SLO-aware Deployment of Web Applications Requiring Strong Consistency using Multiple Clouds

Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya
Cloud Computing and **D**istributed **S**ystems (CLOUDS) Laboratory
The University of Melbourne, Australia
Email: cqu@student.unimelb.edu.au, {rnc, rbuyya}@unimelb.edu.au

***Abstract*—Geographically dispersed cloud data centers (DCs) enable web application providers to improve their services' response time and availability by deploying application replicas in multiple DCs. To allow applications requiring strong consistency to be deployed in multiple clouds, industry and academia have developed various scalable database systems that can guarantee strong inter-DC consistency with alleviated network overhead. For applications using these database systems, it is essential to take both the network latencies to the end users and the communication overhead of the databases into account when selecting the hosting DCs. In this paper, we study how to identify the satisfactory deployment plan (hosting DCs and request routing) considering SLO satisfaction, migration cost, and operational cost for applications using these databases. The proposed approach involves two steps. First, it searches the deployment plan with minimum amount of SLO violations using genetic algorithm when the application is first migrated to the clouds. Then it continuously optimizes the deployment in a certain time interval according to the changing workload and the current deployment plan. We illustrate how our approach works for the applications using two databases (Cassandra and Galera Cluster), and demonstrate the effectiveness of our approach through simulation studies using settings of two example applications (TPC-W and Twissandra). Our solution is extensible to applications using other database systems that have similar properties.**

***Keywords*—Multi-cloud, Geographically Dispersed, Web Application, Database Consistency, Deployment, Request Routing**

I. Introduction

Web application providers always concern about how to provide high Quality of Service (QoS) (i.e., low latency and high availability) to their end users [1]. With the maturity of Cloud computing, it has become the ideal platform for hosting web applications, as it not only enables infrastructure to be scaled up and down according to the real-time traffic, but also allows easy replication of applications in geographically dispersed data centers (DCs) so that customers all around the world can be served with high QoS.

However, when applications are deployed in geographically dispersed cloud DCs, providers need to handle data consistency across DCs (inter-DC consistency), which is challenging for some applications requesting strong consistency, e.g., e-commerce, and banking applications. To make things worse, traditional inter-DC commit (two-phase commit) involves high network cost. Therefore, applications often have to adopt eventual consistency (asynchronous replication) in order to minimize user perceived latencies. This complicates application logic and forces application developers to handle the conflicts and errors caused by the inconsistent data [2], even though such cases are rare in production as data synchronization usually happens quickly in eventual-consistent databases [3].

After realizing that lack of strong consistency has impaired developing productivity, industry and academia shift to develop new databases that can guarantee strong inter-DC consistency [2], [4]–[10] to help relive the programmers' coding burden. Though the inter-DC consistency protocols of these new databases are often optimized in terms of network overhead, the resulted network delays are still significant and cannot be ignored. Thus, to minimize the user perceived response time, it is essential to take the database network delay into account when selecting the hosting DCs and when routing requests originated from different users to the chosen DCs.

In this paper, we aim to minimize the total excess response time the users may perceive beyond the SLO for applications with various inter-DC consistency requirements. The proposed approach benefits the application providers so that they can enjoy the agility of development brought by the new databases, and in the meantime keep the extra latencies as low as possible.

The **key contributions** of the paper are two folded. First, we propose a genetic algorithm (GA) that searches a deployment plan (set of DCs and request routing) with minimum amount of SLO violations when the application is initially migrated to the cloud. After the initial deployment, the application performance may degrade as time passes due to change of workload distribution. Second, we propose a decision-making algorithm that continuously optimizes the deployment in order to balance application performance, redeployment cost, and operational cost under changing workloads. We exemplify how of approach works with two widely used databases (Cassandra [5] and Galera Cluster [4]). To demonstrate the effectiveness of our approach, we conduct simulation studies using settings of two real applications (TPC-W [11], an e-commerce website, and Twissandra [12], a twitter-like social network application).

The rest of the paper is organized as follows. Section II briefly surveys the existing protocols and databases with strong inter-DC consistency support. Then we describe the target applications and their deployment model in Section III. Section IV explains our approach followed by the performance evaluation. After that, we discuss some key issues and pitfalls when extending and using our approach in Section VI. Finally, we present the related work and conclude the paper.

II. Survey of Inter-DC Consistency

A. Consistency Protocols

1) Two-phase Commit: Two-phase commit is the simplest protocol that implements inter-DC transaction commit. Its basic idea is to use one message round to reach consensus of whether to commit or rollback among all the participating processes and another round trip to confirm the action with a

2159-6190/15 $31.00 © 2015 IEEE
DOI 10.1109/CLOUD.2015.118

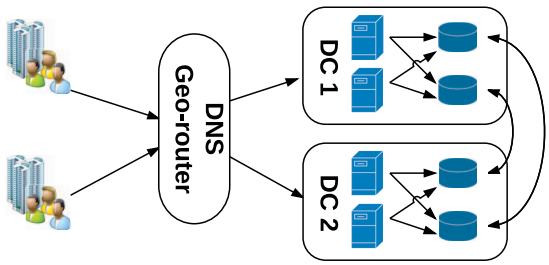


Fig. 1: Deployment using 2 DCs

central coordinator. It is used in many distributed databases, such as Google Spanner [7].

2) Quorum-based Protocols: Quorum-based protocols are used to manage data replication. When writing an object, the system writes to a set of object replicas, called a write quorum. When reading the object, the system fetches it from possibly another set of replicas, called a read quorum. Strong consistency for the object can be guaranteed if the summation of its read and write quorum is larger than the number of replicas. Some quorum databases [5] also allow users to sacrifice consistency to availability and performance by setting a weaker quorum [3]. However, quorum-based protocols alone are not able to support ACID (Atomicity, Consistency, Isolation, Durability) transactions involving multiple objects.

3) Paxos-based Protocols: Paxos [13] is a family of protocols for reaching consensus in an unreliable distributed environment. In database systems, the most common configuration for the protocol is multi-Paxos [14] with each process act as proposer, acceptor, and learner. The Paxos protocol proceeds in rounds. Its basic implementation also involves two steps, a prepare phase and an accept phase, in the successful case.

In database systems, optimized Paxos protocols are commonly combined with two-phase commit to achieve inter-DC transaction commit. A number of inter-DC transaction commit protocols are built upon Paxos, e.g., MegaStore [6], Spanner [7], MDCC [8], Calvin [9], and Replicated Commit [10].

4) Certification-based Commit: Certification-based commit [15] is a synchronous replication protocol developed based on the works by Pedone [16], and Kemme and Alonso [17]. The protocol needs the help of an underlying group communication system to deliver the commit requests originated from distributed processes in total and causal order to each process. When doing write transaction, the request is optimistically executed until commit point. After that, the process sends the write change to the whole communication group. The group then returns a global transaction ID to every process. Since all requests are delivered in the same order, each process can deterministically and independently check potential conflicts in its commit queue using a certification test. The request that passes the test can return immediately.

B. Databases Supporting Strong Inter-DC Consistency

1) Google's Systems: Google have been developing distributed databases that are both highly scalable and strongly consistent. Their first achievement is MegaStore [6]. It implements ACID semantics within each entity group (objects stored together) using synchronous replication based on optimized Paxos, and transaction across entity groups using two-phase commit. The second outcome is Spanner [7], which further supports external consistency (linearizability) with the help of physically synchronized clocks (GPS and atomic clock). Upon Spanner, Google built F1 [2], a distributed relational database system for their critical AdWords platform. It provides more enriched transaction semantics with high availability and scalability. All Google's systems remain proprietary and currently there are no open source analogies available.

2) Open Source Databases: Cassandra [5] is a shared nothing NoSql database using quorum-based protocol for its consistency model. It allows users to set individual read and write quorums at the granularity of query. It also provides limited transaction support (lightweight-transaction) starting from version 2.0 using a heavy-weight Paxos consensus protocol, which requires 4 round-trip messages to complete.

Galera cluster [4] is an open source scalable synchronous replication solution developed and maintained by Codership for MySql. Galera's replication is based on certification-based commit [15]. Since replication is synchronous, read-only queries in Galera are always processed locally.

III. Application and Deployment Model

A. Target Applications

We target session-based Internet applications. We assume the delay of the application is dominated by the round-trip time (RTT) between different parties, as the processing time of the request can be considered constant provided that there are enough computing resources. Therefore, whether the SLOs can be met is largely determined by the involved network latencies.

To benefit from our work, the application should also be deployed on geographically dispersed DCs, and some of its requests should require strong consistency, e.g., a group-working application that always reflects the newest updates to its end-users, a social-network application that consistently and timely shows people's posts and comments, or a distributed banking application that perpetually desires ACID semantics.

B. Deployment Model

We assume the whole software stack of the application (including application servers and underneath databases) is deployed on multiple geographically dispersed DCs and each application replica is able to autonomously scale up and down according to the changing workloads, as shown by the example in Figure 1. Currently, we suppose all chosen DCs have the full copy of data. This approach is commonly adopted by companies, like Facebook [18]. Furthermore, the databases described in this paper, Cassandra and Galera Cluster, support only full replication for multi-DC deployment within the same keyspace or namespace. The target applications should also use shared-nothing multi-master database clusters, which means all database queries originated from any server can be served by database nodes collocated in the same DC. Depending on the database and requested consistency of the query, we also consider there are network delays caused by communications among database cluster nodes in different DCs. All inner-DC communications, otherwise, are omitted.

We classify users into groups according to their geographic locations. All the requests from the same location are routed to the same DC using DNS routing services similar to Amazon Route 53's Geo Routing [19].

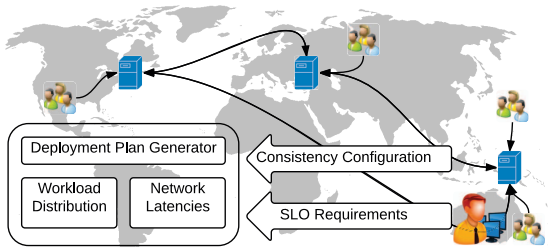


Fig. 2: The Proposed Approach

IV. Proposed Approach

A. Overview

Our approach requires application administrator to provide SLO and consistency configuration for each type of request as illustrated in Figure 2. In addition, it needs the information of the numbers of requests coming from each location. These data can be recorded during production. Furthermore, it wants the network latency data between each DC and each location and latency data between each DC. Since it is difficult to collect all the real-time RTT latencies between each location and DC given the large number of them, we can rely on network predictors, like the one employed by Grozev and Buyya [20], to estimate the unknown latencies. Or we can obtain the latency information from trusted third parties, like NetMetrics [21] which is a global Internet performance database.

The objective of our work is to select and manage a subset of DCs to host the application replicas, and in the meantime, find the optimal request routing according to the chosen DCs, so that the total amount of estimated SLO[1] violations is as small as possible. The approach involves two steps:

a) Initial Deployment: When the application is initially migrated to the clouds, our approach aims to select the hosting DCs and route requests to chosen DCs with minimum amount of total estimated SLO violations according to the current geographical distribution of requests[2].

b) Deployment Optimization: In the second step, our approach continuously attempts to maintain high performance of the application by contracting, optimizing, or expanding the deployment with acceptable migration[3] efforts in response to the changes of the requests distribution.

In this paper, we use the term **expand** and **contract** respectively for increasing and decreasing the number of chosen DCs. We focus on the geographical distribution of resources instead of the total resource amount, for which the term commonly used is scaling up and down.

B. SLO Violation Model

We first propose a model to estimate the amounts of SLO violations incurred by specific deployment plans. It is composed of the general model, which views database network latencies as a black box and extracts the commonality of the target applications, and the database model, which allows providers to plug different databases into the general model.

[1]As we assume the request processing time is constant, the SLO hereafter is referred to as the desirable total network latency.

[2]Here distribution of requests refers to the ratio of requests coming from each geographical location.

[3]In this paper, migration means deployment change that requires moving data to another DC, including moving an existing replica to another DC and deploying a new replica in a DC, but excluding removing an existing replica from a DC.

TABLE I: Symbols of the General Model

Term	Meaning
$\mathbf{M}$	Set of available DCs
$\mathbf{G}$	Set of geographical locations
$\mathbf{I}$	Set of request types
$\mathbf{X}$	Set of the chosen DCs to host application replicas
H	Number of chosen DCs
N_i^l	Number of type l requests from location i
T^l	Latency SLO of type l request
R_{ij}	RTT latency between location i and DC j
s_i	Total estimated SLO violations at location i given $\mathbf{X}$
lt_{ij}^l	Estimated network latency of type l request at location i if that request is served by the application replica placed in DC j
dlt_j^l	Database network latency for type l request served in DC j given $\mathbf{X}$
p^l	Protocol overhead of type l request (number of RTTs)

1) The General Model: For clarity, we introduce a metric **Average Violation Per Request (AVPR)**, calculated as the total estimated excess waiting time beyond defined SLOs (SLO violations) for all requests divided by the total number of requests, as the optimization target. Using the symbols in Table I, the general model then is defined as follows:

$$\begin{aligned} &\text{minimize} \quad f_{AVPR}(\mathbf{X}) = \frac{\sum_i s_i}{\sum_i \sum_l N_i^l} \quad \forall i \in \mathbf{G}, l \in \mathbf{I} \\ &\text{subject to} \quad \mathbf{X} \subset \mathbf{M}, |\mathbf{X}| = H \end{aligned}$$

where s_i is the amount of total SLO violations at location i. N_i^l is the number of type l requests coming from location i. $\mathbf{X}$ and $\mathbf{M}$ respectively represents the set of selected DCs, and the set of available DCs. H is the number of chosen DCs. $\mathbf{G}$ is the set of geographical locations. $\mathbf{I}$ is the set of request types.

When calculating s_i, we first need to determine the routing of requests from each location i according to the chosen $\mathbf{X}$. The DC within $\mathbf{X}$ that incurs the least amount of SLO violations is selected to serve users at location i. The amount of SLO violations incurred by each DC for serving users at location i is computed as the sum of all the excessive latencies that the users are estimated to perceive beyond SLO. Thus, s_i, in formula, can be represented as:

$$s_i = \min_{j \in \mathbf{X}} (\sum\nolimits_l N_i^l (lt_{ij}^l - T^l)) \quad \forall lt_{ij}^l > T^l, l \in \mathbf{I}$$

where T^l is the SLO of type l request. lt_{ij}^l is the latency of type l request perceived by users at location i, if it is served by the replica in DC j, which can be further expanded as:

$$lt_{ij}^l = p^l R_{ij} + dlt_j^l \quad j \in \mathbf{X}, l \in \mathbf{I}$$

lt_{ij}^l is the sum of two parts. The first part is the network latency between the user location i and the corresponding serving DC j ($p^l R_{ij}$). p^l is the communication overhead (number of RTTs) of the communication protocol used by the type l request (e.g., 2 for HTTP and 4 for HTTPS). R_{ij} is the RTT latency between location i and DC j. The second part is the database network latency overhead (dlt_j^l) modelled in the following sub-section.

2) The Database Model: The modelling of database network latency overhead is database-specific. Here we illustrate how to model that of the two widely-used databases. One is Cassandra [5], a NoSql database; the other is Galera Cluster [4], a replication solution for MySql relational database.

TABLE II: Symbols of the Cassandra Model

Term	Meaning
$\mathbf{R}_l$	Set of read queries in type l request
$\mathbf{W}_l$	Set of write queries in type l request
Qr^l_k	Read quorum of the k_{th} read query in request type l
Qw^l_m	Write quorum of the m_{th} write query in request type l
r	The replication factor of DCs
$\alpha(j,k,\mathbf{X})$	The function finds the k_{th} shortest RTT latency among all latencies between each DC within $\mathbf{X}$ and DC j

a) The Cassandra Model: The commonly used replication strategy for Cassandra production cluster using multiple DCs is symmetric replication, where each DC stores the same number of replicas [5]. Cassandra uses quorum-based protocol to implement consistent read/write operations across replicas, and it supports various consistency configurations at the granularity of query. Given the set of selected DCs ($\mathbf{X}$), using the symbols in Table II, its database network overhead dlt^l_j can be modelled as:

$$dlt^l_j = \sum\nolimits_k \alpha(j, \lceil \frac{Qr^l_k}{r} \rceil, \mathbf{X}) + \sum\nolimits_m \alpha(j, \lceil \frac{Qw^l_m}{r} \rceil, \mathbf{X}) \quad \forall k \in \mathbf{R}_l, m \in \mathbf{W}_l \; j \in \mathbf{X}, l \in \mathbf{I}$$

$\mathbf{R}_l$ ($\mathbf{W}_l$) is the set of read (write) queries in type l request, and Qr^l_k (Qw^l_m) is the read (write) quorum of the k_{th} (m_{th}) read (write) query in type l request. r is the replication factor in a DC. The function $\alpha(j,k,\mathbf{X})$ returns the k_{th} shortest RTT latency among all latencies between each DC within $\mathbf{X}$ and DC j. Follow the work by Shankaranarayanan et al. [22], we model the delay of the read/write query as the slowest replica's response time in the quorum. For example, if the read quorum is 3 and each DC holds 1 data replica, Cassandra will wait to receive replies from the 2 replicas located in other DCs as the network delay to the local copy is orders of magnitude smaller. Hence, the resulted delay will normally be the second shortest RTT latency from the local DC j to the other selected DCs.

In Cassandra, the remote replica only replies the digest of the objects. If the local copy is stale, it will send another request to fetch the complete data and update all the stale replicas. Analogy to Shankaranarayanan et al. [22], we ignore this overhead as such case is rare and, thus, only impose minute impact on the average delay of all the requests.

The administrator is responsible for deciding the quorum settings of each query, as besides consistency and performance, there are other concerns in this process that may complicate the decision, such as availability ($Qr = 1, Qw = H$ maximizes the performance for read-intensive applications but is susceptible to failures). For query requiring strong consistency, application administrator should specify its read/write quorum so that the object's Qr and Qw satisfy $Qr + Qw > H$. Certainly it is also possible to set a weaker configuration [3] if strong consistency is unnecessary.

In Cassandra, the legitimate quorum settings are currently limited to **ONE**, **TWO**, **THREE**, **ALL**, and **QUORUM** (simple majority). Some configurations are invalid, e.g., ($H = 5$, $Qr = 2$, and $Qw = 4$), as $Qw = 4$ is not allowed. However, we also include these configurations in our evaluations as we suppose they will be possible in future versions.

b) The Galera Model: In Galera cluster, all read-only transactions are executed locally while transactions with write operations are synchronously replicated to all remote replicas using certification-based commit [15]. As there is no further group communication involved in the protocol [15] after the transaction ID is determined, the network latency is dominated by the DC that has the largest RTT latency to the request originator. Based on symbols in Table III, dlt^l_j, in this case, can be simply formulated as:

TABLE III: Symbols of the Galera Model

Term	Meaning
$\beta(j,\mathbf{X})$	The function finds the largest RTT latency among all latencies between each DC within X and DC j
V^l	Number of transactions that have write operations in request type l

$$dlt^l_j = V^l \beta(j, \mathbf{X}) \quad j \in \mathbf{X}, l \in \mathbf{I}$$

where V^l is the number of database transactions that have write operations in type l request, and the function $\beta(j,\mathbf{X})$ returns the largest RTT latency among all latencies between each DC within $\mathbf{X}$ and DC j.

Galera nodes may queue the messages before delivering them because of the group communication overhead. We neglect this delay because we believe it is unpredictable, application-specific, and also insignificant compared to the network transfer delay. To build a more precise model, application administrators can profile their applications to obtain the average queuing time and add this value to the model.

C. Solution for Initial Deployment

1) Hardness of the problem: The problem of moving several application replicas to cloud at once falls in the category of Facility Location Problems [23], which are usually NP-hard to solve.

Proving by restriction, which consists of showing the target problem contains an already-known NP-hard problem as a special case, is the simplest way to prove a problem is NP-hard. Hereafter, we prove our problem is NP-hard by showing that the NP-hard k-median problem is a special case of our problem.

Proof: Suppose eventual consistency is used by all the request types, then dlt^l_j equals 0 for all j, l. Therefore, $lt^l_{ij} = \min_{j \in \mathbf{X}}(p^l R_{ij})$, which is constant. With fixed amount of estimated SLO violations between any location i and any candidate DC j, selecting H DCs from a set of candidate DCs $\mathbf{M}$ to serve customers at a set of locations $\mathbf{G}$ with minimum amount of SLO violations is exactly the k-median problem.

Since our problem is NP-hard, we refer to heuristic approximation algorithms that can find good enough solutions in polynomial time.

2) Genetic Algorithm Overview: Our solution is based on genetic algorithm (GA). Compared with other heuristics, it has three advantages. The first is that meta-heuristics like GA are more flexible. For each database, the administrator only needs to substitute the representation of dlt^l_j to let the algorithm work. While for other heuristics, such as greedy algorithms, we find that they are often tightly coupled with the database models. The second is that GA produces satisfactory results in our context. We demonstrate that in our experiments. The last but not least is that it is easy for meta-heuristic algorithms to incorporate other selection criteria into the existing model, e.g., number of migrations.

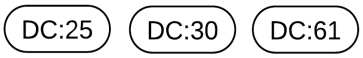


Fig. 3: An Example of the Chromosome with 3 chosen DCs

TABLE IV: Symbols of Deployment Optimization

Term	Meaning
$n\_migration$	The function calculates the number of required migrations
t	Redeployment interval
U	Upper bound of $AVPR$
L	Lower bound of $AVPR$
W	Unit AVPR gain threshold for migration if $AVPR$ of the current deployment is below U
C	Cooling period for contracting the application

3) Genetic Algorithm in Detail: Our GA generates a set of random solutions at the beginning and then iteratively performing crossover, mutation, and selection according to a predefined fitness function. It returns not only the set of chosen DCs but also the optimal request routing regarding the chosen DCs. Before calculating fitness value, it computes the optimal request routing according to the chosen DCs **X** for each location. The fitness function of the algorithm is defined as the f_{AVPR} function in the SLO violation model.

GA requires programmers to encode the solutions using a specific data structure, called chromosome. In our GA, we number all the available DCs and encode the solution as an non-repetitive ascending array, as illustrated in Figure 3. The number of genes in a chromosome equals the total number of DCs the provider wants to choose H. Thus, repetitive genes are not allowed in the chromosome. In addition, we sort the genes in ascending order for the convenience of programming.

We wrote our own initial population generator, crossover, and mutation operators. For **mutation operation**, it first randomly picks one gene in the chromosome. Then it mutates the value of the gene. The mutated gene should be unique to all the genes in the previous chromosome. Finally, the new chromosome is sorted to preserve the ascending property. The **initial population** is generated randomly. The genes in an initial chromosome are generated stochastically one by one unique to the previously generated genes in the same chromosome. After that, the genes are sorted to the ascending order. For **crossover operation**, we randomly swap some genes of the two randomly chosen chromosomes. If the resulted new chromosomes have repetitive genes, we perform extra mutations to eliminate repetitions. Then the resulted new genes are sorted to the ascending order. The algorithm terminates if not enough progress has been made for some time.

D. Solution for Deployment Optimization

1) Decision-making Algorithm: In this step, we aim to optimize the deployment according to the given workload and the current deployment. We propose a decision-making algorithm (Algorithm 1) for deciding whether and how to adjust the deployment so that good enough $AVPR$ can be achieved with acceptable migration effort and operational cost. To realize that, we require the administrator to specify the upper threshold of the acceptable $AVPR$, represented as U, and the lower threshold of the $AVPR$, which is shown as L.

The algorithm uses redeployment heuristics to find the satisfactory redeployment plan. Their objective is to let providers gain more $AVPR$ improvement from unit migration effort, which is defined as:

$$max \quad \frac{U - f_{AVPR}(new)}{n\_migration(new, current)} \qquad f_{AVPR}(current) \geq U \;||\; new.size < old.size$$

$$max \quad \frac{f_{AVPR}(current) - f_{AVPR}(new)}{n\_migration(new, current)} \qquad otherwise$$

Here, $n\_migration(new, current)$ returns the number of migrations required to change the current deployment to the new deployment. The above function means when current

Algorithm 1: Redeployment Decision-making Algorithm

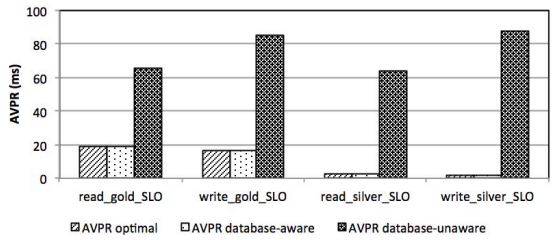
Input: $initial\_dc\_num$, and t

$dc\_num = initial\_dc\_num$;
$current\_plan = first\_step\_deployment(dc\_num)$;
for *every* t **do**
  /* try to contract the application */
  if $f_{AVPR}(current\_plan) <$ L *for consecutively more than* C *rounds* **then**
    new_plan;
    for *each* $dc \in current\_plan$ **do**
      $contracted\_plan = current\_plan.remove(dc)$;
      $tmp\_plan = redeployHeuristic(dc\_num - 1, contracted\_plan)$;
      if $tmp\_plan.isFitter(new\_plan)$ **then**
        $new\_plan = tmp\_plan$;
      end
    end
    if $f_{AVPR}(new\_plan) < U$ **then**
      $current\_plan = new\_plan$;
      $dc\_num - -$;
      $continue$;
    end
  end
  /* try to optimize the deployment with the same number of chosen DCs */
  $new\_plan = redeployHeuristic(dc\_num, current\_plan)$;
  if $worthwhile(current\_plan, new\_plan)$ **then**
    $current\_plan = new\_plan$;
    $continue$;
  end
  /* expand the application */
  if $f_{AVPR}(new\_plan) \geq U$ **then**
    $new\_plan = redeployHeuristic(++dc\_num, current\_plan)$;
    $current\_plan = new\_plan$;
  end
end

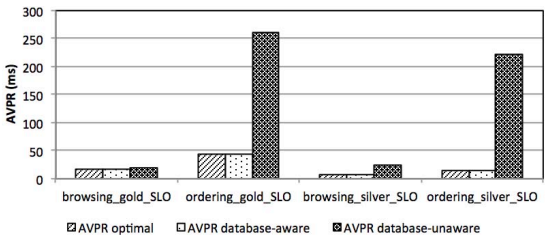
Algorithm 2: Worthwhile Method

Input: $current\_plan$, and new_plan

if $f_{AVPR}(current\_plan) < U$ && $w(new\_plan) > W$ **then**
  return $true$;
end
if $f_{AVPR}(current\_plan) \geq U$ && $f_{AVPR}(new\_plan) < U$ **then**
  return $true$;
end
return $false$;

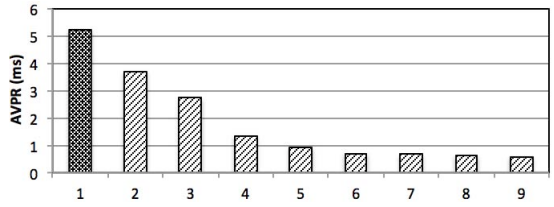


(a) $AVPR$ for Twissandra using various SLOs and workloads

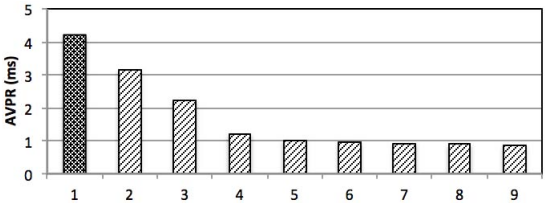


(b) $AVPR$ for TPC-W using different SLOs and workloads

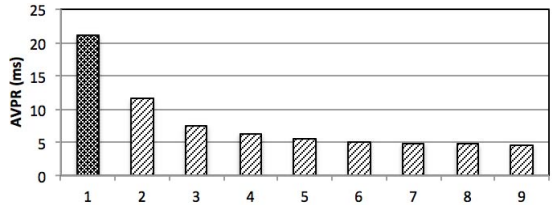
Fig. 4: Comparing performances of different algorithms using 3 DCs



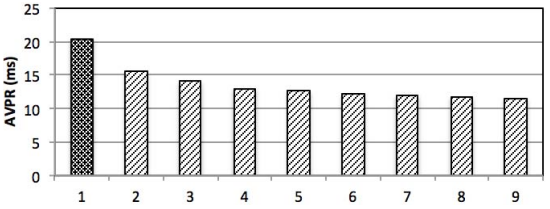
(a) $AVPR$ for Twissandra read-intensive workload



(b) $AVPR$ for Twissandra write-intensive workload



(c) $AVPR$ for TPC-W browsing workload



(d) $AVPR$ for TPC-W ordering workload

Fig. 5: Comparing deployments using multiple DCs and optimal deployments using 1 DC

deployment cannot meet the upper bound of $AVPR$, or the algorithm is trying to contract the application, the optimization target is to maximize the $AVPR$ gain from unit migration effort against the upper bound. Otherwise, it is to maximize the $AVPR$ gain against the $AVPR$ of the current deployment.

As noted in Algorithm 1, our approach first searches if there is chance to contract the application when $AVPR$ has been below the lower bound L for time longer than the cooling period C (Line 4-18). We introduce the cooling period here to alleviate oscillation that would cause frequent contraction and expansion of the application. As removing one DC is not calculated in the migration effort, when contracting the application, the algorithm iterates all the possible contract choices, and tries to find the best redeployment plan. If no contraction is performed, it then endeavours to find a better deployment with the same number of chosen DCs (Line 19-23). Redeployment will only be conducted if $AVPR$ improvement from unit migration effort is beyond an administrator defined threshold W or the new deployment can reduce the $AVPR$ to the acceptable level (Algorithm 2). Suppose no deployment plan that will reduce the $AVPR$ into acceptable level is found, the algorithm then expands the application (Line 24-28).

Administrators can reset constants U, L, C, and W at any time according to their own wishes.

2) Redeployment Heuristics: We have come up with two redeployment heuristics:

Migration-aware Genetic: It simply replaces the fitness function of the GA used in initial deployment phase with the previous defined optimization target.

k-Brute Force: Brute-forcedly find the best plan that is reachable using at most k migrations (feasible for small k if H, $|\mathbf{G}|$, and $|\mathbf{M}|$ are large). In our experiments, we set k to 2.

V. Performance Evaluation

We evaluate our approach using simulations. The settings of the DCs and networks are described in the next sub-section. The workloads and baselines used are explained within each experiment. As GA is stochastic, we run each test 5 times and report the best result. For the parameters of the GA, we set the population size to 1000, the crossover rate to 50%, and the mutation rate to 2%. We select half of the best chromosomes for reproduction after each iteration.

A. DCs and User Settings

We use the data collected by Zhu et al. [24] for our experiments due to lack availability of latency data from real cloud DCs. The dataset uses 307 PlanetLab nodes as the candidate DCs and 1881 web services discovered by a crawler as the user locations. Zhu et al. let the PlatetLab nodes ping the web services and each other to obtain the real RTT latency

data. Though PlanetLab nodes are not commercial cloud DCs, we believe the dataset is still representative to our problems as they are geographically distributed and can be viewed as mimics of cloud DCs in which application replicas interact with each other and end users through WAN.

B. Evaluation of Initial Deployment

1) Workload: We studied two real-world applications and specified the consistency requirements of all their request types according to our own judgement. The first application is the TPC-W workload [11] which mimics an e-business website. The TPC-W implementation we studied uses MySql database, which is compatible to the Galera cluster. The second application is called Twissandra [12], which is an open source copy of Twitter built on Cassandra.

We generate two different workloads for both of the applications from each geographical location using normal distribution. The request mix of TPC-W, is defined by the browsing and ordering workload included in its benchmark suite. Roughly, in browsing workload, 75% requests can be served without inter-DC communications; while in ordering workload, the number decreases to about 38%. For Twissandra, the request mix is the ratio of timeline view and tweet operations. For read-intensive workload, the timeline view/tweet ratio is set to 9:1; for write-intensive workload, it is 7:3. We assume strong consistency is required for both of the operations, which means Qr(timeline view)+Qw(tweet)$> H$. We set $Qr = 2, Qw = H - 1$ for all Twissandra tests using multiple clouds, as Twissandra is generally read-intensive, and $Qr = 2$ can tolerate one DC down when $Qw > 1$ for $Qr + Qw - 1$ DCs.

2) SLO: We specify different latency SLOs as well. The latency constraint for each request type is set according to the total network round-trips it requires. Each request type is respectively given $50ms$ and $100ms$ to perform one network round-trip for **Gold** and **Silver** SLO.

3) Necessity of Considering Database Network Latencies: First, we show that it is important to consider database network latencies when deploying applications requiring strong inter-DC consistency. We fix the number of chosen DCs to 3 in the experiment, and run our consistency-aware GA algorithm with different levels of SLO and workloads. We compare the results with a baseline GA algorithm that only considers network communications between DCs and end users, which is similar to the setting used in Yu et al.'s work [25]. We run the baseline and then evaluate the found solutions using the database latency aware model.

From Figure 4, it is obvious that by omitting database network latencies, the found solutions result to unacceptably higher $AVPR$ compared to the database consistency overhead aware approach, except cases of TPC-W application under browsing workload. The resulted differences of the two algorithms for TPC-W under browsing workload are much smaller because the majority of requests (75%) in browsing workload are served without inter-DC communications. From the results, we can conclude that it is important to take database network latencies into account if the inter-DC communication rate is high in the request mix.

4) Goodness of our GA: We compare the results of our GA with the optimal results. The optimal solutions are derived by traversing all the possible solutions in the search space. In our settings, finding optimal results using 3 DCs is the limit on our desktop testbed, which takes more than 16 hours to finish. As shown in figure 5, our GA finds the exact optimal deployment plan for all 8 settings, but using only around 4 minutes.

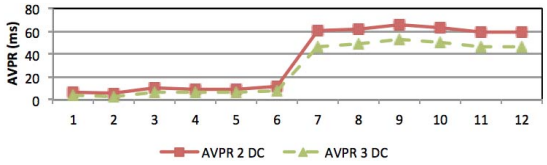


Fig. 6: Results using fixed deployments and silver SLO

5) Usefulness of Deploying Applications in Multiple DCs: Regarding $AVPR$, we discuss whether it is worth to deploy applications requiring strong inter-DC consistencies in multiple DCs. This time, we fix the SLO to silver and run our GA algorithms with different number of DCs and workloads. The results are compared with the optimal deployments using one DC only.

The results presented in Figure 5 indicate that purely from the performance perspective, deploying applications requiring strong consistency in multiple DCs is still beneficial. Using 3 DCs generally can reduce the amount of SLO violations to half of that using optimal single DC deployment. However, the performance gain from increased number of DCs becomes negligible when the number of DCs exceeds 4. This effect justifies our motivation to keep the number of chosen DCs as small as possible in the deployment optimization phase to save operational cost. Even though the performance gain is small, the providers may want to deploy their applications in larger number of DCs for other benefits, such as availability and fault-tolerance, which is out of the scope of this paper.

C. Evaluation of Deployment Optimization

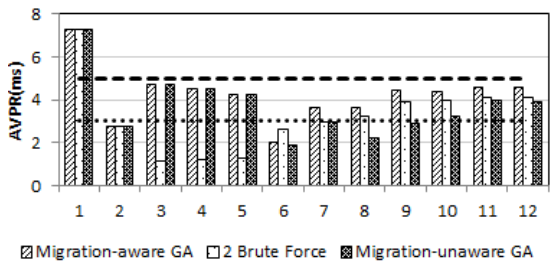
1) Workload: We generate a series of workloads of Twissandra to simulate the expansion of business and workload increase for the test of our redeployment decision-making algorithm. We classify the user locations into 11 geographic categories based on their latencies to all the 307 DCs using K-means, and we add one category of locations into the workload per redeployment round. The numbers of requests at the added locations continuously grow in the following rounds in our settings to mimic workload increase.

2) Necessity of Deployment Optimization: To illustrate the necessity of deployment optimization along with the business expansion and workload increase, we run experiments using 2, and 3 DCs according to the initial workloads and observe how the performances of the fixed deployments will change in the later times under varied workload distributions.

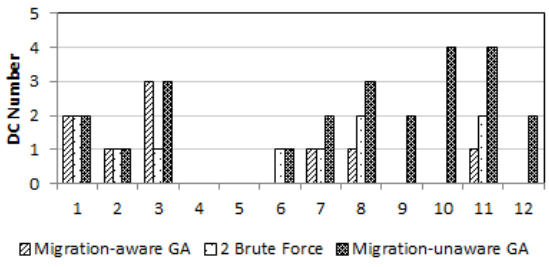
As Figure 6 shows, the fixed deployments incur unacceptably high $AVPR$ when the workload has been expanded and increased, which indicates that redeployment is essential to maintain acceptable QoS under changing workloads.

3) Our Approach: We test our decision-making algorithm using the two proposed redeployment heuristics. We compare the results with one baseline algorithm.

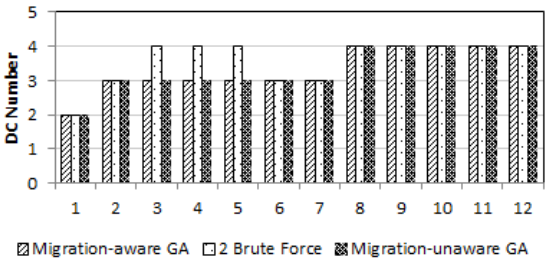
Similar to Algorithm 1, the baseline algorithm tries to contract the application when $AVPR$ is below the lower bound L for time longer than the cooling period, and expands



(a) AVPR of Deployments



(b) Number of Migrations



(c) Number of Chosen DCs

Fig. 7: Results with $initial\_dc = 2$, $U = 5$, $L = 3$, $W = 0.5$, $C = 1$, and silver SLO

the application when U cannot be met. However, it performs migrations as long as it finds a better deployment plan, no matter whether the improvement is significant or not compared to the current deployment. The redeployment heuristic used in the baseline is called Migration-unaware Genetic. Its target is always to find the deployment with minimum $AVPR$, which is the same GA used in the initial deployment phase.

We run the test with $initial\_dc = 2$, $U = 5$, $L = 3$, $W = 0.5$, $C = 1$, and silver SLO. Figure 7 shows the comparison of the $AVPR$, the number of migrations, and the number of chosen DCs within each redeployment round using the proposed and the baseline approaches. Except round 3-5, the baseline algorithm finds the deployment plans with the smallest $AVPR$. On the other hand, it incurs the largest number of migrations. Our approaches, though results a little more $AVPR$, manage to maintain the performance under the upper bound with much less total numbers of migrations (9:10:24), which shows the effectiveness of our decision-making algorithm in balancing $AVPR$ and the migration cost.

Comparing the two redeployment heuristics used in our algorithm, at the 3rd round, 2 brute force heuristic failed to find a valid redeployment plan using 3 DCs due to its limitation that, within each round, maximum 2 migrations can be conducted. Though it greatly outperforms other approaches in $AVPR$ during round 3 to round 5, it uses one more DC, which increases the operational cost. The 2 brute force approach finally takes the chance to contract the application at round 6 after long passing the cooling period. This is also due to its limitation, which results it not being able to find a redeployment plan with $AVPR$ below the upper bound using 3 DCs after the cooling period during round 4 to round 5.

Choosing the right redeployment heuristic is always context specific. If the workload distribution does not change abruptly, k brute force is the better choice, as it provides higher chance for the providers to find the redeployment plan with optimal $AVPR$ gain from unit migration. For application providers that have tight operational budget or providers that are expanding quickly, then migration-aware GA is possibly the right choice.

VI. Discussions

Our approach can be applied to applications using other database systems, as long as they adopt shared-nothing architecture, store full copy of data at each site, and employ a known inter-DC consistency protocol. Though, currently, not many commercial databases support inter-DC consistency, there are some emerging ones that satisfy the prerequisites and can be incorporated in our approach in the future, such as MDCC [8], Calvin [9], and Replicated Commit [10].

Furthermore, it is difficult to build a precise database latency model which considers all the factors. Like what we have done with the Cassandra and Galera model, we believe a close approximation is enough to meet the purpose as the special cases only have minute influence on the aggregated results when the traffic is huge.

Our approach aims to provide performance boost in long-term. Handling performance issues caused by short-term network instability is out of our scope, and there is no solution can realize that if application cannot be migrated in short time. As long as the network latency data used for deployment plan calculation is close to the network performance in normal status, our approach is sure to be beneficial.

Providers can either use the current workload or the predicted workload to generate the redeployment plan. If predictions can be done accurately, using predicted workload can further improve the performance by preparing for the workload changes during the redeployment intervals in advance.

VII. Related Work

Previous works about inter-DC consistency mostly focus on the database layer. Besides developing databases that are

capable of supporting strong inter-DC consistency [2], [4]–[10], some works have also focused on optimizing the placement of data replicas and their consistency configurations to reduce the response time and cost in the database layer. These works usually target quorum-based systems, as they are more flexible in consistency configurations. SPANStore [26] is a multi-DC key-value store with quorum consistency. It is able to transparently place data replicas across geo-distributed DCs so that total cost of storage and I/O operations is minimized, and meanwhile, it still can meet its latency, fault-tolerance, and consistency goals. Shankaranarayanan et al. [22] proposed an approach to find the optimal configuration of quorum-based data across multiple DCs (number of replicas, replica placement, Qr, and Qw) so that read/write latency is minimized in normal case and bounded when one DC is lost. Our work is different to theirs as we aim to optimize the performance of the whole application instead of just the read/write latencies in database. In addition, we strive to build a general approach that is extensible to support multiple databases.

Many works have studied the application placement problem in multi-DC context. However, none of them have considered inter-DC consistency. Yu et al. [25] explored how to deploy and redeploy standalone application replicas to minimize total response time or maximize user satisfaction in changing workload. Zhang et al. [27] proposed an approach to dynamically place applications in geographically distributed cloud DCs with limited capacities and volatile costs using control and game theory. Wu et al. [28] targeted the deployment of social media applications (e.g., Youtube) using multiple clouds. Their approach employed a social influence model to predict the future demand, and then judiciously place the media files and servers in cloud DCs with minimum cost under latency, bandwidth, and availability constraints.

VIII. Conclusions

We proposed an approach to help web application providers deploy their applications with various inter-DC consistency requirements across multiple cloud DCs. It generates deployment plan with minimum amount of SLO violations when the application is first moved to the cloud using our genetic algorithm (initial deployment phase), and then it continuously optimizes the deployment considering SLO satisfaction, migration cost, and operational cost along with the change of workload distribution (deployment optimization phase). We proposed an extensible SLO violation model so that besides the illustrated database systems (Cassandra and Galera Cluster), other databases that satisfy certain requirements can be easily adapted to our approach. To demonstrate the effectiveness of our approach, we conducted simulation experiments using settings of two applications (TPC-W and Twissandra).

References

[1] "Latency - it costs you," http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[2] J. Shute *et al.*, "F1: A distributed SQL database that scales," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1068–1079, 2013.

[3] P. Bailis *et al.*, "Quantifying eventual consistency with PBS," *The VLDB Journal*, vol. 23, no. 2, pp. 279–302, 2015.

[4] Codership, "Galera cluster," http://galeracluster.com/products/, 2015.

[5] Apache, "Cassandra," http://cassandra.apache.org/, 2015.

[6] J. Baker *et al.*, "Megastore: Providing scalable, highly available storage for interactive services," in *Porceddings of the 5th biennial Conference on Innovative Data Systems Research (CIDR)*, vol. 11, 2011, pp. 223–234.

[7] J. C. Corbett *et al.*, "Spanner: Google's globally-distributed database," in *Proceedings of OSDI*, vol. 1, 2012.

[8] T. Kraska *et al.*, "MDCC: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, pp. 113–126.

[9] A. Thomson *et al.*, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, pp. 1–12.

[10] H. Mahmoud *et al.*, "Low-latency multi-datacenter databases using replicated commit," *Proceedings of the VLDB Endowment*, vol. 6, no. 9, pp. 661–672, 2013.

[11] Transaction Processing Performance Council, "TPC-W Workload," http://www.tpc.org/tpcw/, 2015.

[12] Twissandra, "Twissandra," https://github.com/twissandra/twissandra.

[13] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[14] T. D. Chandra *et al.*, "Paxos made live: an engineering perspective," in *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*. ACM, pp. 398–407.

[15] Codership, "Certification-based commit," http://galeracluster.com/documentation-webpages/certificationbasedreplication.html, 2014.

[16] F. Pedone, "The database state machine and group communication issues," Ph.D. dissertation, 1999.

[17] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *VLDB*, 2000, pp. 134–143.

[18] R. Nishtala *et al.*, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, Conference Proceedings, pp. 385–398.

[19] Amazon, "Route 53 Update - Geo Routing," http://aws.amazon.com/blogs/aws/route-53-domain-reg-geo-route-price-drop/, 2015.

[20] N. Grozev and R. Buyya, "Multi-cloud provisioning and load distribution for three-tier applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 3, p. 13, 2015.

[21] "NetMetrics," https://www.ookla.com/netmetrics, 2015.

[22] P. Shankaranarayanan *et al.*, "Performance sensitive replication in geo-distributed cloud datastores," in *Proceedings of 44th International Conference on Dependable Systems and Networks (DSN)*, 2014.

[23] D. B. Shmoys *et al.*, "Approximation algorithms for facility location problems," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. ACM, Conference Proceedings, pp. 265–274.

[24] Z. Jieming *et al.*, "Scaling service-oriented applications into geo-distributed clouds," in *proceedings of the 7th International Symposium on Service Oriented System Engineering (SOSE)*, pp. 335–340.

[25] K. Yu *et al.*, "A user experience-based cloud service redeployment mechanism," in *Proceedings of the 2011 IEEE International Conference on Cloud Computing (CLOUD)*, pp. 227–234.

[26] Z. Wu *et al.*, "SPANStore: cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[27] Q. Zhang *et al.*, "Dynamic service placement in geographically distributed clouds," in *Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)*, pp. 526–535.

[28] Y. Wu *et al.*, "Scaling social media applications into geo-distributed clouds," in *Proceedings of the 2012 IEEE INFOCOM*. IEEE, pp. 684–692.