Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/080154

A Two-layer Geo-cloud based Dynamic Replica Creation Strategy

Zhen Ye[*], *Shanping Li and Junzan Zhou*

Colleague of Computer Science and Technology, Zhejiang University, Hangzhou 310013, P. R. China

Received: 14 Jul. 2013, Revised: 27 Nov. 2013, Accepted: 28 Nov. 2013
Published online: 1 Jan. 2014

Abstract: To improve data availability and reduce user access latency, geo-cloud based data replication is widely used in large global Web sites, such as Facebook. However, as the popularity of data is different and will change as time goes by, simple static replica creation strategies that assign the same number of replicas to all data, never changing thereafter, are not suitable. To this issue, we propose a two-layer geo-cloud based dynamic replica creation strategy called TGstag. TGstag addresses the issue with twofold: policy constraint heuristic inter-datacenter replication and load aware adaptive intra-datacenter replication. TGstag aims to minimize both cross-datacenter bandwidth consumption and average access time with constraints of policy and commodity node capacity. To evaluate the effectiveness of our strategy, we've conducted comprehensive experiments to compare TGstag with other approaches. The results show TGstag significantly reduces cross-datacenter bandwidth and average read access time.

Keywords: Geo-cloud, Data Replication, Dynamic Replica Creation

1 Introduction

As cloud computing is becoming increasingly popular [20], many cloud service providers use tens of geographically dispersed datacenters [1,2]. To increase the availability and improve the performance of applications that are deployed over the geo-cloud, cross datacenter data replication was introduced and has been widely used.

Obviously, with more replicas, higher availability can be achieved and more loads can be served. However, due to constraints on storage capacity, cost and replica consistency, it is unrealistic to assign replicas to all datacenters. Thus, in many cloud-based applications [2, 3], all data have a fixed number of replicas; each replica is assigned to specific datacenters according to the corresponding replica placement algorithms, which we call static replication strategy. In these static approaches, the number of replicas is predefined and the locations of those replicas will never change once they are determined.

However, static replica creation strategy is not suitable in some scenarios. First, every piece of data is unique. Some data may be more important than others and require higher availability or other QoS requirements; some data are more popular than others. Thus it is not suitable to assign the same number of replicas among them. Second, user access patterns will change over time. For example, some data that were important or popular in the past may no longer be, or vice versa. In such situations, increasing or reducing the replica number dynamically will be a good way. In addition, access locations often change from one place to another, which means data replicas need to be reallocated accordingly. Because of the above-mentioned issues, dynamic replication algorithms are widely studied [12,13,14].

When dealing with data replication, compliance in accordance with policy constraints is not unusual. Frequently, in order to achieve certain levels of availability, we need to assign at least n replicas for each data in different datacenters. Additionally, sometimes it is forbidden to place certain data in certain areas or it is necessary for some data to be placed in specific areas.

The current geo-cloud infrastructure usually has two layers: many datacenters are located in different regions or even on different continents; within one datacenter there are thousands of commodity machine nodes. In each node, both storage and load capacity are limited.

With the observation and the consideration of above constraints and scenarios, we propose a two-layer

[*] Corresponding author e-mail: yezhen@zju.edu.cn

geo-cloud based dynamic replica creation strategy called TGstag. As far as we know, we are the first team that has given consideration to both policy and node capacity constraint and has utilized the two-layer datacenter infrastructure when designing a dynamic replica creation algorithm. As regards worldwide systems, cross datacenter bandwidth is a scarce resource and user-perceived access latency is one of the most important factors to be considered. Therefore, TGstag is aiming to minimize cross datacenter bandwidth consumption and average user access latency. Our contribution includes: 1. Proposal of a novel dynamic replica creation strategy composed of two parts: policy constraint heuristic inter-datacenter (Inter-DC) replication and load aware adaptive intra-datacenter (Intra-DC) replication. 2. Extensive experiments have been conducted which prove our strategy quite effective.

The remainder of the paper is organized as follows: Section 2 surveys the related work. Section 3 introduces the background. Section 4 presents TGstag and how the access be served. Section 5 describes our simulation experiment and the obtained result. Finally, the conclusion is provided in section 6.

2 Related Work

Data replica creation strategy has been investigated extensively in the literature. In Windows Azure Storage [2], data is stored durably using local intra-stamp replication, while the geographic inter-stamp replication is used to facilitate disaster recovery. Intra-stamp replication is synchronous replication that is focused on making sure all the data written into a stamp is kept durable within that stamp. It is used to balance a read load. Inter-stamp replication is asynchronous replication used for both keeping a copy of accounts' data in other locations for disaster recovery and migrating accounts' data between stamps.

Kadambi et al. [4] presents mechanisms for selectively replicating a large-scale web database on a record-by-record basis. Its goal is to minimize replication costs while respecting policy constraints. It designs a fine coarse data level policy language that describes which kind of policy constraints the data should obey. Then it proposes a minimal bookkeeping dynamic replication algorithm to migrate replicas dynamically from their original locations to the location with a higher read rate. However, it does not utilize any past access pattern, nor does it consider the constraints on node capacity, thus the prediction is not as accurate as ours.

ecStore [5] is an elastic cloud storage system that provides effective load balancing schema using self-tuning replication technology that is specially designed for large-scale data. ecStore adopts two-tier partial replication to provide high availability and to balance the load. In the first tier, it replicates a small number of replicas for all data objects. In the second layer, ecStore provides additional slave replicas for those hot data. To avoid inefficiency in maintaining access statistics on the data, it uses bucket-based technology to record a suitable range of data statistics, and then uses this record to perform the self-tuning slave replica creation and clearance. In ecStore, the number of secondary replicas is the same for all data. Additionally, it does not consider any policy constraints.

DHR [9] is a dynamic hierarchical replication algorithm that aims to reduce the file access time due to limited storage space in the data grid environment. In DHR, a three-tier hierarchical network topology is presented: different regions, the LANs within each region, and nodes within the same LAN. When the replica is not stored in local nodes, DHR selects the best site with the most number of accesses and then replicates the file remotely into this site. If there is not enough storage, DHR removes the least recently used data replicas that are available both in the current site as well as the local LAN, repeating this process until enough storage is made available. It selects the site that has the fewest number of requests if there is more than one replica in the same level.

In PRCR [10], data are divided into types based on their importance and the storage duration request. One type of data is critical and would be reused over the long term, while another type of data is only used for a short time and has no long-term value. For the first type of data, PRCR stores two replicas in the Cloud where the management of two replicas is based on a proactive checking methodology. For the second type of data, only one replica is stored in the cloud.

DPRSKP [11] is a periodic replication strategy that aims to select the best candidate files for replication, placing them in the best locations, assuming limited storage for replicas. For each site, the DPRSKP strategy selects replicas to be created or deleted based on the knapsack algorithm.

3 Background

In this paper, we considered a scenario that there are several datacenters distanced from each other (in different regions or even on different continents) and each data center contains many commodity nodes. Each node has constraints on both storage and load capacity. Once the storage or load exceeds its capacity, it replicates data and forwards a request to other nodes within the same datacenter. Figure 1 presents this geo-cloud network topology.

The data replicas are divided into two categories from the view at the datacenter level: the primary replica and the secondary replica. All write requests to a data object are first sent to the primary replica and then propagate to the secondary replicas. These propagation processes are all asynchronous. In this paper we do not consider replica consistency issue. However, read requests can be served

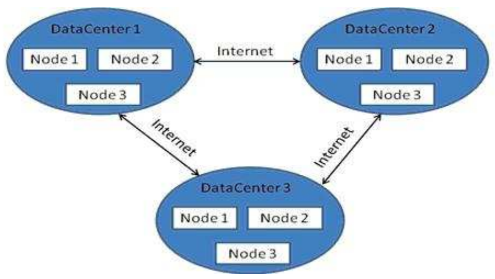


Fig. 1: Geo-cloud topology

by any replica. In this situation, write request latency is determined only by the datacenter to which the primary replica is assigned. However, our main concern in this paper is when to create secondary replicas (and slave replicas, which we will introduce in section 4.2) and how many replicas we need to create, which only affect read access latency. Thus we have chosen to optimize read access latency as one of our goals. As we can see in the formula (1), total cross datacenter bandwidth is the sum of update propagation bandwidth and remote read access bandwidth. Update propagation bandwidth is the bandwidth produced when the primary replica replicates data to secondary replicas. Remote read access bandwidth is yielded when the local datacenter cannot serve the read request and needs to retrieve data from remote datacenters.

$$BW_{total} = BW_{update} + BW_{read} \tag{1}$$

Our work is based on the observation that most applications have temporal and geographical locality [7]. In the theory of temporal locality, the data accessed most recently will likely be accessed again in the near future. The geographical locality states that once a client requests the data, it is possible that clients nearby will also request it. By taking into account these two localities, we can utilize past access statistics to predict the near future users' access patterns.

4 TGstag

TGstag is comprised of two parts: policy constraint heuristic Inter-DC replication and load aware adaptive Intra-DC replication. We will introduce these parts more in detail in 4.1 and 4.2. We will also present in section 4.3 how users' write and read access being handled.

4.1 Policy constraint heuristic Inter-DC replication

The first portion of TGstag is to create data replicas between different datacenters, aiming to increase both availability and read access speed. When replicating data into different datacenters, we also need to obey predefined policies.

In this paper, we consider three types of policies: 1. A policy regarding the minimal number of replicas to ensure availability and cross datacenter latency. The more important data will have the greater number of replicas; 2. An include list policy that defines where in the datacenters the data replicas must be placed; 3. An exclude list policy that forbids placing data replicas into specific datacenters. Our policy solutions are flexible thus can support more policies with very little extra effort if necessary.

There were no access statistics at the time we created the data object, thus we used a simple heuristic algorithm to assign the replicas of this object into different datacenters initially.

Algorithm 1 shows the detail procedure to create initial replicas. For each data object *o*, first check if any policies related to *o* conflict with each other. E.g., if the value of the required minimal replicas of *o* is larger than the total number of datacenters, or if the include list and the exclude list have any datacenters in common. If there are conflicted policies, we have two options: 1. Tell user to modify the policies in the configuration file to solve these conflicts; 2. Set priority to each policy and the higher priority policy will override the lower one when they are in conflict. If there is no conflict, create a primary replica in the local datacenter of *o* and select a node with enough storage that has the lowest load to store this primary replica. Then we find $n-1$ number of datacenters nearest to this datacenter and obey the policies, where n is the minimal number of replicas of *o*. Finally, we propagate *o* into those $n-1$ datacenters.

Algorithm 1 Initialize replicas

```
1: $result \leftarrow policyValidation(o)$
2: if result is true then
3:     $createPrimaryReplica(o)$
4:     $n \leftarrow getMinReplicasNumber(o)$
5:     $selectedDCs \leftarrow$
           $findNearestDCs(n-1, policies, dcs, currentDC)$
6:     $propagateReplica(o, selectedDCs)$
7: end if
```

Once object *o* has been created, the number and locations of its replicas will be re-determined every specific time α, by running the Inter-DC replication process. Access statistics regarding *o* will be sent to the node that contains the primary replica of *o*. Then this node will run the Inter-DC replication.

Algorithm 2 describes the Inter-DC replication procedure. Firstly, the datacenters that exist on the include list will be added into R'_o, which are the datacenters where object *o* will be replicated after the procedure. It then checks the remaining datacenters that are also not on the exclude list. Two factors will be used to consider whether a replica will be created in datacenter dc. The first is the read request number $readNumber_{dc,o}$, and the other is the latency between dc and the datacenter where the primary replica of o exists, called l_{dc,dc^o}. The larger $readNumber_{dc,o}$ is, the greater the chance it will get a replica. For l_{dc,dc^o}, the larger value means if there is no replica in dc, it will cost more to retrieve the data from dc^o, so we need to give it a greater chance to be replicated. The algorithm adds dc to R'_o only if $readNumber_{dc,o}*log(l_{dc,dc^o})$ is larger than $writeNumber_o*remoteReadRatio_o*c$, where $writeNumber_o$ is the number of write requests against o within the specific α time, $remoteReadRatio_o$ is the remote read ratio among all read access, c is the adjustable Inter-DC replication threshold factor. If after that, the number of R'_o is still lower than its minimal requirement, the remaining ones will be selected from the nearest datacenters. After we get this R'_o, we then compare it with the original datacenters where o has been replicated before. Finally, replicas of o will be propagated to the newly selected datacenters and removed from datacenters that have not been chosen.

Algorithm 2 Inter-DC replication

```
1:  $R'_o \leftarrow getIncludeList(o)$
2:  for each $dc \in datacenters$ do
3:      if $dc \notin getIncludeList(o)$ and $dc \notin getExcludeList(o)$ then
4:          if $readNumber_{dc,o} * log(l_{dc,dc^o})$ larger than $writeNumber_o * remoteReadRatio_o * c$ then
5:              $R'_o \leftarrow R'_o \bigcup dc$
6:          end if
7:      end if
8:  end for
9:  if (number of $R'_o$) less than $minReplicasNumber_o$ then
10:     $R'_o \leftarrow minRep(R'_o)$
11: end if
12: for each $dc \in R'_o$ and $dc \notin R_o$ do
13:     $propagateReplica(o,dc)$
14: end for
15: for each $dc \in R_o$ and $dc \notin R'_o$ do
16:     $removeReplica(o,dc)$
17: end for
```

In the Inter-DC replication, the value of c has a significant impact. If this value is set large, it means only those datacenters that have a larger number of read requests or are far away from the primary replica will have a replica. This leads to the replica numbers becoming smaller, which will result in higher read access latency and read access cross-datacenter bandwidth. At the same time, the storage costs and the updated propagation bandwidth will have a relatively low value. If we set c to be a small value, the situation will be the opposite. This is a trade off decision; in the experiment we will see how this c value affects the TGstag's result.

4.2 Load aware adaptive Intra-DC replication

Once the replica is assigned to a specific datacenter, it will be stored in a node with sufficient storage that has the lowest load. We refer to this kind of replica as the secondary replica.

When a node's load exceeds its threshold value, it triggers an Intra-DC replication process. We call the replica created by Intra-DC replication the slave replica.

Before performing the Intra-DC replication, we rate each data based on three factors: the total request number of the data, the access frequency of the data and last access time of the data. Formula (2) is used to calculate the weight of each data o.

$$weight_o = c_1 * rn_o + c_2 * rnw_o/interval + c_3/(currentTime - lastAccessTime_o) \tag{2}$$

rn_o is the request number of o since the last Inter-DC replication. This number will be reset to 0 once the Inter-DC replication is done. $interval$ is the specific time interval used for our frequency calculation usage. rnw_o is the request number within that interval. c_1, c_2 and c_3 are constant coefficients used to do the normalization.

Algorithm 3 shows how Intra-DC replication works. To begin with, all data will be sorted by their weight values. Then we choose the top k ratio of the object as the candidate to replicate, e.g. the top 5%. The selected data will be replicated to the nodes with both sufficient storage and the lowest loads. If the current node is the slave replica, then the information in this newly created slave replica must be sent to the secondary replica, so the secondary replica can propagate updated values to the new slave replica when it receives one from the primary replica.

When one node's data access rate falls below a specific level, it will remove this slave replica and notify the related secondary replica.

As in the data storage node, the bottleneck is I/O; using CPU resources to calculate weight will not cause serious issues to the entire system. The exceeded threshold scenario will not occur very often, so it will not significantly impact the system's overall performance. In order to make these per record statistics less expensive, we use Hierarchical Timing Wheels [6] to reduce storage overhead.

Algorithm 3 Intra-DC replication

```
1: $objectIDs \leftarrow sortByWeight(weights)$
2: $candidateIDs \leftarrow selectObjectIDs(objectIDs, topNRatio)$
3: $newSlaveReplicaNode \leftarrow selectCandidateNode()$
4: if current node is slave replica then
5:     $notifySndReplica(candidateIDs, newSlaveReplicaNode)$
6: end if
7: $replicate(candidateIDs, newSlaveReplicaNode)$
```

4.3 Replica access

As discussed in section 3 and section 4.2, there are three types of replicas: the primary replica, the secondary replica and the slave replica. Once there is an update request for data o, it will update the primary replica first and then propagate the update message to all secondary replicas in other datacenters. Once the secondary replica receives this update, it forwards the update to any slave replicas within that datacenter.

Regarding read access, the situation becomes more complex. When user contact node n_i in datacenter dc_j for data o, first it checks if there exists some replicas of o in dc_j. If find, it will request the one with the lowest load. If there is no replica within dc_j, then it will go to the nearest datacenter that has the replica of o. At last, it needs to record related statistics for the future usage. Algorithm 4 presents the detail procedure.

Algorithm 4 Replica read access

```
1: if there is a replica of $o$ in local datacenter then
2:     $node \leftarrow findSuitableNode(o)$
3:     $forwardRequest(o, node)$
4: else
5:     $dcID \leftarrow findNearestDC(o)$
6:     $forwardRequest(o, dcID)$
7: end if
8: $updateStat(o)$
```

5 Experiments

We have conducted several experiments to compare different replica creation strategies. This chapter describes our experiments and the results in detail.

We modeled a social network application and used it as our workload. The number of users' friends follows a zipf distribution. Their write and read access frequency also follows this distribution, which matches the real workloads [15]. We set all data with an identical size of 1k.

Our experiment is based on CloudSim [8], a widely used framework for the modeling and simulation of cloud computing platforms and services. We created eight for our experiments, to emulate the number of Amazon EC2 datacenters [17], and to simulate a geo-cloud environment that spans the world. The access latencies among these datacenters are all the real values we obtained from the Amazon EC2 platform by creating a micro instance node in each region, all of which ping each other. The detailed values can be seen in Table 1. In CloudSim, we made each datacenter with as many nodes as could be created on demand. Each node has fixed storage size and limited access frequency capacity.

Table 1: Latencies between Datacenters in Amazon EC2

	DC1	DC2	DC3	DC4	DC5	DC6	DC7	DC8
DC1	0	87	83	90	255	201	271	150
DC2	87	0	21	177	213	139	219	203
DC3	85	22	0	171	198	129	216	197
DC4	90	175	171	0	354	285	445	223
DC5	256	221	198	360	0	81	221	370
DC6	201	137	119	285	91	0	172	310
DC7	151	197	184	412	189	141	0	330
DC8	142	203	197	224	374	403	368	0

We created a default policy wherein every object must have a minimum of two replicas; also, we have specified the primary replica must be included in the Include List.

We compared our approach to two other approaches: Fixed Number replication (Fix Number) and Dynamic Constraint-based replication (DCR). In the Fixed Number approach [16], each object has a fixed number of replicas, named n (we set n=3 as the default value) as used in Dynamo [1], Cassandra [3], and many other large applications. Each datacenter is identified by a 128-bit integer and is then mapped into an integer ring. The data will be assigned to one datacenter as the primary replica based on its ID and is replicated to the following (n-1) datacenters in that ring. In the DCR approach [4], data will be retrieved from the remote primary replica; a secondary replica will be created when there is a read request from that datacenter. There is retention interval I; within I the replica will always exist. After I, if the next request in this datacenter is read, the last access time of that replica will be updated. If the next request is an update request from the primary replica, then this replica will be removed. This I has an important effect on the result. If it is too short, the replica will be created and removed too often, which will result in higher costs. If the value is too long, the updates need to propagate data to a number of different datacenters even if there is not another read in those datacenters. In this experiment, we adjust I into a suitable value based on this scenario, making it reach an optimal point from both a read access latency perspective and cross datacenter bandwidth consumption. In our experiment, we modeled the arrival of the requests as a Poisson Process, which is a common way [18, 19].

In this experiment, we set the following default values. As the same with other work [4], the write access proportion is set to 0.1, and the probability of remote friends is set to 0.1. We conservatively set the overload request percentage to 0, as we can see in section 5.6, when this overload ratio value increases, TGstag can perform even better compared to other algorithms. Because a large access pattern shift in a short amount of time is not likely [4], we set 10% as the default access pattern change percentage. We varied one factor value and kept the other factors unchanged in one group of the experiment.

In this paper, we used average read latency and cross datacenter bandwidth as our evaluation criteria.

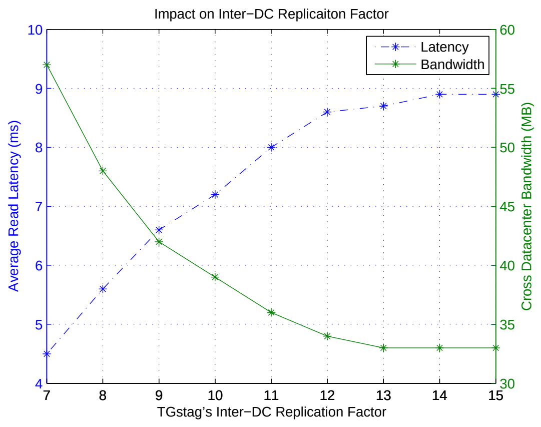


Fig. 2: Impact on Inter-DC Replication Factor

5.1 Impact on Inter-DC replication factor

The first experiment we conducted was to observe the impact on the Inter-DC replication factor. In this experiment we varied the Inter-DC replication factor *c*, thus varying the number of replicas for each object remaining in the write/read request ratio and remote read access percentage. Since it is not applicable to the Fix Number and DCR approach, we've only presented how this change affects TGstag here.

From Figure 2 we can see as factor *c* increases, TGstag's access latency increases while the bandwidth consumption decreases. This is because when c increases, the average object replica numbers decrease, thus more requests have to contact remote datacenters, resulting in the increase of average read access latency. However, as the replica numbers decrease, the update propagation bandwidth consumption likewise decreases. Also, when the replica numbers decrease, the storage needs also decrease. There is a resulting tradeoff between latency and bandwidth & storage when choosing this value.

We also can see in Figure 2, after *c* reaches a certain value, even if we continue increasing the value of *c*, there will be no obvious changes to the latency or bandwidth. This is because once *c* exceeds a specific value, the replica numbers do not decrease as they need to obey the minimal replica number policy. In order to see how the value of *c* affects the result, in the following experiment we picked three values (*c*=9,10,11 as default) and applied them in TGstag. So in each experiment, we have 5 algorithms to compare: Fix Number, DCR and three TGstag algorithms with different values of *c*.

5.2 Impact on write access proportion

As different kinds of applications have different write/read proportions, in this experiment we tried 3 groups with write proportions to be 0.025, 01 and 0.2 respectively.

Table 2: Result on different write proportion

	Fix Number	DCR	TGstag20	TGstag30	TGstag40
Latency(ms)	19.3	11.2	2.2	2.9	4.8
Bandwidth(MB)	34	23	20	19	17

(a) Write proportion is 0.025

	Fix Number	DCR	TGstag20	TGstag30	TGstag40
Latency(ms)	19.0	16.0	6.6	7.2	8.0
Bandwidth(MB)	63	43	42	39	36

(b) Write proportion is 0.1

	Fix Number	DCR	TGstag20	TGstag30	TGstag40
Latency(ms)	21.5	18.5	8.9	10	10.4
Bandwidth(MB)	102	69	70	63	58

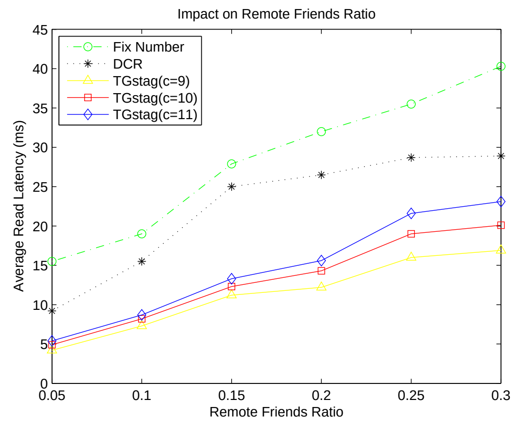
(c) Write proportion is 0.2

We can know from the Inter-DC replication algorithm introduced in section 4.1 and described in Algorithm 2, when a write proportion is set to a large value, in order to reach the same value of the replica number, the threshold factor c should also be set to a relatively large value. For a low write proportion, the value should be a small one. In this experiment we selected the Inter-DC replication factor *c* from different ranges of values in different write proportion scenarios.

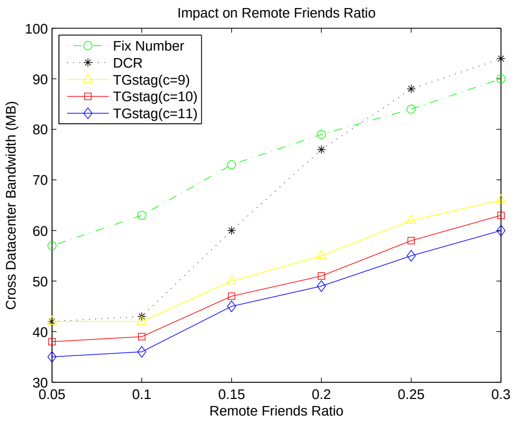
Tables 2(a), (b) and (c) present the detail results. We've named the TGstag series of algorithms as "TGstag" + "value of *c*", e.g. TGstag20 means TGstag with *c*=20. From Table 2 we can tell for each experimental group, TGstag performs better than the Fix Number and DCR approaches after adjusting *c* to suitable values.

The second row of Tables 2(a), (b) and (c) show the latency results. For the Fix Number replication, the average read latency only increased a little with an increased write proportion. This is because neither its replica number nor the locations of these replicas have anything to do with the write proportion. For the DCR, as the write proportion increases, when replica exceeded its retain interval *I*, the chance that next operation it meet is write request will increase, in this situation the replica are prong to be removed and thus need to contact remote primary replica more frequently, which result the higher read access latency. For TGstag, as the write proportion increases, the numbers of datacenters with read request frequencies that exceed the threshold then decreases, thus the number of Inter-DC replicas decrease. This means more read requests need to contact remote datacenters, resulting in an increase of read request latency.

The third row of Tables 2(a), (b) and (c) presents the bandwidth results. For Fix Number approach, as the write ratio increases, the update propagation bandwidth likewise increases, while the remote read access bandwidth remains the same; thus the total cross datacenter bandwidth increases at a constant speed. For both DCR and TGstag, as the write proportion increases, more read requests then contact the remote datacenter, which makes the remote read access bandwidth grow more rapidly. As a result, although the update propagation

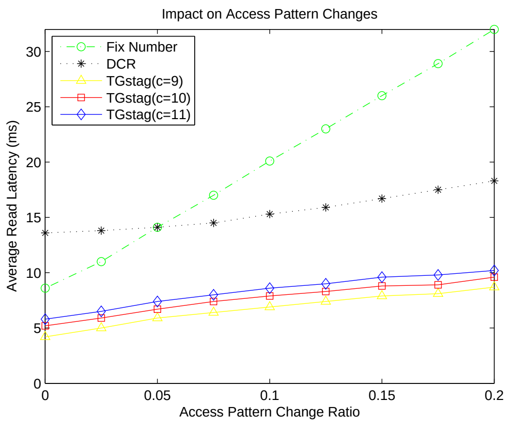


(a) Latency

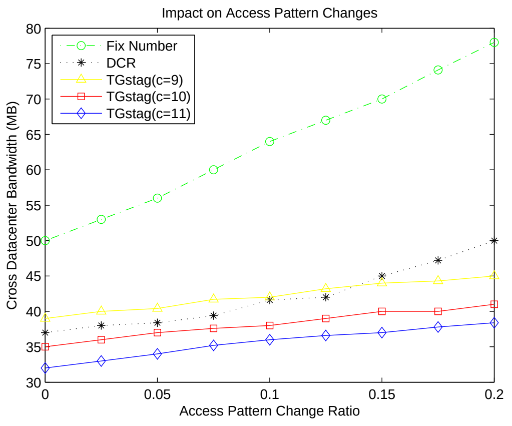


(b) Bandwidth

Fig. 3: Impact on Remote Friends' Ratio



(a) Latency



(b) Bandwidth

Fig. 4: Impact on Access Pattern Changes

bandwidth consumption may not increase or decrease, the total bandwidth they use still grows very quickly.

5.3 Impact on remote friends' percentage

In this experiment, we increased the remote friends' percentage gradually from 0 to 0.3, without changing other factors, seeing how this factor affects the result.

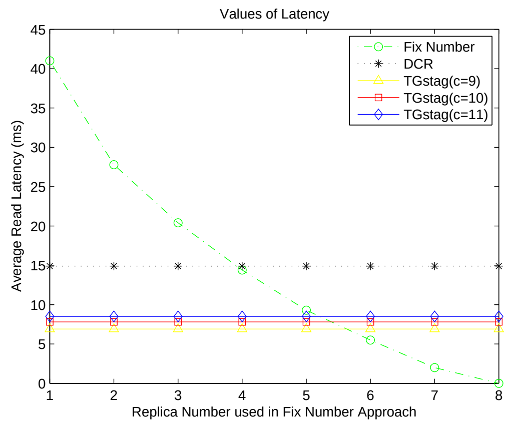
Figure 3(a) shows the latency result. For Fix Number replication, replica numbers and locations remain the same. As the ratio of remote friends' increase, remote read requests likewise increase, resulting in the average read latency to increase linearly. For the DCR approach, at first the remote read access numbers increase. However, as this ratio keeps increasing, when a read request comes from a specific datacenter, it is likely there has already been a secondary replica retrieved and stored in that datacenter before. Therefore, there is no need to contact a remote datacenter to store the primary replica. As a result, the read access latency growth rate becomes slower. For TGstag, the situation is similar. At first, while the remote friends' ratio increases, the average read latency also increases linearly. But when it reaches a certain level, while it is still increasing, it creates more Inter-DC replicas, resulting in the read latency growth occurring much more slowly than the Fixed Number approach.

Figure 3(b) shows bandwidth consumption results. As the ratio of remote users increase, the update propagation bandwidth consumption and remote read access bandwidth of all three approaches increase.

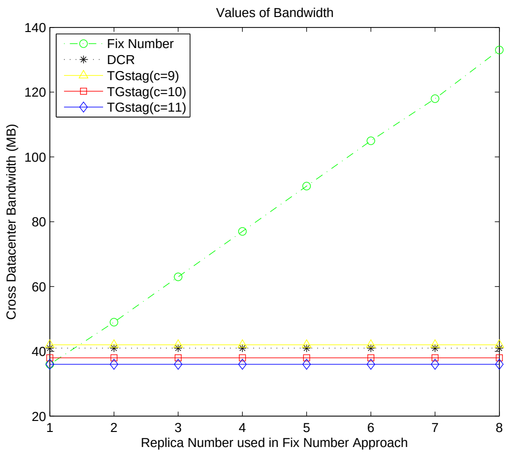
5.4 Impact on user access pattern changes

In this experiment we see how changes in user access patterns affect the result.

By referring to an access pattern change, we mean the location of a user's friends' changes from one datacenter to another. We varied the percentage of total read requests that change from 0.0 to 0.2. From Figure 4(a), we can see that as this access pattern change ratio increases, the Fix Number replication's average latency increases quickly.

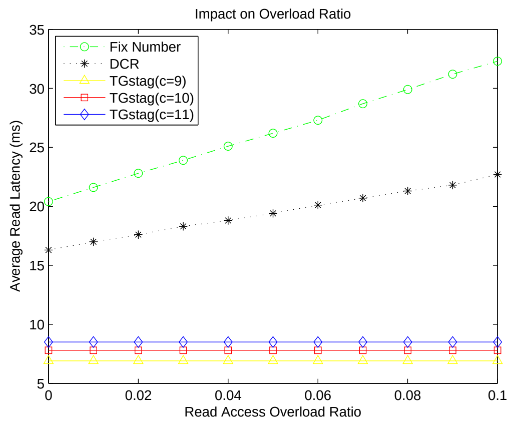


(a) Latency

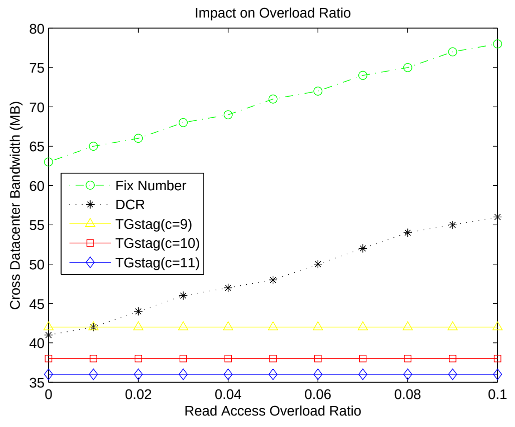


(b) Bandwidth

Fig. 5: Impact on Number of Fix Replicas



(a) Latency



(b) Bandwidth

Fig. 6: Impact on Overload Ratio

For DCR, because it is an online algorithm, it can adjust dynamically as the access pattern changes. But based on the definition of access pattern changes we have used here, when this value becomes larger, the percentage of remote friends increases slowly, so the latency of the DCR approach still increases slowly. For TGstag, when the access pattern changes increase, more read requests, which could have been served locally before, now need to contact a remote datacenter, resulting in the average latency increase.

Figure 4(b) presents the impact on cross datacenter bandwidth consumption. For Fixed Number replication, as the change percentage increases, the remote read access number increases greatly; therefore the total bandwidth consumption likewise increases quickly. For DCR and TGstag, because they can adjust both the number and the location of replicas dynamically, their bandwidth consumption grows much more slowly than Fix Number replication.

5.5 Vary number of replicas in Fix Number replication

For Fix Number replication, as the number of replicas increase, the average read latency reduces, but the cross datacenter bandwidth and storage costs both increase. In this experiment, we see the impact of changing the numbers of these fixed replicas.

We varied the number of replicas in the Fix Number replication from 1 to 8, and then compared it with DCR and TGstag. In reality, this experiment is not applicable to DCR and TGstag because they do not adopt the static fixed number replica strategy; however, in order to make the comparison results intuitive, we must include their results in Figure 5. From Figure 5(a), we can see when the fixed replica numbers are 4 and 5, the average read latency is almost the same as DCR and TGstag.

Figure 5(b) displays the bandwidth used by Fixed Number approach as the replica numbers increase. As it increase, the Fixed Number approach uses more cross datacenter bandwidth. This is because in every write request, the primary replica needs to propagate more secondary replicas when the fixed number increases. When the replica number is 5, the cross datacenter bandwidth Fixed Number approach used is much higher than that of TGstag (in this situation, their average read latency is the same as described in previous paragraph).

5.6 Impact on overload ratio

In the above experiment, we set the overload request percentage to 0, which means we will not utilize TGstag's Intra-DC replication. In this experiment, we varied the overload ratio from 0 to 0.1 and compared TGstag to the other two algorithms in each situation to see how this affects the result. Figure 6(a) shows the latency results. As the overload ratio increases, the Fixed Number and DCR approaches require that more local requests be made to other datacenters, causing an increase in latency. For TGstag, when the overload situation occurs, it replicates the object into other nodes within the same datacenter without having to contact other datacenters, so the average read latency remains the same. The results for bandwidth are similar to latency. Figure 6(b) presents these results in detail.

6 Conclusion

In this paper, we explored a two-layer geo-cloud based dynamic replica creation strategy and proposed a novel approach called TGstag. TGstag includes two parts: policy constraint heuristic Inter-DC replication and load aware adaptive Intra-DC replication. It aims to minimize cross-datacenter bandwidth consumption and average read access times with constraints of policy and commodity node capacity. The experiment results prove the effectiveness of TGstag.

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, Dynamo: Amazon's Highly Available Key-value Store, SOSP, (2007).

[2] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Fahim ul Haq, M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency, SOSP, (2011).

[3] A. Lakshman, P. Malik, Cassandra: A Decentralized Structured Storage System, SIGOPS, (2010).

[4] S. Kadambi, J. Chen, B. F Cooper, D. Lomax, R. Ramakrishnan, A. Silberstein, E. Tam, and H. Garcia-molina, Where in the World Is My Data?, VLDB Endowment, (2011).

[5] H. Vo, C. Chen and B. Ooi, Towards Elastic Transactional Cloud Storage with Range Query Support, Proc. VLDB Endow., **3**, 506-514 (2010).

[6] G. Varghese, Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility, IEEE/ACM Transactions on Networking, **5**, 824-834 (1997)

[7] D Sun, G. Chang, S. Gao, L. Jin, and X. Wang, Modeling a Dynamic Data Replication Strategy to Increase System Availability in Cloud Computing Environments, Journal of Computer Science and Technology, **27**, 256-272 (2012).

[8] R. Calheiros1, R. Ranjan, A. Beloglazov, C. De Rose, and R. Buyya1, CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms, Software: Practice and Experience, **41**, 23-50 (2011).

[9] N. Mansouri, and G. Hosein Dastghaibyfard, A dynamic replica management strategy in data grid, Journal of Network and Computer Applications, **35**, 1297-1303 (2012).

[10] W. Li, Y. Yang, J. Chen, D. Yuan, A cost-effective mechanism for Cloud data reliability management based on proactive replica checking, 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, (2012).

[11] H. Chettaoui, and F. Charrada, A Decentralized Periodic Replication Strategy based on Knapsack Problem, Grid Computing, (2012).

[12] V. Andronikou, K. Mamouras, K. Tserpes, D. Kyriazis, and T. Varvarigou. Dynamic QoS-aware Data Replication in Grid Environments Based on Data importance, Future Generation Computer Systems, **28**, 544-553 (2012).

[13] Q. Wei, B. Veeravalli, B.o Gong, L. Zeng, and D. Feng. CDRM: A Cost-Effective Dynamic Replication Management Scheme for Cloud Storage Cluster, IEEE International Conference on Cluster Computing, 188-196 (2010).

[14] Z. Wang, T. Li, N. Xiong, and Y. Pan, A novel dynamic network data replication scheme based on historical access record and proactive deletion, The Journal of Supercomputing, **62**, 227-250 (2012)

[15] A. Silberstein, J. Terrace, B. Cooper, and R. Ramakrishnan, Feeding frenzy: selectively materializing users' event feeds, SIGMOD, 831-842 (2010)

[16] G. Candia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, Dynamo: Amazon's Highly Available Key-value Store, SOSP, 205-220 (2007)

[17] Amazon, EC2, http://aws.amazon.com/ec2/, (2013).

[18] H. Chihoub, S. Ibrahim, G. Antoniu, M. Prez, Consistency in the Cloud:When Money Does Matter!, Technical Report: http://hal.archives-ouvertes.fr/hal-00756314/, (2012).

[19] T. Kraska, M. Hentschel, G. Alonso and D. Kossmann, Consistency Rationing in the Cloud: Pay only when it matters, Proceedings of the VLDB Endowment, **2**, 253-264 (2009)

[20] W. Chen, K. Yin, D. Yang, M. Hung, Data Migration from Grid to Cloud Computing, Appl. Math. Inf. Sci., **7**, 399-406 (2013)

Zhen Ye received B.S. degree in computer science and technology from Zhejiang University, Hangzhou, China in 2007. He is currently a Ph.D candidate under supervision of Prof. Shanping Li. His research interests include Geo-cloud replication, NoSQL and distributed consensus algorithms.

Shanping Li received the M.S. and Ph.D. degrees in the computer science and technology from Zhejiang University, Hangzhou, China in 1987 and 1993, respectively. He is currently a professor in the College of Computer Science and Technology at Zhejiang University. His research interests include the distributed computing, very large information system, operation system and software enginering.

Junzan Zhou received B.S. degree in computer science and technology from Zhejiang University, Hangzhou, China in 2008. He is currently a Ph.D candidate under supervision of Prof. Shanping Li. His research interests include performance testing, anomaly detection and cloud computing.