{net} t}$. Similarly, we can get the exponential probability function for storage failure. Assuming that both network and storage faults are independent of each other, we can get the reliability of the j^{th} server as[8]

$$R_j = 1 - \int_0^t f_{nf}\, dt - \int_0^t f_{sf}\, dt \tag{3.1}$$

which comes out to be

$$R_j = e^{-(f_j^{net} + f_j^{stor})t} \tag{3.2}$$

3.3 Problem Statement

We aim to maximize the profit for the front-end server, which schedules the requests, i.e., we want to schedule **all** the requests generated from the tenants on the servers under the constraints of reliability and deadline while maximizing the profit earned. Revenue generated by executing a request can be formulated as

$$revenue_i = \frac{(\sum_p^{D_i} h_p) * (1 + e^{\epsilon_k})}{(1 + e^{d_k})} \tag{3.3}$$

where h_p is the cost fixed by the front-end server to serve data partition p and D_i is the set of partitions requested. ϵ_k and d_k are the tenant's reliability and deadline requirements, respectively.

The cost incurred by the front-end server to serve a request $cost_i$ depends on the maintenance and operational cost of the servers on which the request is scheduled. The cost can be formulated as follows

$$cost_i = \sum_p^{D_i} MC_j^p + SC_j^p \tag{3.4}$$

where D_i is the set of all data partitions requested. MC_j^p is the maintenance cost of the server and SC_j^p is the service cost of the server from which the p^{th} data partition is fetched. The problem can now be defined as follows:-

$$Maximize \sum_i^{requests} revenue_i - cost_i \tag{3.5}$$

under the constraints that:-

1. $R_j \geqslant \epsilon_k$

2. $Num_i \leqslant D_i$

The first constraint restricts the request allotment algorithm to allocate requests to the servers whose server reliability is greater than the reliability request by the tenant.

The second constraint restricts the number of servers a request is scheduled on to be less than or equal to the number of data partitions requested by the tenant, i.e., a data partition can only be requested from a single server.

Notation	Definition
ϵ_k	Expected Reliability of the k^{th} tenant.
d_k	Expected Deadline of the k^{th} tenant.
λ'_j	Deadline Guaranteed service rate for server j
λ_j	service rate for server j
a_j	Available service capacity of the j^{th} server
μ_j	Utilization of the j^{th} server
c_p	Size of p^{th} data partition
h_p	Base cost of accessing data partition p.
D_i	Data partitions requested by the i^{th} request
Num_i	Number of servers on which i^{th} request is scheduled
f_j^{net}	Network Fault Rate of the j^{th} server.
f_j^{stor}	Storage Fault Rate of the j^{th} server.
R_j	Reliability of the j^{th} server.
SC_j	Service Cost per unit for the j^{th} server.
MC_j	Maintenance Cost per request for the j^{th} server.
$revenue_i$	Revenue earned by executing the i^{th} request
$cost_i$	Cost of executing the i^{th} request

Table 3.1: Notations used in this work

Chapter 4

Solution Approach

The flow diagram given shows how the front-end server processes an incoming request. The front-end server runs the ***Request Allotment*** algorithm to allocate the request to a data server. In case no server is capable of executing the request, then the ***Replica Creation*** algorithm is executed to create a new replica of the data partition requested. If a new replica is created, then the request is scheduled on the server; else, a new server is woken up to execute the request. **Workload Consolidation** runs at fixed intervals to optimize energy consumption.

4.1 Calculating Deadline Guaranteed service rate (λ'_j):-

Deadline Guaranteed service rate is the upper bound on the service rate of a server such that it can satisfy the deadline of all the requests allocated to it.
Before computing λ'_j, we first compute P^i_k, which is the probability of any request R_i from tenant k meeting the deadline requirement. Based on this probability, we then **derive the deadline-guaranteed service rate** for s_j.From [1], we get

$$\lambda'_{j,k} = \mu_j - (ln(1 - \sqrt[\alpha]{(1-\epsilon_k)/p'}/d_k \qquad (4.1)$$

where p' is the probability of a request simultaneously requesting no larger than α partitions. The tenant with the lowest value of $\lambda'_{j,k}$ has the strictest deadline requirements and thus $\lambda'_j = Min(\lambda'_{j,k})$. Therefore whenever a new tenant is allocated to a server, we must calculate the value of $\lambda'_{j,k}$.

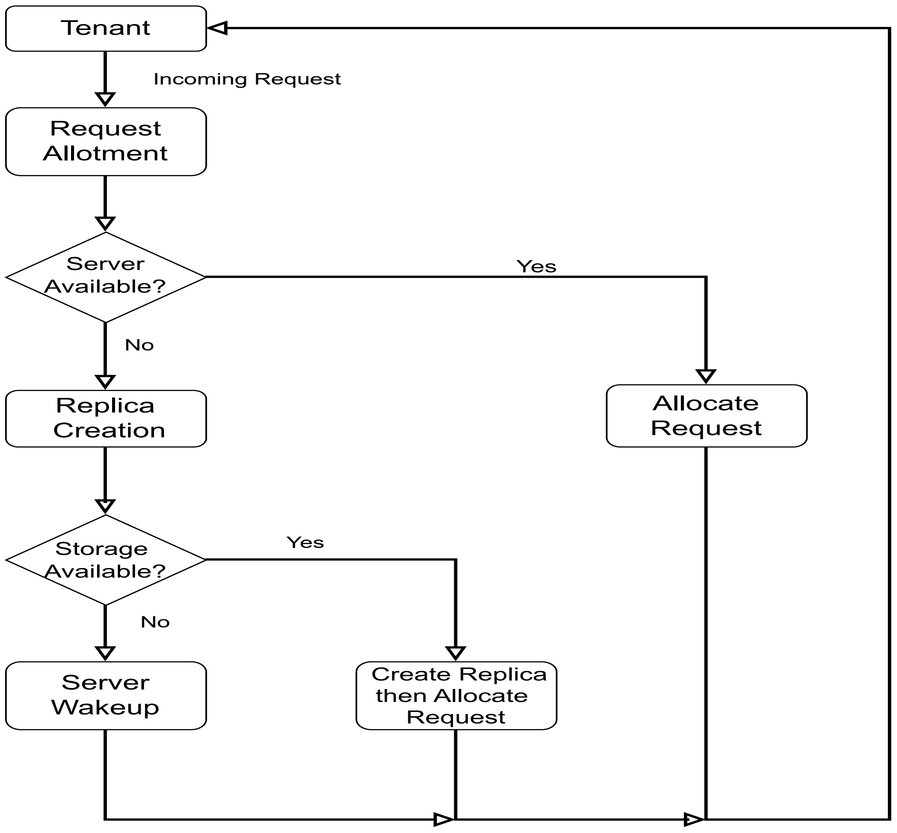

Fig. 4.1: Flow for an incoming request

4.2 Request Allotment

This algorithm runs whenever a new request enters the system and it allocates the request to a set of data servers. Inputs to the algorithm are the request characteristics(tenant's deadline, reliability requirements, and requested data partitions) and the server information.

The algorithm goes through the set of all servers for every data partition requested and chooses the server with the maximum reliability, given that the server's reliability is greater than the request's reliability and the available service capacity of the server after request allocation is non-negative. If the server is not currently serving the tenant, then its deadline-guaranteed service rate is calculated, assuming the server serves the tenant.

If no server can serve the request, the request allotment algorithm calls the replica creation algorithm.

Algorithm 1 Request Allotment Algorithm

Input: Incoming request, Server Information

Output: Allocation of request to servers

```
1: Sort the set of servers in non-increasing order of their reliability
2: server_allocated=false
3: for each server $j$ in allocatable server list do
4:     if server has data partition $p$ and $R_j \geqslant \epsilon_k$ then
5:         recalculate $\lambda'_j$
6:         if $\lambda'_j \geqslant 0$ then
7:             allocate request to server $j$
8:             server_allocated=true
9:             break
10:        end if
11:    end if
12: end for
13: if server_allocated=false then
14:     ReplicaCreation(Incoming Request)
15: end if
```

Time Complexity:- $O(nk)$ where n is the number of servers and k is the number of data partitions requested.

4.3 Replica Creation

This algorithm finds the most suitable server to create a new replica for a data partition when no server containing that data partition can serve an incoming request. It first sorts the servers in non-increasing order of available service capacity (a_j).This is done so that a new data replica is created on the server with the highest available service capacity and meets the deadline requirements so that for any future requests accessing the same data partition, the chances of creating another data replica are minimized.

The algorithm goes through all the servers in order of decreasing available service capacity and finds the first server with reliability greater than the required reliability and available storage Sto_j greater than the size of data partition c_p which is to be replicated. A data replica is created on this server, and the request is allocated to it.

If no server meets the requirements, the server wakeup algorithm is called to add a new server to the server list.

Algorithm 2 Replica Creation Algorithm

```
Input: Server Workload Information, Incoming Request
Output: Allotment of Request, Updated Servers list
 1: sort servers in non-increasing order of available service capacity
 2: replica_created=false
 3: for each server $j$ in allocatable servers list do
 4:     if server has available storage and $R_j \geqslant \epsilon_k$ then
 5:         create data replica on server $j$
 6:         replica_created=true
 7:         allocate request to server $j$
 8:         break
 9:     end if
10: end for
11: if replica_created=false then
12:     ServerWakeup(Request)
13: end if
```

Time Complexity:- $O(nlog(n))$ where n is the number of servers.

4.4 Server Pairing

Each server has a list of three backup servers, which enable a faster recovery whenever the server crashes, as all the crashed server's requests are redirected to these servers. For our work, each server can only have three backup servers, and a server can act as a backup for a maximum of three servers.

For every pair of servers i and j, we define a favourability coefficient O_{ij}, which measures how favorable a server pairing is. The coefficient is defined as

$$O_{ij} = \frac{|a_i - a_j|}{Dist_{ij}} \tag{4.2}$$

where $Dist_{ij}$ is the euclidean distance between server i and j and a_j is the available service capacity for j^{th} server. The higher the value of the coefficient more favorable it is for the server i to pair with server j as it will reduce the transmission cost and decrease the time required to recover from the crash. In our algorithm, for every server, we iterate through all other servers and calculate this coefficient. Of these values, the three pairs with the highest values are selected, given that the servers in pairs are not yet paired with three other servers, and the reliability of these servers is greater.

Algorithm 3 Server Pairing Algorithm

Input: Server Information

Output: Server Backup List

```
1: for each server $i$ in the server list do
2:     for each server $j$ in the server list do
3:         if $flag_j \leqslant 2$ and $R_i \leqslant R_j$ then
4:             calculate $O_{ij}$ and put it in list $L_i$
5:         end if
6:         sort $L_i$ in decreasing order
7:         Select the top three servers from $L_i$ and add them to server list of $i$
8:         flag++ for these three servers
9:     end for
10: end for
```

Time Complexity:- $O(n^2)$ where n is the number of servers.

4.5 Workload Consolidation

The workload consolidation algorithm reduces the number of servers running, thus reducing the operational cost and making the system more energy-efficient. First, we calculate the server utilization μ_j for each server, which is defined as the ratio of the service rate to the deadline guaranteed service rate (λ_j/λ'_j). Lower the value of μ_j means the more under-loaded the server is. Using these values of μ_j, we sort the servers according to their utilization. We then iterate through this list, starting from the most underloaded server and redirect its requests to the most overloaded servers, given that the server reliability of the new server is always greater than or equal to the requested reliability.

Algorithm 4 Workload Consolidation Algorithm

Input: Server Information

Output: Updated Request Allotment

```
 1: for each server $s_j$ in the server list do
 2:    Calculate server utilization $(\lambda_j/\lambda'_j)$ and store in list L
 3:    Sort L in increasing order
 4: end for
 5: for each server $s_j$ in list L do
 6:    while server $s_j$ request queue is not empty do
 7:       for each server $s_i$ in list L in reverse order do
 8:          if $R_i \geqslant R_j$ then
 9:             Move requests from server $s_j$ to server $s_i$
10:          end if
11:          if all requests from $s_j$ rescheduled then
12:             Sleep $s_j$
13:          end if
14:       end for
15:    end while
16: end for
```

Time Complexity:- $O(n^2m)$ where n is the number of servers and m is the maximum number of requests allocated to a server.

4.6 Server Wakeup

This algorithm runs if, during replica creation, we get no server to create a new data replica. The algorithm sorts the sleeping servers according to their cost. Now it iterates through the list and selects the first server that meets the reliability requirements and contains the data partition whose replica was being created. If no such server is found, then the lowest-cost server satisfying the reliability constraints is chosen, and a data replica is created on it.

Algorithm 5 Server Wakeup Algorithm

Input: Data Partition d_p,Sleeping Servers Information

Output: Updated Request Allotment

```
1: sort the sleeping servers list in increasing order of their cost
2: server_found=false
3: for each server $s_j$ in the asleep server list do
4:    if $R_j \geqslant \epsilon_k$ and data partition $p$ present on the server then
5:       Wake up $s_j$ and allocate request.
6:       server_found=true
7:       break
8:    end if
9: end for
10: if server_found=false then
11:    ReplicaCreation(asleep server list, data partition $d_i$)
12: end if
```

Time Complexity:- $O(nlog(n))$ where n is the number of servers.

Chapter 5

Experimental Evaluations

Evaluations were done on a homegrown simulator using C++. The simulator was implemented on a Windows OS with i5-11400H, 16GB RAM and 512GB SSD.

5.1 Synthetic Data Generation

5.1.1 Servers

The grid position of the servers (x,y) is uniformly distributed on a 100x100 grid. The storage capacity and service capacity of each server and the size of data partitions were initialized randomly. The server network and storage fault rates follow a Poisson Distribution. The maintenance cost and the service cost for each server depend on its reliability, whereas the initialization cost depends on its storage and service capacities.

5.1.2 Tenants

The number of tenants was fixed at 100. The arrival of new requests from tenants follows a Poisson Distribution. 75% of the requests have a reliability requirement between [0.9-1.0]. 15% of the requests have a reliability requirement between [0.8-0.9], and the remaining 10% have a reliability requirement of less than 0.8.

5.2 Publicly Available Data

5.2.1 Servers

The server information is the same as that generated for synthetic data.

5.2.2 Tenants

Every request at a particular timestamp comes from a different tenant. The tenants are divided into three categories in the case of Netflix with the deadline and reliability requirements as (0.6,0.8), (0.75,0.6), and (0.9,0.45) respectively and in the case of Spotify, the tenants are divided into just two categories with the deadline and reliability requirements as (0.6,0.8) and (0,9,0.45) respectively.

5.2.3 Incoming Requests

Evaluations were done on publicly available data of traffic on Netflix and Spotify. The trace consisted of the Tenant Id, Data Partition requested and the Timestamp. All requests coming on a particular day were assumed to come simultaneously.

5.3 Approaches Evaluated

Following request allotment algorithms were evaluated.

5.3.1 Lowest Cost First (Greedy)

This schedules the request on the lowest-cost server that meets the deadline and reliability requirements.

5.3.2 Maximum Available Service Capacity

This schedules the request on the server with the maximum available service capacity.

5.3.3 First Come First Serve

This schedules the request on the first server in the allocatable server list that meets the deadline and reliability requirements.

5.3.4 Random

In this approach, the request is scheduled on a random server in the allocatable server list that meets the deadline and reliability requirements.

5.3.5 Maximum Reliability First

In this approach, the request is scheduled on the server with maximum reliability.

5.4 Success Rate vs Number of Requests

In Fig. 4, we can see the success rate is the highest of our proposed approach of max reliability first and the lowest for lowest cost first as it schedules the job on the lowest cost servers, which have a higher probability of crashing since the server cost is directly proportional to its reliability. The state-of-the-art approach of First Come First Serve also has a lower success rate (93.6%) than Maximum Reliability First (99.8%).

5.5 Profit per Request vs Number of Requests

In Fig. 5, the profit per request of our proposed approach of maximum reliability first always stays higher than any other approach. This is mainly because of the fact that the success rate is very high for our proposed approach, as shown in Fig.5. and thus, a higher percentage of the total incoming requests are satisfied.

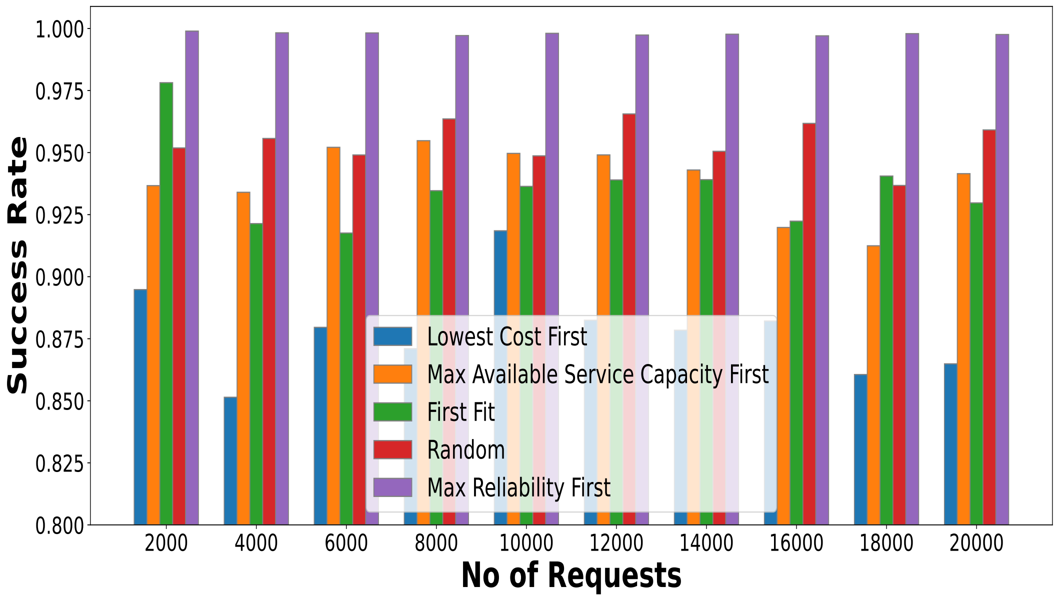

Fig. 5.1: Success Rate vs Number of Requests

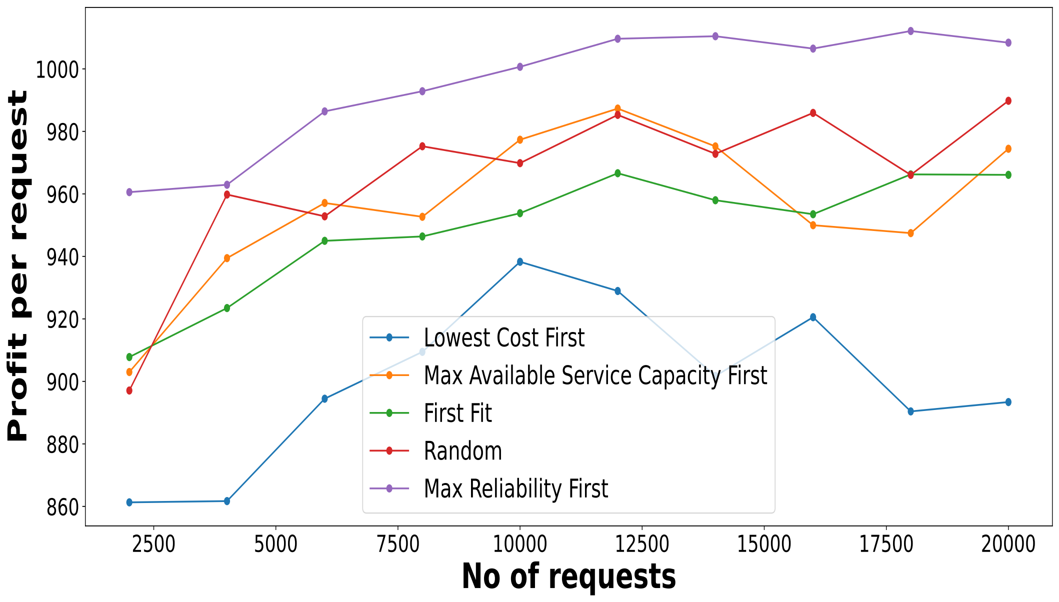

Fig. 5.2: Profit per Request vs Number of Requests

5.6 Success Rate vs Requested Reliability

In Fig. 6 we can see that our proposed algorithm performs best but the difference decreases with increasing required reliability.

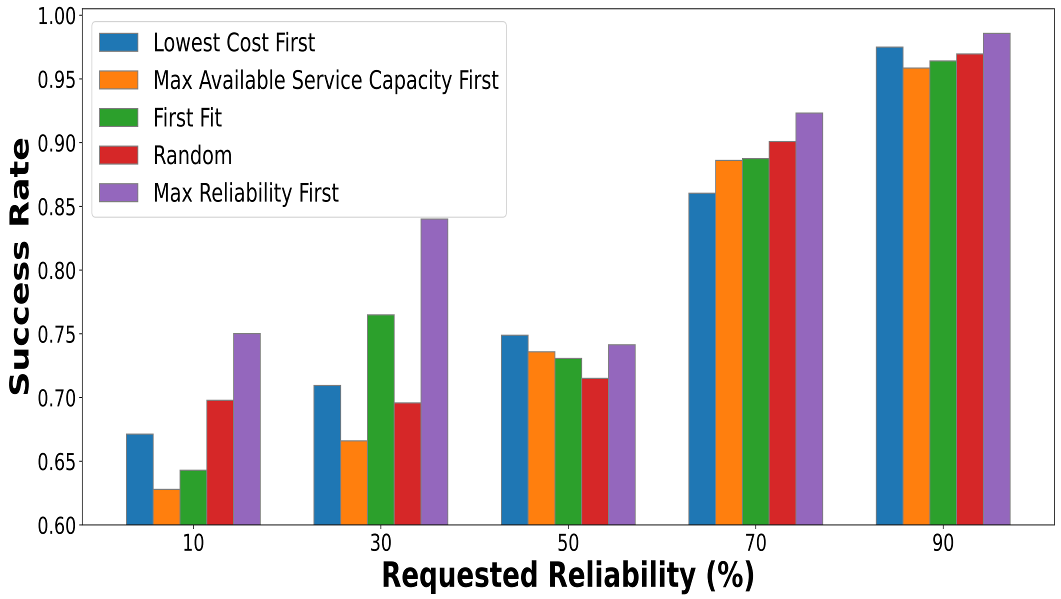

Fig. 5.3: Success Rate vs Requested Reliability

5.7 Profit Per Request vs Requested Reliability

In Fig. 7, we can see that our proposed algorithm Max Reliability First outperforms other approaches. The difference is large when the request reliability is low because, in the greedy or first fit approach, lower quality servers are allocated to the requests, which leads to a low success rate, as shown in Fig. 6, which leads to a lower profit. However, as the request reliability by the tenant increases, max reliability first continues to outperform other approaches, but the difference reduces as the other approaches also start scheduling the requests on higher-quality servers.

5.8 Success Rate vs Requested Deadline

In Fig. 8, again, it can be seen that the success rate of our proposed approach is way higher than the success rate of the State of the Art (First Fit) or any other approach.

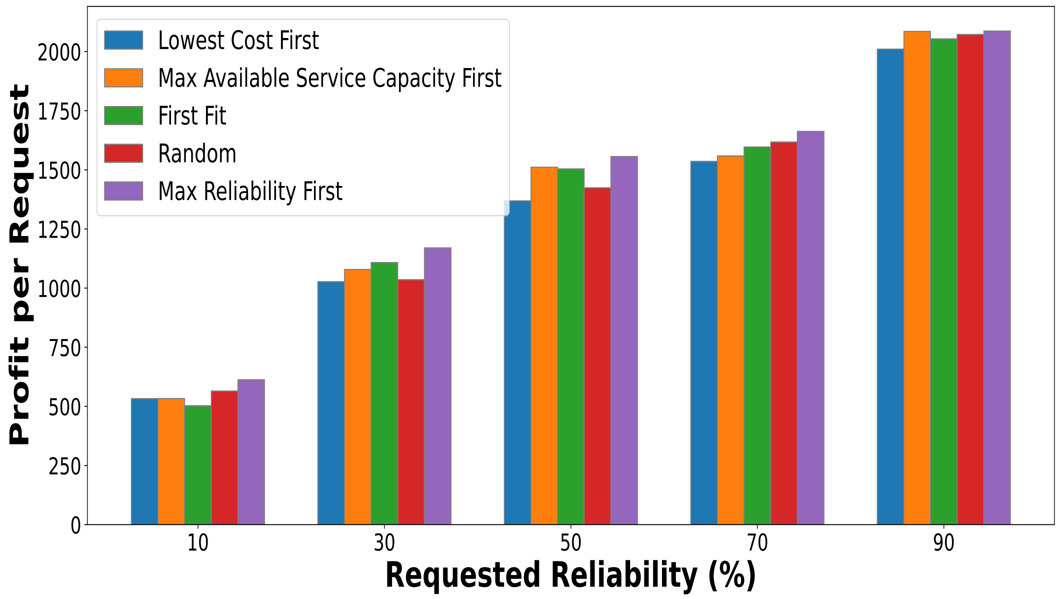

Fig. 5.4: Profit Per Request vs Requested Reliability

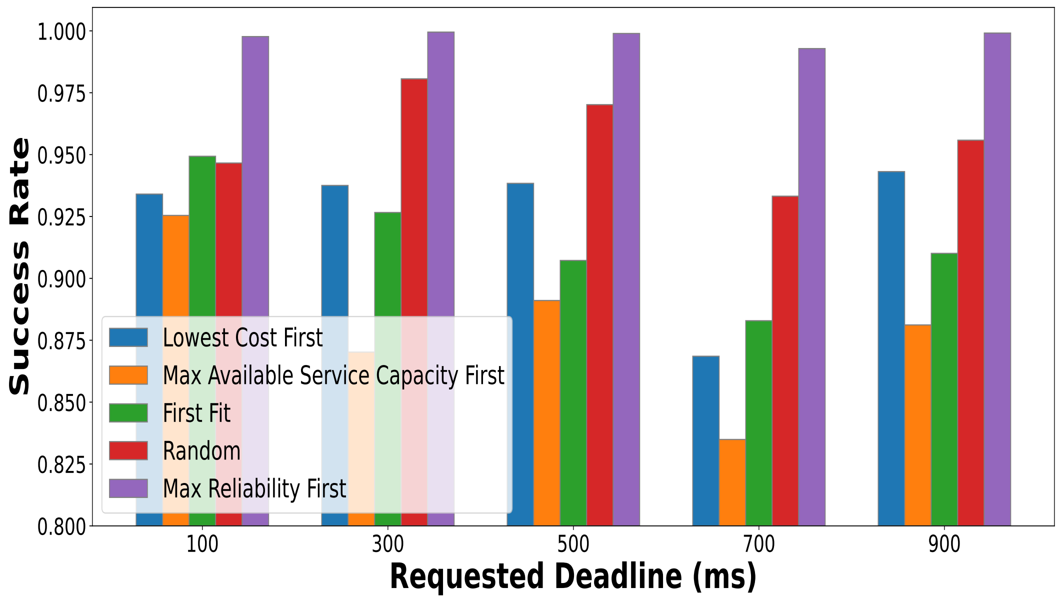

Fig. 5.5: Success Rate vs Requested Deadline

5.9 Profit per Request vs Requested Deadline

Experiments were performed by changing the average deadline of requests, and the results in Fig. 9 show that the profits decrease with an increasing deadline as the profits are inversely proportional to the requested deadline in our cost model. Max Reliability First approach outperforms all the other approaches as it successfully executes the highest number of requests (high success rate).

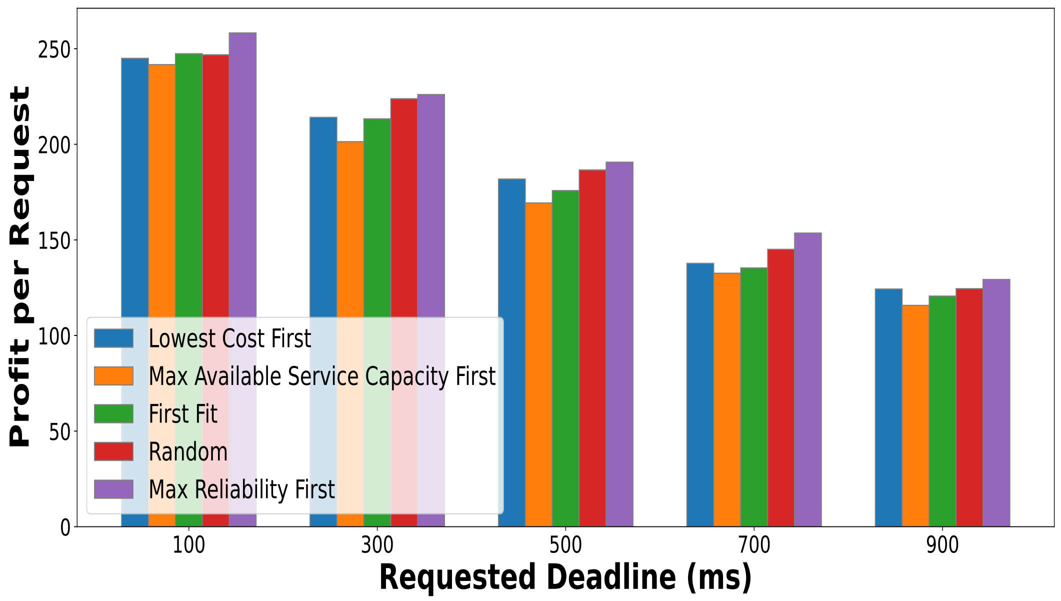

Fig. 5.6: Profit Per Request vs Request Deadline

5.10 Spotify

In Fig. 10 and Fig. 11, we have plotted the results from the publicly available trace for Spotify and it can be clearly seen that our proposed approach *Max Reliability First* outperforms all other approaches both in terms of **profit generated per request** and **percentage of requests successfully satisfied**.

5.11 Netflix

In Fig. 12 and Fig. 13, we have plotted the results from the publicly available trace for Netflix and again it can be observed that our proposed approach *Max Reliability First* does

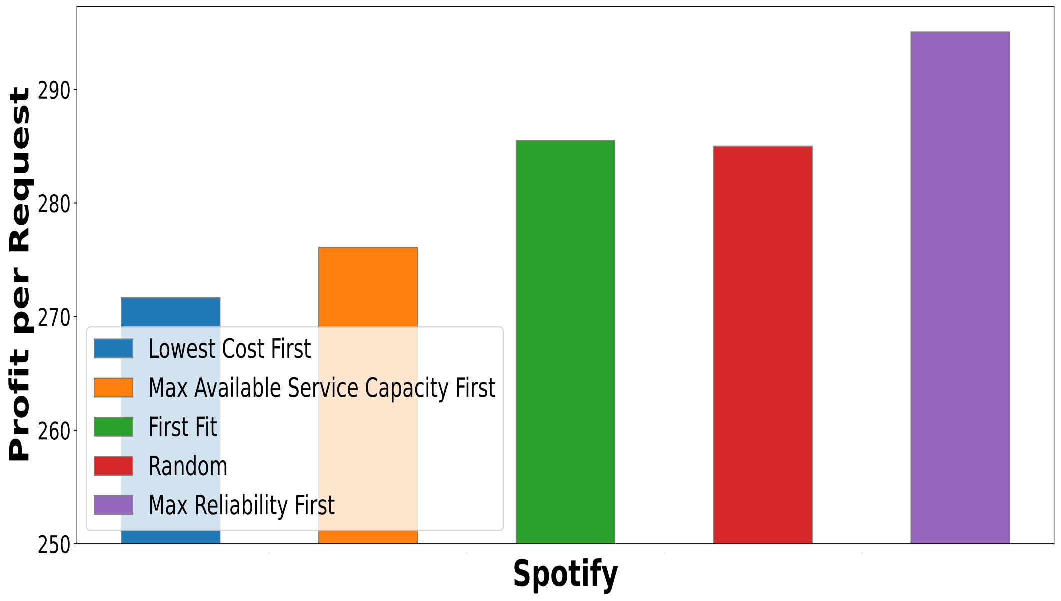

Fig. 5.7: Profit Per Request for Spotify

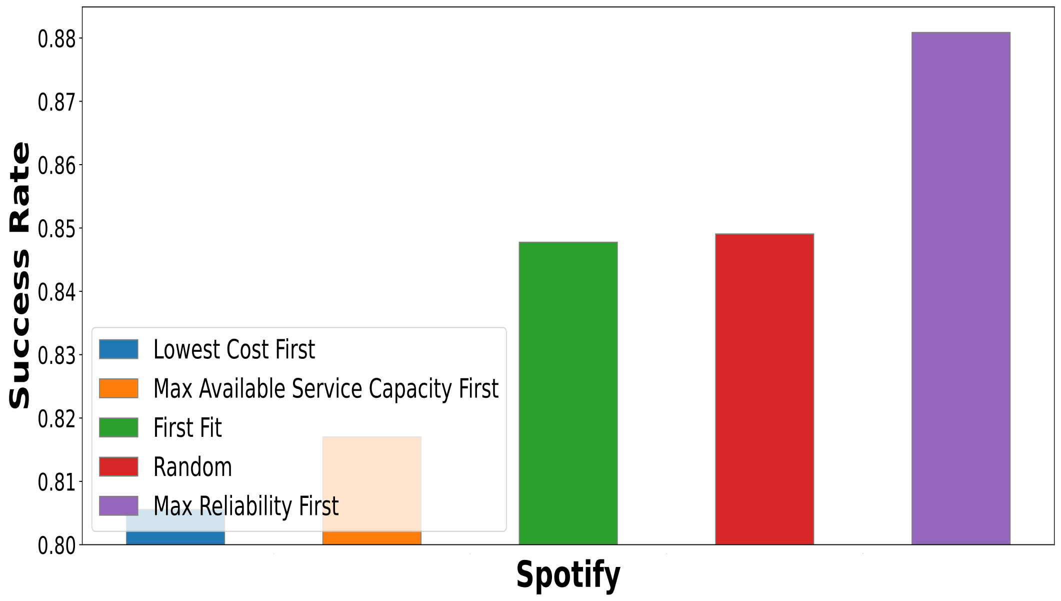

Fig. 5.8: Request Success Rate for Spotify

slightly better than all other approaches both in terms of **profit generated per request** and **percentage of requests successfully satisfied**.

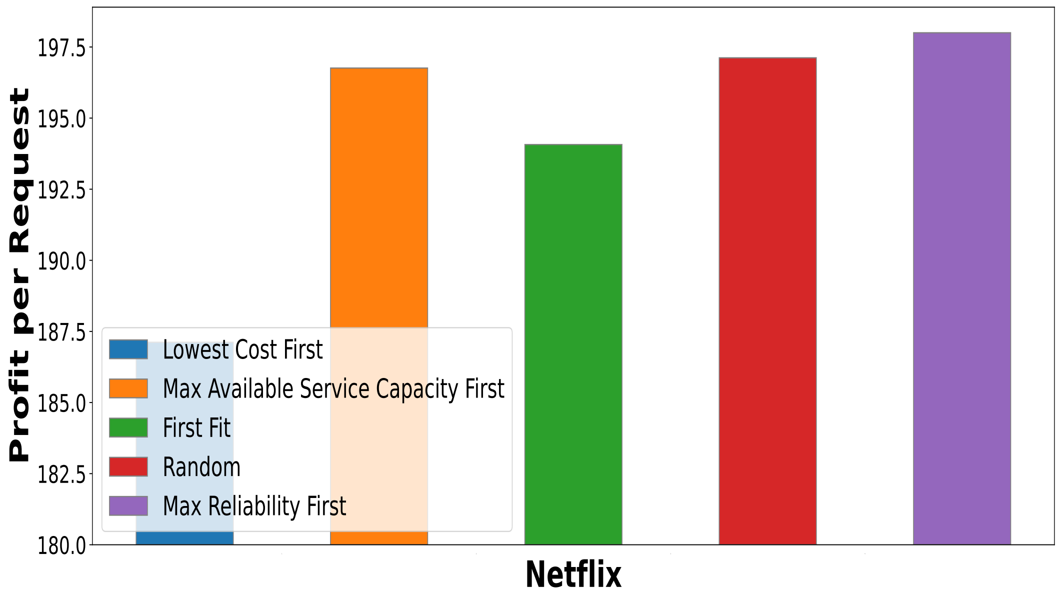

Fig. 5.9: Profit Per Request for Netflix

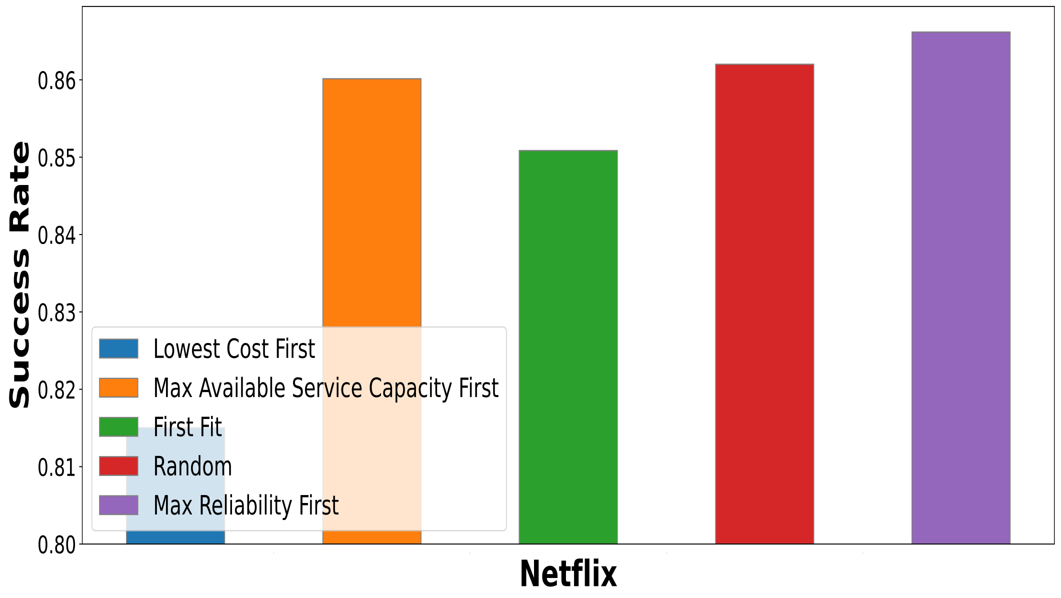

Fig. 5.10: Request Success Rate for Netflix

Chapter 6

Conclusion and Future Work

We studied eight latest research papers on cloud systems, primarily covering the domain of QoS aware scheduling, profit maximization techniques, multi-tier cloud systems, and vehicular networks. After understanding the currently active research topics, we formulated the problem statement of **maximizing the profit for the cloud providers in a reliability-aware and deadline-guaranteed cloud storage system**.

As an outcome of this research work, we have proposed a **Maximum Reliability First** request allotment approach that outperforms the State-of-the-Art first fit approach in both the key metrics of Profit per Request and Success Rate. Furthermore, to reduce operational costs and improve energy efficiency, we devised a workload consolidation algorithm. We have also proposed a server pairing algorithm that optimizes transmission time and enables faster crash recovery.

Software fault prediction using TDLSTM was included in the server reliability. Server reliability was included in the mathematical model of calculating the deadline-guaranteed service rate (λ'_j).The Head of Line Blocking issue in data servers was also solved.

Testing was done on both synthetically generated as well as publicly available data to showcase the superiority of our approach.

In the future, we aim to improve the results even more by working on a way to model our work for M/M/c queuing model.

Chapter 7

References

1. G. Liu, H. Shen, H. Wang and L. Yu, ”Towards Deadline Guaranteed Cloud Storage Services,” in IEEE Transactions on Services Computing, vol. 14, no. 3, pp. 915-929, 1 May-June 2021, doi: 10.1109/TSC.2018.2824831.

2. C. Peng, M. Kim, Z. Zhang, and H. Lei, “Dynamo: Amazon’s highly available key-value store,” in Proc. ACM SIGOPS Symp. Operating Syst. Principles, 2007, pp. 205–220.

3. A. K. Singh, X. Cui, B. Cassell, B. Wong, and K. Daudjee, “MicroFuge: A middleware approach to providing performance isolation in cloud storage systems,” in Proc. IEEE Int. Conf. Distrib. Comput. Syst., 2014, pp. 503–513.

4. A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, “Copysets: Reducing the frequency of data loss in cloud storage,” in Proc. USENIX Conf. Annu. Tech. Conf., 2013, pp. 37–48

5. E. Thereska, A. Donnelly, and D. Narayanan, “Sierra: Practical power-proportionality for data center storage,” in Proc. 6th Conf. Comput. Syst., 2011, pp. 169–182.

6. A. J. Gonzalez, B. E. Helvik, P. Tiwari, D. M. Becker, and O. J. Wittner, “GEARSHIFT: Guaranteeing availability requirements in SLAs using hybrid fault tolerance,” in Proc.

IEEE INFOCOM, 2015, pp. 1373–1381.

7. G. L. Stavrinides and H. D. Karatza, ”An energy-efficient, QoSaware and cost-effective scheduling approach for real-time workflow applications in cloud computing systems utilizing DVFS and approximate computations,” Future Generation Computer Systems, vol. 96, pp. 216-226, Jul. 2019.

8. Tang, Xiaoyong and Liu, Yi and Zeng, Zeng and Veeravalli, Bharadwaj. (2021). Service Cost Effective and Reliability Aware Job Scheduling Algorithm on Cloud Computing Systems. IEEE Transactions on Cloud Computing. PP. 1-1. 10.1109/TCC.2021.3137323.

9. P.K. Sahoo, C.K. Dehury, and B. Veeravalli, ”LVRM: On the Design of Efficient Link Based Virtual Resource Management Algorithm for Cloud Platforms,” IEEE Trans. Parallel Distributed Syst., vol. 29, no. 4, pp. 887-900, Apr. 2018

10. S. Long, W. Long, Z. Li, K. Li, Y. Xia, and Z. Tang, ”A Game-Based Approach for Cost-Aware Task Assignment With QoS Constraint in Collaborative Edge and Cloud Environments,” IEEE Trans. Parallel and Distributed Systems, vol. 32, no. 7, pp. 1629-1640, July 2021.

11. X. Zhu, J. Wang, H. Guo, D. Zhu, L. T. Yang, and L. Liu, ”Fault-Tolerant Scheduling for Real-Time Scientific Workflows with Elastic Resource Provisioning in Virtualized Clouds,” IEEE Trans. Parallel and Distributed Systems, vol. 27, no. 12, pp. 3501-3517, Dec. 2016

12. X. Li, Y. Liu, R. Kang, and L. Xiao, ”Service reliability modeling and evaluation of active-active cloud data center based on the IT infrastructure,” Microelectronics Reliability, vol. 75, pp. 271-282, Aug. 2017.

13. A. Dogan and F. Özgi̇̈er, ”Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing,” IEEE Trans. Parallel and Distributed Systems, vol. 13, no. 3, pp. 308-323, Mar. 2002.

14. X. Tang, K. Li, and G. Liao, "An effective reliability-driven the technique of allocating tasks on heterogeneous cluster systems," Cluster Computing, vol. 17, no. 4, pp. 1413-1425, Dec. 2014.