This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems

Deadline Guaranteed Service for Multi-Tenant Cloud Storage

Guoxin Liu, *Student Member, IEEE*, Haiying Shen*, *Senior Member, IEEE* and Haoyu Wang

Abstract—It is imperative for cloud storage systems to be able to provide deadline guaranteed services according to service level agreements (SLAs) for online services. In spite of many previous works on deadline aware solutions, most of them focus on scheduling work flows or resource reservation in datacenter networks but neglect the server overload problem in cloud storage systems that prevents providing the deadline guaranteed services. In this paper, we introduce a new form of SLAs, which enables each tenant to specify a percentage of its requests it wishes to serve within a specified deadline. We first identify the multiple objectives (i.e., traffic and latency minimization, resource utilization maximization) in developing schemes to satisfy the SLAs. To satisfy the SLAs while achieving the multi-objectives, we propose a Parallel Deadline Guaranteed (*PDG*) scheme, which schedules data reallocation (through load re-assignment and data replication) using a tree-based bottom-up parallel process. The observation from our model also motivates our deadline strictness clustered data allocation algorithm that maps tenants with the similar SLA strictness into the same server to enhance SLA guarantees. We further enhance *PDG* in supplying SLA guaranteed services through two algorithms: i) a prioritized data reallocation algorithm that deals with request arrival rate variation, and ii) an adaptive request retransmission algorithm that deals with SLA requirement variation. Our trace-driven experiments on a simulator and Amazon EC2 show the effectiveness of our schemes for guaranteeing the SLAs while achieving the multi-objectives.

Keywords: Cloud storage, Service level agreement (SLA), Deadline, Resource utilization.

1 INTRODUCTION

Cloud storage (e.g., Amazon Dynamodb [1], Amazon S3 [2] and Gigaspaces [3]) is emerging as a popular business service with the pay-as-you-go business model [4]. Instead of maintaining private clusters with vast capital expenditures, more and more enterprises shift their data workloads to the cloud. In order to supply a cost-effective service, the cloud infrastructure is transparently shared by multi-tenants in order to fully utilize cloud resources, which however leads to unpredictable performance of tenants' service. Indeed, tenants often experience significant performance variations, e.g., in service latency of data requests [5–7].

Such unpredictable performance hinders tenants from migrating their workload to cloud storage systems since the data access latency is important to their commercial business. Experiments at Amazon portal [8] demonstrated that increasing page presentation time by as little as 100ms significantly reduces user satisfaction, and degrades sales by one percent. For data retrieval in the web presentation process, the typical latency budget inside a storage system for a web request is only 50-100 ms [9].

Therefore, the unpredictable performance without the deadline guaranteed services decreases the quality of service to clients, reduces the profit of the tenants, prevents tenants from using the cloud storage systems, and hence reduces the profit of the cloud providers. Therefore, ensuring service deadline is critical for application performance guarantee of tenants. For this purpose, we argue that cloud storage systems should have service level agreements (SLAs) [10] baked into their services as other online services. In such an SLA, the cloud storage guarantees that the data requests of a tenant will be responded by a specific latency target (i.e., deadline) with no less than a pre-promised probability. The deadline and probability in an SLA are specified by the tenant in the SLA with the cloud provider based on the tenant's provided services to the clients. For example, the SLA can be specified as *99.9%* of web page presentation need to be completed within a deadline of *200-300ms* [11, 10]. A key cause for high data access latency is excess loads on cloud storage servers. Many requests from different tenants targeting a workload-intensive server may be blocked due to the server's limited service capability, which causes unexpected long latency. Therefore, to guarantee such SLAs, a challenge is how to allocate data partitions among servers (i.e., *data allocation*) under the multiplexing of tenants' workloads to avoid overloaded servers. A server is called an overloaded server if the request arrival rate on it exceeds its service capability so that it cannot supply an SLA guaranteed data access service; otherwise, it is called an underloaded server. However, previous deadline aware solutions neglect this overload problem in cloud storage systems that prevents providing the deadline guaranteed services; most of them focus on scheduling work flows or resource reservation in datacenter networks [10, 12–15]. Therefore, in this paper, we propose our Parallel Deadline Guaranteed scheme (*PDG*) to ensure the SLAs for multiple tenants in a cloud storage system.

Avoiding service overload to ensure the SLAs is a non-trivial problem. A data partition request is served by one of the servers that hold the data replicas. Each replica server has a serving ratio (i.e., the percentage of requests directed to the server) assigned by the cloud storage load balancer. We avoid service overload by *data reallocation* including the reassignment of serving ratios among replica servers and creating data replicas. This process

- * *Corresponding Author. Email: shenh@clemson.edu; Phone: (864) 656 5931; Fax: (864) 656 5910.*
- *Haiying Shen, Guoxin Liu and Haoyu Wang are with the Department of Electrical and Computer Engineering, Clemson University, Clemson, SC, 29634. E-mail: {guoxinl, shenh, haoyuw}@clemson.edu*

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems

is complex and challenging due to the heterogeneity of server capacities, tenant deadline requirements and variations of request rates of tenants.

We first formulate this data reallocation problem by identifying the multiple objectives in developing a data reallocation scheme, including traffic minimization, resource utilization maximization and scheme execution latency minimization. To solve this problem, we then build a mathematical model to measure the SLA performance under a specific data-server allocation given predicted data request workloads from tenants. The model helps to derive the upper bound of request arrival rate on each server to guarantee the SLAs. To guarantee the SLAs while achieving the multi-objectives, *PDG* schedules data reallocation (through load re-assignment and data replication), through a tree-based bottom-up parallel process in the system load balancer. The parallel process expedites the scheduling procedure; load migration between local servers reduces traffic load, and server deactivation increases resource utilization.

Our mathematical model also indicates that placing the data of two tenants with greatly different SLAs to the same server would reduce resource utilization, which motivates our deadline strictness clustered data allocation algorithm that maps tenants with the same SLA into the same server during data reallocation scheduling. We further enhance *PDG* in supplying SLA guaranteed services through two algorithms: i) a prioritized data reallocation algorithm, and ii) an adaptive request retransmission algorithm. The prioritized data reallocation algorithm handles the situation that the request rate may vary greatly over time and even experience sharp increase, which would lead to SLA violations. In this algorithm, highly overloaded servers autonomously probe nearby servers and the load balancer instantly handles highly overloaded servers without delay. The adaptive request retransmission algorithm handles the situation that tenants' SLA requirements may vary over time. In this algorithm, when a queried server does not reply in time, the frond-end server waits for a time period before retransmitting the request to another server. The waiting time is determined so that the SLA requirement can be met and the communication overhead is minimized.

We summarize our contribution below:

- Data reallocation problem formulation for SLA guarantee with multi-objectives in a multi-tenant cloud storage system.
- A mathematical model to measure the SLA performance, which gives an upper bound of the request arrival rate of each server.
- The *PDG* scheme to ensure SLA guarantee while achieving the multi-objectives.
  (1) Tree-based parallel processing;
  (2) Data reallocation scheduling;
  (3) Server deactivation.
- *PDG* enhancement algorithms to avoid SLA violations under request arrival rate and SLA requirement variation with low overhead.
  (1) Deadline strictness clustered data allocation;
  (2) Prioritized data reallocation;
  (3) Adaptive request retransmission.
- Trace-driven experiments that show the effectiveness and efficiency of our schemes in achieving deadline guarantees and the multi-objectives on both a simulator and Amazon EC2 [16].

The rest of the paper is organized as follows. Section 2 depicts the system model and the problem. Section 3 presents the prediction of the SLAs' performance in future. Based on this prediction, Section 4 and Section 5 present our parallel deadline guaranteed scheme and its enhancement in detail. Section 6 presents the performance evaluation of our methods compared with other methods. Section 7 presents the related work. Section 8 concludes the paper with remarks on our future work.

2 Problem Statement

2.1 System Model and A New SLA

We consider a heterogeneous cloud storage system consisting of N tenants and M data servers of the same kind, which may have different serving capabilities and storage capacities but supply the same storage service. As shown in Figure 1, tenant t_1 operates an online social network (OSN) (e.g., WeChat), t_2 operates a portal (e.g., Netflix) and t_N operates a file hosting service (e.g., Dropbox). A *data partition* is a unit for data storage and replication. One server may store the data partitions from different tenants and a tenant's data partitions may be stored in different servers, e.g., s_2 stores the data replicas of t_1 and t_2. Each data partition may have multiple replicas across different servers. We assume that each data partition has at least r ($r > 1$) replicas.

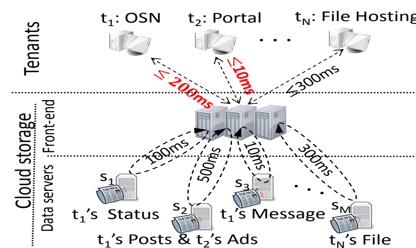


Fig. 1: Multi-tenant cloud storage service.

A data request from a tenant targets a set of data partitions in several servers, such as a News Feed request in Facebook targeting all recent posts. The request arrives at the front-end server of the cloud first, and then is redirected according to the load balancing strategy in the load balancer to servers, each of which hosts a replica of the requested data partition. The service latency of a request is the longest respond time in all target servers. As in [17], we assume that the arrival of data requests from a tenant follows a Poisson distribution, where the average request rate of tenant t_k is λ_{t_k}. Each data server has a single queue for requests from all tenants.

As shown in Figure 1, t_1's deadline is d_{t_1}=200ms and its request is served by s_1 and s_2. Though s_1's response latency is 100ms, s_2 produces 500ms response latency due to the colocation of data request intensive data partitions of t_1 and t_2 on s_2. To provide the deadline guaranteed service to tenants, we introduce a new form of SLAs for cloud storage service. That is, for any tenant t_k, no more than ϵ_{t_k} percent of all requests have service latency larger than a given deadline, denoted as d_{t_k}. We use $\mathcal{P}_{t_k}$ to denote the probability of t_k's request having service latency no longer than d_{t_k}, then the SLA

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems

is denoted by $(\epsilon_{t_k}, d_{t_k})$; $\mathcal{P}_{t_k} \geq 1 - \epsilon_{t_k}$. The probability ϵ_{t_k} and deadline d_{t_k} are specified by the tenants in their SLAs with the cloud provider. For simplicity, we only consider a common SLA for all requests from a tenant t_k, which can be easily extended for multiple SLAs for different types of requests from t_k. If there are multiple types of requests from t_k that have different SLAs, t_k can be treated as several different sub-tenants. We assume that the data request responses are independent, which means the servers work independently for data requests.

2.2 Problem Formulation

In this section, we formulate the problem of data reallocation for the SLA guarantee service in a cloud storage system. Recall that the *serving ratio* of a data partition $\mathcal{D}_i$'s replica is the percentage of requests targeting $\mathcal{D}_i$ that are served by this replica. We define *data allocation* as the allocation status for data partition placement in servers and serving ratios of data partition replicas. We use $X_{s_n}^{\mathcal{D}_i}$, a binary variable, to denote the existence of $\mathcal{D}_i$'s replica on server s_n. We use $\mathcal{H}_{s_n}^{\mathcal{D}_i}$ to denote the serving ratio of the replica of $\mathcal{D}_i$ in s_n. Then, the data allocation (denoted by f) can be presented as a set of mappings:

$$f = \{\langle s_1, (X_{s_1}^{\mathcal{D}_1} \cdot \mathcal{H}_{s_1}^{\mathcal{D}_1}, X_{s_1}^{\mathcal{D}_2} \cdot \mathcal{H}_{s_1}^{\mathcal{D}_2}, ..., X_{s_1}^{\mathcal{D}_k} \cdot \mathcal{H}_{s_1}^{\mathcal{D}_k})\rangle, ..., \langle s_n, (X_{s_n}^{\mathcal{D}_1} \cdot \mathcal{H}_{s_n}^{\mathcal{D}_1}, X_{s_n}^{\mathcal{D}_2} \cdot \mathcal{H}_{s_n}^{\mathcal{D}_2}, ..., X_{s_n}^{\mathcal{D}_k} \cdot \mathcal{H}_{s_n}^{\mathcal{D}_k})\rangle\},$$

We use $\mathcal{P}_{t_k}$ to denote the probability of t_k's request having service latency no longer than d_{t_k}. In order to ensure the SLAs, we should have $\forall t_k, \mathcal{P}_{t_k}/(1 - \epsilon_{t_k}) \geq 1$. Thus, for data partition replicas on overloaded servers, we either reduce their serving ratios or create new replicas in underloaded servers. Such data reallocation leads to a new data allocation among servers.

To avoid disrupting cloud storage service, we identify the objectives during data reallocation. To maximize resource utilization for energy-efficiency, we aim to minimize the number of servers. We name a server in use as an *active server*, and denote the whole set of active servers ($\mathcal{M}_u$) as

$$\mathcal{M}_u = \{s_n : \sum_{\mathcal{D}_i \in \mathbf{D}} X_n^{\mathcal{D}_i} \cdot \mathcal{H}_{s_n}^{\mathcal{D}_i} > 0 \wedge s_n \in \mathcal{M}\},$$

where $\mathbf{D}$ is the set of all data partitions.

Another important issue is the traffic load (replication cost through network), caused by replicating data partitions to underloaded servers. We use the product of data size ($S_{\mathcal{D}_i}$) and the number of transmission hops (i.e., switches in the routing path) between servers s_m and s_n ($I_{s_n}^{s_m}$) to measure the traffic load ($\xi_{\mathcal{D}_i}^{s_n}$) for replicating $\mathcal{D}_i$ from s_m to s_n [18, 19]; $\xi_{\mathcal{D}_i}^{s_n} = S_{\mathcal{D}_i} \cdot I_{s_n}^{s_m}$. Suppose f is the original data allocation, and f' is a new data allocation to ensure the SLAs. $s_n^f = \{\mathcal{D}_i : X_{s_n}^{\mathcal{D}_i} = 1\}$ denotes the set of data partitions contained in s_n in f. Thus, the total traffic load for a specific f' is

$$\Phi_{f'} = \sum_{s_n \in \mathcal{M}} \sum_{\mathcal{D}_i \in s_n^{f'} \wedge \mathcal{D}_i \notin s_n^f} \xi_{\mathcal{D}_i}^{s_n}.$$

We aim to find a new data allocation f', so that the traffic load that converts f to f' is minimized. The conversion from f to f' also introduces data access workload on servers. In order not to interfere in tenants' data requests, each server maintains a priority queue, where the data transmission for conversion has a lower priority than customer's requests. Also, since the conversion time is very small compared to the time for data allocation f', the effect of conversion on SLA can be ignored.

Finally, we formulate the problem of *data reallocation for deadline guarantee* as a nonlinear programming by simultaneousness achieving these two goals as:

$$\min \ (|\mathcal{M}_u| + \beta\Phi_{f'}) \tag{1}$$

$$\text{subject to} \quad \forall t_k, \mathcal{P}_{t_k}/(1 - \epsilon_{t_k}) \geq 1 \tag{2}$$

$$\sum_{s_n \in \mathcal{M}_u} X_{s_n}^{\mathcal{D}_i} \cdot \mathcal{H}_{s_n}^{\mathcal{D}_i} = 1 \quad \forall \mathcal{D}_i \in \mathbf{D} \tag{3}$$

$$\sum_{\mathcal{D}_i \in \mathbf{D}} S_{\mathcal{D}_i} \cdot X_{s_n}^{\mathcal{D}_i} \leq C_{s_n} \quad \forall s_n \in \mathcal{M} \tag{4}$$

$$\sum_{s_n \in \mathcal{M}} X_{s_n}^{\mathcal{D}_i} \geq r \quad \forall \mathcal{D}_i \in \mathbf{D} \tag{5}$$

$$X_{s_n}^{\mathcal{D}_i} \in \{0, 1\} \quad \forall s_n \in \mathcal{M}, \forall \mathcal{D}_i \in \mathbf{D} \tag{6}$$

$$0 \leq \mathcal{H}_{s_n}^{\mathcal{D}_i} \leq 1 \quad \forall s_n \in \mathcal{M}, \forall \mathcal{D}_i \in \mathbf{D} \tag{7}$$

where C_{s_n} denotes the storage capacity of s_n.

In Formula (1), β is a relative weight between the two objectives. If β is larger, the data reallocation tends to reduce the traffic load more than the number of active servers, and vice versa. Constraint (2) ensures the SLAs. Constraint (3) ensures that all data requests targeting any data partition can be successfully served. Although the storage capacity of a datacenter can be increased infinitely, the storage capacity of a server in the datacenter is still limited. Constraint (4) ensures that the storage usage cannot exceed storage capacity in any server. Constraint (5) guarantees that there are at least r replicas for each data partition in the system in order to maintain data availability. Constraint (6) guarantees that each data partition is either stored at most once or not stored in a data server. Constraint (7) guarantees that each replica's serving ratio is between 0 and 1.

Beside the two objectives, the execution time of creating f' is important to constantly maintain the SLA guarantee over time. Thus, another objective is to minimize the execution time of the data reallocation scheme.

Lemma 1. The problem of data reallocation for deadline guarantee is NP-Hard.

Proof: The *service rate* of a server is the average number of requests served by it per unit time. Suppose that all servers are homogeneous with equal service rate and storage capacity. Assume that the servers' service rate is large enough to ensure the SLAs, and we do not consider the traffic load cost, which means $\beta = 0$. Then, the deadline guarantee problem is to create a data allocation with the minimum number of active servers under storage capacity constraints of all servers, which is a bin packing problem [20]. Since the bin packing problem is NP-hard, our problem is also NP-hard. □

We then propose our heuristic PDG scheme to solve this problem. To achieve the condition in Equation (2), in Section 3, we build a mathematical model to derive the upper bound of request arrival rate at each server to satisfy Equation (2), which is named as *deadline guaranteed arrival rate*, denoted by $\lambda_{s_n}^g$. Then, in Section 4, we present PDG to constrain the request arrival rate in each server below $\lambda_{s_n}^g$ through data reallocation.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems

3 Prediction of SLA Performance

According to [15], the response time of workflows follows a long tail distribution with low latency in most cases. Thus, we assume that the service latency follows an exponential distribution. In addition, we assume that the arrival rate of requests follows the Poisson process as in [17], and each server works independently with a single queue. Therefore, each server can be modeled as an M/M/1 queuing system [21]. In an M/M/1 queuing system, there is a single server, where arrivals follow a Poisson process and the job service time follows an exponential distribution.

To calculate the parameters, we profile the average service latency T^{s_n} of a request of a server, and then calculate its service rate $\mu_{s_n} = \frac{1}{T^{s_n}}$. In each short period T, the system monitor tracks the request arrival rate of each data partition by $\lambda'_{\mathcal{D}_i} = N_{\mathcal{D}_i}/T$, where $N_{\mathcal{D}_i}$ is the number of requests on this partition. Then, we can forecast $\lambda_{\mathcal{D}_i}$ for the next period, as $\lambda_{\mathcal{D}_i} = g(\lambda'_{\mathcal{D}_i})$, where $g(\lambda)$ is a demand forecasting method as introduced in [22]. Thus, the request arrival rate of s_n:

$$\lambda_{s_n} = \sum_{\mathcal{D}_i \in \mathbf{D}} \lambda_{\mathcal{D}_i} \cdot X_{s_n}^{\mathcal{D}_i} \cdot \mathcal{H}_{s_n}^{\mathcal{D}_i}.$$

Based on the forecasted λ_{s_n} and $\lambda^g_{s_n}$ given by our mathematical model, the available service capacity of s_n is calculated by $\mu^a_{s_n} = \lambda^g_{s_n} - \lambda_{s_n}$.

s_n is an *overloaded server* if $\mu^a_{s_n} < 0$; an *underloaded server* if $\mu^a_{s_n} > 0$; and an *idle server* if $\lambda_{s_n} = 0$. PDG then conducts data reallocation to eliminate the overloaded servers. Below, we build the mathematical model to calculate $\lambda^g_{s_n}$.

Suppose $T^{s_n}_{t^i_k}$ is t_k's request i's service latency on server s_n. According to [23], the corresponding cumulative distribution function of $T^{s_n}_{t^i_k}$ for t_k's request i in an M/M/1 queuing system is:

$$F(t)_{s_n} = 1 - e^{-(\mu_{s_n} - \lambda_{s_n}) \cdot t}. \tag{8}$$

For a request i, targeting a set of data partitions in several servers, the request's service latency depends on the longest service latency among all target servers. Then, the corresponding probability that the service latency meets the deadline requirement is

$$\mathcal{P}_{t^i_k} = p(\max\{T^{s_n}_{t^i_k}\}_{s_n \in \Re(t^i_k)} \leq d_{t_k}), \tag{9}$$

where $\Re(t^i_k)$ is the set of target data servers for the request i, and each request partition is served by a server. In Equation (9), $\max\{T^{s_n}_{t^i_k}\}_{s_n \in \Re(t^i_k)} \leq d_{t_k}$ 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 can have

$$\mathcal{P}_{t^i_k} = \prod_{s_n \in \Re(t^i_k)} F(d_{t_k})_{s_n}. \tag{10}$$

Event A means that d_{t_k} is satisfied. Event B means that the data request has a target server set from ϕ_{t_k}={$\Re_1, \Re_2, ...\Re_j, ...$}. We use event B_j to mean that the target server set is $\Re_j$.

$\mathcal{P}_{t_k} = p(A \cap B) = p(B|A) \cdot p(A) = p((\cup_{\Re_j \in \phi_{t_k}} B_j)|A) \cdot p(A)$.

Assuming each B_j is independent to each other, we have

$$\mathcal{P}_{t_k} = \sum_{\Re_j \in \phi_{t_k}} p(B_j|A) \cdot p(A) = \sum_{\Re_j \in \phi_{t_k}} p(A|B_j) \cdot p(B_j).$$

According to Equation (10), the deadline satisfying probability can be rewritten as

$$\mathcal{P}_{t_k} = \sum_{\Re_j \in \phi_{t_k}} (\prod_{s_n \in \Re_j} F(d_{t_k})_{s_n}) \cdot p(B_j). \tag{11}$$

However, $|\phi_{t_k}|$ grows exponentially, so tracking all $p(B_j)$ to calculate $\mathcal{P}_{t_k}$ is impractical. Then, for tenant t_k, we define

$$b_{t_k} = \min\{F(d_{t_k})_{s_n}\}. \tag{12}$$

Thus, we can rewrite Equation (11) by combining different B_j with same cardinality as

$$\mathcal{P}_{t_k} \geq \sum_{\Re_j \in \phi_{t_k}} (b_{t_k})^{|\Re_j|} \cdot p(B_j) = \sum_{j \in [1,n]} b^j_{t_k} \cdot F_{t_k}(j), \tag{13}$$

where $F_{t_k}(j)$ is the probability density function that t_k's request targets j servers in the next period, and n is the maximum cardinality of $\Re_j$, which can be derived from the trace of the previous period. Combing Formulas (13) and (2), we get $f(b_{t_k}) = \sum_{j \in [1,n]} b_{t_k} \cdot F_{t_k}(j) = 1 - \epsilon_{t_k}$. We use x_{t_k} to denote the solution for $b_{t_k} \in (0,1)$, and call it the *supportive probability* of tenant t_k

Lemma 2. If $\forall t_k \forall s_n, s_n \in \Re_{t_k} \Rightarrow F(d_{t_k})_{s_n} \geq x_{t_k}$, then the SLAs are guaranteed.

Proof: Based on this condition and Equation (12), we can get $b_{t_k} \geq x_{t_k}$. Due to monotone increasing of $f(b_{t_k})$ when $b_{t_k} \in (0,1)$, we can get that $f(b_{t_k}) \geq f(x_{t_k}) = 1 - \epsilon_{t_k}$. According to Equation (13), for any t_k, we can get $\mathcal{P}_{t_k} \geq f(x_{t_k}) = 1 - \epsilon_{t_k}$. Thus, each t_k's SLA is ensured. □

According to Lemma 2 and Equation (8), for each tenant t_k, we can get a upper bound of λ_{s_n} to satisfy the SLAs: $\lambda_{s_n} \leq \mu_{s_n} - |ln(1 - x_{t_k})/d_{t_k}|$.

Definition 1. *We use $\mathcal{K}_{t_k}$ to denote $|ln(1 - x_{t_k})/d_{t_k}|$, and call $\mathcal{K}_{t_k}$ the* deadline strictness *of tenant t_k, which reflects the hardness of t_k's deadline requirement.*

Then, in order to ensure the SLAs, the deadline guaranteed arrival rate should satisfy:

$$\lambda^g_{s_n} = \mu_{s_n} - \max\{\mathcal{K}_{t_k} : s_n \in \Re(t_k)\}. \tag{14}$$

If $\forall s_n, \lambda_{s_n} \leq \lambda^g_{s_n}$ is satisfied in a specific data allocation, the SLAs are ensured. This is the goal in data reallocation in PDG to satisfy the SLAs.

4 Parallel Deadline Guaranteed Scheme

4.1 Overview

Figure 2 shows an overview of the parallel deadline guarantee scheme (*PDG*). It consists of three basic components and three components for enhancement. When a server's workload does not satisfy $\lambda_{s_n} \leq \lambda^g_{s_n}$, it is an overloaded server and its excess workload needs to be offloaded to other underloaded servers. The tree-based parallel processing algorithm builds servers to a logical tree. It enables the information of servers to be collected in the bottom-up manner and arranges the workload transfer from overloaded servers to underloaded servers. The data reallocation scheduling algorithm is executed in each parent node in the tree to arrange the workload transfer through load re-assignment and data replication. Finally, the server deactivation algorithm aims to minimize the number of active servers.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems

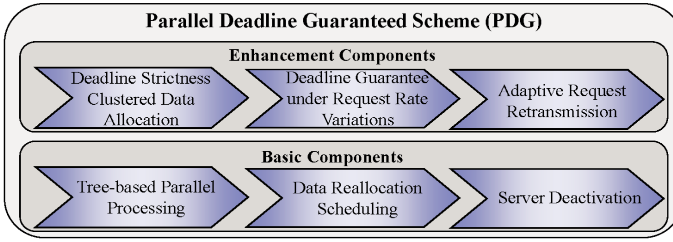


Fig. 2: Overview of PDG.

The three enhancement algorithms improve the performance of *PDG*. The deadline strictness clustered data allocation algorithm groups the tenants with similar deadline strictness and places their data partitions to the same server in order to increase tenants' deadline strictness, hence reduces the probability of SLA violations. The prioritized data reallocation algorithm enables overloaded servers to probe nearby servers to offload their excess loads without waiting for the next time period for the data reallocation scheduling based on the tree. In the adaptive request retransmission algorithm, the front-end server retransmits a request targeting an overloaded server to other servers storing the requested data partition's replicas in order to guarantee SLAs.

4.2 Tree-based Parallel Processing

The load balancer in the system conducts the SLA performance prediction in the next period and triggers data reallocation process if $\exists s_n, \lambda_{s_n} > \lambda^g_{s_n}$. The load balancer is a cluster of physical machines that cooperate to conduct the load balancing task. In order to reduce the execution time of data reallocation scheduling, we propose a concept of tree-based parallel processing. We assume a main tree topology for the servers [24] in the cloud. The load balancer abstracts a tree structure from the topology of data servers and switches (routers), with all data servers as leaves and switches (routers) as parents (Figure 3 (a)). To abstract the tree structure from any topology of data servers and switches (routers), such as a fat tree [24], PDG selects one of the core routers as the source, and finds the shortest paths from it to all data servers to build the tree structure. It then creates a number of virtual nodes (VNs). The VNs form a tree that mirrors the parent nodes in the topology tree and still uses the servers as leaves as shown in Figure 3(b). Each VN is mapped to a physical machine in the load balancer; that is, the VN's job is executed by its mapped physical machine.

The parallel data reallocation scheduling is conducted based on the tree structure in a bottom-up manner. The bottom-up process reduces the traffic load generated during the conversion to a new data allocation, by reducing the number of transmission hops for data replication. The VNs in the bottom level are responsible for collecting the following information for their children (i.e., servers): $\langle s_n, (X^{\mathcal{D}_1}_{s_1} \cdot \mathcal{H}^{\mathcal{D}_1}_{s_1}, X^{\mathcal{D}_2}_{s_1} \cdot \mathcal{H}^{\mathcal{D}_2}_{s_1}, ..., X^{\mathcal{D}_k}_{s_1} \cdot \mathcal{H}^{\mathcal{D}_k}_{s_1})\rangle$, the request arrival rate and the number of replicas of each data partition, and each t_k's supportive probability. Then, they calculate $\mu^a_{s_n}$ for their servers and classify them to overloaded, underloaded and idle servers. After that, it conducts the data reallocation scheduling, which moves data service load from overloaded servers to underloaded or idle servers. We will explain the details

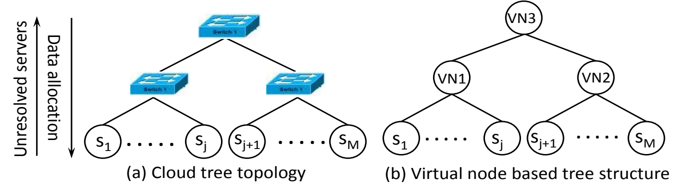


Fig. 3: Tree structure for parallel data reallocation scheduling.

of this process later. After the scheduling, if some servers are still overloaded or are still underloaded, the parent forwards the information of these servers to its parent. This process repeats until the root node finishes the scheduling process. Therefore, the scheduling for servers in the same sub-tree is conducted in parallel, which expedites the scheduling process of the data reallocation.

4.3 Data Reallocation Scheduling

Each VN groups overloaded servers, underloaded servers and idle servers into an *overloaded list* ($\mathcal{L}_o$), an *available list* ($\mathcal{L}_a$) and an *idle list* ($\mathcal{L}_i$), respectively. In the data reallocation scheduling algorithm, the lists are sorted in order to move the load from most overloaded servers to the most underloaded servers to quickly improve their service latency. In the data reallocation, each VN first conducts the serving ratio reassignment algorithm and then conducts the new replica creation algorithm to release the load of overloaded servers.

In the serving ratio reassignment algorithm, the VN fetches each s_n from $\mathcal{L}_o$ and releases its extra load $|\mu^a_{s_n}|$ to servers in $\mathcal{L}_a$ by reassigning the serving ratios on its data partitions to the same partitions' replicas in underloaded servers. In s_n, the data partitions $\mathcal{D}_i$ that have higher request rate ($\lambda^{\mathcal{D}_i}_{s_n}$) should be selected in order to more quickly release the extra load. Also, larger data partitions should be selected first because it can proactively reduce the traffic load in the subsequent data replication phase. To consider both factors, we use the harmonic mean metric $2 \cdot \lambda^{\mathcal{D}_i}_{s_n} \cdot S_{\mathcal{D}_i}/(\lambda^{\mathcal{D}_i}_{s_n} + S_{\mathcal{D}_i})$ to sort $\mathcal{D}_i$ in decreasing order. It tends to quickly release the load of overloaded servers, and reduce the traffic load in the new replica creation algorithm by avoiding replicating partitions with a larger arrival rate and data size. Then, the partial of the serving ratio $\min\{|\mu^a_{s_n}|, \mu^a_{s_m}, \lambda^{\mathcal{D}_i}_{s_n}\}$ on the replica in s_n is moved to the replica in s_m. This process repeats until s_n releases all $|\mu^a_{s_n}|$ or cannot find an underloaded server to release load.

In the new replica creation algorithm, each unsolved overloaded server in $\mathcal{L}_o$ replicates its data partitions to underloaded servers in $\mathcal{L}_a$. The data partitions with higher $\lambda^{\mathcal{D}_i}$ should be selected first to replicate since they can more quickly release the extra load. Also, the replication of $\mathcal{D}_i$ that has larger size will generate higher traffic load. To consider these two factors, we propose a metric of $\lambda^{\mathcal{D}_i}_{s_n}/S_{\mathcal{D}_i}$. The $\mathcal{D}_i$ in an overloaded server s_n are sorted in decreasing order of $\lambda^{\mathcal{D}_i}_{s_n}/S_{\mathcal{D}_i}$. It aims to quickly release the load of overloaded servers while reducing both the number and the data size of replicas. Also, with the proximity consideration, s_m replicates $\mathcal{D}_i$ from the closest server with a replica of $\mathcal{D}_i$ in the current subtree to reduce the traffic load by reducing the number of transmission hops in replication. If s_n cannot release all of its extra load, it replicates its data partitions to the servers in the idle list.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

4.4 Server Deactivation

This algorithm aims to deactivate as many servers to sleep as possible in order to maximize resource utilization while ensuring the SLAs. In each period, when the data reallocation successfully achieves the SLA guarantee, then the server deactivation can be triggered if $\sum_{s_n \in \mathcal{M}_u}(\lambda^g_{s_n} - \lambda_{s_n}) \geq \min\{\lambda_{s_n}\}_{s_n \in \mathcal{M}_u}$, i.e., the sum of the available service capacities of all active servers is no less than the minimum value among all servers' request arrival rates. In this case, the workload on the server introduced by the minimum request arrival rate may be supported by other servers.

This algorithm is conducted by the root. It first sorts active servers s_n in ascending order of $\lambda^g_{s_n}$. Then, starting from the first active server s_n, it sets its $\lambda^g_{s_n}$ to 0, and runs the data reallocation scheduling offline. If the data reallocation is successful, i.e., s_n's all workload can be offloaded to other servers while ensuring the SLAs, the root conducts the data reallocation, and deactivates s_n to sleep. Otherwise, the process terminates. Then, the system has the new data allocation satisfying the SLAs with the minimum number of active servers.

5 PDG Enhancement

5.1 Deadline Strictness Clustered Data Allocation

Different tenants have different deadline strictness (denoted by $\mathcal{K}_{t_k}$), where $\mathcal{K}_{t_k} = F^{-1}_{s_n}(d_{t_k}, \epsilon_{t_k})$. Intuitively, a tenant with a short deadline (d_{t_k}) and small exception probability (ϵ_{t_k}) has a higher $\mathcal{K}_{t_k}$, which leads to a small deadline guaranteed arrival rate ($\lambda^g_{s_n}$) given the service rate of s_n. We use $\mathcal{M}_{t_k}$ to denote the set of all servers serving the data requests from t_k. Based on Formula (14), if we place the data partitions of tenants with greatly different $\mathcal{K}_{t_k}$ in the same server, many tenants' deadline strictness are much larger than $min\{F^{-1}_{s_n}(d_{t_k}, \epsilon_{t_k})\}_{\forall t_k,\ s_n \in \mathcal{M}_{t_k}}$, which leads to low resource utilization of all servers with small guaranteed arrival rates. By isolating the services of groups of tenants having different deadline strictness, we can reduce the average $\max\{\mathcal{K}_{t_k} : s_n \in \Re(t_k)\}$ of all underloaded servers, which leads to a higher potential resource utilization.

To avoid this problem, each VN classifies all tenants into different groups (G_i) according to their $\mathcal{K}_{t_k}$:

$$t_k \in G_i \quad iff \quad \bar{\mathcal{K}}_{t_k} \in [\tau \cdot i, \tau \cdot (i+1)), \tag{15}$$

where τ is the $\mathcal{K}_{t_k}$ range of each group, and $\bar{\mathcal{K}}_{t_k}$ is the average of $\mathcal{K}_{t_k}$ in previous data reallocation operations. After classification, the VN avoids placing the data partitions of tenants from different groups in the same server. To this end, it conducts data reallocation in Section 4 separately for different groups. That is, a VN runs one data reallocation process for individual groups only with the servers for the group tenants and idle servers. Then, this algorithm increases the $\lambda^g_{s_n}$ of a server by reducing the variance of $\mathcal{K}_{t_k}$ of tenants having data partitions on it, which increases resource utilization of the system. In our future work, we will investigate the resource multiplexing among different groups while increasing resource utilization.

5.2 Deadline Guarantee under Request Rate Variations

Within a period, the request arrival rates may vary greatly over time and sometimes even experience sharp increases, which would violate the SLAs. When a heavily overloaded server waits for the periodical data reallocation from the load balancer, it may experience the overload for a relatively long time, which exacerbates the SLA violation. In order to constantly ensure the SLAs dynamically within each period, we can use the highest arrival rate in a certain previous time period as the predicted rate for the next period. However, it will lead to low resource utilization by using more servers. Thus, we propose prioritized data reallocation algorithm that quickly releases the load on the heavily overloaded servers in order to guarantee the SLAs.

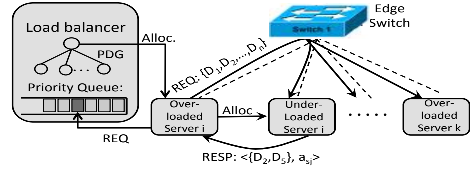


Fig. 4: Prioritized data reallocation for deadline guarantee.

The overloaded server s_n autonomously probes nearby servers to quickly release its load. s_n selects data partitions with the largest request arrival rates, sum of which is larger than $|\mu^a_{s_n}|$. It then broadcasts the information of these selected data partitions to nearby data servers. When an underloaded server, say s_m, receives this message, it responds with its available service capacity $\mu^a_{s_m}$ and the information of duplicated data partitions in it. When s_n receives the responses from its nearby servers, it conducts the serving ratio reassignment algorithm and notifies the data reallocation information to the load balancer and participating servers. If s_n is still overloaded, s_n sends a load releasing request to the load balancer. Inside the load balancer, we set a threshold $T_r \leq 0$ for the available service capacity $\mu^a_{s_n}$ of overloaded servers. When $\mu^a_{s_n} < T_r$, i.e., the overload degree is high, the request is put into a priority queue maintained by the root VN in Figure 4. Once the root VN notices the existence of such a server, it handles the server with the smallest $\mu^a_{s_n}$ using the data reallocation scheduling algorithm instantly.

5.3 Adaptive Request Retransmission

Within a period, tenant t_k may make its SLA requirement more rigid by requiring a smaller d_{t_k} or ϵ_{t_k}, so that its deadline strictness becomes more rigid. According to Formula (14), a more rigid deadline strictness of a tenant leads to a smaller deadline guaranteed arrival rate $\lambda^g_{s_n}$, which is the upper bound of request arrival rate at s_n without SLA violations. Thus, servers serving this tenant's requests may become overloaded. We can depend on the data reallocation scheduling algorithm in Section 4.3 to achieve load balance again. However, it needs to replicate data partitions from overloaded servers to underloaded servers, and introduces a certain traffic load. In order to save the traffic load, we rely on a request retransmission algorithm running on the front-end server without depending on data reallocation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems

In a request retransmission algorithm, the front-end server retransmits a request to other servers storing requested data partition's replicas in order to guarantee SLAs. This way, although some of the servers cannot supply an SLA guarantee service independently to t_k, the earliest response time among them may satisfy the SLA requirement. Once there is a response, the front-end server cancels all other redundant requests [25]. Then, the cancelled requests will not be served, and the request arrival rate of the requested data partition will not be changed.

Intuitively, we can simultaneously transmit a request of data partition $\mathcal{D}_i$ to all servers that store a replica of $\mathcal{D}_i$ in order to achieve a low response latency with a high probability. However, it generates high communication overhead due to many transmitted messages and request cancelation. To reduce the communication overhead, we can retransmit requests to servers sequentially. In *Percentile* [25], a front-end server transmits requests to servers storing the requested data partition one by one and waits a fixed percentile of the CDF of the response latencies of all the servers in the system after each request transmission until it receives a response. However, since it determines the waiting time without deadline awareness, if the percentile is high, it may not guarantee the SLA (i.e., a probability higher than $1-\epsilon_{t_k}$ to receive a response within the deadline); otherwise, it may generate high communication overhead. Also, due to the fixed waiting time, it cannot constantly supply an SLA guaranteed service when the SLA requirement varies. A challenge here is to adaptively determine the waiting time before retransmission so that the SLA requirement still can be satisfied and the communication overhead is minimized.

To tackle this challenge, we propose an *adaptive request retransmission* algorithm. In this algorithm, the waiting time, named as *adaptive waiting time* (denoted by τ_{t_k}) is specified to be the longest delay with deadline awareness, so that it can supply an SLA guaranteed service to tenant t_k and meanwhile minimize the communication overhead. That is, the setting of τ_{t_k} can ensure that the response is received by deadline d_{t_k} with a probability equal to $1-\epsilon_{t_k}$ while minimizing the communication overhead. We use $\mathcal{L}_{\mathcal{D}_i}$ to denote the list of servers (that store a replica of $\mathcal{D}_i$) ordered in ascending order of their request arrival rates with index starting from 0. The front-end sequentially sends the requests for $\mathcal{D}_i$ to the servers in $\mathcal{L}_{\mathcal{D}_i}$ one by one so that more loaded servers will be requested later. We assume that each server responds the request independently. Given the CDF of the response latency of each server s_n serving the request from tenant t_k and the t_k's SLA requirement $\langle d_{t_k}, \epsilon_{t_k}\rangle$, the probability that all servers do not respond the requests within the deadline should be equal to ϵ_{t_k}:

$$\prod_{s_n\in\mathcal{L}_{\mathcal{D}_i}} F_{s_n}(\lambda_{s_n}, d_{t_k}-\tau_{t_k}\cdot I(s_n,\mathcal{L}_{\mathcal{D}_i}))=\epsilon_{t_k} \tag{16}$$

where $I(s_n,\mathcal{L}_{\mathcal{D}_i}) \in [0, |\mathcal{L}_{\mathcal{D}_i}|-1]$ is a function that returns the index of server s_n's position in list $\mathcal{L}_{\mathcal{D}_i}$. $F_{s_n}(\lambda_{s_n}, d_{t_k}-\tau_{t_k}\cdot I(s_n,\mathcal{L}_{\mathcal{D}_i}))$ represents the probability of receiving the response from server s_n at position $I(s_n,\mathcal{L}_{\mathcal{D}_i})$ in list $\mathcal{L}_{\mathcal{D}_i}$. Since the front-end server waits for time $\tau_{t_k}\cdot I(s_n,\mathcal{L}_{\mathcal{D}_i})$ before the retransmission to s_n, s_n should respond the request before $d_{t_k}-\tau_{t_k}\cdot I(s_n,\mathcal{L}_{\mathcal{D}_i})$ in order to meet the deadline. By solving this equation, we can derive the adaptive waiting time τ_{t_k} that satisfies the rigid SLA requirement of tenant t_k and also saves the communication overhead maximally.

Based on the adaptive determination of τ_{t_k}, we then present the adaptive request retransmission algorithm. Starting from the first server s_n in $\mathcal{L}_{\mathcal{D}_i}$, the front-end server waits for an adaptive waiting time τ_{t_k}, after transmitting a request from tenant t_k to s_n. If there is a response during the waiting time, all requests not responded yet are canceled and the process is terminated; otherwise, the front-end server sends the request to the next server in $\mathcal{L}_{\mathcal{D}_i}$.

6 Performance Evaluation

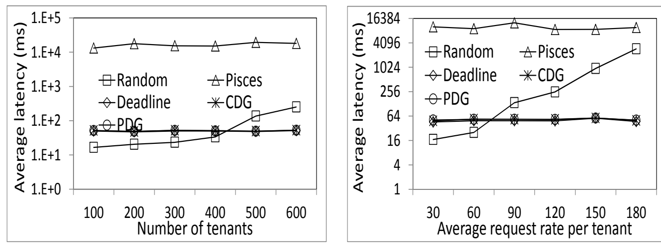
In simulation. We conducted a trace-driven simulation on both a simulator and Amazon EC2 [16] to evaluate the performance of *PDG* in comparison with other methods. In the simulation, there were 3000 data servers, each of which has a storage capacity randomly chosen from {6TB, 12TB, 24TB} [26, 27]. Each ten servers were simulated by one node in the Palmetto Cluster [28], which has 771 8-core nodes. The topology of the storage system is a typical fat tree with three levels [24]. In each rack, there were 40 servers, and each aggregation switch linked to five edges. In the experiments, each server was modeled as an M/M/1 queuing system [29, 21]. In an M/M/1 queuing system, there is a single server, where the arrivals of requests follow a Poisson process and the job service time follows an exponential distribution. According to [23], the corresponding inverse function of the CDF of the response latency distribution is $F_{s_n}^{-1}(d,\epsilon)=\mu_{s_n}-|ln(1-\epsilon)/d|$. The service rate μ of each server was randomly chosen from [80,100]. According to Equation (14), we can derive $\lambda_{s_n}^g$.

The default number of tenants was 500. For each tenant, the number of its data partitions was randomly chosen from [100,900]. Each partition has the size randomly chosen from [6GB,36GB], and the request arrival rate in the Poisson process was generated as 10 times of a randomly selected file's visit rate from the CTH trace [30] as in Section 5.3. For each tenant's SLA, d_{t_k} was randomly chosen from [100ms, 200ms] [31], and ϵ_{t_k} was set to 5% referring to 95th-percentile pricing [32]. We set the minimum number of replicas of each partition as 2. Initially, each replica of a partition has the same serving ratio.

On Amazon EC2. We repeated the experiments in simulation in a real-world environment consisting of 30 nodes in an availability zone of EC2's US west region [16]. We chose all nodes as front-end servers on EC2, and the request arrival rate of each data partition requested has the same visit rate as in [30]. Each node in EC2 simulates 10 data servers in order to enlarge system scale, and each data server has a service rate randomly chosen from $[8, 10]$. Due to the local storage limitation of VMs in EC2, the partition size and server storage capacity were set to $1/3000$ of the settings in Section 6. The default number of tenants was 10. We measured the distance of any pair of data servers by the average ping latency, based on which we mapped all simulated

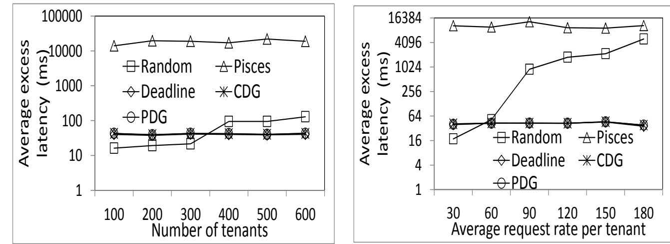
1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems



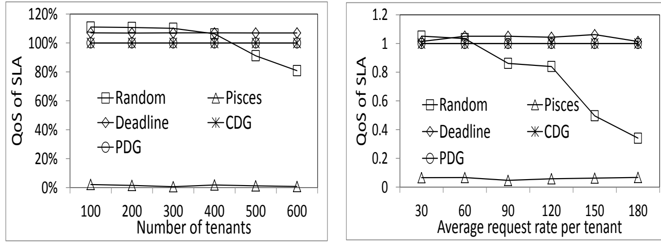
(a) In simulation (b) On Amazon EC2

Fig. 5: Average latency.



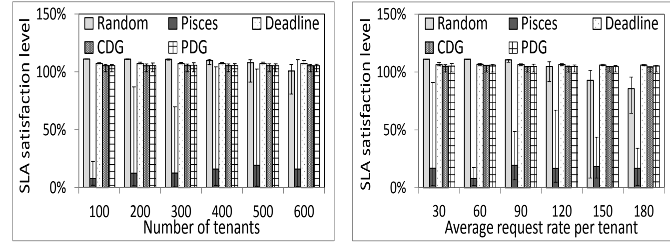
(a) In simulation (b) On Amazon EC2

Fig. 6: Average excess latency.



(a) In simulation (b) On Amazon EC2

Fig. 7: QoS of SLA.



(a) In simulation (b) On Amazon EC2

Fig. 8: SLA satisfactory level.

storage servers into a typical three layer fat-tree with 20 severs in a rack. According to the setting, the average request rate per tenant is around 30 requests per second. In all experiments, we enlarged the request arrival rates of each partition by one to six times. Thus, the average request rate per tenant was increased from 30 to 180 requests per second with 30 increase at each step. The default average request rate per tenant was set to 120.

We compared *PDG* with *CDG*, which runs *PDG* in a centralized manner without the tree structure. We also compared *PDG* with a deadline unaware strategy, which places replicas greedily and sequentially to servers with constraints of each server's storage capacity and service rate. It is adopted by [33] to allocate data partitions to different servers, so we denoted it by *Pisces*. In order to compare the performance of our strategies in Section 4.3, we provided an enhanced *Pisces* strategy (named as *Deadline*) for comparison, which additionally ensures that the request arrival rate on a server cannot exceed its deadline guaranteed arrival rate $\lambda^g_{s_n}$. We also added another comparison method (denoted by *Random*), which randomly places data replicas to servers that have enough storage capacity. We set the SLA prediction period to one hour. We conducted each experiment 10 times with an hour running and report the average experimental results.

6.1 Performance of Deadline Guarantee

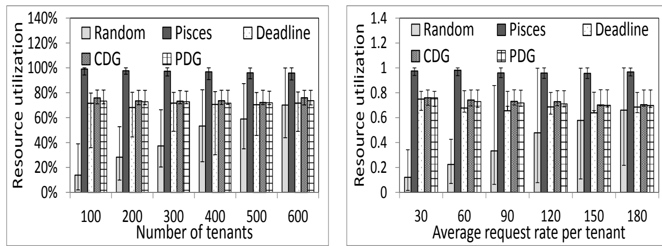
In this experiment, each tenant stores all of its data partition replicas randomly into the system. Figures 5(a) and 5(b) show the average latency of requests of all tenants versus the number of tenants in simulation and on testbed, respectively. They show that *Pisces*>*PDG*≈*CDG*≈*Deadline*>*Random* when there are no larger than 500 tenants. With 600 tenants, the average latency of *Random* is larger than all three methods with deadline awareness. With fewer tenants, *Random* uses all servers, so the load of a server is the smallest. When the system has a heavy data request load from 600 tenants, *Random* produces unbalanced utilization among servers, and some overloaded servers have much larger latency than the deadlines. Since *PDG*, *CDG* and *Deadline* supply deadline guaranteed services, they produce similar average latencies. *Pisces* does not consider deadline, and distributes more load on a server, which leads to a much longer service latency than all other methods. The figures also show that the average latency of *Random* increases proportionally to the number of tenants, while other methods have nearly stable average latency. The methods except *Random* constrain the request arrival rate on a server below $\lambda^g_{s_n}$, and try to fully utilize the active servers. Thus, their expected load on an active server is nearly stable as the number of tenants increases. In *Random*, more replicas of partitions are allocated to a server, which leads to an increasing average latency as the number of tenants increases. Figures 5(a) and 5(b) indicate that *PDG* and *CDG* can supply deadline guaranteed service with stable and low average latency to tenants even under a heavy system load.

We also evaluate the *excess latency* of a data request, which is defined as the extra service latency time beyond the deadline for a request. Figures 6(a) and 6(b) show the average excess latency of all requests. They show a similar curve and relationship for all methods as Figure 5(a) due to the same reasons. A noteworthy difference is that unlike the average latency, the average excess latency of *Random* is larger than *Deadline*, *PDG* and *CDG* when the number of tenants exceeds 300 or 60 due to its neglect of SLAs in simulation and on testbed, respectively. Also, *Random* generates an average excessive latency larger than 100ms with 400 or more tenants, which will degrade the sale of customers [8] and prevent them to shift workload to cloud storage systems. Figures 6(a) and 6(b) also indicate that *PDG* and *CDG* can provide a lower average excess latency, which means a lower average excess latency when the SLA is violated.

We define *QoS of SLA* as $\min\{\forall t_k, \mathcal{P}_{t_k}/(1-\epsilon_{t_k})\}$. Figures 7(a) and 7(b) show the QoS of each method. They show that all *Deadline*, *CDG* and *PDG* can supply a deadline-aware service with a QoS slightly larger than 1, which means SLAs of all tenants are satisfied. Due

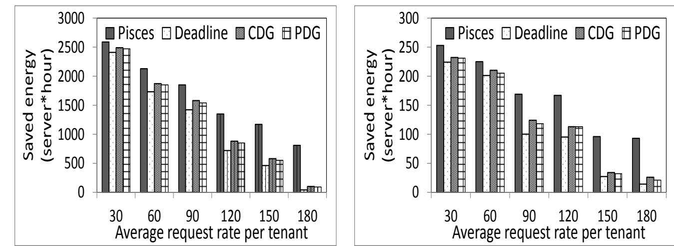
1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems



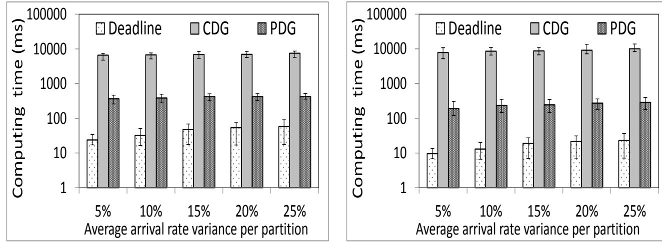
(a) In simulation (b) On Amazon EC2

Fig. 9: Resource utilization.



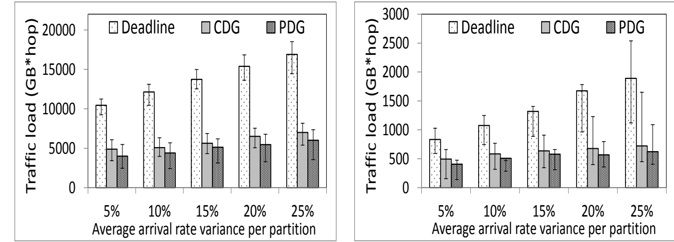
(a) In simulation (b) On Amazon EC2

Fig. 10: Saved energy.



(a) In simulation (b) On Amazon EC2

Fig. 11: Computing time.



(a) In simulation (b) On Amazon EC2

Fig. 12: Traffic load.

to the worst performance on overloaded servers for the same reason as in Figure 5(a), *Random* cannot supply a deadline guaranteed service when the request load is heavy. Its QoS is reduced to 80% when there are 600 tenants. Since the QoS is very important for tenants operating web applications, this is a big obstacle for customers to shift their workload to cloud storage systems. Also, *Random* always uses all servers even when the number of tenants is small. Since the servers can supply SLA guaranteed services to 600 tenants as shown in *PDG* and *CDG*, *Random* wastes at least 83%, 67% and 50% resources to supply a deadline guaranteed service when there are 100, 200 and 300 tenants, respectively in simulation. Also, due to the same reason as Figure 5(a), *Pisces* has a much worse QoS than other methods. Although *Deadline* can supply a deadline-aware service, its QoS is larger than *PDG*'s and *CDG*'s. That is because it uses more servers to supply the deadline-aware service, which means *Deadline* wastes system resources to supply over-satisfied services. Figures 7(a) and 7(b) indicate that *PDG* and *CDG* achieve QoS of SLA larger than and closer to 100%, respectively, which are higher than those of all other methods.

Figures 8(a) and 8(b) show the median, 5th and 95th percentiles of all tenants' SLA satisfaction level, defined as $\mathcal{P}_{t_k}/(1-\epsilon_{t_k})$, in simulation and on testbed, respectively. Due to the same reason as Figure 7(a), the median satisfaction level follows *Random*>*Deadline*≈*PDG*≈*CDG*≈1>*Pisces*, when the number of tenants is no larger than 500 (90), and *Random* supplies a worse performance than *PDG*, *CDG* and *Deadline* in simulation (on testbed). *Random* exhibits larger variances between the 5th and 95th percentiles than the three deadline-aware methods when the request load is heavy. They indicate that *Random* supplies unfair deadline guaranteed service among all tenants with different SLAs. Also, *Pisces* produces the largest variance, because the requests from tenants with looser deadline requirements can be more easily satisfied. Also, *Deadline* can supply SLA guaranteed services for all tenants, but it uses more system resources than *PDG* and *CDG* due to the same reasons as in Figure 7(a). Figures 8(a) and 8(b) indicate that *PDG* and *CDG* can constantly supply SLAs guaranteed services using less system resources.

6.2 Performance for Multiple Objectives

In this section, we measure the performance of all systems in achieving the multi-objectives including resource utilization maximization, traffic load and scheme execution latency minimization. Figures 9(a) and 9(b) show the median, the 5th and 95th percentiles of the server resource utilization, calculated by $\rho_{s_n} = \lambda_{s_n}/\mu_{s_n}$. The median server utilization follows *Random*<*Deadline*< *PDG*<*CDG*<*Pisces*. *Random* generates the smallest utilization by using all servers, and *Pisces* generates the highest utilization by fully utilizing the service rates of servers with the greedy strategy, but at the cost of a very low QoS as shown in Figure 7(a). *PDG* and *CDG* produce higher resource utilization than *Deadline*. *PDG* and *CDG* fully utilize available service capacities of active servers by serving ratio reassignment and data replication. When *Deadline* tries to allocate a partition replica with a request arrival rate, it choose a server that must be able to support this request arrival rate without considering the distribution of the load among several servers, thus leading to lower server utilization. Also, by balancing the load between most overloaded and underloaded servers, *PDG* and *CDG* have smaller variances between the 5th and 95th percentile of resource utilization than *Deadline*. *CDG* has higher resource utilization than *PDG* (1.3% more on average). This is because in *CDG*, the centralized load balancer can deactivate a server with the highest service rate among all sleeping servers, which leads to fewer active servers to support the deadline-awareness service. Thus, *CDG* has higher utilization than *PDG*. The experimental results indicate that *PDG* can achieve comparable resource utilization as *CDG*, and both of them have higher and more balanced resource utilization than *Deadline*, which also offers a deadline-aware service.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems

As in [34], we measured the energy savings in $server \times hour$ by counting the sleeping time of all servers. Since *Random* uses all servers without energy consideration, we only measured the performance of all other methods. Figures 10(a) and 10(b) show that saved energy follows *Deadline*<*PDG*<*CDG*<*Pisces* due to the same reason as in Figure 9(a). *PDG* can save up to 95 $server * hour$ more than *Deadline* on average. The figures indicate that both *PDG* and *CDG* can save more energy than *Deadline*.

Even though *CDG* saves more energy than *PDG*, *CDG* uses much more computing time and introduces more traffic load than *PDG*. In order to measure these overheads, we set the request arrival rate of each partition, λ_p, to a value randomly chosen from $[\lambda_p \cdot (1 - 2x), \lambda_p \cdot (1 + 2x)]$, where x is the average arrival rate variance, and is increased from 5% to 25% by 5% at each step. *Random* and *Pisces* cannot supply deadline guaranteed services, and they do not schedule data reallocation after the request arrival rate varying. Therefore, we compare the performance of *PDG* with *CDG* and *Deadline*. Figures 11(a) and 11(b) show the median, the 5th and 95th percentiles of algorithm computing time. We see that the computing time and its variance follows *Deadline*<*PDG*<*CDG*. This is because the data reallocation algorithm in both *PDG* and *CDG* has higher time complexity than a greedy algorithm in *Deadline*. In *PDG*, the tree-based parallel processing shortens the computing time. Thus, *PDG* only takes around 5.8% computing time of *CDG*.

We measured the traffic load in $GB \cdot hop$ as introduced in Section 2.2. Figures 12(a) and 12(b) show the median, the 5th and 95th percentiles of traffic load, which follows *PDG*<*CDG*<*Deadline*. Since *PDG* and *CDG* try to reduce traffic load in data reallocation, they produce lower traffic load than *Deadline*. *PDG* has lower traffic load than *CDG* because *PDG* has lower expected transmission path length than *CDG* by resolving the overloaded servers locally first. The figures indicate that *PDG* introduces the lowest traffic load to the system, which produces the least interruption to the cloud data storage service. We also measured the conversion time of data reallocation schedule, as the time that all servers finish conversion to the new data allocation. Figure 13 shows the average conversion time of all systems, which shows a similar curve and relationship for all methods as Figure 12(a) due to the same reason. It indicates that *PDG* achieves the lowest conversion time, no longer than 5 seconds, causing the fewest effects on the SLA. All Figures 12 and 13 show that *PDG* achieves a better performance in minimizing the traffic load.

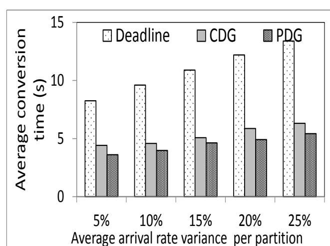


Fig. 13: Conversion time.

6.3 Performance of Deadline Guarantee Enhancement

In this section, we present the performance of each of the PDG enhancement algorithms individually.

6.3.1 Performance of Deadline Strictness Clustered Data Allocation

In order to make the deadline strictness of tenants having data on the same server vary greatly, different from the scenario in Section 6.1, in this experiment, tenants add data replicas to servers in turn and each tenant adds one data replica to a server at each time. Since this method does not affect the performance of *Random* and *Pisces*, which do not consider tenant deadline requirements, we compared the performance of *Deadline*, and *PDG* with and without the deadline strictness clustered data allocation algorithm, denoted by *PDG* (w/ c) and *PDG* (w/o c), respectively. *PDG* (w/ c) groups all tenants into 5 different clusters.

Figures 14(a) and 14(b) shows the median, the 5th and 95th percentiles of the server resource utilization versus the number of tenants. Due to the same reasons as in Figure 9(a), *Deadline* generates lower resource utilization than *PDG*. Also, *PDG* (w/ c) generates higher utilization than *PDG* (w/o c). This is because the data partitions with strict SLAs increase the deadline strictness requirement of data partitions on servers, and hence reduces $\lambda^g_{s_n}$ of the servers, and then decreases the resource utilization. Thus, *PDG* (w/o c) supplies overqualified service of higher $\mathcal{P}_{t_k}$ to the tenants with lower deadline strictness, while *PDG* (w/ c) isolates the deadline service performance of tenants with different deadline strictness. Without supplying overqualified service, *PDG* (w/ c) produces higher utilization than *PDG* (w/o c). Figure 14(a) indicates that the deadline strictness clustered data allocation algorithm can help *PDG* achieve higher resource utilization when the tenant deadline strictness varies greatly. By rationally utilizing the system resources, *PDG* (w/ c) can still supply a deadline guaranteed service when there are 600 tenants, while others cannot. The experimental results indicate that *PDG* can achieve higher resource utilization with the deadline strictness clustered data allocation algorithm. Figures 15(a) and 15(b) show the extra energy saved by *PDG* (w/o c) and *PDG* (w/ c) versus the number of tenants. *PDG* (w/ c) saves more energy than *PDG* (w/o c). These results indicate that the deadline strictness classification strategy is effective in helping *PDG* maximize the resource utilization and minimize the number of active servers.

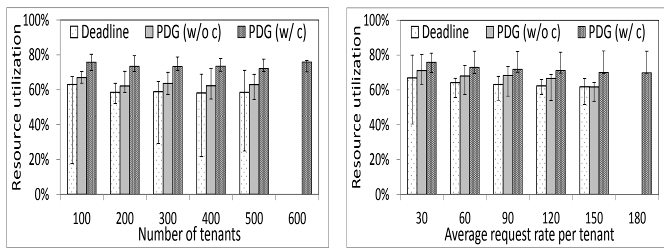
6.3.2 Performance of Prioritized Data Reallocation

We measured the effectiveness of the prioritized data reallocation algorithm in satisfying the SLAs of all tenants. In this experiment, each data partition's request arrival rate varies once at a randomly selected time during the experiment time. The variation of request arrival rates is the same as in Figure 11. We use *PDG_R* and *PDG_NR* to denote *PDG* with and without this algorithm. We set $T_r = 0$ in *PDG_R*. We use *PDG_H* to denote *PDG* that uses the highest arrival rate in a previous time period as the predicted rate for the next period.

Figures 16(a) and 16(b) show the median, the 5th and 95th percentiles of QoS of SLA of each method. They show that the QoS follows *PDG_NR*<1≈*PDG_R*<*PDG_H*. *PDG_NR* cannot supply a deadline guaranteed service with varying data request

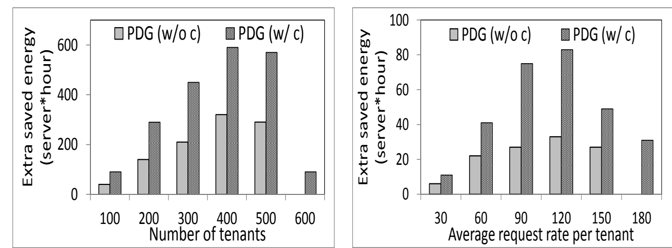
1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems



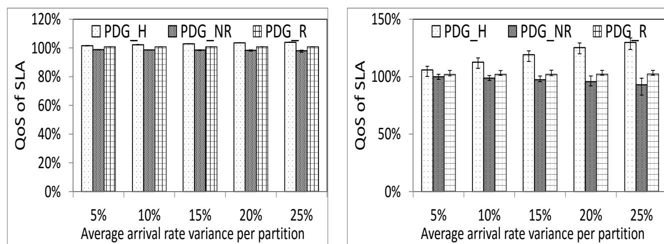
(a) In simulation (b) On Amazon EC2

Fig. 14: Resource utilization improvement of the deadline strictness clustered algorithm.



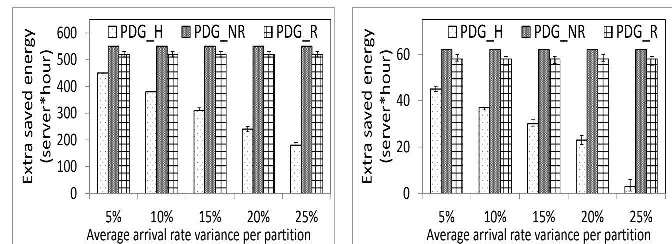
(a) In simulation (b) On Amazon EC2

Fig. 15: Extra saved energy of the deadline strictness clustered algorithm.



(a) In simulation (b) On Amazon EC2

Fig. 16: QoS of SLA enhancement of the prioritized data reallocation algorithm.



(a) In simulation (b) On Amazon EC2

Fig. 17: Extra saved energy of the prioritized data reallocation algorithm.

arrival rates in next period. The QoS of *PDG_NR* decreases when the variance increases. With greater request arrival rate variance, the overloaded servers with larger arrival rates may supply longer latency to more requests, which leads to QoS lower than 100%. *PDG_R* instantly reallocates the data replicas of high request arrival rate, which can always supply a deadline guaranteed service with no less than 100% QoS. The *PDG_H* uses the past highest request arrival rate of each data partition as the predicted value, so it supplies a deadline guaranteed service. Figures 16(a) and 16(b) indicate that the prioritized data reallocation algorithm helps *PDG* supply a deadline guaranteed service with varying request arrival rates.

Figure 17(a) and 17(b) show the energy saved by different methods versus the average arrival rate variance per partition. It shows that the saved energy follows *PDG_NR*>*PDG_R*>*PDG_H*. Because both *PDG_R* and *PDG_NR* have the same data allocation initially, and *PDG_R* needs to additionally execute data reallocation algorithm for the prioritized servers experiencing severe SLA violations, so it saves less energy than *PDG_NR*. Due to the same reason as Figure 16(a), *PDG_H* produces more active servers than other two methods. These figures indicate that the prioritized data reallocation algorithm saves more energy than simply using the largest data request arrival rate in handling the request burst, while ensuring the SLAs.

6.3.3 Performance of Adaptive Request Retransmission

In order to show the individual performance of the adaptive request retransmission algorithm, we measure its performance in Amazon EC2 [16] without PDG's other enhancement algorithms in Section 5. In this experiment, by default, we tested the performance of data request from one tenant t_k, and the number of t_k's data partitions were set to 1000. The distributions of the size and the visit rate of a data partition are the same as before. We used two nodes in Amazon EC2 [16] to be the front-end servers. By default, we chose $r = 6$ other nodes in the same region of Amazon EC2 [16] as replica servers, each of which stores the replicas of all data partitions.

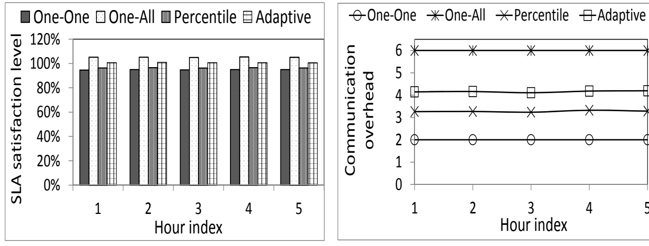
In this experiment, we first show the effectiveness of the sequential retransmission in meeting SLA requirements and saving communication overhead. We use *One-One* to denote the algorithm that randomly selects one server to request the data partition, and use *One-All* to denote the algorithm that simultaneously sends requests to all servers storing replicas of the requested data. We use $p = x\%$ to denote the *Percentile* [25] algorithm with waiting time equals the $x\%$ of the response latencies of all requests of data partitions in the system in the last period. We conducted the experiment for one hour to get the CDF of the response latency of each server and then evaluated the performance of all algorithms during the next hour in Amazon EC2 US East and West (Oregon), separately.

We then measured the effectiveness of our adaptive request retransmission algorithm (denote by *Adaptive*) in satisfying the SLA and reducing communication overhead. In this experiment, after one-hour running, tenant t_k among all tenants reduced its d_{t_k} to 40ms from 50ms and kept $\epsilon_{t_k} = 95\%$ the same as before. τ_{t_k} was only calculated once after the first hour. We compared *Adaptive* with the *One-One* and *One-All* algorithms. We also compared it with *Percentile* [25], in which the front-end server retransmits the request after 95% of the response latencies of all responses from all servers for all requests in the last period if there is no response. We measured the performance of each algorithm during each hour of consecutive 5 hours after one hour running.

Figure 18(a) shows the user satisfaction level of different algorithms during each hour. It shows that the user satisfaction level follows *One-All*> *Adaptive*≈1>*Percentile*>*One-One*. *One-All* submits the requests to all servers containing the replica of the requested

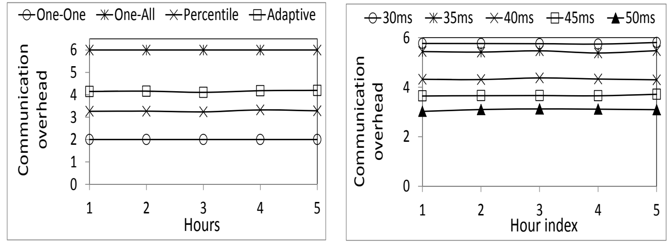
1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems



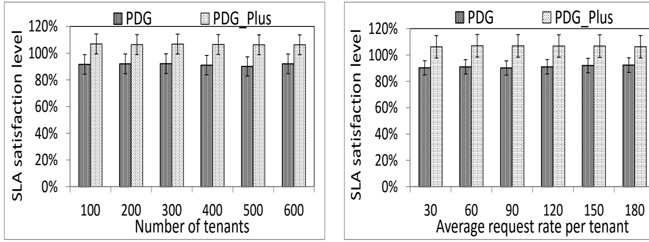
(a) SLA satisfaction level (b) Communication overhead

Fig. 18: Performance of the adaptive request retransmission algorithm.



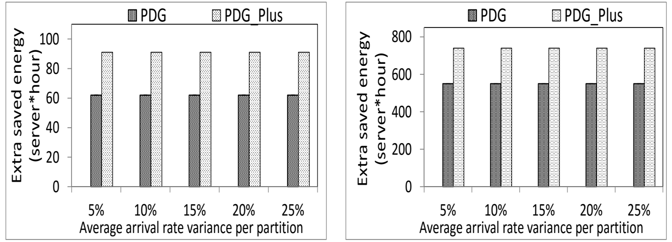
(a) SLA satisfaction level (b) Communication overhead

Fig. 19: Performance of the adaptive request retransmission algorithm with different deadline requirements.



(a) In simulation (b) On Amazon EC2

Fig. 20: SLA satisfaction level of enhanced PDG.



(a) In Simulation (b) On Amazon EC2

Fig. 21: Saved energy of enhanced PDG.

data partition simultaneously, which can be regarded as *Adaptive* with $\tau_{t_k} = 0$. *Adaptive* retransmits the request to servers one by one after adaptive waiting time τ_{t_k}, which satisfies the SLA. Since a lower τ_{t_k} leads to a higher probability to receive a response within the deadline, *One-All* generates a higher satisfaction level than *Adaptive*. The 95% of the response latency used in *Percentile* is much longer than τ_{t_k} in *Adaptive*. Therefore, it has a lower probability to receive the response by the deadline, and cannot supply an SLA guaranteed service. *One-One* does not have the retransmission, and hence produces a lower probability to receive the response within the deadline than *Percentile*. This figure indicates that *Adaptive* and *One-All* can supply an SLA guaranteed service. However, *One-All* generates much higher transmission overhead, which is shown in the following.

Figure 18(b) shows the communication overhead of different algorithms during each hour. It shows that the communication overhead follows *One-All>Adaptive>Percentile>One-One*. *One-All* submits the requests to all servers containing the requested data partition simultaneously, leading to the highest communication overhead. Other algorithms send requests to servers one by one after a certain waiting time, during which a response may be received. Thus, they generate lower communication overhead than *One-All*. *Adaptive* adaptively sets the waiting time τ_{t_k} to guarantee the SLA while minimizing the number of retransmission messages. The waiting period used in *Percentile* is much longer than the adaptive waiting time τ_{t_k}. Thus, it saves more retransmission messages than *Adaptive*. *One-One* selects only one server to request the data partition without retransmission, resulting in the lowest communication overhead. Figures 18(b) indicates that *Adaptive* generates lower transmission overhead than *One-All*, though both of them can supply deadline guaranteed service. Although *Percentile* and *One-One* generate lower communication overhead than *Adaptive*, they cannot provide SLA guaranteed service. Figures 18(a) and 18(b) together indicate that *Adaptive* can supply an SLA guaranteed service while maximally saving communication overhead.

We then measured the *Adaptive*'s performance under different SLA requirement changes. We tested the performance of the data requests of 5 tenants, each having 1000 data partitions. After one-hour running, each tenant reduces the deadline from 50m to a lower value (indicated in the figure). Figure 19(a) shows the $(100\%-\epsilon)$ (95%) of response latency of the data requests of each of the 5 tenants in *Adaptive* after each of total 5 hours. From the figure, we can observe that the 95% of response latencies are all below the required deadline, which means that *Adaptive* can receive at least $(100\%-\epsilon)$ of requests within each different d_{t_k}. *Adaptive* changes the adaptive waiting time τ_{t_k} according to Equation (16) under different SLA requirements. The figure indicates that *Adaptive* can always supply an SLA guaranteed service even when a tenant has shorter deadline requirement.

Figure 19(b) shows the communication overhead of the data requests of each of the 5 tenants in *Adaptive* after each of total 5 hours. It shows that a lower deadline requirement leads to a larger communication overhead. This is because a lower deadline requirement leads to a shorter adaptive waiting time τ_{t_k} according to Equation (16). The experimental results indicate that *Adaptive* can save communication overhead by adaptively adjusting the adaptive waiting time τ_{t_k} when the deadline is decreased and supply an SLA guaranteed service as shown in Figure 19(a).

6.3.4 Performance of Enhanced PDG

We then measure the performance of *PDG* with all the three enhancement algorithms (denoted by *PDG_Plus*) including the deadline strictness clustered data allocation, the regular prioritized data reallocation with $T_r = 0$ and the adaptive request retransmission. We measured the SLA satisfaction performance and energy saving since they are the most important metrics. In this experiment, tenants add data replicas to servers in turn and each tenant adds one data replica to a server

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems

at each time. Also, each data partition's request arrival rate varies once at a randomly selected time during the experiment time.

Figures 20(a) and 20(b) show the median, 5th and 95th percentiles of all tenants' SLA satisfaction level in simulation and on testbed, respectively. In this experiment, *PDG* cannot supply a deadline guaranteed service while *PDG_Plus* can, which show the combined effectiveness of the three enhancement algorithms in improving the SLA satisfaction performance. Figure 21(b) and 21(b) show the extra saved energy versus the average arrival rate variance per partition. They show that the extra saved energy follows *PDG_Plus*>*PDG*. The results also confirm the combined effectiveness of the three enhancement algorithms in reducing the energy consumption. For the details of the reasons, please refer to the previous sections in Section 6.3. The results suggest that 1) grouping tenants with similar deadline strictness can lead to higher resource utilization; 2) when the request arrival rates vary greatly, distributed load balancing method can offload the excess load from overloaded servers more quickly; and 3) the assigned servers for a data request can be adaptively determined in order to improve the SLA satisfaction performance.

7 RELATED WORK

Recently, several works [10, 12–15] have been proposed on deadline-aware network communications in datacenters. Since bandwidth fair sharing among network flows in the current datacenter environment can degrade application deadline awareness performance, Wilson *et al.* [10] proposed D_3 explicit rate control to apportion bandwidth according to flow deadlines instead of fairness. Hong *et al.* [12] proposed a distributed flow scheduling protocol. A flow prioritization method is adopted by all intermediate switches based on a range of scheduling principles, such as EDF (Earliest Deadline First) and so on. Earliest Deadline First (EDF) [13] is one of the earliest packet scheduling algorithms. It assigns a dynamic priority to each packet to achieve high resource utilization and satisfy the deadline. Vamanan *et al.* [14] 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. In [15], a new cross-layer network stack was proposed to reduce the long tail of flow completion times. Our work shares a similar goal of deadline guarantee as the above works. However, they focus on scheduling work flows for deadline-aware network communications rather than cloud storage systems. Spillane *et al.* [35] used advanced caching algorithms, data structures and Bloom filters to reduce the data Read/Write latencies in a cloud storage system. However, it cannot quantify the probability of guaranteed latency performance without considering request rates of stored data partitions in a server.

To reduce the service latency of tenants, *Pisces* [33] assigns the resources according to tenant loads and allocates the partitions of tenants using a greedy strategy that aims not to exceed storage capacity and service capacity of servers. In [36], the authors improve Best-Fit scheduling algorithm to achieve throughput-optimal. Wei *et al.* [37] proposed a cost-effective dynamic replication management scheme to ensure the data availability. It jointly considers the average latency and failure rate of each server to decide optimal replica allocation. Wang *et al.* [26] proposed a scalable block storage system using pipelined commit and replication techniques to improve the data access efficiency and data availability. In [38–40], 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. Ford *et al.* [41] proposed a replication method over multiple geo-distributed file system instances to improve data availability by avoiding concurrent node failures. However, these methods cannot guarantee SLAs of tenants without considering the request rates of stored data in a server and its service rate. There are related works in datacenter focusing on topology improvement/management to improve the bisection bandwidth usage of the network to increase the throughput, such as FatTrees [24], VL2 [42], BCube [43], and DCell [43], which finally reduce the average latency. However, none of them can guarantee the deadlines of data requests.

8 CONCLUSION

In this paper, we propose the parallel deadline guaranteed scheme (*PDG*) for cloud storage systems, which dynamically moves data request load from overloaded servers to underloaded servers to ensure the SLAs for tenants. *PDG* incorporates different methods to achieve SLA guarantee with multi-objectives including low traffic load, high resource utilization and fast scheme execution. Our mathematical model calculates the extra load that each overloaded server needs to release to meet the SLAs. The load balancer builds a virtual tree structure to reflect the real server topology, which helps schedule load movement between close servers in a bottom-up parallel manner, thus reducing traffic load and expedite scheme execution. The scheduling considers data partition size and request rate to more quickly resolve the overloaded servers. A server deactivation method also helps minimize the number of active servers while guaranteeing the SLAs. *PDG* is further enhanced by the deadline strictness clustered data allocation algorithm to increase resource utilization, a prioritized data reallocation algorithm and an adaptive request retransmission algorithm to dynamically strengthen SLA guarantee under the variances of request arrival rates and SLA requirements, respectively. Our trace-drive experiments on both a simulator and Amazon EC2 [16] show that *PDG* outperforms other methods in guaranteeing SLA and the multi-objectives. In our future work, we will implement our scheme in a cloud storage system to examine its real-world performance.

ACKNOWLEDGEMENTS

This research was supported in part by U.S. NSF grants NSF-1404981, IIS-1354123, CNS-1254006, and Microsoft Research Faculty Fellowship 8300751. An early version of this work is presented in the Proc. of P2P'15 [44].

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

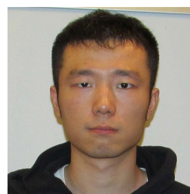
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2513054, IEEE Transactions on Parallel and Distributed Systems

REFERENCES

[1] Amazon DynnamoDB. http://aws.amazon.com/dynamodb/, [Accessed in Nov. 2015].
[2] Amazon S3. http://aws.amazon.com/s3/, [Accessed in Nov. 2015].
[3] Gigaspaces. http://www.gigaspaces.com/, [Accessed in Nov. 2015].
[4] H. Stevens and C. Pettey. Gartner Says Cloud Computing Will be as Influential as E-Business. Gartner Newsroom, Online Ed., 2008.
[5] S. L. Garfinkel. An Evaluation of Amazons Grid Computing Services: EC2, S3 and SQS. *Technical Report TR-08-07*, 2007.
[6] N. Yigitbasi A. Iosup and D. Epema. On the Performance Variability of Production Cloud Services. In *Proc. of CCGrid*, 2011.
[7] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of OSDI*, 2008.
[8] R. Kohavl and R. Longbotham. Online Experiments: Lessons Learned. *http://exp-platform.com/Documents/IEEEComputer2007OnlineExperiments.pdf.*, 2007.
[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannona, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *Proc. of VLDB*, 2008.
[10] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proc. of SIGCOMM*, 2011.
[11] M. Alizadeh, A. Greenberg, D. A. Maltz, P. Patel J. Padhye, B. Patel, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proc. of SIGCOMM*, 2010.
[12] C. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proc. of SIGCOMM*, 2012.
[13] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 1973.
[14] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). In *Proc. of SIGCOMM*, 2012.
[15] 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.
[16] Amazon EC2. http://aws.amazon.com/ec2/, [Accessed in Nov. 2015].
[17] D. Wu, Y. Liu, and K. W. Ross. Modeling and Analysis of Multichannel P2P Live Video Systems. *TON*, 2010.
[18] A. Beloglazov and R. Buyya. Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient dDnamic Consolidation of Virtual Machines in Cloud Data Centers. *CCPE*, 2011.
[19] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual Machine Image Distribution Network for Cloud Data Centers. In *Proc. of INFOCOM*, 2012.
[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
[21] L. Kleinrock. *Queueing Systems*. John Wiley & Sons, 1975.
[22] N. Bobroff, A. Kochut, and K. Beaty. Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Proc. of IM*, 2007.
[23] W. J. Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling*. Princeton University Press, 2009.
[24] M. AI-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of SIGCOMM*, 2008.
[25] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.
[26] 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.
[27] Apache Hadoop FileSystem and its Usage in Facebook. http://cloud.berkeley.edu/data/hdfs.pdf.
[28] Palmetto Cluster. http://http://citi.clemson.edu/palmetto/, [Accessed in Nov. 2015].
[29] N. B. Shah, K. Lee, and K. Ramchandran. The MDS Queue: Analysing Latency Performance of Codes and Redundant Requests. *arXiv:1211.5405*, 2013.
[30] CTH Trace. http://www.cs.sandia.gov/Scalable_IO/SNL _Trace_Data/, [Accessed in Nov. 2015], 2009.
[31] H. Medhioub, B. Msekni, and D. Zeghlache. Ocni-open cloud networking interface. In *Proc. of ICCCN*, 2013.
[32] R. Stanojevic, N. Laoutaris, and P. Rodriguez. On Economic Heavy Hitters: Shapley Value Analysis of 95th-Percentile Pricing. In *Proc. of IMC*, 2010.
[33] D. Shue and M. J. Freedman. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proc. of OSDI*, 2012.
[34] 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 DecentralizedWakeup Service. In *Proc. of NSDI*, 2012.
[35] R. P. Spillane, P. Shetty, E. Zadok, S. Dixit, and S. Archak. An Efficient Multi-Tier Tablet Server Storage Architecture. In *Proc. of SoCC*, 2011.
[36] S. T. Maguluri, R. Srikant, and L. Ying. Stochastic Models of Load Balancing and Scheduling in Cloud Computing Clusters. In *Proc. of INFOCOM*, 2012.
[37] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng. CDRM: A Cost-Effective Dynamic Replication Management Scheme for Cloud Storage Cluster. In *Proc. of Cluser*, 2010.
[38] 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.
[39] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: Practical Power-Proportionality for Data Center Storage. In *Proc. of Eurosys*, 2011.
[40] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proc. of SIGMOD*, 2011.
[41] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of OSDI*, 2010.
[42] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A Scalable and Flexible Data Center Network. In *Proc. of SIGCOMM*, 2009.
[43] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. In *Proc. of SIGCOMM*, 2009.
[44] G. Liu and H. Shen. Deadline Guaranteed Service for Multi-Tenant Cloud Storage. In *Proc. of P2P*, 2015.

Guoxin Liu received the BS degree in BeiHang University 2006, and the MS degree in Institute of Software, Chinese Academy of Sciences 2009. He is currently a Ph.D. student in the Department of Electrical and Computer Engineering of Clemson University. His research interests include distributed networks, with an emphasis on Peer-to-Peer, datacenter and online social networks.

Haiying Shen received the BS degree in Computer Science and Engineering from Tongji University, China in 2000, and the MS and Ph.D. degrees in Computer Engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Associate Professor in the ECE Department at Clemson University. Her research interests include distributed computer systems and computer networks with an emphasis on P2P and content delivery networks, mobile computing, wireless sensor networks, and grid and cloud computing. She was the Program Co-Chair for a number of international conferences and member of the Program Committees of many leading conferences. She is a senior member of the IEEE and a member of the ACM.



Haoyu Wang received the BS degree in University of Science & Technology of China, and the MS degree in Columbia University in the city of New York. He is currently a Ph.D student in the Department of Electrical and Computer Engineering of Clemson University. His research interests include datacenter, cloud and distributed networks.

1045-9219 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.