# Application-Managed Replication Controller for Cloud-Hosted Databases

Liang Zhao *†, Sherif Sakr *†, Anna Liu *†

*National ICT Australia (NICTA)
†School of Computer Science and Engineering
University of New South Wales, Australia
{firstname.lastname}@Nicta.com.au

***Abstract*—Data replication is a well-known strategy to achieve the availability, scalability and performance improvement goals in the data management world. However, the cost of maintaining several database replicas always strongly consistent is very high. The CAP theorem shows that a shared-data system can choose at most two out of three properties: consistency, availability, and tolerance to partitions. In practice, most of the cloud-based data management systems tend to overcome the difficulties of distributed replication by relaxing the consistency guarantees of the system. In particular, they implement various forms of weaker consistency models such as eventual consistency. This solution is accepted by many new Web 2.0 applications (e.g. social networks) which could be more tolerant with a wider window of data staleness (replication delay). However, unfortunately, there are no generic *application-independent* and *consumer-centric* mechanisms by which software applications can specify and manage to what extent inconsistencies can be tolerated.**

**We introduce an adaptive framework for database replication at the middleware layer of cloud environments. The framework provides flexible mechanisms to enable software applications of keeping several database replicas (that can be hosted in different data centers) with different levels of service level agreements (SLA) for their data freshness. The experimental evaluation demonstrates the effectiveness of our framework in providing the software applications with the required flexibility to achieve and optimize their requirements in terms of overall system throughput, data freshness and invested monetary cost.**

## I. Introduction

The cloud is an increasingly popular platform for hosting software applications in a variety of domains such as e-retail, finance, news and social networking. Cloud computing simplifies the time-consuming processes of hardware provisioning, hardware purchasing and software deployment. Therefore, it promises a number of advantages for the deployment of data-intensive applications such as elasticity of resources, pay-per-use cost model, low time to market and the perception of (virtually) unlimited resources and infinite scalability. Hence, it becomes possible, at least theoretically, to achieve unlimited throughput by continuously adding computing resources (e.g. database servers) if the workload increases.

In general, virtual machine technologies are increasingly being used to improve the manageability of software systems and lower their total cost of ownership. Resource virtualization technologies add a flexible and programmable layer of software between applications and the resources used by these applications. One approach for deploying data-intensive applications in cloud platforms is the *application-managed* approach which takes an existing application designed for a conventional data center, and then port it to virtual machines in the public cloud [1], [2]. Such a migration process usually requires minimal changes in the architecture or the code of the deployed application. In this approach, *database servers*, like any other software components, are migrated to run in virtual machines. One of the major advantages of the *application-managed* approach is that the application can have full control in dynamically allocating and configuring the physical resources of the database tier (database servers) as required [3], [4], [5]. Hence, software applications can fully utilize the elasticity feature of the cloud environment to achieve their defined and customized scalability or cost reduction goals. In addition, this approach enables the software applications to build their geographically distributed database clusters. Without the cloud, building such an in-house cluster would require a self-owned infrastructure which represents an option that only big enterprises can afford to put in place.

In general, stateless services are easy to scale in the cloud since any new *replicas* of these services can operate completely independently of other instances. In contrast, scaling stateful services, such as a *database system*, needs to guarantee a consistent view of the system for users of the service. The **CAP** theorem [6] shows that a shared-data system can only choose at most two out of three properties: ***C***onsistency (all records are the same in all replicas), ***A***vailability (all replicas can accept updates or inserts), and tolerance to ***P***artitions (the system still functions when distributed replicas cannot talk to each other). In practice, it is highly important for cloud-based applications to be always available and accept update requests of data and at the same time cannot block the updates even while they read the same data for scalability reasons. Therefore, when data is replicated over a wide area, this essentially leaves just consistency and availability for a system to choose between. Thus, the **C** (consistency) part of **CAP** is typically compromised to yield reasonable system availability [7]. Hence, most of the cloud data management overcome the difficulties of distributed replication by relaxing the consistency guarantees of the system. In particular, they implement various forms of weaker consistency models (e.g. eventual consistency [8]) so that all replicas do not have to agree on the same value of a data item at every moment of

978-0-7695-4755-8/12 $26.00 © 2012 IEEE
DOI 10.1109/CLOUD.2012.35

time. In particular, the eventual consistency policy guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the *inconsistency window* can be determined based on factors such as communication delays, the load on the system and the number of replicas involved in the replication scheme.

While traditional transactional data management applications (e.g. banking, stock trading) usually tend to rely on strong consisteny guarantees and require microsecond precision for their read operations, eventual consistency model is more favorable to the new generation of many Web 2.0 applications (e.g. social network applications) which could be more tolerant with a wider window of *data staleness* (replication delay). In practice, several very large Web-based systems such as *Amazon*, *Google* and *eBay* have relied on the eventual consistency model for managing their replicated data over distributed data centers [9], [10]. However, unfortunately, there are no general, *application-independent* and *consumer-centric* mechanisms by which software applications can manage to what extent inconsistencies can be tolerated. In other words, replication for scalability is inherently coupled to the access, usage and semantics of the data that are being replicated.

In this paper, we consider the problem of adaptive *consumer-centric* management for replicated databases in (multi) data center cloud platforms. In practice, this environment is characterized by high latency communication between data centers and significant fluctuations in the performance of underlying virtual machines [11], [12]. In addition, the cost of maintaining several database replicas that are always strongly consistent is very high. Therefore, keeping several database replicas with different levels of freshness can be highly beneficial in the cloud environment since freshness can be exploited as an important metric of replica selection for serving the application requests as well as optimizing the overall system performance and monetary cost. We introduce an adaptive framework for database replication at the middleware layer of cloud environments that enables keeping several replicas of the database in different data centers to support the different availability, scalability and performance improvement goals. The framework provides the software applications with flexible mechanisms for specifying different service level agreements (SLA) of data freshness for the underlying database replicas. In particular, the framework allows specifying an SLA of data freshness for each database replica and continuously monitor the replication delay of each replica so that once a replica violates its defined SLA, the framework automatically activates another replica at the closest geographic location in order to balance the workload and re-satisfy the defined SLA.

The remainder of this paper is structured as follows. Section II gives a brief overview about database replication strategies in cloud environments. Section III details the implementation of the different components of our framework. The results of the experimental evaluation for the performance of our approach are presented in Section IV. Section V summarizes the related work before we conclude the paper in Section VI.

## II. Database Replication in the Cloud

Data replication [13] is a well-known strategy to achieve the availability, scalability and performance improvement goals in the data management world. In particular, when the application load increases, there are two main options for achieving scalability at the database tier and for enabling the software applications to cope with more client requests:

1) *Scaling up*: aims at allocating a bigger machine with more horsepower (e.g. more processors, memory, bandwidth) to act as a database server.
2) *Scaling out*: aims at *replicating* the database tier across more machines (servers).

The scaling up option has the main drawback that large machines are often very expensive and eventually a physical limit is reached where a more powerful machine cannot be purchased at any cost. Alternatively, it is both *extensible* and *economical* to scale out by adding another commodity server. Therefore, cloud platforms are generally relying on the scaling out model in their services as it fits well with the *pay-as-you-go* pricing philosophy.

In practice, two database replication architectures are commonly used: the *multi-master* replication and the *master-slave* replication. On one side, the multi-master replication allows each replica to serve both read and write requests. In this architecture, write-write conflicts are resolved by the replication middleware, so that each replica executes write transactions in the same order. On the other side, the master-slave replication is more adequate for improving read throughput where read transactions are served by slaves while all the write requests are only served by the master. The replication middleware is in charge of passing writesets from the master to slaves in order to keep the database replicas up-to-date. The write-write conflicts are all resolved on the master side. Our approach relies on the master-slave replication architecture as it is the most commonly employed approach by many web applications.

In general, the replication architectures expose how read and write transactions are assigned across replicas while the synchronization models reveal how data is committed across replicas. For example, the *synchronous* replication blocks a successful response to the client until the write transaction is committed on the updated replica and writesets are duplicated over all other replicas. Therefore, it makes sure that all replicas are consistent during the time, however traversing all replicas potentially incurs high latency on write transactions. Furthermore, the availability of the system may be affected if unreachable replicas due to network partitioning cause suspension of synchronization. The *asynchronous* replication sends a successful response once the write transaction is committed on updated replica. The writesets will be propagated to other replicas at a later time. It avoids high write latency over networks in exchange of stale data. Moreover, once the updated replica goes offline before duplicating data, data loss may occur. Due to the existence of such a replication delay, read transactions on database replicas are not expected to

return consistent results all the time. However, it is guaranteed that the database replicas will be eventually consistent [8].

Florescu and Kossmann [14] have argued that in the new large scale web applications, the requirement to provide 100 percent read and write availability for all users has overshadowed the importance of the ACID paradigm as the gold standard for data consistency. In these applications, no user is ever allowed to be blocked. Hence, while having strong consistency mechanisms has been considered as a hard and expensive constraint in traditional database management systems, it has been turned into an optimization goal (that can be relaxed) in cloud-based database systems.

Kraska et al. [15] have presented a mechanism that allows software designers to define the consistency guarantees on the data instead of at the transaction level. In addition, it allows the ability to automatically switch consistency guarantees at runtime. They described a dynamic consistency strategy, called *Consistency Rationing*, to reduce the consistency requirements when possible (i.e., the penalty cost is low) and raise them when it matters (i.e., the penalty costs would be too high). Keeton et al. [16] have proposed a similar approach in a system called *LazyBase* that allows users to trade off query performance and result freshness. LazyBase breaks up metadata processing into a pipeline of ingestion, transformation and query stages which can be parallelized to improve performance and efficiency. LazyBase uses models of transformation and query performance to determine how to schedule transformation operations to meet users' freshness and performance goals, and to utilize resources efficiently.

While *Consistency Rationing* and *LazyBase* represent two approaches for supporting adaptive consistency management for cloud-hosted databases, they are more focused on the perspective of cloud service provider. On the contrary, the framework presented in this paper is more focused on the cloud consumer perspective. In particular, it provides the cloud consumer (software application) with flexible mechanisms for specifying and managing the extent of inconsistencies that they can tolerate. In addition, it allows the creation and monitoring of several replicas of the database with different levels of freshness across the different virtualized servers.

## III. Framework Architecture

Figure 1 shows an overview of our framework architecture which consists of three main modules: the *monitor module*, the *control module* and the *action module*. In this architecture, the monitor module is responsible for continuously tracking the replication delay of each database replica and feeding the control module with the collected information. The control module is responsible for continuously checking the replication delay of each replica against its associated application-defined SLA of data freshness and triggers the action module to scale out the database tier with a new replica in case of SLA-violation of any current replica is detected.

The design principles of our framework architecture are to be *application-independent* and to require *no code modification* on the consumer software applications that the framework will support. In order to achieve these goals, we rely on a database proxying mechanism which provides the ability to forward database requests to the underlying databases using an intermediate piece of software, the proxy, and to return the results from those request transparently to the client program without the need of having any database drivers installed [17]. In particular, a database proxy software is a simple program that sits between the client application and the database server that can monitor, analyze or transform their communications. Such flexibility allows for a wide variety of uses such as load balancing, query analysis and query filtering. In the following subsections we describe the implementation details for each of the three main modules of our framework architecture.

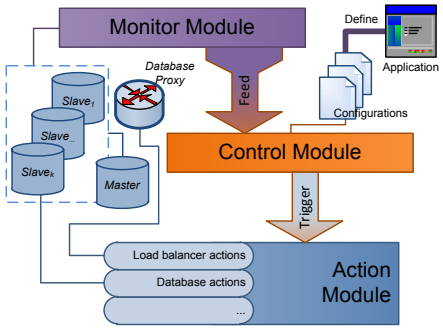



Fig. 1: Framework Architecture

### A. Monitor module

The monitor module is responsible for tracking the replication delay between the master database and each database replica. The replication delay (also know as data staleness window) for each replica is computed by measuring the time difference of two associated local timestamps committed on the master and the database replica. Therefore, a *Heartbeats* database is created in the master and each synchronized slave database server. Each *Heartbeats* database maintains a '*heartbeat*' table with two fields: an *id* and a *timestamp*. A database request to insert a new record with a global id and a local timestamp is periodically sent to the master. Once the insert record request is replicated to the slaves, every slave re-executes the request by committing the same global id and its own local timestamp. The update frequency of record in the master is configurable, named as *heartbeat interval* in millisecond unit (the default configuration of the heartbeat interval is set to be 1000 milliseconds in our experiments). While records are updated in the master database and propagated over all slaves periodically, the monitor module maintains a pool of threads that are run frequently to read up-to-date records from the master and slaves. The read frequency is also a configurable parameter in millisecond unit, known as *monitor interval*. In order to reduce the burden of the repetitive read request on the database replicas, all records are only fetched

once, and all local timestamps extracted from records are kept locally in the monitor module for further calculation.

The replication delay calculation between the master and a slave is initiated by the corresponding thread of the slave every time after fetching the records. In the general case of assuming there are $n$ and $k$ local timestamps in total in the master array ($timestamps_m$) and the slave array ($timestamps_s$), the slave's $i^{\text{th}}$ replication delay $delay[i]$ is computed as follows:

$$delay[i] = timestamps_s[i] - timestamps_m[i] \quad (1)$$

where $i \leq k = n$ and the master and the slave databases are fully synchronized. In the case of $k < n$ and there is a partial synchronization between the master and the slave databases which composes of consistent part and inconsistent part, the computation of the $delay[i]$ of the slave breaks into two parts: The delay of the consistent part with $i \leq k$ is computed using Equation(1). The delay of the inconsistent part with $k < i \leq n$ is computed as follows:

$$\begin{aligned} delay[i] \quad = \quad & timestamps_s[k] - timestamps_m[k] \\ & + timestamps_m[i] - timestamps_m[k] \quad (2) \end{aligned}$$

In the case of $n < k$ where indeterminacy could happen due to the missing of $k + 1^{\text{th}}$ local timestamp and beyond (this situation could happen when a recent fetch of the slave occurs later than the fetch of the master), the $delay[i]$ of the slave uses Equation (1) for $i \leq n$ and the $delay[i]$ of the slave for $n < i \leq k$ will be neglected as there is no appropriate local timestamps of the master that can be used for calculating the replication delay. The neglected calculations will be carried out later after the array of the master is updated.

### B. Control module

The control module maintains the information about the configurations of the load balancer (e.g. proxy address, proxy script), the configurations of the monitor module (e.g. heartbeat interval, monitor interval), the access information of each database replica (e.g. host address, port number, user name, and password), the location information of each replica (e.g. us-east, us-west, eu-west) in addition to the application-defined SLA of the tolerated replication delay (data freshness) of each replica. In practice, the SLA of the replication delay for each replica ($delay_{sla}$) is defined as an integer value in the unit of millisecond which represents two main components:

$$delay_{sla} = delay_{rtt} + delay_{tolerance}$$

where the round-trip time component of the SLA replication delay ($delay_{rtt}$) is the average round-trip time from the master to the database replica. In particular, it represents the minimum delay cost for replicating data from the master to the associated slave. The tolerance component of the replication delay ($delay_{tolerance}$) is defined by a constant value which represents the tolerance limit of the period of the time for the replica to be inconsistent. This tolerance component can vary from one replica to another depending on many factor such as the application requirements, the geographic location of the replica, and the workload characteristics and the load balancing strategy of each application.

One of the main responsibilities of the control module is to trigger the action module for adding a new database replica, when necessary, in order to avoid any violation in the application-defined SLA of data freshness for the active database replicas. In our framework implementation, we follow an intuitive strategy that triggers the action module for adding a new replica when it detects a number of continuous up-to-date monitored replication delays of a replica which exceeds its application-defined threshold ($T$) of SLA violation of data freshness. In other words, for a running database replica, if the latest $T$ monitored replication delays are violating its SLA of data freshness, the control module will trigger the action module to activate the geographically closest replica (for the violating replica). It is worthy to note that the strategy of the control module in making the decisions (e.g. the timing, the placement, the physical creation) regarding the addition a new replica in order to avoid any violence of the application-defined SLA can play an important role in determining the overall performance of the framework. However, it is not the main focus of this paper to investigate different strategies for making these decisions. We leave this aspect for future work.

In our previous work, it has been noted that the effect of the configurations of geographic location of the database replica is not as significant as the effect of the overloading workloads in increasing the staleness window of the database replicas [2]. Therefore, the control module can decide to stop an active database replica when it detects a decreasing workload that can be served by less number of database replicas without violating the application-defined SLAs in order to reduce the monetary cost of the running application.

### C. Action module

The action module is responsible for adding a new database replica when triggered by the action module. In general, adding a new database replica involves extracting database content from an existing replica and copying that content to a new replica. In practice, the time of executing these operations mainly depends on the database size. To provision database replicas in a timely fashion, it is necessary to periodically snapshot the database state in order to minimize the database extraction and copying time to that of only the snapshot synchronization time. Apparently, there is a tradeoff between the time to snapshot the database, the size of the transactional log and the amount of update transactions in the workload. This tradeoff can be further optimized by applying recently proposed live database migration techniques [5], [18]. In order to keep our experiments focused on the main concerns of our framework, we simply rely on a set of hot backups which are originally not used for serving the application requests, but kept synchronized, and then can be turned for being active and used by the load balancer for serving the application requests when the action module is triggered for adding a new replica. We leave the study of the cost and effect of the live database migration activities for our future work.

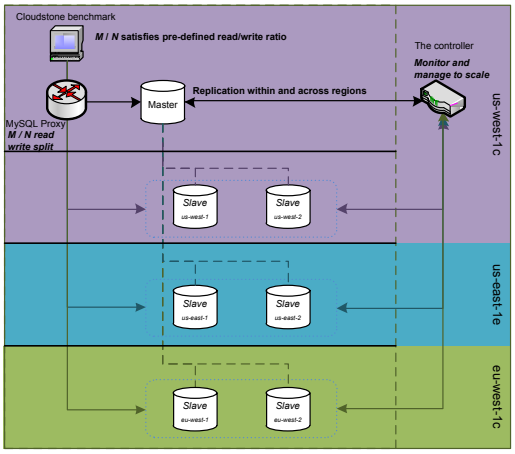



Fig. 2: Experiment setup

## IV. Experimental Evaluation

### A. Experiment design

The Cloudstone benchmark[1] mimics a Web 2.0 social events calendar that allows users to perform individual operations (e.g. browsing, searching and creating events) as well as social operations (e.g. joining and tagging events)[19]. The original benchmark has been designed as a performance measurement tool for all components of a Web 2.0 application (e.g. web application, database and etc.). We implemented a database-focused version of the Cloudstone benchmark[2] which only keeps and enhances the database tier [2]. In particular, we re-implemented the business logic of the application in a way that a user's operation can be processed directly at the database tier without any intermediate interpretation at the web server tier. We also included a connection pool (i.e. DBCP[3]) to reuse connections that have been released by other users who have completed their operations in order to save the overhead of creating a new connection for each operation.

On the database tier, we are using a MySQL Cluster which is deployed in Linux environment where every database replica contains an *Olio* database of the Cloudstone benchmark and a *Heartbeats* database for monitoring the replication delay in addition to our plug-in software which is used for achieving fine-grained measurements of the replication delay that consists of a user defined time/date function at microsecond resolution and a forced clock synchronization with multiple time servers every second via NTP[4] protocol. MySQL Proxy[5] with read and write split enabled resides in the middle between the benchmark and the database replicas and acts as a load balancer to forward write and read operations to the master and slaves correspondingly.

[1]http://radlab.cs.berkeley.edu/wiki/Projects/Cloudstone
[2]http://code.google.com/p/clouddb-replication/
[3]http://commons.apache.org/dbcp/
[4]http://www.ntp.org/
[5]https://launchpad.net/mysql-proxy

### B. Experiment setup

Figure 2 illustrates the setup of our experiments in Amazon EC2 platform. The experiment setup is a multiple-layer implementation. The first layer represents the Cloudstone benchmark which generates an increasing workload of database requests with fixed read/write ratio. The benchmark is deployed in a large instance to avoid any overload on the application tier. The second layer hosts the MySQL Proxy and our application-managed replication controller. The third layer represent the database tier (MySQL Cluster) that consists of all the database replicas where the master database receives the write operations from the load balancer and then it becomes responsible for propagating the writesets to all the slave replicas. The master database runs in a small instance so that an increasing replication delay is expected to be observed along with an increasing workload [2]. The master database is closely located to the benchmark, the load balancer and the replication controller. They are all deployed in the location of *us-west*. The slave replicas are responsible for serving the read operations and updating the writesets. They are deployed in three regions, namely: *us-west*, *us-east-1e* and *eu-west*. All slaves run in small instances for the same reason of provisioning the master instance.

We implemented two sets of experiments in order to evaluate the effectiveness of our application-managed replication controller in terms of its effect on the end-to-end system throughput and the replication delay for the underlying database replicas. In the first set of experiments, we fix the value of the tolerance component ($delay_{tolerance}$) of the SLA replication delay to 1000 milliseconds and vary the monitor interval ($intvl_{mon}$) among the following set of values: 60, 120, 240 and 480 seconds. In the second set of experiments, we fix the monitor interval ($intvl_{mon}$) to 120 seconds and adjusts the SLA of replication delay ($delay_{sla}$) by varying the tolerance component of the replication delay ($delay_{tolerance}$) among the following set of values: 500, 1000, 2000 and 4000 milliseconds. We have been evaluating the round-trip component ($delay_{rtt}$) of the replication delays SLA ($delay_{sla}$) for the database replicas in the three geographical regions of our deployment by running *ping* command every second for a 10 minutes period. The resulting average three round-trip times ($delay_{rtt}$) are 30, 130 and 200 milliseconds for the master to slaves in *us-west*, *us-east*, and *eu-west* respectively. Every experiment runs for a period of 3000 seconds with a starting workload of 220 concurrent users and database requests with read/write ratio of $80/20$. The workload gradually increases in steps of 20 concurrent users every 600 seconds so that each experiment ends with a workload of 300 concurrent users. Each experiment deploys 6 replicas in 3 regions where each region hosts two replicas: the first replica is an active replica which is used from the start of the experiment for serving the database requests of the application while the second replica is a hot backup which is not used for serving the application requests at the beginning of the experiment but can be added by the action module, as necessary, when triggered by the control module. Finally, in addition to the two

TABLE I: The effect of the the adaptive replication controller on the end-to-end system throughput

| Experiment Parameters | The monitor interval ($intvl_{mon}$) (in seconds) | The tolerance of replication delay ($delay_{tolerance}$) (in milliseconds) | Number of running replicas | Running time of all replicas (in seconds) | End-to-end throughput (operations per seconds) | Replication delay |
|---|---|---|---|---|---|---|
| Baselines with fixed number of replicas | *N/A* | *N/A* | 3 | 9000 | 22.33 | Fig. 3a |
| | *N/A* | *N/A* | 6 | 18000 | 38.96 | Fig. 3b |
| Varying the monitor interval ($intvl_{mon}$) | 60 | 1000 | $3 \rightarrow 6$ | 15837 | 38.43 | Fig. 3d |
| | 120 | 1000 | $3 \rightarrow 6$ | 15498 | 36.45 | Fig. 3c |
| | 240 | 1000 | $3 \rightarrow 6$ | 13935 | 34.12 | Fig. 3e |
| | 480 | 1000 | $3 \rightarrow 6$ | 12294 | 31.40 | Fig. 3f |
| Varying the tolerance of replication delay ($delay_{tolerance}$) | 120 | 500 | $3 \rightarrow 6$ | 15253 | 37.44 | Fig. 3g |
| | 120 | 1000 | $3 \rightarrow 6$ | 15498 | 36.45 | Fig. 3c |
| | 120 | 2000 | $3 \rightarrow 6$ | 13928 | 36.33 | Fig. 3h |
| | 120 | 4000 | $3 \rightarrow 6$ | 14437 | 34.68 | Fig. 3i |

sets of experiments, we conducted two experiments without our adaptive replication controller in order to measure the end-to-end throughputs and replication delays of 3 (minimum number of running replicas) and 6 (maximum number of running replicas) slaves in order to measure the baselines of our comparison. For all experiments, we have set the value of the *heartbeat interval* ($intvl_{heart}$) to 1 second and we set the value of the threshold ($T$) for the maximum possible continuous SLA violations for any replica using the following formula: $T = \frac{intvl_{mon}}{intvl_{heart}}$.

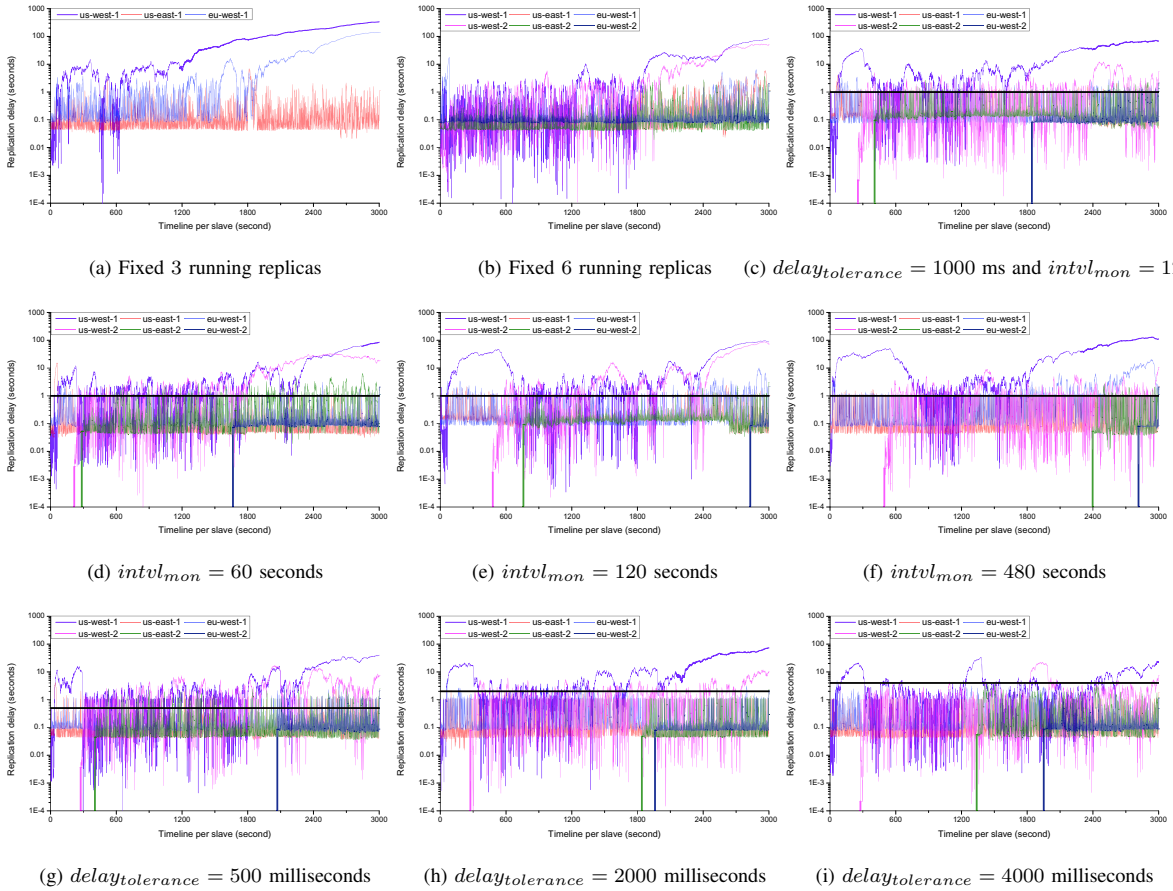
### C. Experiment results

*1) End-to-end throughput:* Table I presents the end-to-end throughput results for our set of experiment with different configuration parameters. The *baseline* experiments represent the *minimum* and *maximum* end-to-end throughput results with 22.33 and 38.96 operations per second, respectively. They also represent minimum and maximum baseline for the running time of all database replicas with 9000 (3 running replicas, with 3000 seconds running time of each replica from the beginning to the end of the experiment) and 18000 (6 running replicas, with 3000 seconds running time of each replica) seconds, respectively. The end-to-end throughput of the other experiments fall between the two baselines based on the variance on the monitor interval ($intvl_{mon}$) and the tolerance of replication delay ($delay_{tolerance}$). Each experiment starts with 3 active replicas then the number of replicas gradually increases during the experiments based on the configurations of the monitor interval and the SLA of replication delay parameters until it finally ends with 6 replicas. Therefore, the total running time of the database replicas for the different experiments fall within the range between 9000 and 18000 seconds. Similarly, the end-to-end throughput delivered by the adaptive replication controller for the different experiments fall within the end-to-end throughput range produced by the two baseline experiments of 22.33 and 38.96 operations per second. However, it is worth noting that the end-to-end throughput can be still affected by a lot of performance variations in the cloud environment such as hardware performance variation, network variation and warm up time of the database replicas. In general, the relationship between the running time of all slaves and end-to-end throughput is not straightforward. Intuitively, a longer monitor interval or a longer tolerance of replication delay usually postpones the addition of new replicas and consequently reduces the end-to-end throughput. The results show that the tolerance of the replication delay parameter ($delay_{tolerance}$) is more sensitive than the monitor interval parameter ($intvl_{mon}$). For example, having the values of the tolerance of the replication delay equal to 4000 and 1000 result in longer running times of the database replicas than having the values equal to 2000 and 500. On the other side, the increase of running time of all replicas shows a linear trend along with the increase of the end-to-end throughput. However, a general conclusion might not easy to draw because the trend is likely affected by the workload characteristics.

*2) Replication delay:* Figure 3 illustrates the effect of the adaptive replication controller on the performance of the replication delay for the cloud-hosted database replicas. Figure (3a) and Figure (3b) show the replication delay of the two baseline cases for our comparison. They represent the experiments of running with a fixed number of replicas (3 and 6 respectively) from the starting times of the experiments to their end times. Figure (3a) shows that the replication delay tends to follow different patterns for the different replicas. The two trends of *us-west-1* and *eu-west-1* surge significantly at 260 and 280 users, respectively. On the same time, the trend of *us-east-1* tends to be stable through out the entire running time of the experiment. The main reason behind that is the performance variation between the hosting EC2 instances for the database replicas[6]. Due to the performance differences between the physical CPUs specifications, *us-east-1* is able to handle the amount of operations that saturate *us-west-1* and *eu-west-1*. Moreover, with an identical CPU for *us-west-1* and *eu-west-1*, the former seems to surge at an earlier point than the latter. This is basically because of the difference in the geographical location of the two instances. As illustrated in Figure (2), the MySQL Proxy location is closer to *us-west-1* than *eu-west-1*. Therefore, the forwarded database operations by the MySQL Proxy take less time to *us-west-1* than to *eu-west-1* which leads to more congestion on the *us-west-1* side. Similarly, in Figure (3b), the replication delay tends to surge in both *us-west-1* and *us-west-2* for the same reason of the difference in the geographic location of the underlying database replica.

Figures (3c), and (3g) to (3i) show the results of the replication delay for our experiments using different values for the monitor interval ($intvl_{mon}$) and the tolerance of replication delay ($delay_{tolerance}$) parameters. For example, Figure (3c)

[6]Both *us-west-1* and *eu-west-1* are powered by Intel(R) Xeon(R) CPU E5507 @ $2.27GHz$, whereas *us-east-1* is deployed with a better CPU, Intel(R) Xeon(R) CPU E5645 @ $2.40GHz$




(a) Fixed 3 running replicas (b) Fixed 6 running replicas (c) $delay_{tolerance} = 1000$ ms and $intvl_{mon} = 120$ sec

(d) $intvl_{mon} = 60$ seconds (e) $intvl_{mon} = 120$ seconds (f) $intvl_{mon} = 480$ seconds

(g) $delay_{tolerance} = 500$ milliseconds (h) $delay_{tolerance} = 2000$ milliseconds (i) $delay_{tolerance} = 4000$ milliseconds

Fig. 3: The performance of the the adaptive management of the replication delay for the cloud-hosted database replicas.

shows that the *us-west-2*, *us-east-2*, and *eu-west-2* replicas are added in sequence at $255^{th}$, $407^{th}$ and $1843^{th}$ seconds, where the drop lines are emphasized. The addition of the three replicas are caused by the SLA-violation of the *us-west-1* replicas at different periods. In particular, there are four SLA-violation periods for *us-west-1* where the period must exceed the monitor interval, and all calculated replication delays in the period must exceed the SLA of replication delay. These four periods are: 1) 67:415 (total of 349 seconds). 2) 670:841 (total of 172 seconds). 3) 1373:1579 (total of 207 seconds). 4) 1615:3000 (total of 1386 seconds). The adding of new replicas is only triggered on the $1^{st}$ and the $4^{th}$ periods based on the time point analysis. The $2^{nd}$ and the $3^{rd}$ periods do not trigger the addition of any new replica as the number of detected SLA violations does not exceed the defined threshold ($T$).

Figures (3c), and (3d) to (3f) show the effect of varying the monitor interval ($intvl_{mon}$) on the replication delay of the different replicas. The results show that *us-west-2* is always the first location that add a new replica because it is the closest location to *us-west-1* which hosts the replica firstly violates its defined SLA data freshness. The results also show that as the monitor interval increases, the triggering points for adding new replicas are usually delayed. On the contrary, the results of Figure (3c) and Figures (3g) to (3i) show that increasing the value of the tolerance of the replication delay parameter ($delay_{tolerance}$) does not necessarily cause a delay in the triggering point for adding new replicas.

In general, the results of our experiments show that the adaptive replication controller can play an effective role on reducing the replication delay of the underlying replicas by adding new replicas when necessary. It is also observed that with more replicas added, the replication delay for the overloaded replicas can dramatically drop. Moreover, it is more cost-effective in comparison to the over-provisioning approach for the number of database replicas that can ensure low replication delay because it adds new replicas only when necessary based on the application-defined SLA of data freshness for the different underlying database replicas.

## V. Related Work

The *application-managed* approach presented in this paper is not the only option to have scalable persistent data store in the cloud. Two other approaches are in widespread use. The *platform-supplied storage layer* approach relies on a new wave of storage platforms named as key-value stores or NoSQL (Not

Only SQL) systems. These systems are designed to achieve high throughput and high availability by giving up some functionalities that traditional database systems offer such as joins and ACID transactions [20]. For example, most of NoSQL systems offer weaker consistency properties (e.g. eventual consistency [8]). Cloud offerings of this scenario include Amazon S3, Amazon SimpleDB and Microsoft Azure Table Storage. In practice, migrating existing software application that uses relational database to NoSQL offerings would require substantial changes in the software code due to the differences in the data model, the query interface and the support of transaction management. In addition, developing applications on top of an eventually consistent datastore requires a higher effort compared to that of traditional databases [21].

The *relational database as a service* is another approach in which a third party service provider hosts a relational database as a service [22]. Such services alleviate the need for their users to purchase expensive hardware and software, deal with software upgrades, and hire professionals for administrative and maintenance tasks. Cloud offerings of this scenario include Amazon RDS and Microsoft SQL Azure. For example, Amazon RDS provides access to the capabilities of MySQL or Oracle database while Microsoft SQL Azure has been built on Microsoft SQL Server technologies. The migration of the database tier of any software application to a relational database service is supposed to require minimal effort if the underlying RDBMS of the existing software application is compatible with the offered service. However, many relational database systems are not, yet, supported by the DaaS paradigm (e.g. DB2, Postgres). In addition, some limitations or restrictions might be introduced by the service provider for different reasons[7].

A common feature to the different cloud offerings of the *platform-supplied storage services* and the *relational database services* is the creation and management of multiple replicas of the stored data while a replication architecture is running behind-the-scenes to enable automatic failover management and ensure high availability of the service. In general, replicating for performance differs significantly from replicating for availability or fault tolerance. The distinction between the two situations is mainly reflected by the higher degree of replication, and as a consequence the need for supporting weak consistency when scalability is the motivating factor for replication [23]. In addition, the *platform-supplied storage layer* and the *relational database as a service* are mainly designed for multi-tenancy environment and they are more centric to the perspective of the cloud provider. Therefore, they do not provide support for any flexible mechanisms for scaling a single-tenant system (consumer perspective).

## VI. Conclusion

We presented an adaptive application-managed controller for replicating cloud-hosted relational databases using virtual machine technology. The controller provides the software applications with flexible mechanisms for maintaining several database replicas in different data centers with different levels of service level agreements (SLA) for their data freshness. The presented replication controller represents a main component of our CloudDB AutoAdmin project[8] [1]. As a future work, we plan to extend our SLA-focused management for cloud databases to a more fine grained level, i.e. query-focused SLA. In particular, we will extend our framework to support a variety of read operations with different SLA of data freshness (e.g. read the most recent data, read data not older than a consistency window of $t$ time units, or read data with a consistency window between $t_1$ and $t_2$ time units).

[7]http://msdn.microsoft.com/en-us/library/windowsazure/ee336245.aspx

[8]http://cdbslaautoadmin.sourceforge.net/

## References


[1] S. Sakr, L. Zhao, H. Wada, and A. Liu, "CloudDB AutoAdmin: Towards a Truly Elastic Cloud-Based Data Store," in *ICWS*, 2011.

[2] L. Zhao, S. Sakr, A. Fekete, H. Wada, and A. Liu, "Application-Managed Database Replication on Virtualized Cloud Environments," in *Data Management in the Cloud (DMC), ICDE Workshops*, 2012.

[3] A. A. Soror and al., "Database Virtualization: A New Frontier for Database Tuning and Physical Design," in *ICDE Workshops*, 2007.

[4] ——, "Automatic virtual machine configuration for database workloads," in *SIGMOD*, 2008.

[5] E. Cecchet and al., "Dolly: virtualization-driven database provisioning for the cloud," in *VEE*, 2011.

[6] E. Brewer, "Towards robust distributed systems," in *PODC*, 2000.

[7] D. J. Abadi, "Data Management in the Cloud: Limitations and Opportunities," *IEEE Data Eng. Bull.*, vol. 32, no. 1, 2009.

[8] W. Vogels, "Eventually consistent," *Queue*, vol. 6, pp. 14–19, October 2008. [Online]. Available: http://doi.acm.org/10.1145/1466443.1466448

[9] 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," in *SOSP*, 2007.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," in *OSDI*, 2006.

[11] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.

[12] J. Schad, J. Dittrich, and J. Quiané-Ruiz, "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance," *PVLDB*, vol. 3, no. 1, 2010.

[13] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez, *Database Replication*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[14] D. Florescu and D. Kossmann, "Rethinking cost and performance of database systems," *SIGMOD Record*, vol. 38, no. 1, 2009.

[15] T. Kraska and al., "Consistency Rationing in the Cloud: Pay only when it matters," *PVLDB*, vol. 2, no. 1, 2009.

[16] K. Keeton, C. B. M. III, C. A. N. Soules, and A. C. Veitch, "LazyBase: freshness vs. performance in information management," *Operating Systems Review*, vol. 44, no. 1, 2010.

[17] S. Sakr and A. Liu, "SLA-Based and Consumer-Centric Dynamic Provisioning for Cloud Databases," in *IEEE CLOUD*, 2012.

[18] A. Elmore and al., "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in *SIGMOD*, 2011.

[19] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," in *Proc. of Cloud Computing and Its Applications (CCA)*, 2008.

[20] S. Sakr, A. Liu, D. M. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *IEEE Communications Surveys and Tutorials*, vol. 13, no. 3, 2011.

[21] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective," in *CIDR*, 2011, pp. 134–143.

[22] D. Agrawal and al., "Database Management as a Service: Challenges and Opportunities," in *ICDE*, 2009.

[23] E. Cecchet and al., "Middleware-based Database Replication: The Gaps Between Theory and Practice," in *SIGMOD*, 2008.