1-2017

Towards Deadline Guaranteed Cloud Storage Services

Guoxin Liu
Clemson University

Haiying Shen
Clemson University, shenh@clemson.edu

Lei Yu
Georgia Institute of Technology

Towards Deadline Guaranteed Cloud Storage Services

Guoxin Liu, Haiying Shen
Department of Electrical and Computer Engineering
Clemson University
{guoxinl, shenh}@clemson.edu

Lei Yu
School of Computer Science
Georgia Institute of Technology
leiyu@gatech.edu

***Abstract*—More and more organizations move their data and workload to commercial cloud storage systems. However, the multiplexing and sharing of the resources in a cloud storage system present unpredictable data access latency to tenants, which may make online data-intensive applications unable to satisfy their deadline requirements. Thus, it is important for cloud storage systems to provide deadline guaranteed services. In this paper, to meet a current form of service level objective (SLO) that constrains the percentage of each tenant's data access requests failing to meet its required deadline below a given threshold, we build a mathematical model to derive the upper bound of acceptable request arrival rate on each server. We then propose a Deadline Guaranteed storage service (called *DGCloud*) that incorporates three algorithms. Its deadline-aware load balancing scheme redirects requests and creates replicas to release the excess load of each server beyond the derived upper bound. Its workload consolidation algorithm tries to maximally reduce servers while still satisfying the SLO to maximize the resource utilization. Its data placement optimization algorithm re-schedules the data placement to minimize the transmission cost of data replication. Our trace-driven experiments in simulation and Amazon EC2 show the higher performance of *DGCloud* compared with previous methods in terms of deadline guarantees and system resource utilization, and the effectiveness of its individual algorithms.**

I. Introduction

Cloud storage (e.g., Amazon Dynamo and Gigaspaces) is emerging as a popular business service. Currently, more and more companies and organizations shift their data and workloads to cloud in a pay-as-you-go manner to avoid large capital expenditures in infrastructure [1]. However, cloud storage services face unpredictable performance due to the multiplexing of resources between tenants for higher utilization of servers and network infrastructure. Tenants often experience significant performance variations in data access latency [2–4]. Figure 1 shows an example of the cloud storage system with multiple tenants. Tenant T_1 operates an online social network (OSN) (e.g., Google+), T_2 operates a portal (e.g., BestBuy) and T_n operates a file hosting service (e.g., Dropbox). Each server possibly stores data from different tenants, e.g., s_2 stores replicas of data from T_1, T_2 and T_n. The front-end servers direct a data request to the servers storing corresponding data replicas. The storage sharing on s_2 from T_1, T_2 and T_n can overload s_2 and introduce a significant access latency for the tenants.

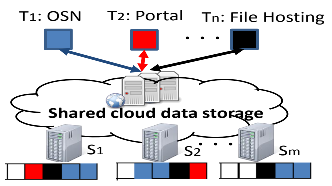


Figure 1: Multi-tenant cloud storage service.

The issue of unpredictable performance limits the types of applications that can migrate to multi-tenant clouds. Most online data-intensive applications, including web search, online retail and advertising, operate under soft real-time constraints (e.g., $\leq$ 300ms latency) for good user experience [5]. Experiments at the Amazon portal [6] demonstrated that a small increase of 100ms in webpage presentation time significantly reduces user satisfaction, and degrades sales by 1%. For a data retrieval request during the web presentation process, the typical latency budget inside a storage system is only 50-100ms [7], which requires strict deadlines for data access in data storage services.

One of the key reasons for high data access latency is server overload. The workloads caused by data requests among servers are largely skewed [8]. Requests to workload-intensive servers may be blocked due to their capacity constraints, causing unexpected data access latency and hence violations of the deadline requirements. Balancing data request workload among servers reduces the data access latency. However, current load balancing schemes [9] in the cloud do not provide deadline guarantees. The works on deadline-aware solutions for cloud focus on scheduling work flows in datacenter networks [5, 10], and little research has been devoted to the load balancing problem to supply deadline-aware service in cloud storage systems. In this paper, we use load balancing to satisfy the heterogeneous deadline requirements from multiple tenants with minimized energy and transmission cost for a commercial cloud storage service. The novelty of this work lies in providing a load balancing method that supplies deadline-aware service in cloud storage systems.

Specifically, we propose a Deadline Guaranteed storage service (called *DGCloud*) that satisfies a current form of service level objective (SLO) [11], i.e., constrains the percentage of each tenant's data access requests failing to meet its required deadline below a given threshold. This objective is non-trivial because the request distribution and replica allocation among servers are complex, and data popularity, server capacities and tenant deadline requirements are heterogeneous. To handle this challenge, based on queueing theory, we mathematically derive the **upper bound** of acceptable request arrival rate on each server to satisfy the SLOs of all tenants. We then propose three algorithms in *DGCloud*:

(1) A load balancing algorithm to ensure that that the request arrival rate on each server is no higher than its **upper bound**. This algorithm incorporates data request redirection and new replica allocation to move load from overloaded servers to underloaded servers.

(2) A workload consolidation algorithm to maximize the system utilization and energy-efficiency. It adjusts the data placement schedule determined by our load balancing algorithm, which determines the request redirection (i.e., which server serves a request) and data placement (i.e., which servers stores a data replica) to minimize the number of servers in use.

(3) A data placement optimization algorithm to minimize the transmission cost for data replication. It regards the data placement optimization problem as a minimum-weight perfect matching problem [12] by considering the new replica allocation as the transformation of data placement schedule to find the optimal solution.

The rest of the paper is organized as follows. Section II describes the system model and our problems. Sections III and IV present our deadline-aware load balancing scheme and its enhancement in detail. Section V presents the performance evaluation of our methods in simulation and on Amazon EC2. Section VI presents the related work. Section VII concludes the paper with remarks on our future work.

II. System Model and Problem Statement

A. System Model and Assumptions

We consider a heterogeneous cloud storage system consisting of M data servers, which may have different service capability and storage capacity. We assume that there are N tenants sharing the system. A data item consists of a number of data partitions. Each server may host a certain number of data partitions. Each data partition may have multiple replicas across several data servers to enhance the access efficiency and data availability [13], and each replica can be stored in any server. We assume that each data partition has at least r ($r > 1$) replicas. We suppose that the system maintains the consistency among replicas as [7], which is orthogonal to this work.

A data request from a tenant arrives at the front-end servers of the cloud storage system first, and the load-balancer assigns the request to the servers which hold the replicas of the requested data partitions. The service latency of a request is the duration between the arrival time at the front-end servers and the time when the response is returned to the front-end servers. For a data request involving multiple data partitions, its latency is the longest service latency of a partition among all the data partitions. Each tenant t_k ($1 \leq k \leq N$) has a deadline requirement for requests, denoted by d_{t_k}, which means that t_k requires service latency on its requests to be no larger than d_{t_k}. If there are multiple types of requests from t_k that have different deadlines, t_k can be treated as several different tenants with different deadline requirements. As in [14], we assume that the arrival of data requests from a tenant follows a Poisson process, where the average request rate of tenant t_k is λ_{t_k}. Each data server has a single queue for queuing arriving requests from all tenants.

B. Problem Statement

In this paper, we introduce a form of SLOs [11] with deadline guarantees for cloud storage services. That is, for any tenant t_k, no more than ϵ_{t_k} percent of all data requests have service latency longer than a given deadline d_{t_k}. Such an SLO is denoted by $(\epsilon_{t_k}, d_{t_k})$. For example, Amazon Dynamo should guarantee that no more than 0.1% of its requests have a response time exceeding 300ms [11].

In order to satisfy the SLOs of all tenants, our *deadline-aware load balancing problem* is how to dynamically create data replicas in servers and redirect the requests to replicas such that the service latency of any request from tenant t_k satisfies $(\epsilon_{t_k}, d_{t_k})$. We present our solution to this problem in Section III. In order to save system resources and improve resource utilization, we further address two optimization problems for performance enhancements as follows:

(1) To find the minimum number of data servers to support SLOs of all tenants for their requests, such that the server resource utilization is maximized, and other idle data servers can sleep to save energy cost and wake up whenever the system is overloaded [15];

(2) To find an optimal data placement for data replication with the goal to minimize the transmission cost. The transmission cost of one data replication operation is measured by the product of data size and the number of transmission hops (i.e., the number of switches in the routing path) [16, 17].

C. Overview of Our Deadline Guaranteed Cloud Storage

Our deadline guaranteed cloud storage system (*DGCloud*) incorporates three algorithms: deadline-aware load balancing algorithm (Section III), workload consolidation (Section IV-A), and data placement optimization (Section IV-B). *DGCloud* is employed in the load balancer in current commercial cloud storage systems. The load balancer needs to periodically estimate the parameters in Section III-A and check if the deadline-aware load balancing algorithm should be activated. If yes, the load balancer runs this algorithm off-line to generate the data placement schedule. Then if the system resource utilization is low, it runs the workload consolidation algorithm to reduce the number of active servers in order to increase the system resource utilization while still guaranteeing the required SLOs of tenants. After the algorithm execution, a new data placement schedule and new request rate on each data replica are generated. To minimize the transmission cost for data replication, the data placement optimization algorithm is conducted to find the optimal transformation. Finally, data is replicated based on the final data placement schedule. We introduce each algorithm of *DGCloud* in detail as below.

III. Deadline-aware Load balancing

To satisfy the deadline (SLO) requirements of all tenants, we need to balance the workload among all servers to avoid overloaded servers. In this paper, we define *overloaded servers* as the servers that cannot satisfy the SLOs of all tenants, and define *underloaded servers* as the servers that can accept more data requests under the SLO requirements of all tenants. The basic idea of our load balancing scheme is to shift requests from the most overloaded servers to the most underloaded servers. The workload shifting is achieved by redirecting requests and creating new data replicas in other servers. First, the load balancer attempts to redirect arriving requests originally targeting overloaded servers to underloaded servers. After the request redirection scheduling, if some servers still cannot satisfy tenants' SLOs, new replicas are created in other servers, which will handle part of the requests.

To design such a scheme, a critical problem is how to quantify the service capability of servers with regard to the tenants' SLOs. To handle this problem, we introduce two concepts:

Definition 1: Deadline-guaranteed request arrival rate: Deadline-guaranteed request arrival rate of a server s_n, denoted by λ'_{s_n}, is the upper bound of request arrival rates at s_n, with which the server can still satisfy the SLO requirements $(\epsilon_{t_k}, d_{t_k})$ for all the tenants served by it.

Definition 2: Available service capacity: Available service capacity of a server s_n, denoted by a_{s_n}, is defined as $a_{s_n} = \lambda'_{s_n} - \lambda_{s_n}$, where λ_{s_n} is the average request arrival rate at server s_n.

According to the above definitions, if a server has available service capacity no less than zero, it can satisfy SLOs of all tenants served by it; otherwise, it cannot. Then, the load balancing scheme moves load from overloaded serves to underloaded servers. Calculating λ'_{s_n} is not trivial. Below, we build a theoretical model to derive λ'_{s_n} in Section III-A, and then introduce our load balancing scheme in Section III-B.

A. Deadline-Guaranteed Request Arrival Rate Derivation

Our load balancing scheme requires the following parameters: i) the request arrival rate at each server s_n and at each data partition replica c_i in s_n, denoted by λ_{s_n} and $\lambda^{c_i}_{s_n}$, respectively, for computing available service capacity, ii) s_n's service rate denoted by μ_{s_n}, and iii) the deadline-guaranteed request arrival rate λ'_{s_n}. In this section, we introduce how to estimate and compute these parameters.

1) Estimating λ_{s_n}, $\lambda^{c_i}_{s_n}$ and μ_{s_n}: We estimate λ_{s_n} and μ_{s_n} from the historical records of data requests. We assume that the cloud storage system monitors each user's data request activity. In particular, the system periodically records the number of data requests on each server for each tenant during a certain period of time named *checking period* (denoted by T). T is a tradeoff between the system overhead and the sensitivity of request rate variation. A smaller T could be sensitive to the variation of request rates, leading to larger system overhead for load balancing, and vice versa. Let N_{s_n} be the number of data requests targeting s_n during T. Then, we estimate $\lambda_{s_n} = \frac{N_{s_n}}{T}$. Similarly, let $N^{c_i}_{s_n}$ denote the number of data requests targeting c_i in s_n during T, and then $\lambda^{c_i}_{s_n} = \frac{N^{c_i}_{s_n}}{T}$. When a partition is removed from s_n or a new partition is added to s_n, λ_{s_n} should be updated by subtracting or adding the request arrival rate of the corresponding partition. If a new partition c_i is added by tenant t_k, since its request arrival rate is not known yet, we use the average request arrival rate of all partitions of tenant t_k as the estimation of c_i's request arrival rate. To estimate μ_{s_n}, we profile the average service latency T_{s_n} of the server, and then calculate $\mu_{s_n} = \frac{1}{T_{s_n}}$.

2) Computing λ'_{s_n}: Next, we compute the probability of any request R_i from tenant t_k meeting the requirement of deadline requirement d_{t_k} in a checking period, denoted by $\mathbf{P}^i_{t_k}$. Based on this probability, we then derive the deadline-guaranteed request arrival rate λ'_{s_n}. According to [18], the response time of workflows follows a long tail distribution with low latency in most cases. Thus, we approximate service time with an exponentially distributed random variable. In addition, the arrival of requests follows the Poisson process [14], so each server can be modeled as M/M/1 queueing system [19].

Calculating $\mathbf{P}^i_{t_k}$. Suppose that $T^{s_n}_{t^i_k}$ is the tenant t_k's request R_i's service latency at server s_n. According to [20], the corresponding cumulative distribution of service latency of s_n, denoted as $F(t)_{s_n}$, is

$$F(t)_{s_n} = 1 - e^{-(\mu_{s_n} - \lambda_{s_n}) * t}. \tag{1}$$

For a request targeting a data partition in s_n, to guarantee the request to be finished before d_{t_k} with probability no less than $1-\epsilon_{t_k}$, we have $F(d_{t_k})_{s_n} \geq 1-\epsilon_{t_k}$. For a request targeting multiple data partitions in several servers, the data request's service latency depends on the longest service latency among all accessed servers. Then, the corresponding probability that the service latency meets the deadline requirement is

$$\mathbf{P}^i_{t_k} = p(Max\{T^{s_n}_{t^i_k}\}_{s_n \in \Re(t^i_k)} \leq d_{t_k}), \tag{2}$$

where $\Re(t^i_k)$ is a set of target data servers for the request. In Equation (2), $Max\{T^{s_n}_{t^i_k}\}_{s_n \in \Re(t^i_k)} \leq d_{t_k}$ also means that $\forall s_n \in \Re(t^i_k)$, $T^{s_n}_{t^i_k} \leq d_{t_k}$. Since $T^{s_n}_{t_k}$ is an independent variable for different servers, we also have

$$\mathbf{P}^i_{t_k} = \prod_{s_n \in \Re(t^i_k)} F(d_{t_k})_{s_n}. \tag{3}$$

Calculating λ'_{s_n}. Based on $\mathbf{P}^i_{t_k}$, we then compute λ'_{s_n} on server s_n defined in Definition 1. Suppose that p' is the probability of a request simultaneously accessing no larger than α data partitions. Given a value of p', such that $p' > \max\{1-\epsilon_{t_k} \mid t_k \in J\}$, we estimate α with CDF (Cumulative Distribution Function) of simultaneously accessing a number of partitions from previous running logs of the cloud storage system. Let p_α be the probability of any data request accessing no more than α servers and having service latency no longer than the deadline. Since $p_\alpha = \lim_{x\to\infty}\sum_{i\in[1,x]\wedge|\Re(t^i_k)|\le\alpha}(\mathbf{P}^i_{t_k} * \frac{1}{x})$, where x is the total number of requests, and $\Re(t^i_k)$ denotes the set of servers visited by request R_i. If $p_\alpha \geq 1-\epsilon_{t_k}$, we can guarantee that the probability of the service latency satisfying $\leq d_{t_k}$ is larger than $1-\epsilon_{t_k}$. Let p_l be the lower bound of $\mathbf{P}^i_{t_k}$ for any request R_i with no more than α accessed servers. We can get $p_\alpha \geq p_l * p'$, and if $p_l * p' \geq (1-\epsilon_{t_k})$, the deadline requirement of t_k is satisfied. Then, according to Equation (3), for any server of t_k, we can derive the lower bound for $F(d_{t_k})_{s_n}$ as $\sqrt[\alpha]{(1-\epsilon'_{t_k})/p'}$. Finally, we derive the upper bound of request arrival rate on s_n that satisfies the deadline requirement of t_k as:

$$\lambda'_{s_n,t_k} = \mu_{s_n} - |(ln(1-\sqrt[\alpha]{(1-\epsilon_{t_k})/p'}))/d_{t_k}|. \tag{4}$$

Given p' and α, the upper bound of λ'_{s_n} of a server is decided by d_{t_k} and ϵ_{t_k}. In order to supply deadline guaranteed service to all tenants having partitions in this server, we calculate λ'_{s_n} as the lowest value of the upper bounds of all tenants having partitions in this server. We refer to the $|(ln(1-\sqrt[\alpha]{(1-\epsilon_{t_k})/p'}))/d_{t_k}|$ of each t_k as its deadline strictness, denoted by $\mathbf{K}_{t_k}$. Then, the tenant with the highest value of $\mathbf{K}_{t_k}$ has the strictest deadline requirement, whose λ'_{s_n,t_k} equals λ'_{s_n}. Then we can get:

$$\lambda'_{s_n} = Min\{\lambda'_{s_n,t_k}\} = \mu_{s_n} - Max\{\mathbf{K}_{t_k}\}_{s_n\in\Re(t_k)}. \tag{5}$$

Here λ'_{s_n} is used to check whether s_n is overloaded or underloaded and calculate the excess load of an overloaded server in load balancing introduced in Section III-B.

B. Scheme Description

The load balancing scheme first computes each server s_n's deadline-guaranteed request arrival rate λ'_{s_n} and then available service capacity a_{s_n}. The data servers that have positive a_{s_n} values are stored into a list named *allocatable server list* in descending order of the a_{s_n}. Giving higher priority to servers with higher a_{s_n} in assigning overloaded servers' excess load helps quickly release their load. For the servers having the same a_{s_n}, they are stored in ascending order of their available storage capacity. This way, we can avoid storage fragmentation and the situation in which a server cannot utilize its available storage capacity due to lack of service capacity. The data servers with negative a_{s_n} values are stored into a list named *overloaded server list* in ascending order of the a_{s_n}. Then, starting from the beginning of the overloaded server list, i.e., the most overloaded server, we try to distribute the workload of each overloaded server in turn to the servers in the allocatable server list until its $a_{s_n} \geq 0$. In particular, to release the excess workload of an overloaded server s_n, our scheme conducts step (1) and (2) below in order.

Algorithm 1: Request Redirection Algorithm

```
1  Generate $\mathbf{C}_{s_n} = \{c_1, c_2, \ldots, c_I\}$;
2  for each $c_i$ in $\mathbf{C}_{s_n}$ do
3      for each $s_m$ in the allocatable server list do
4          if $s_m$ has replica of data partition $c_i$ then
5              Shift the requests for $c_i$ at rate $Min\{a_{s_m}, \lambda^{c_i}_{s_n}\}$ from $s_n$ to $s_m$;
6              $\lambda^{c_i}_{s_n} \leftarrow \lambda^{c_i}_{s_n} - Min\{a_{s_m}, \lambda^{c_i}_{s_n}\}$;
7              $a_{s_m} \leftarrow a_{s_m} - Min\{a_{s_m}, \lambda^{c_i}_{s_n}\}$ ;
8              $a_{s_n} \leftarrow a_{s_n} + Min\{a_{s_m}, \lambda^{c_i}_{s_n}\}$;
9          if $\lambda^{c_i}_{s_n} = 0$ then
10             break;
11         if $a_{s_n} \geq 0$ then
12             return;
```

(1) **Request Redirection**. We use $\lambda^{c_i}_{s_n}$ to denote the request arrival rate for data partition c_i at server s_n. Let $\mathbf{C}_{s_n} = \{c_1, c_2, \ldots, c_I\}$ be the set of all data partitions with $\lambda^{c_i}_{s_n} > 0$. For each data partition c_i, the scheme tries to distribute all of the requests for c_i to the allocatable servers having c_i's replica. Request redirection is conducted for data partitions in $\mathbf{C}_{s_n}$ in descending order of their request arrival rates, such that fewer partitions need to be redirected to release the overloaded server's excess load, thus expediting the process. The complete request redirection procedure for an overloaded s_n is shown in Algorithm 1.

When Algorithm 1 finishes, if $a_{s_n} > 0$, we add s_n into the allocatable server list. If $a_{s_n} < 0$, step (2) is conducted for s_n to allocate new replicas in other allocatable servers and then redirect requests to them in order to release its excess load.

Algorithm 2: New Replica Allocation Algorithm

```
1  Generate $\mathbf{C}_{s_n} = \{c_1, c_2, \ldots, c_I\}$;
2  for each $c_i$ in $\mathbf{C}_{s_n}$ do
3      while $\lambda^{c_i}_{s_n} > 0 \wedge a_{s_n} < 0$ do
4          Select next server $s_m$ in the allocatable server list;
5          if no next server in the list then
6              Add a spare data server to the allocatable server list as $s_m$;
7          if $s_m$ has enough storage to store $c_i$ then
8              Compute $a^{c_i}_{s_m}$ ;
9              if $a^{c_i}_{s_m} > 0$ then
10                 A new replica of $c_i$ is allocated in $s_m$ and the requests for $c_i$ are assigned to $s_m$ with an arrival rate of $Min\{a^{c_i}_{s_m}, \lambda^{c_i}_{s_n}\}$;
11                 $\lambda^{c_i}_{s_n}$, $a_{s_n}$ and $a_{s_m}$ are updated as Algorithm 1 ;
```

(2) **New Replica Allocation**. The new replica allocation algorithm is shown in Algorithm 2. In $\mathbf{C}_{s_n}$, data partitions with equal request arrival rates are sorted in ascending order of data size, such that the data partitions with smaller sizes

are given higher priorities for replica creation to reduce the transmission cost. For each partition c_i in $\mathbf{C}_{s_n}$, the algorithm checks the servers in the allocatable server list in order and attempts to create c_i's replica in them.

Finally, after shifting c_i's data access workload on server s_n, if $\lambda_{s_n}^{c_i} = 0$ and the number of replicas of c_i is larger than the minimum requirement (e.g., Amazon DynamoDB requires 3 copies for each data [13]), c_i can be removed from s_n. By releasing excess workload of all data servers in the overloaded server list, all data servers' available service capacities are no less than zero. Then, the SLOs of the tenants on these severs are satisfied according to the Definitions 1 and 2. Note that replicas are created only when request redirection cannot release a server's excess load. Therefore, the number of replicas created is the minimum to achieve load balance.

C. Load Balancing Activation

Since each tenant's data retrieval activity varies over time, and the data partitions are continually added and removed, the SLO of a tenant may no longer be satisfied by the servers after load balancing and the scheme needs to run again. To determine whether to activate the load balancing scheme at the end of each checking period, we introduce a measurement referred to as *system service satisfaction level* (denoted by Υ). It is defined as the minimum of the satisfaction levels of all tenants. The *satisfaction level* of tenant t_k is measured by

$$\mathbf{S}_{t_k} = \begin{cases} \mathbf{P}_{t_k}/(1-\epsilon_{t_k}) & \text{if } \mathbf{P}_{t_k} < (1-\epsilon_{t_k}) \\ 1 & otherwise \end{cases} \tag{6}$$

where $\mathbf{P}_{t_k}$ is the ratio of data requests of t_k with service latency no longer than deadline d_{t_k} during a checking period T. Then, Υ is computed by

$$\Upsilon = Min\{\mathbf{S}_{t_k}\}_{t_k \in J}. \tag{7}$$

At the end of every checking period, the load balancer calculates Υ to measure the current system service satisfaction level and compares it with a given threshold T_R. If $\Upsilon < T_R$, the load balancer activates the deadline-aware load balancing scheme to increase the system service satisfaction level. The threshold value is determined based on the commercial contracts between the cloud storage service provider and tenants, which declares the deadline miss ratio and benefit loss. We need to set a relative large value for T_R, in order to trigger the load balancing scheme before the system is highly overloaded. T_R affects the tradeoff between the tenant satisfaction level and system cost.

IV. PERFORMANCE ENHANCEMENT

A. Workload Consolidation to Maximize System Utilization

Suppose $x_{s_n}^{c_i}$ is a binary variable; if server s_n has the replica of data partition c_i, $x_{s_n}^{c_i} = 1$, otherwise, $x_{s_n}^{c_i} = 0$. We use $\rho_{s_n} = \lambda_{s_n}/\mu_{s_n} > 0$ to denote server s_n's utilization, and use M_S to represent the set of active servers whose $\rho_{s_n} > 0$. Let $\mathbf{H}_{s_n}^{c_i}$ be the access ratio of c_i's replica on s_n among all c_i's replicas. Then, for any accessed partition c_i, we have $\sum_{s_n \in M_S} \mathbf{H}_{s_n}^{c_i} * x_{s_n}^{c_i} = 1$. We define the system utilization as:

$$U_s = \sum_{s_n \in M_S} \rho_{s_n}/|M_S| \tag{8}$$

We can see that a higher U_s indicates fewer more highly utilized active servers. Besides the system utilization, we also consider the total replication cost, i.e., the transmission cost for transforming the data placement schedule, denoted by L. Thus, our problem aims to maximize U_s while minimizing L, which is is NP-hard.

To maximize system utilization, we propose a workload consolidation algorithm. After the data placement schedule is determined by the deadline-aware load balancing scheme, the workload consolidation algorithm is executed. Basically, it tries to shift all workload from the most underloaded servers to other servers to minimize the number of active servers. It follows the similar procedure as in Section III-B. All the active servers are put into two lists one in ascending order and the other in descending order of their current request arrival rates λ_{s_n}, respectively. Then, we try to shift all workload on each server in the ascending-order list to the servers in the descending-order list. The workload shifting is conducted in the order of the list since the server with the smallest request arrival rate has the largest probability to successfully release all of its workload. If all the workload of a server is released, this server is removed from the list and set to sleep mode; otherwise, the server is kept as-is and the algorithm stops.

B. Minimize Replication Cost

The workload consolidation runs off-line and finally generates a new data placement schedule in the system. Suppose $f = \{< s_{0,f}, (c_i, c_j, ..., c_k) >, ..., < s_{m,f}, (c_x, c_y, ..., c_z) >\}$, where each tuple indicates the server and the data replicas it stores, is the original data placement schedule which uses m servers and $f' = \{< s_{0,f'}, (C'_i, C'_j, ..., C'_k) >, ..., < s_{n,f'}, (C'_x, C'_y, ..., C'_z) >\}$ is the data placement schedule generated by Algorithm 2 which uses n servers. The transformation f to f' results from data replication. We aim to find the optimal data replication schedule that transforms f to f' with the minimum transmission cost.

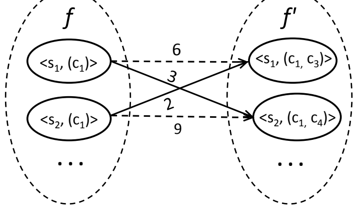


Figure 2: Optimal data placement.

We use $\mathbf{C}_{s,f}$ and $\mathbf{C}_{s,f'}$ to denote the data partition set of server s in f and f', respectively. To create a replica in a server s, we choose the closest existing replica of the data partition to s to transmit the data to s in order to reduce the transmission cost. A straightforward method to transform f to f' (which is used in our previous load balancing scheme) is to let each server in f' replicate the absent data partitions $\mathbf{C}_{s,f'} \setminus \mathbf{C}_{s,f}$. However, such a method may not be optimal in minimizing the transmission cost of all replication operations. We use an example shown in Figure 2 to explain it. Originally, both servers s_1 and s_2 have partition c_1. In the new replica placement schedule f', s_1 has $\{c_1, c_3\}$ and s_2 has $\{c_1, c_4\}$. In the straightforward method, s_1 copies c_3 with transmission cost 6, and s_2 copies c_1 with transmission cost 9. The total transmission cost is 15. However, s_1 only generates transmission cost 3 for copying c_4, and s_2 only generates transmission cost 2 for copying c_3. To reduce total transmission cost, the second replication schedule can be used instead and s_1 and s_2 are switched accordingly in the storage service if they are homogeneous (with equal service and storage capacity), and the switch will not degrade performance.

The problem is to find data replication schedule that achieves an optimal mapping between homogeneous server pairs for data replication to minimize the total transmission cost of transforming f to f'. We solve this problem by reducing it to the minimum-weight perfect matching problem for a bipartite graph which has polynomial time algorithms [12]. Considering f and f' defined above, if $m > n$, $m-n$ empty servers $\{< s_{n+1}, \emptyset >, ..., < s_m, \emptyset >\}$ are added to f' such that f and f' have the same number of servers. Similarly, empty data servers are added to f if $m < n$. Then, we consider each server in f and f' as a vertex. The edges are only constructed between each pair of homogeneous servers $< s_{i,f}, s_{j,f'} >$. The weight on an edge $< s_{i,f}, s_{j,f'} >$ is the transmission cost of transforming $s_{i,f}$ to $s_{j,f'}$. We denote the data partition set stored in $s_{i,f}$ by $\mathbf{C}_{i,f}$ and that in $s_{j,f'}$ by $\mathbf{C}_{j,f'}$. To transform server $s_{i,f}$ to server $s_{j,f'}$, the replicas in $\mathbf{C}_{j,f'} \setminus \mathbf{C}_{i,f}$ need to be created by copying from the nearest server and the replicas in $\mathbf{C}_{i,f} \setminus \mathbf{C}_{j,f'}$ need to be removed. Thus, the transmission cost for transforming $s_{i,f}$ to $s_{j,f'}$ is the transmission cost of copying $\mathbf{C}_{j,f'} \setminus \mathbf{C}_{i,f}$, denoted as L_{s_i}, and the transmission cost of one data replication operation is measured by the product of data size and the number of transmission hops. The total transmission cost of the data replication schedule is $L = \sum_{s_i \in M} L_{s_i}$. With the minimized transmission cost by *DGCloud*, considering the commercial datacenter's bandwidth today, such as 10Gb/s, 40Gb/s and 100Gb/s [21], the replication load can be ignored.

V. Performance Evaluation

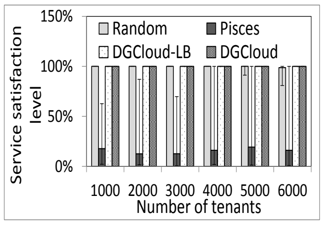
In simulation. In this section we measure the performance of *DGCloud* in a simulator. There were 30000 data servers in the cloud storage system. The storage capacity of each server was randomly chosen from {6TB, 12TB, 24TB} as [22, 23]. The topology of the storage system is a fat tree structure with two levels. The service rate μ of each server was randomly chosen from the range of [60,120], which indicates the number of requests it can serve per second. The default number of tenants was 1000. We randomly selected X data items from [24] as each tenant's data, and X was randomly chosen from the range of [100,900]. The dataset in [24] includes 4-hour data request log in a file system generated from large scale computing applications, which are also typical types of applications in Cloud. The request arrival rate on a data item was set to 10 times the data's real visit rate [24]. Each tenant's data item consists of x partitions, where x is randomly chosen from the range of [1,4]. The size of a data partition was randomly chosen from the range of [1.5,15]GB. d_{t_k} of each tenant was randomly chosen from the range of [100, 200]ms [6], and ϵ_{t_k} was set to 5%. In Equation (4), $p' = 99\%$ and $\alpha = 4$. We set the minimum number of replicas of each partition to 2, and set $T_R = 1$ and $T_U = 0.7$ by default. The service latency of a server is determined by Equation (1).

On Amazon EC2. We repeated the experiments in a real test environment consisting of 33 nodes in an availability zone of EC2's US west region [25]. We randomly chose 3 nodes as front-end servers on EC2, which generate the visits with the same rates as in [24]. The size of read/write has the same distribution as in [24]. The others are used as data servers with service rate randomly chosen from the range of [6,12], and particularly each node in EC2 simulates 10 data servers for enlarging scale. Due to the storage limitation of VMs in the testbed, the size of a partition and the storage capacity of a data server in our cloud storage system are reduced to 1/3000 of their previous settings to fit into the limited hard disk storage. The default number of tenants is 10. We measured the distance of any pair of data servers by the average ping latency.

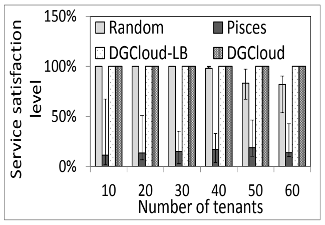
We compared *DGCloud* with a deadline unaware scheme, which places replicas greedily and sequentially to servers with constraints of server storage capacity and service rate. It does not sort the partitions and the underloaded server list in load balancing, and also does not provide deadline guarantees. This scheme is adopted by *Pisces* [26] to allocate data to different servers, so we denote it by *Pisces*. We further extended this method that additionally ensures that the request arrival rate on a server does not exceed its λ'_{s_n}, and denote it by *DGCloud-LB*. We also compared *DGCloud* with a scheme, denoted by *Random*, which randomly moves the data partitions from overloaded servers to servers that have enough storage capacity without considering service capacity.

A. Latency and Deadline Guaranteed Service

Figures 3(a) and 3(b) show the median, 5th and 95th percentiles of all tenants' satisfaction levels defined in Equation (6). It shows that the median satisfaction level follows 100%=*DGCloud*=*DGCloud-LB*>*Random*>*Pisces*. The 5th and 95th percentiles of *DGCloud-LB* and *DGCloud* also maintain at 100%. With deadline consideration, both *DGCloud* and *DGCloud-LB* provide high tenant satisfaction levels. When the number of tenants is larger than 4000, *Random* exhibits a larger variance in tenant satisfaction

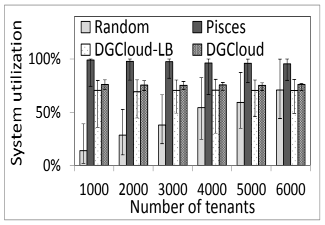


(a) Simulation

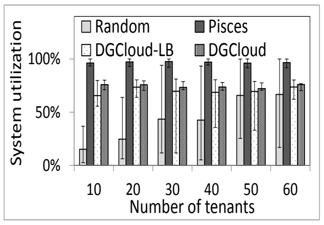


(b) Amazon EC2

Figure 3: Tenant satisfaction level vs. the number of tenants.

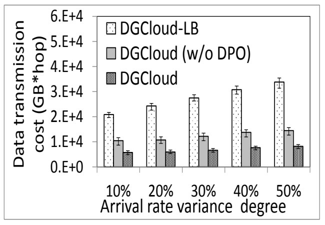


(a) Simulation

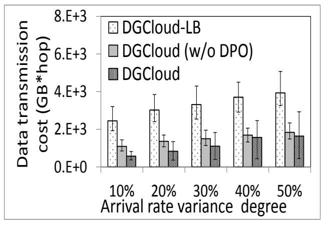


(b) Amazon EC2

Figure 4: Server utilization vs. the number of tenants.



(a) Simulation



(b) Amazon EC2

Figure 5: Minimize data transmission cost.

level. *Pisces* generates the largest variance, and the variance increases as the number of tenants increases. Both methods do not consider deadline guarantee. As the workload increases, the tenant satisfaction level decreases due to the longer service latency. As *Random* distributes workload to more servers, while *Pisces* accumulates workloads in as few servers as possible, *Pisces* generates a larger variance. Figures 3(a) and 3(b) indicate that *DGCloud* constantly supplies services with high tenant satisfaction levels even under heavy workload.

B. System Utilization

Figures 4(a) and 4(b) show the median, 5th and 95th percentiles of the server utilization defined in Section II-B. We see the system utilization follows *Random*<*Deadline*<*DGCloud*<*Pisces*. *Random* distributes workload randomly among all servers while other methods try to allocate workload to as few servers as possible, so *Random* generates the smallest median server utilization. Though *Pisces* generates the highest median server utilization, since it does not consider deadline guarantee, it produces very low system service satisfaction level as shown in Figure 3(a). It is worth noting that *DGCloud* produces a higher system utilization than *DGCloud-LB*. This is because with the workload consolidation algorithm, *DGCloud* minimizes the number of active servers. Also, when allocating excess load, *DGCloud* gives higher priority to servers with higher available service capacity, which helps increase server resource utilization and reduce the number of active servers. This is also the reason that *DGCloud* has smaller variances in server utilizations than other methods. These methods may allocate partitions with small request rates to servers with large service capacities, leading to low system utilization.

C. Transmission Cost

In the following experiments, there were 5000 tenants in simulation and 50 tenants on EC2 in the system and the average request rate of tenants was set to 1500 and 150 in simulation and EC2, respectively. We use *DGCloud* (w/o DPO) to denote *DGCloud* without the *data placement optimization* algorithm. We varied the request arrival rate of each data item, λ^c, to a value randomly chosen from $[\lambda^c * (1-\beta), \lambda^c * (1+\beta)]$, where β was varied from 10% to 50% with step size of 10%.

We measured the transmission cost in $GB{*}hop$ as defined in Section IV-B. Figures 5(a) and 5(b) show the median, 5th and 95th percentiles of data transmission cost. Each result follows *DGCloud-LB*>*DGCloud* (w/o DPO)>*DGCloud*. *DGCloud* produces lower transmission cost than *DGCloud* (w/o DPO), because the data placement optimization algorithm helps reduce the communication cost in data replication. *DGCloud-LB* does not have the data placement optimization algorithm. Thus, *DGCloud* (w/o DPO) generates lower transmission cost than *DGCloud-LB*. These results verify the low transmission cost of *DGCloud* and the effectiveness of the data placement optimization algorithm.

VI. Related Work

Recently, several works [5, 10, 18, 27] have been proposed on deadline guaranteed services in datacenters by focusing on scheduling work flows. Vamanan *et al.* [5] proposed a Deadline-aware Datacenter TCP protocol, which handles bursts of traffic by prioritizing near-deadline flows over far-deadline flows in bandwidth allocation to avoid congestion. Hong *et al.* [10] proposed a distributed flow scheduling protocol, in which all intermediate switches adopt a flow prioritization method based on a range of scheduling principles. Zats *et al.* [18] proposed a cross-layer network stack to reduce the long tail of flow completion times. Wang *et al.* [27] proposed Cake to guarantee service latency SLO and achieve high throughput using a two-level scheduling scheme of data requests within a datacenter. Corral [28] places a job and its requested data into the same rack, and different jobs into different racks to avoid resource competition to expedite job execution. Zhao *et al.* [29] proposed scheduling jobs onto geo-distributed datacenters by leveraging their different pricing policies and resource availabilities to minimize payments to cloud providers and meet job deadlines. In [30], the lower bound of work flow deadline violation probability is predicted, and a heuristic scheduling algorithm is proposed to achieve this lower bound. Though our work shares a similar goal of meeting service deadlines, the above works focus on scheduling work flows or workloads while our work focuses on a load balancing problem.

Bonvin *et al.* [31] proposed a cost-efficient self-organized data replication method to ensure the data availability by adaptively adding new storage according to node failures. Wang *et al.* [22] proposed a scalable block storage system using pipelined commit and replication techniques to improve the data access efficiency and data availability. In [32, 33], the data availability is improved by selecting data servers inside a datacenter to allocate replicas in order to reduce data loss due to simultaneous server failures. In [34], the failure rate and payment cost of different fault tolerance techniques are modeled, so that users can choose from different techniques to support a required service availability with minimized cost. To reduce the service latency of tenants, *Pisces* [26] allocates the data partitions of tenants to underloaded servers without exceeding storage and service capacity of servers. However, the above methods cannot guarantee the deadline SLOs of tenants of cloud storage systems.

VII. Conclusions

In order to provide deadline guaranteed cloud storage services, in this paper, we first propose a deadline-aware load balancing scheme. It dynamically redirects requests and creates data replicas in servers to ensure a current form of SLO, i.e., the deadlines of the requests from tenants are met with a guaranteed probability. We mathematically derive the extra load that a server needs to move out to meet the SLOs of all tenants. We further enhance our scheme with work consolidation to maximize the system resource utilization, and data placement optimization to minimize the transmission cost in data replication. The trace-driven experiments in simulation and Amazon EC2 show that our scheme provides deadline-guaranteed service while achieving high system resource utilization compared with other methods. Our enhancement methods also reduce energy cost and transmission cost of data replication. In our future work, we will design a load balancing scheme that dynamically redirect requests and replicate data to ensure SLO under a request burst.

Acknowledgements

This research was supported in part by U.S. NSF grants NSF-1404981, IIS-1354123, CNS-1254006, IBM Faculty Award 5501145 and Microsoft Research Faculty Fellowship 8300751.

References

[1] H. Stevens and C. Pettey. Gartner Says Cloud Computing Will Be As Influential As E-business. Gartner Newsroom, Online Ed., 2008.

[2] N. Yigitbasi A. Iosup and D. Epema. On the Performance Variability of Production Cloud Services. In *Proc. of CCGrid*, 2011.

[3] S. L. Garfinkel. An Evaluation of Amazons Grid Computing Services: EC2, S3 and SQS. *Technical Report TR-08-07*, 2007.

[4] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of OSDI*, 2008.

[5] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). In *Proc. of SIGCOMM*, 2012.

[6] R. Kohavl and R. Longbotham. Online Experiments: Lessons Learned., 2007. http://exp-platform.com/Documents/IEEEComputer2007OnlineExperiments.pdf.

[7] B. F. Cooper and et al. PNUTS: Yahoo!s Hosted Data Serving Platform. In *Proc. of VLDB*, 2008.

[8] G. You, S. Hwang, and N. Jain. Scalable Load Balancing in Cluster Storage Systems. In *Proc. of Middleware*, 2011.

[9] Amazon Elastic Load Balancing. http://aws.amazon.com/documentation/elasticloadbalancing/.

[10] C. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. of SIGCOMM*, 2012.

[11] C. Peng, M. Kim, Z. Zhang, and H. Lei. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of SOSP*, 2007.

[12] W. Cook and A. Rohe. Computing Minimum-Weight Perfect Matchings. *INFORMS Journal on Computing*, 1999.

[13] Amazon DynamoDB. http://aws.amazon.com/dynamodb/.

[14] D. Wu, Y. Liu, and K. W. Ross. Modeling and Analysis of Multichannel P2P Live Video Systems. *TON*, 2010.

[15] S. Seny, J. R. Lorch, R. Hughes, C. G. J. Suarez, B. Zill, W. Cordeiroz, and J. Padhye. Don't Lose Sleep Over Availability: The GreenUp Decentralized Wakeup Service. In *Proc. of NSDI*, 2012.

[16] A. Beloglazov and R. Buyya. Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. *CCPE*, 2011.

[17] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual Machine Image Distribution Network for Cloud Data Centers. In *Proc. of INFOCOM*, 2012.

[18] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proc. of SIGCOMM*, 2012.

[19] L. Kleinrock. *Queueing Systems*. Wiley-Interscience, 1975.

[20] W. J. Stewart. Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling. *Princeton University*, 2009.

[21] H. Liu, C. F. Lam, and C. Johnson. Scaling Optical Interconnects in Datacenter Networks Opportunities and Challenges for WDM. In *Proc. of HOTI*, 2010.

[22] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus scalable block store. In *Proc. of NSDI*, 2013.

[23] Apache Hadoop FileSystem and its Usage in Facebook. http://cloud.berkeley.edu/data/hdfs.pdf.

[24] CTH. http://www.cs.sandia.gov/Scalable_IO/SNL_Trace_Data/.

[25] Amazon EC2. http://aws.amazon.com/ec2/.

[26] D. Shue and M. J. Freedman. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proc. of OSDI*, 2012.

[27] A. Wang, S. Venkataraman, S. Alspaugh, R. H. Katz, and I. Stoica. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *Proc. of SoCC*, 2012.

[28] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *Proc. of SIGCOMM*, 2015.

[29] J. Zhao, H. Li, C. Wu, Z. Li, Z. Zhang, and F. C. M. Lau. Dynamic Pricing and Profit Maximization for the Cloud with Geo-Distributed Data Centers. In *Porc. of INFOCOM*, 2014.

[30] H. Wu, X. Lin, X. Liu, and Y. Zhang. Application-Level Scheduling with Deadline Constraints. In *Porc. of INFOCOM*, 2014.

[31] N. Bonvin, T. G. Papaioannou, and K. Aberer. A Self-Organized, Fault-Tolerant and Scalable Replication Scheme for Cloud Storage. In *Proc. of SoCC*, 2010.

[32] 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. of USENIX ATC*, 2013.

[33] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: Practical Power-Proportionality for Data Center Storage. In *Proc. of Eurosys*, 2011.

[34] 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. of INFOCOM*, 2015.