2012 41st International Conference on Parallel Processing

RFH: A Resilient, Fault-Tolerant and High-efficient Replication Algorithm for Distributed Cloud Storage

Yanzhen Qu, Naixue Xiong
School of Computer Science, Colorado Technical University, CO, USA
Email: {yqu, nxiong}@coloradotech.edu

***Abstract*—To avoid failure and achieve higher availability, replication scheme is now widely used in distributed Cloud storage systems [25]. However, most of them only statically replicate data on some randomly chosen nodes for a fixed number of times and it is obviously not enough for more reasonable resource allocation. Moreover, query load for Web application is highly irregular. It throws us into a dilemma to always maintain maximum number of replicas in case of explosive query load outburst or save resources with fewer replicas at the expense of performance. In this paper, we present a Resilient, Fault-tolerant and High-efficient global replication algorithm (RFH) for distributed Cloud storage systems. RFH is especially efficient facing 'flash crowd' problem. Each data partition is represented by a virtual node. Each virtual node itself decides whether to replicate, migrate or suicide by weighing up the pros and cons. It is based on the evaluation of traffic load of all nodes, and selects among physical nodes with the most traffic (traffic hub) to replicate or migrate on. After that, it takes into account blocking probability to achieve quicker response and better load balance performance. Extensive simulations have been conducted and the results have demonstrated that the proposed scheme RFH outperforms the main existing algorithms (the request-oriented algorithms [16] [5], the owner-oriented algorithms [7] [11] [12] [13] and the random algorithms [4] [21] [22] in terms of high replica utilization rate, high query efficiency and reasonable path length at a low cost while maintaining high availability.**

***Index Terms*—Data replication, Distributed Cloud storage, Fault-tolerance, High-efficient.**

I. Introduction

Distributed Cloud storage is now widely used by Cloud service providers, such as Amazon S3, Google, App iCloud and DropBox. Based on Cloud storage e-commerce platforms, these companies serve thousands of millions customers using tens of thousands of servers located distributed in many datacenters world widely. However, hardware failure, power failure and network failure [1] in current datacenters are becoming more and more frequent as storage capacity scales up [23]. To avoid access failure and data loss caused by these failures, or by natural disasters, such as earthquake or tornado which may destroy a whole datacenter, replication scheme is widely employed by Cloud storage systems, such as Amazon Dynamo, to guarantee its reliability and availability by replicating data partitions across different datacenters.

Nevertheless, to the best of our knowledge, most of the current Cloud storage systems replicate each data item at a fixed number of physically distinct nodes in a static way, without taking replication cost and geographical diversity into account. More importantly, all of them never consider design from a traffic-based angle. For example, hot spot change, flash crowd, lookup skew and load imbalance are also important factors that impact customer's experience and raise a problem for replication scheme. On the one hand, it is reported by Amazon that a Service Level Agreement (SLA) should guarantee a response within 300ms for 99.9% of its requests a peak client load of 500 requests per second [4]. Given that the slightest outage will impact customers' trust and has significant financial consequences, a system should be built to provide all customers with a good experience, rather than just the majority. On the other hand, "Slashdot effect" indicates that the query rate for Web application data is highly irregular. If a hot partition and its replicas receive too many requests at a time, they could become overloaded and consequently cannot response to the clients within time limit. But if a partition is seldom accessed or a hot partition is cooled down as time passes by, too many replicas of it will become a waste of resource and causes maintenance overhead. A new replication algorithm is in dire need to address these problems. Therefore, we propose RFH, a resilient, fault-tolerant global replication algorithm, which can flexibly replicate data according to changing query load, with high efficiency.

Clearly, to achieve high availability while maintaining low replication cost, it is better to choose a different datacenter close to the primary partition owner to replicate on. We call it owner-oriented. The advantage is relatively lower replication cost and consequently lower replication failure possibility. However, the lookup path length and response time cannot be significantly reduced, especially when most of the large amount of queries is from far away continents. Another method is request-oriented, which encourages replicating data on datacenters near to the requesters with the highest query rate. This method is able to reduce lookup path length dramatically and consequently improve query efficiency. However, it cannot guarantee replica utilization rate since those other requesters will have a lower chance to access these replicas. Moreover, query interest for Web application data varies dramatically. It can cause a massive increase in traffic within a few minutes, and it can also pass into silence after peak time. So the replicas remained close to the requesters become useless. Even if migration and suicide functions can be employed to dynamically relocate or kill these replicas, the migration cost or the re-replication cost are high, let alone the possible failure rate that will increase with the distance.

To address the above issues, in this paper, we propose a

0190-3918/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPP.2012.3

Resilient, Fault-tolerant and High-efficient global replication algorithm (RFH) for Cloud storage systems. The RFH algorithm is a better way to address this challenge, which can achieve high replica utilization rate, high query efficiency and reasonable path length at a low cost while maintaining high availability. Rather than owner-oriented or request-oriented, it replicates data on nodes with the most forwarding traffic, such as conjunction nodes of many necessary routing paths. We call it traffic-oriented. Unlike request-oriented method, its replicas can serve for most requesters, resulting in higher utilization rate. To deal with query interest variance, RFH employs flexible migration scheme together with replication algorithm to dynamically change replica number and location to meet different needs. It also adopts suicide function to reclaim resources in time. At the meanwhile, it can achieve quicker response by reducing lookup path length compared to other three algorithms. The concept of virtual ring and virtual node is put to use in the RFH algorithm. A virtual ring is consist of a ring of virtual nodes based on consistent harshing [24]. A virtual node itself can decide whether to replicate, migrate or suicide according to the RFH algorithm decision agent. The advantage is that each node is highly independent and node join and departure only impacts its immediate neighbors.

The rest of the paper is organized as follows: In Section II, the design and analysis for our RFH replication algorithm are presented from various aspects. Section III shows experimental performance results of the RFH algorithm in comparison with other main existing algorithms under a variety of metrics. Finally, we conclude this paper and propose the future works in Section V.

II. RFH REPLICATION ALGORITHM

In this section, we will describe the proposed RFH algorithm. First, we discuss main problems involved in replication requirement for Cloud Storage systems. Then, we give out the design and analysis of the RFH algorithm from various aspects.

A. Problem and strategies

Unpredictable query rate, especially under massive increase at an explosive speed, raises a great challenge to Cloud storage systems. For example, in Fig. 1, there are 10 datacenters in different countries of different continents. Datacenter A holds a hot partition, which is frequently requested by many clients from different locations, and 80% of the queries are from the clients near to datacenters I, J and H. Many Cloud storage systems, such as Amazon Dynamo, employ distributed key-value store, which will replicate data at the N-1 clockwise successor nodes [4]. Although adjacent in node ID space, these replicas are actually randomly chosen considering geographical location. In Fig. 1, replicas will be distributed to any other datacenters with a random manner.

If with an owner-oriented manner, the coordinator will consider maximizing availability while minimizing replication cost. Replication cost relates to partition size s_i, failure rate f_i, replication bandwidth b_i and distance d_i, between the source and the destination. Thus replication cost c_i is defined as:

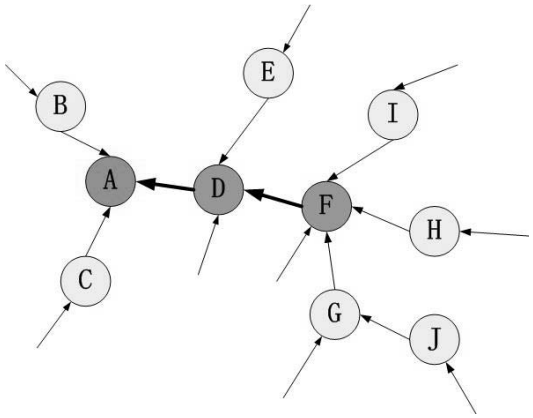

Fig. 1. Client Requests in Global Distributed Cloud Storage System

$$c_i = \frac{d_i.f_i.s_i}{b_i} \tag{1}$$

To measure availability, each physical node (i.e., storage hosts or servers in a rack) has a label of the form "continent-country-datacenter-room-rack-server" in order to identify its geographical location [5]. For example, in Fig. 1, a server located in Datacenters A is possibly labeled as "NA-USA-GA1-C01-R02-S5". We measure availability level according to geographical diversity. If two servers are in different datacenters, they are of the highest availability level, Level 5. If two servers are in the same datacenter, but different rooms, their availability level is 4. Correspondingly, the lowest level is Level 1, which means the two replicas are in the same server. Thus, in Fig. 1, replicas will be placed on B and C, which are in the same country of A, or it will replicate on D, which is in the same continent of A, with relatively low replication cost but high availability. Migration function is employed when higher availability versus cost can be obtained.

If with a request-oriented manner, it will choose among datacenters closest to the clients, where most of the queries come from. As mentioned above, 80% of the queries are from clients near Datacenters I, J and H. Thus, in Fig. 1, Datacenters H, I and J, which are the closet to the query clients, become replication preference. It will randomly choose a node among the top 3 ones to replicate on. The migration process is started when another node without any replica joins in the list of the top 3.

The RFH algorithm, which employs a traffic-oriented manner, chooses a server in datacenters with the most forwarding traffic to replicate on. In Fig. 1, it prefers to replicate on datacenters D and F, which are in necessary routing paths of many queries from the clients to the hot partition holder A, consequently shoulder most traffic. The condition of migration or suicide is also based on traffic load, which will be further discussed in the following.

B. Partitioning and Routing

We employ the similar partitioning algorithm as it's in Amazon Dynamo. The partitioning scheme of RHF is built using a variant of consistent hashing [6]. Data is dynamically partitioned or stripped over the set of storage hosts or physical

nodes in the system. A ring topology, which is treated as a fixed circular space, is employed as the output range of a hash function. A ring consists of several virtual nodes. Each node is assigned a random value within the hashing space to represent its position. A physical node hosts an amount of virtual nodes within its capacity limit, such as maximum disk storage and processing speed. However, instead of a coordinator node, a virtual node itself can decide to whether to replicate, migrate or suicide according to RFH decision tree described in Section II-E. Apparently, the advantage is that node join or departure, failure or recovery only affects its immediate neighbors, and keep other nodes unaffected. Note that maintaining data consistency is not the focus of this work.

Routing is similar to Oceanstore [7] in RHF. Entities are free to reside on any physical node. Every physical server manages the routing table of all its virtual nodes. The routing layer of RFH is built on top of IP and provides additional functionality. The routing protocol messages are labeled with a destination ID. It routes messages directly to the closest node which has the desired ID and matches the prefix. A virtual node periodically calculates its traffic load, replication storage capacity and bandwidth for a replica. If it's overloaded by its traffic and has enough storage and bandwidth capacity, it will add its replication request and other information, such as its ID, holder ID and IP address to the tail of the received query, and forward it to the next hop. The cost of routing is $\mathcal{O}(log\ n)$. To avoid a huge amount of manual input for widely distributed nodes, we employ automatic address configuration for Data Center networks [2]. It can automatically configures Data Centers within seconds, including traditional Data Centers and BCube [3].

C. Traffic determination

Given that the RFH algorithm is traffic-oriented, we should first discuss query and traffic. Query is different from traffic. Query is the request information that a client sends to request for a certain data partition. In the routing process, a forwarded query will generate some traffic which is different according to different routing algorithms. Here, we only calculate valid traffic that routes directly to the destination, not including forwarding information to all its neighbors. From every requester j to the holder of partition B_i, there's a routing path, and the set of all nodes in this path is denoted as A_{ij}, and

$$A_{ij} = \big\{N_k | N_k\ is\ at\ the\ routing\ path\ from\ N_i\ to\ N_j\big\}.$$

For each virtual node, the processing capacity is limited. And for each physical node, the number of virtual nodes holding a certain replica varies. So if the query load is out of its capacity, the rest of the queries that cannot be handled will be forwarded to the next physical node. During a unit time epoch *t*, suppose that the processing capacity of node N_k for a replica of B_i is C_{ikl}, $0 < l < m_{ik}$, and m_{ikt} is the number of total replicas of partition B_i that are now in physical node N_k. Let tr_{ijkt} represent the traffic load of node N_k for partition B_i and it's from requester *j*. Node $N_{k'}$ is the node immediate before N_k in the routing path, so its traffic, processing capacity and total replica number is $tr_{ijk't}$, $C_{ik'l}$ and $m_{ik't}$ respectively. Thus,

$$\begin{aligned} tr_{ijkt} &= \begin{cases} tr_{ijk't} - \sum_{l=0}^{m_{ik't}} C_{ik'l} & tr_{ijk't} > \sum_{l=0}^{m_{ik't}} C_{ik'l} \\ 0 & tr_{ijk't} \le \sum_{l=0}^{m_{ik't}} C_{ik'l} \end{cases} \\ &= max(0, tr_{ijk't} - \textstyle\sum_{l=0}^{m_{ik't}} C_{ik'l}). \end{aligned} \tag{2}$$

If node $N_{k'}$ is not the first node in a given routing path, let $N_{k''}$ represents the node before $N_{k'}$. So the traffic of node $N_{k'}$ can be given as:

$$\begin{aligned} tr_{ijk't} &= \begin{cases} tr_{ijk''t} - \sum_{l=0}^{m_{ik''t}} C_{ik''l} & tr_{ijk''t} > \sum_{l=0}^{m_{ik''t}} C_{ik''l} \\ 0 & tr_{ijk''t} \le \sum_{l=0}^{m_{ik''t}} C_{ik''l} \end{cases} \\ &= max(0, tr_{ijk''t} - \sum_{l=0}^{m_{ik''t}} C_{ik''l}). \end{aligned} \tag{3}$$

Thus, based on (2) and (3), we can reach

$$\begin{aligned} tr_{ijkt} &= max(0, tr_{ijk't} - \sum_{l=0}^{m_{ik't}} C_{ik'l}) \\ &= max\big(0, max(0, tr_{ijk''t} - \sum_{l=0}^{m_{ik''t}} C_{ik''l}) - \sum_{l=0}^{m_{ik't}} C_{ik'l}\big) \\ &= \dots \\ &= max\big(0, \dots, max(0, tr_{ijjt} - \sum_{l=0}^{m_{ijt}} C_{ijl})\big). \end{aligned} \tag{4}$$

For simplicity and without losing generality, we regard queries closest to datacenter j as from requester j. We define the number of queries for a partition B_i, during a unit time period T, from requester j, as q_{ijt}. Assume that N_{k^x} is any node before N_k in the routing path, so $N_{k^x} \in A_{ij}$. The processing capacity and replica number of node N_{k^x} are m_{ik^x} and $C_{ik^xl} (0 < l < m_{ik^x})$ respectively. Because,

$$tr_{ijjt} = q_{ijt}, \tag{5}$$

tr_{ijkt} can be given as:

$$\begin{aligned} tr_{ijkt} &= max\big(0, \dots, max(0, tr_{ijjt} - \sum_{l=0}^{m_{ijt}} C_{ijl})\big) \\ &= max(0, q_{ijt} - \sum_{k^x \in A_{jk}} \sum_{l=0}^{m_{ik^xt}} C_{ik^xl}). \end{aligned} \tag{6}$$

For each node N_k, it may be at the routing path from requester j to the holder of partition B_i, or it may not. Thus, the probability that it's at the path is p_{ijk}. The value of p_{ijk} is either 1 or 0, say

$$p_{ijk} \in \big\{0, 1\big\}.$$

Correspondingly, during period t, for partition B_i, the traffic of a forwarding or a host node N_k is denoted as tr_{ikt}. Therefore, the traffic of node N_k for partition B_i is given as:

$$tr_{ikt} = \sum_{j=1}^{N} tr_{ijkt}.p_{ijk}. \tag{7}$$

We replace tr_{ijkt} with (20), then we get:

$$tr_{ikt} = \sum_{j=1}^{N} max(0, q_{ijt} - \sum_{k^x \in A_{jk}} \sum_{l=0}^{m_{ik^xt}} C_{ik^xl}).p_{ijk} \tag{8}$$

The system average query for partition B_i during epoch t is:

$$\overline{q}_{it} = \frac{\sum_{j=1}^{N} q_{ijt}}{N}. \quad (9)$$

In order to compensate for steep changes of the query rate, we take historical data into account and use a smoothing factor α. The average system query is then presented as:

$$\overline{q}_{it} = \alpha.\overline{q}_{i(t-1)} + (1-\alpha).\overline{q}_{it},\ 0 < \alpha < 1. \quad (10)$$

We also use the same smoothing factor for traffic evaluation. So,

$$tr_{ikt} = \alpha.tr_{ik(t-1)} + (1-\alpha).tr_{ikt},\ 0 < \alpha < 1. \quad (11)$$

For the holder of partition B_i, if its traffic is larger than β times of the average system query, it is regarded as overloaded, i.e.:

$$tr_{iit} \geq \beta.\overline{q}_{it},\ \beta > 1. \quad (12)$$

For a forwarding node, if its traffic is larger than γ times of the average system query, it is regarded as overloaded and is marked as traffic hub node, i.e.:

$$tr_{ikt} \geq \gamma.\overline{q}_{it},\ \gamma > 1. \quad (13)$$

D. Availability lower limit

Query rate is changing in Web applications. Sometimes the query rate is very large and the system is overloaded, but it may be not that crowded at some other times. However, the minimum availability should still be satisfied for the sake of good user experience. Suppose that f_i is the failure probability of a virtual node, which is the i_{th} replica of an original node j. And r_j is its replica number. According to [8], to obtain the expected availability A_{expect}, the minimum replica number r_{min} should satisfy the following inequation:

$$1 - \sum_{j=1}^{m} (-1)^{j+1} C_m^j (\prod_{i=1}^{r_j})^j \geq A_{expect}. \quad (14)$$

For a given minimum expected availability requirement, we can get the minimum replica number, say r_{min}. For example, if the system requires a minimum availability of 0.8 and the failure probability is 0.1, then the minimum replica number is 2 according to this inequation.

E. RFH decision tree

Based on traffic determination, we discuss how the RFH algorithm works, in this subsection. Fig. 2 gives out the decision tree of RFH algorithm. Every node is self-organized. They replicate, migrate or choose to suicide with a decentralized manner. For each epoch, every node calculates availability according to (14). If the minimum availability is not reached for a primary partition holder, it will replicate to its most forwarding nodes, even if all the nodes are not overloaded. If the minimum availability is reached, but there's still too much traffic, it will force the scheme to start relieving load. Besides availability, query and traffic are also calculated during each epoch. Query is calculated according to (9) and (10). For a node holding an original partition, if its traffic satisfies condition (12), it's regarded as overloaded and then enters a status waiting for replication request from other nodes. For a node not holding an original partition, if the traffic value can satisfy condition (13), it will be regarded as a traffic hub. It will send a replication request with its ID, traffic value and other needed information, to the partition holder along the routing path. Once the node holding the original partition receives requests from traffic hubs, it will choose a node among the 3 nodes with the largest amount of traffic. In the next step, if there's any replica of it is not at these three nodes, it will check the migration condition according to (16) and sends a migration request to the node holding this replica. Otherwise, it will replicate to the chosen traffic hub node.

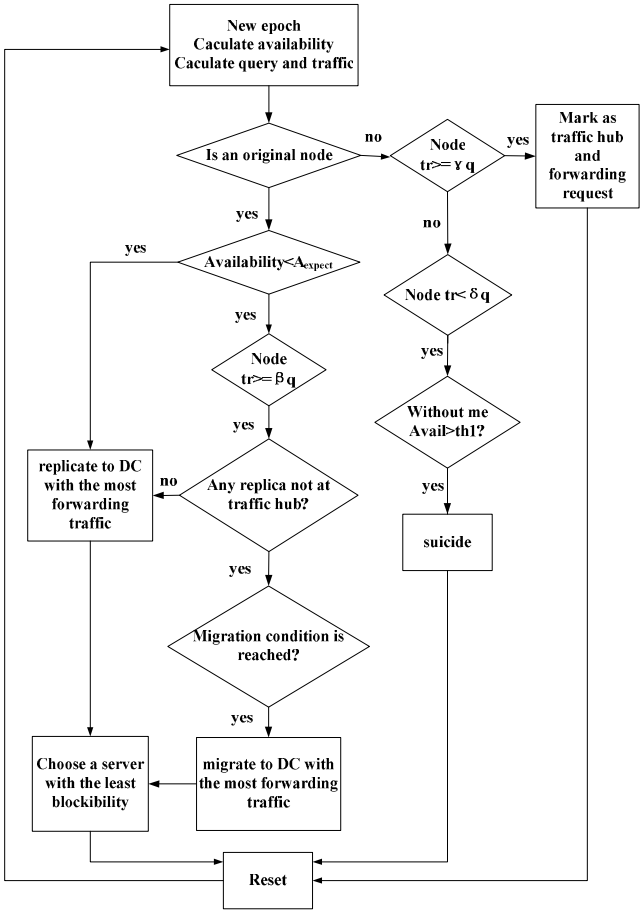

Fig. 2. Decision tree of the RFH algorithm

Query rate is always changing, so some replicas may become rarely visited. To maintain these replicas is time-consuming and may increase failure rate. To avoid maintenance overhead and resource waste, for a node holding a replica, if the traffic load is very light, say

$$tr_{ikt} \leq \delta.\overline{q}_{it}, \quad (15)$$

It will calculate the availability without itself. If the minimum availability is still satisfied without it, it will commit suicide.

In migration process, to guarantee enough benefit and avoid failure, a threshold of benefit is set to determine whether to migrate or not. It's given below:

$$tr_{ij} - tr_{ik} \geq \mu.\overline{tr}_i, \tag{16}$$

here, $\overline{tr}_i$ is the average traffic load of all nodes, which is for partition B_i.

$$\overline{tr}_i = \frac{\sum_{j=1}^{N} tr_{ij}}{N}. \tag{17}$$

Among the physical nodes in the same datacenter, RHF chooses a node with the lowest blocking probability. Let's assume λ_i is the arrival rate of Poisson process. τ_i is the average service time of a node i, and c_i is the upper processing limit of it. According to [10], the blocking probability BP_i is defined as follows:

$$BP_i = \frac{(\lambda_i \tau_i)^{c_i}}{c_i!} \left[\sum_{k=0}^{c_i} \frac{(\lambda_i \tau_i)^k}{k!} \right]^{-1}. \tag{18}$$

This model is a $M/G/c_i$ model. In each epoch, each physical node i leverages its computational ability and also records query information. It calculates the average value of λ_i and τ_i and then gets blocking probability BP_i periodically. The value of BP_i will be piggybacked into a replication request if there's any. A virtual node will choose a node which has the lowest value of BP_i to replicate or migrate. Thus, it can dynamically balance workload among all the physical nodes.

Another condition that a virtual node can choose a server is no less than its disk storage limit. If the current storage rate of a server is the upper limit, any replication or migration request will not be allowed. Certainly, on the virtual node side, it will not choose a crowded server either. This condition is given as:

$$S_i < \phi. \tag{19}$$

By default, the storage upper limit ϕ is set as 70%. Based on these two conditions, the RFH algorithm can achieve good load balance performance which will be shown in the experimental section.

F. Facing flash crowd

Web application query is highly irregular. There're mainly two types of query surge. The first one is query location changes. For example, in Fig. 1, most of the queries for a certain partition may first come from Tokyo, Japan, near Datacenter I, and then become very few. At the same time, queries for the same partition, which come from Beijing, China, is keeping increasing. After a short period of time, most of the queries is from Beijing, China, near Datacenter H. In this case, it has little impact on the RFH algorithm, which is traffic-oriented, because the traffic hub nodes are still D and E, replicas on which can still serve for most queries. Also, it has little impact on the owner-oriented algorithm, because replication decision is made according to availability versus cost. Higher availability cannot be obtained by increasing traffic or queries, so the owner-oriented algorithm will not be affected. However, replicas have to migrate or be added to H according to the request-oriented algorithm, resulting in relatively low efficiency and high cost.

The second type of query surge is that the popularity of a partition changes over time. For example, a hot partition in Datacenter A may become cool while another cool partition in Datacenter E becomes hot. For simplicity and without loss of generality, we suppose that most of the queries is from Beijing, China, near Datacenter H. For the request-oriented algorithm, the replicas of a former hot partition will become a waste of resource remaining on H, because of low utilization rate. For the owner-oriented algorithm, the performance counts on the location of the hot partition holder and Cloud network deployment. The replication cost of the holder that has limited number of neighbors is larger than that with more and close neighbors. The RFH algorithm can adapt the replica number according to changing traffic. If a partition becomes hot, more replicas will be replicated to meet the need of service, or replicas will migrate to achieve higher utilization. Otherwise, unwanted replicas will commit suicide to save resources.

G. Replica utilization rate

Replica utilization rate is a good criterion to measure the performance of replication algorithm. The more the replicas are, the less workload each one might shoulder, and the less the collapse probability may occur. However, too many replicas will result in low utilization rate and a waste of resource. Our proposed algorithm can satisfy service need with fewer replicas than other existing algorithms, so that the replica utilization rate is higher. Another factor impacts replica utilization rate is the placement of these replicas. If some replicas are in a location that is seldom visited by queries, the utilization rate is obviously lower. But if all or most of the replicas are in key locations, such as traffic hub, they will be fully utilized. The RFH algorithm place its replicas in conjunctions of many necessary routing paths, which definitely results in more hitting chances, and consequently higher utilization rate. This can be seen in our experiments, which will be further discussed in the following experimental section.

Now let's analyze replica utilization rate of the random algorithm, the owner-oriented algorithm, the request-oriented algorithm and the proposed RFH algorithm. During epoch t, tr_{ijkt} represents the traffic value of node N_k for partition B_i and it's from requester j. Suppose that the processing capacity of node N_k for a replica of B_i is C_{ikl}, $0 < l < m_{ikt}$, and m_{ikt} is the total replica number of partition B_i in node N_k. Thus, replica utilization rate of a replica on node N_k is denoted as

$$U_{iklt}, \text{ and } U_{iklt} = \begin{cases} 0 & tr_{ikt} \leq \sum_{n=0}^{l-1} C_{ikn} \\ \frac{tr_{ikt} - \sum_{n=0}^{l-1} C_{ikn}}{C_{ikl}} & \sum_{n=0}^{l-1} C_{ikn} < tr_{ikt} < \sum_{n=0}^{l} C_{ikn} \\ 1 & tr_{ikt} \geq \sum_{n=0}^{l} C_{ikn} \end{cases}$$
$$= min(1, max(0, \frac{tr_{ikt} - \sum_{n=0}^{l-1} C_{ikn}}{C_{ikl}})). \tag{20}$$

And the average replica utilization rate in epoch t is:

$$\overline{U}_t = \frac{\sum_{i=1}^{N} \sum_{k=1}^{N} \sum_{l=1}^{m_{ikt}} U_{iklt}}{\sum_{i=1}^{N} \sum_{k=1}^{N} m_{ikt}}. \tag{21}$$

Because

$$tr_{ikt} = \sum_{j=1}^{N} max(0, q_{ijt} - \sum_{k^x \in A_{jk}} \sum_{l=0}^{m_{ik^x t}} C_{ik^x l}).p_{ijk}$$
$$= \sum_{j=1}^{N} p_{ijk}.max(0, q_{ijt} - \sum_{k^x \in A_{jk}} \sum_{l=0}^{m_{ik^x t}} C_{ik^x l}),$$

then we get

$$\begin{aligned} U_{iklt} &= min(1, max(0, C_{ikl}^{-1}.(tr_{ikt} - \sum_{n=0}^{l-1} C_{ikn}))) \\ &= min\left[1, max\left[0, C_{ikl}^{-1}.\sum_{j=1}^{N} p_{ijk}.max\,(0, q_{ijt}\right.\right. \\ &= -\left.\left.\sum_{k^x \in A_{jk}} \sum_{l=0}^{m_{ik^x t}} C_{ik^x l}\right) - \sum_{n=0}^{l-1} C_{ikn}\right]\right]. \end{aligned} \tag{22}$$

Replace U_{iklt} in (21), for average replica utilization rate $\overline{U}_t$, we get:

$$\begin{aligned} \overline{U}_t = \Bigg\{ \sum_{i=1}^{N}\sum_{k=1}^{N}\sum_{l=1}^{m_{ikt}} min\Bigg[1, max\Bigg[0, C_{ikl}^{-1}.\Bigg(\sum_{j=1}^{N} p_{ijk}. \\ max(0, q_{ijt} - \sum_{k^x \in A_{jk}} \sum_{l=0}^{m_{ik^x t}} C_{ik^x l}) - \sum_{n=0}^{l-1} C_{ikn}\Bigg)\Bigg]\Bigg]\Bigg\} \\ \Big(\sum_{i=1}^{N}\sum_{k=1}^{N} m_{ikt}\Big)^{-1}. \end{aligned} \tag{23}$$

Replica utilization rate decreases with the increase of replica number, so replica number should be constrained in order to achieve higher utilization rate. Furthermore, with the same replica number and other parameters, if a replica could be in a physical node that belongs to more routing paths, there'll be more chances for the value of the parameter p_{ijk} to be 1, not 0. From (23), we can learn that if the parameter p_{ijk} has more chances be 1, then the value of $\overline{U}_t$ will be larger with the same replica number. Thus, the average utilization rate of all replicas is higher. Therefore, the replica utilization rate of RFH is higher than other three algorithms.

H. Load imbalance

Load balance is very important for replication algorithm. The goal of replication is to avoid data loss by backing up, and also to obtain quicker response by splitting workload using parallel processing. If a physical node holding a replica shoulders too much load while others are idle, this node may become too busy to response quickly. Thus, all the efforts of replication are in vain. To achieve better load balance performance, a virtual node that migrates or is replicated to a certain datacenter will choose a physical node with the lowest blockibility probability according to (18). And this is also for a consideration of quicker response.

To measure load balance, we assume that the workload of each virtual node is l_i, thus the average workload of the system is:

$$\bar{l} = \frac{\sum_{i=1}^{n} l_i}{n}. \tag{24}$$

Standard deviation is employed, and hence, the load imbalance Lb is denoted as:

$$Lb = \sqrt{\frac{\sum_{i=1}^{n} (l_i - \bar{l})^2}{n}}. \tag{25}$$

Put (24) into this formula, Lb is then given as:

$$Lb = \Big[\sum_{i=1}^{n} (l_i - \sum_{j=1}^{n} l_j.n^{-1})^2.n^{-1}\Big]^{1/2}. \tag{26}$$

Obviously, the lower the value of Lb is, the better the load balance performance can be achieved.

III. PERFORMANCE EVALUATION

A. Experiment setup

We assume a simulated Cloud storage environment similar to Fig. 1. It consists of 10 datacenters geographically distributed in different countries, different continents. Three of them are in America, two of them are in Canada, and two are in Swiss. The rest three are in China and Japan. Initially, each datacenter contains one room and there are two racks in each room. For each rack, it consists of 5 servers, or physical nodes. Each server has a fixed storage capacity, and it has a fixed bandwidth and processing capacity to serve a certain number of queries in each epoch. It also has fixed replication and migration bandwidth capacities. However, for every server, their capacities are different from each other, according to their own physical condition and other settings. Data are stripped into partitions, which are managed by virtual nodes. The size of every data partition is 512KB. At each epoch, the number of generated queries follows a Poisson distribution with a mean rate λ, which may vary to meet different needs. The detailed setting of environment and parameters can be seen in Table I.

TABLE I
ENVIRONMENT AND PARAMETERS SETTING

Parameter	Default value
Max server storage capacity	10GB
Server storage rate limit	70%
Replication bandwidth	300MB/epoch
Migration bandwidth	100MB/epoch
Epoch	10 seconds
Queries per epoch	Poisson($\lambda = 300$)
Partitions	64
Partition size	512K
Failure rate	0.1
Minimum availability	0.8
α	0.2
β	2
γ	1.5
δ	0.2
μ	1

Besides experiment with random and even query rate, we test our proposed algorithm under flash crowd, and compare its performance with other algorithms. There are four stages in the flash crowd setting. Each stage lasts a quarter of the whole time. In the first stage, 80% of queries are from areas near datacenters H, I and J. And then dramatic change happens. 80% of all queries are near datacenters A, B and C, in the second stage. It moves to the areas near E, F and G in the third stage, and then becomes random and even distributed in the last stage.

B. Replica utilization rate

This experiment is to test the replica utilization rate under different algorithms. Fig. 3(a) illustrates average replica utilization rate in each epoch with random query setting. We can

observe that the random algorithm has the lowest utilization rate, while the RFH algorithm has the highest rate. Because it replicates on nodes which have the most traffic, its replicas can be fully utilized. The replica utilization rate of the request-oriented algorithm is lower than the traffic-oriented one, but it's higher than the owner-oriented and the random algorithm. It replicates on nodes mostly near to frequent requesters, so its replica utilization rate is relatively high. However, obviously, it cannot handle flash crowd well, which we can see in Fig. 3(b).

Fig. 3(b) demonstrates replica utilization rate of all four algorithms under flash crowd setting. The random algorithm has the lowest utilization rate, as it's under random query setting. The request-oriented algorithm achieves poorer performance than that it's under random query setting. The replica utilization rate is lower than that of the owner-oriented algorithm. But worse was to follow, when query changes dramatically, after the 100th epoch, it decreases very obviously. The replica utilization rate drops from about 70% to below 10%, and becomes the lowest of all four algorithms. The reason is that the request-oriented algorithm replicates data on datacenters H, I and J during the first 100 epoch, because most of the queries are near these three datacenters. However, when queries move to other areas, these replicas on datacenters H, I and J have less opportunity to be reached. Although, after adjusting for a long period of time, the replica utilization rate increases, but it's still very low. The owner-oriented algorithm and the traffic-oriented algorithm, apparently, have better performance dealing with flash crowd. They both don't drop at the first query change. For the owner-oriented algorithm, it chooses some nodes near to the ones holding original partition, but not in the same datacenter, so that it can achieve high availability while maintaining relatively low replication cost. Thus, the replica utilization rate of the owner-oriented algorithm counts on network topology a little bit. Ideally, if a node replicates a partition on all its neighbors, all queries for this partition will definitely meet a replica before it reaches the partition holder node. Therefore, the replica utilization rate would be high.

For the RFH algorithm, it achieves the best performance under flash crowd. The replica utilization rate decreases only once, when the "traffic hub" changes. It's because RFH replicates data on those nodes with the most traffic. So if those nodes change in a condition under which a large amount of queries changing in a very short period of time, the rate will drop with it. But it adjusts very quickly, and the rate increases sharply resulting in almost the same good performance as initial.

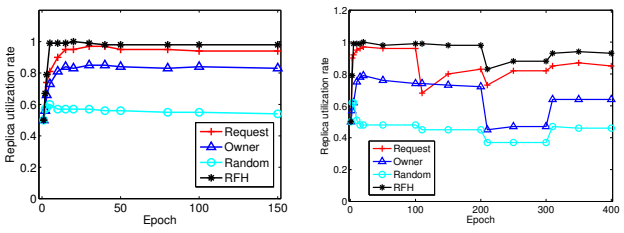

Fig. 3. Replica utilization rate: (a) Under random query. (b) Under flash crowd.

C. Replication cost

This experiment is to test the replica number and corresponding replication cost under two different settings - the random query setting and the flash crowd setting. Fig. 4(a) and Fig. 4(b) illustrate replica number and average replica number per partition, respectively, under random query. To meet the need of the same query workload, the random algorithm has over 500 replicas and has to replicate about 8 times for a partition in average, which achieves the worst performance among all the algorithms. The owner-oriented algorithm performs better than the random algorithm, with about 300 in total and 4.5 in average. The request-oriented algorithm has the best performance with both the lowest total replica number and average number. The curve of the RFH algorithm is quite close to the request-oriented algorithm's, but a little higher. It can address the workload with about 250 replicas in total and each partition has about 4 replicas in average. However, under flash crowd, its performance remains almost the same good as it's under random query, which can be seen in Fig. 4(c) and Fig. 4(d). In the meanwhile, the rest three algorithms, apparently, have to adapt to changing query with more replicas. This result shows that the RFH algorithm has good performance in replica number and has good adaptability under flash crowd especially. Figs. 5(a) and

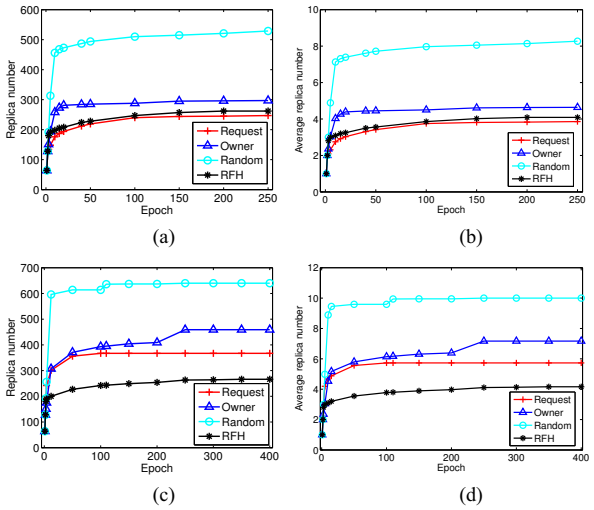

Fig. 4. Replica number. (a)Total replica number under random query. (b)Average replica number per partition under random query. (c)Total replica number under flash crowd. (d)Average replica number per partition under flash crowd.

(b) illustrate replication cost under random query. Fig. 5(a) is of total replication cost while Fig. 5(b) is the average replication cost per replica. We can see that the random algorithm has the highest total and average replication cost, which is much higher than other three algorithms. The total replication cost of the owner-oriented algorithm is close to the request-oriented algorithm's. However, the average cost of the request-oriented algorithm is much higher. It's because the request-oriented algorithm replicates data on nodes near to the requesters, while the owner-oriented algorithm replicates data

on nodes near to the primary partition owners, the replication cost of the former one is definitely higher than the later one. The RFH algorithm achieves both the lowest total replication cost and average replication cost among all four algorithms. It seems unreasonable that the RFH algorithm can achieve lower replication cost than the owner-oriented algorithm, because the later one places replicas mostly on its close neighbors. However, we find it's reasonable after deep analysis. Some replicas are placed on the same datacenter of the primary partition holders, but in different servers. Thus, the replication cost is even lower than replicating on neighbors. It counts on the locations of traffic hub nodes, and therefore counts on the locations of queries based on which the traffic nodes form. We can see this point from the result of flash crowd setting experiment, illustrated in Figs. 5(c) and (d).

Figs. 5(c) and (d) illustrate replication cost under flash crowd. Fig. 5(c) is of total replication cost while Fig. 5(d) is the average replication cost per replica. The performance of the request-oriented algorithm is worse than it is under random query because of long distance replication. The total replication cost of it is lower than the random algorithm's, but the average replication cost is much higher. The average cost of the RFH algorithm is higher than the owner-oriented algorithm's, since the traffic nodes are different from those under random query experiment, resulting in higher cost. Basically, if the primary partition holder is not a traffic node, the average replication cost of the RFH algorithm will be higher than the owner-oriented one's. However, the total replication cost is still the lowest because of fewer replicas.

D. Migration cost

Except the random algorithm, all other algorithms employ migration function. According to the request-oriented algorithm, a virtual node migrates to a server that has much more queries than the former one, and also the server is the closest to the requesters. The migration condition of the RFH algorithm is based on traffic load together with other constrains and benefits. The owner-oriented algorithm chooses migration action when a higher value of availability versus cost can be achieved, and this actually happens only when physical nodes are added into or removed from the system. Fig. 6 is the result of migration times experiment. Figs. 6(a) and (b) are under random query setting, and Figs. 6(c) and (d) are under flash crowd setting. It shows that the request-oriented algorithm has more migration times whether in total times or average times, and whether it's under random query or under flash crowd. Therefore, the RFH algorithm has better performance in migration times experiment.

Figs. 7(a) and (b) demonstrate migration cost under random query. The total migration cost of all four algorithms is illustrated and compared in Fig. 7(a). The request-oriented algorithm has the highest migration cost because it chooses to migrate data to nodes near the requesters. The cost of random algorithm is zero, because no migration function is employed. The cost of the owner-oriented algorithm is zero, for that the migration condition is not reached. The total cost of the proposed RFH algorithm is very low. Fig. 7(b) shows the comparison of the average migration cost per replica among

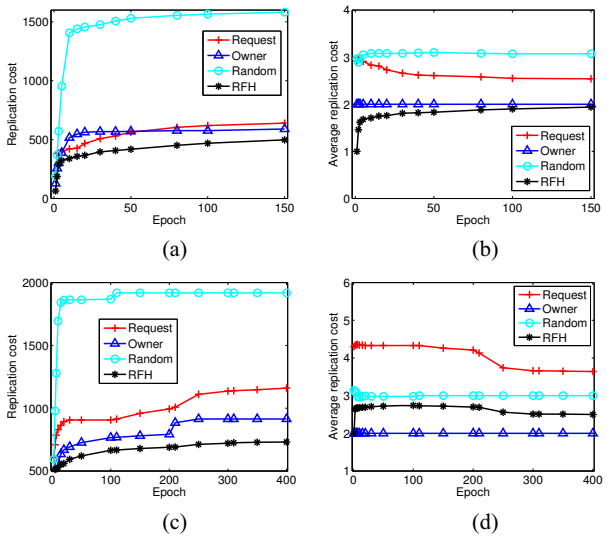

Fig. 5. Replication cost. (a)Total replication cost under random query. (b)Average replication cost per replica under random query. (c)Total replication cost under flash crowd. (d)Average replication cost per replica under flash crowd.

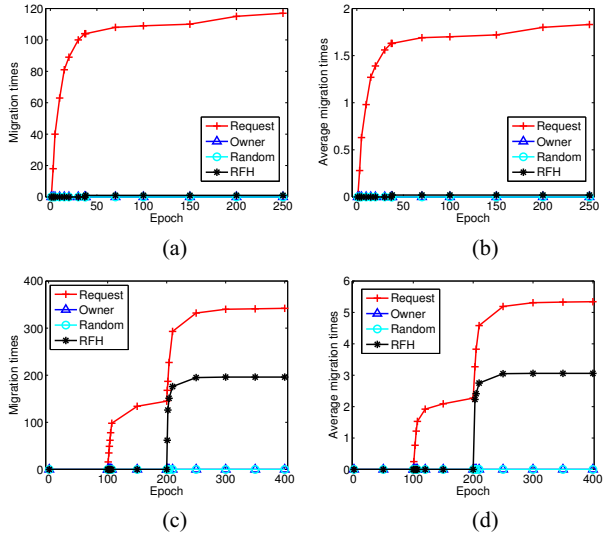

Fig. 6. Migration times. (a)Total migration times under random query. (b)Average migration times per replica under random query. (c)Total migration times under flash crowd. (d)Average migration times per replica under flash crowd.

four algorithms. We can see that the cost of the RFH algorithm is lower than the request-oriented algorithm's.

Under flash crowd setting, Fig. 7(c) shows the total migration cost of different algorithms while Fig. 7(d) is the average cost. By comparing with Figs. 7(a) and (b), we can see that both the total and average migration cost under flash crowd is higher than under random query. Obviously, the reason is that migrating operation increases in order to meet the need of the changing query. The performance of the proposed algorithm is still better than others, note that the random algorithm has no migration function and the migration condition of the owner-oriented algorithm is to achieve maximum availability versus minimum migration cost.

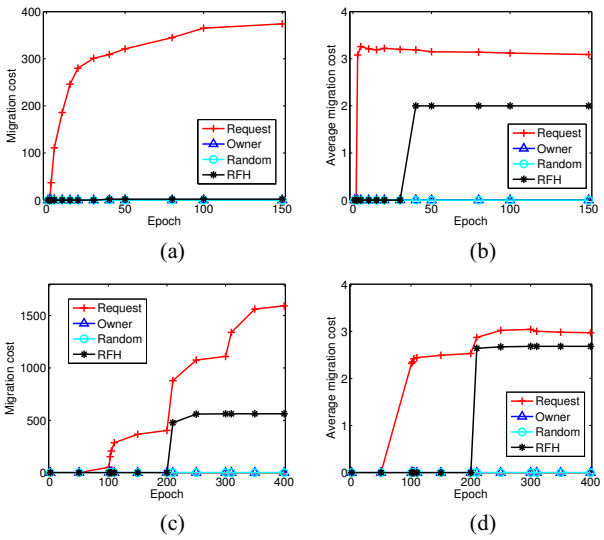

Fig. 7. Migration cost. (a)Total migration cost under random query. (b)Average migration cost per replica under random query. (c)Total migration cost under flash crowd. (d)Average migration cost per replica under flash crowd.

E. Load imbalance

Fig. 8(a) and Fig. 8(b) demonstrate load imbalance performance of each algorithm in each epoch, under random query and flash crowd respectively. With the owner-oriented algorithm, data is replicated on servers to achieve maximum availability, so it would like to choose a rack different from another replica, or at least chooses a different server. The request-oriented algorithm employs random choosing method, which is the same as the random algorithm. The RFH algorithm chooses a server with the least blockibility. So its load balance performance is the best among all the four algorithms. And it achieves better result under flash crowd than under random query, while the performances of other algorithms become worse.

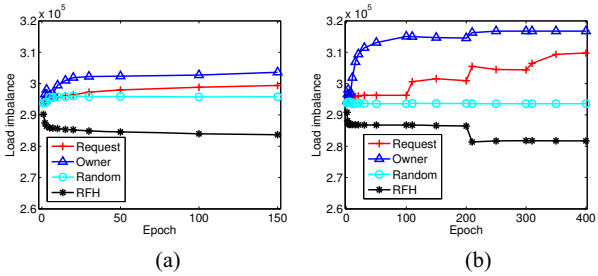

Fig. 8. Load imbalance. (a)Load imbalance under random query. (b)Load imbalance under flash crowd.

F. Lookup path length

Fig. 9 illustrates lookup path length experimental result. Fig. 9(a) is under random query while Fig. 9(b) is under flash crowd. Initially, all curves drop very sharply in both two settings, because replica number increases with the replication action resulting in higher replica hit chance and shorter lookup path length correspondingly. The owner-oriented algorithm is of the longest path length in random query experiment. And it's also the worst under flash crowd, except in the third stage. The reason why it has relatively better performance in the third stage is because most of the queries are near the replicas. But it's still longer than the RFH algorithm's. The RFH algorithm achieves the best performance among all algorithms in both settings, except in the first stage of flash crowd. It has a little longer path length than the request-oriented algorithm's for that most of the queries have a path length of 0 according to the request-oriented algorithm. During the few epochs after epoch 200, there's a sharp angle in the curve of the RFH algorithm. In these epochs, the traffic hub nodes change according to the query condition, so migration process is started by the RFH algorithm and it adjusts very well.

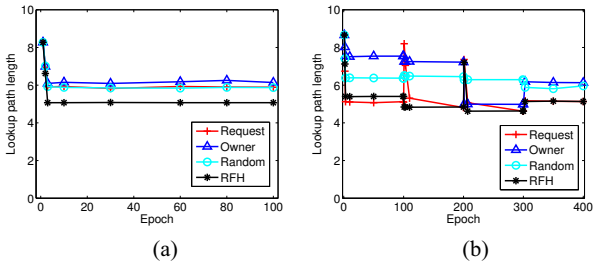

Fig. 9. Lookup path length. (a)Lookup path length under random query. (b)Lookup path length under flash crowd.

G. Node failure and recovery

Node failure is very common in Cloud storage system, and the main purpose of replication is to avoid unwilling consequence, such as data loss, which is brought by a server failure, a rack failure or even a whole datacenter's out of work. And also to allow physical nodes freely join or depart the system is another goal.

Fig. 10 shows the experimental result of node join, failure and recovery of the RFH algorithm. The number of replicas is keep increasing to meet the need of query load at first. Then when the replicas number becomes stable, 30 servers are randomly removed at epoch 290, resulting in a sharp decrease of replicas number. The experiment proves the robustness of the RFH algorithm in the later epochs. The replica number increases as time passes by, and reaches the same level as initial.

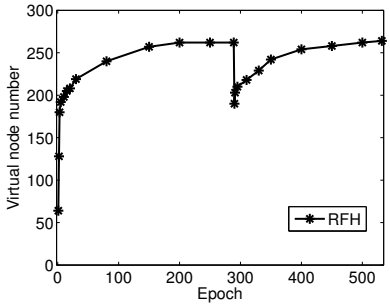

Fig. 10. Node failure and recovery

IV. RELATED WORK

Replication algorithm is widely studied in Cloud storage systems, distributed databases and distributed file systems.

CFS is a cooperative file system that is built on Chord [14]. It replicates blocks of an original file on nodes of the Chord ring, which are immediate after the block's holder. Oceanstore is an architecture designed for global-scale persistent storage which is built on Tapestry [15]. It addresses the consistency problem by serializing replicas updates before applying them atomically. Gnutella [16] is a famous P2P protocol that provides a simple reliable distribution system. When an original file owner is overloaded, it will replicate the file on the file requesters. Gnutella's replication scheme is request-oriented, based on which the proposed RFH algorithm is compared with. Haiying Shen carried out the concept of traffic hub and further proposed an algorithm named EAD to address the problem of data replication in P2P systems [17].

Work on Data Center traffic analysis has been done in [9], which provides us a way to learn its feature of highly irregular. It also introduce how to extend tomography methods for traffic measurement in Data Center Networks. Amazon Dynamo [4] is designed for Amazon e-commerce platform. Distributed file systems, such as Ficus [19] and Coda [20], have replication scheme which can achieve high availability at the expense of relatively lower consistency. To host the state of Googles internal applications, the Google File System [21] uses a single master node to store all the metadata chunks.

Distributed database systems employ data replication algorithms to provide data consistency and efficient data management. In [18], replication techniques based on snapshot isolation are discussed. It points out the conflict between the local concurrency controls providing snapshot isolation and transaction inversions.

V. Conclusion

In this paper, we propose the RFH replication algorithm for distributed Cloud storage systems, which is high-efficient, fault-tolerant and suitable for global wide replication. Different from traditional existing algorithms which randomly replicate to other nodes, the RFH algorithm employs traffic load evaluation to figure out the nodes that are in the traffic hub, and then the decision whether to replicate, migrate or suicide is up to every individual distributed virtual node. We first describe the flash crowd problem and how the RFH algorithm can work to solve it. Then theoretical analysis is given out to prove its effectiveness and also to analyze its performance together with other compared-with algorithms. The simulation results of some experiments confirm our hypothesis and analysis from various aspects, such as replicas utilization rate, migration cost, to name a few. It proves that the RFH algorithm has much better performance than other three algorithms, which can provide Cloud storage systems with high-efficient, fault-tolerant and resilient global replication service. As a future work, we will further study the effectiveness of RFH in real business cases and plan to focus on the research of consistency maintenance.

Acknowledgments

We thank many colleagues for their constructive criticism for this paper, Naixue Xiong is the corresponding author.

References

[1] P. Gill, N. Jain, N. Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *ACM SIGCOMM'11*, Canada, 2011.

[2] K. Chen, C. Guo, H. Wu, J. Yuan, Z. Feng, Y. Chen, S. Lu, W. Wu. DAC: Generic and Automatic Address Configuration for Data Center Networks. *IEEE/ACM Transactions on Networking,* vol.19, no.6, pp.1, 2011.

[3] C. Guo, Guohan Lu, D. Li, H. Wu, X. Zhang, Y. Shi. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *ACM SIGCOMM'09*, Spain, Aug. 2009.

[4] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazons highly available key-value store. In *Proceeding of ACM Symposium on Operating Systems Principles (SOSP)*, New York,USA, 2007.

[5] N. Bonvin, Thanasis G. Papaioannou and K. Aberer. A Self-Organized, Fault-Tolerant and Scalable Replication Scheme for Cloud Storage. In *ACM SoCC'10*, Indianapolis, USA, 2010.

[6] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM Symposium on Theory of Computing*, pp.654–663, May 1997.

[7] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton,D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, vol.35, no.11, pp.190–201, 2000.

[8] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, D. Feng. CDRM: A Cost-effective Dynamic Replication Management Scheme for Cloud Storage Cluster. In *IEEE International Conference on Cluster Computing*, Heraklion, Greece, 2010.

[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *USENIX Internet Measurement Conference (IMC)*, Chicago, USA, 2009.

[10] D. Feng, L. Qin. Adaptive Object Placement in Object-Based Storage Systems with Minimal Blocking Probability. In *Proceeding of the 20th international conference on Advanced Information Networking and Applications (AINA)*, 2006.

[11] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceeding Symp. Operating Systems Principles (SOSP)*, 2001.

[12] F. Dabek et al.. Wide Area Cooperative Storage with CFS. In *Proceeding Symp. Operating Systems Principles (SOSP)*, 2001.

[13] M. Theimer and M. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proceeding Intl Conference Distributed Computing Systems (ICDCS)*, 2002.

[14] I. Stoica et al.. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Networking*, 1(1):17–32, Feb. 2003.

[15] Tapestry, http://tapestry.apache.org, 2011.

[16] Gnutella, http://www.gnutella.com, 2008.

[17] H. Shen. An Efficient and Adaptive Decentralized File Replication Algorithm in P2P File Sharing Systems. *IEEE Trans. On Parallel and Distributed System*, vol.21, no.6, pp.827–840, June 2010.

[18] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proceeding of the 32nd International Conference on Very large data bases (VLDB)*, pp.715-726, Seoul, Korea, 2006.

[19] R. G. Guy, J. S. Heidemann, and J. T. W. Page. The ficus replicated file system. *ACM SIGOPS Operating Systems Review*, vol.26, no.2, 1992.

[20] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, vol.39, no.4, pp.447–459, 1990.

[21] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, New York, USA, Oct. 2003.

[22] D. Borthakur. HDFS Architecture. http://cdh3u0.cloudera.com/cdh/3/hadoop-0.20.2+320/hdfs_design.pdf, April 2009.

[23] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proceeding of 5th USENIX Conference on File and Storage Technologies*, San Jose, USA, Feb. 2007.

[24] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on theory of Computing (STOC)*, New York, USA, 1997.

[25] N. Xiong, A. V. Vasilakos, L. T. Yang, L. Song, Y. Pan, R. Kannan, Y. Li. Comparative Analysis of Quality of Service and Memory Usage for Adaptive Failure Detectors in Healthcare Systems. IEEE JSAC, 27(4): 495 - 509, May 2009.