2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing

Towards Elastic Multi-Tenant Database Replication with Quality of Service

Flávio R. C. Sousa, Javam C. Machado
Departament of Computer Science
Federal University of Ceara
Fortaleza, Brazil
{sousa,javam}@ufc.br

***Abstract*—Database systems serving cloud platforms must handle a large number of applications or tenants. Multi-tenancy database has been prevalent for hosting multiple tenants within a single DBMS while enabling effective resource sharing. Providing such performance goals is challenging for providers as they must balance the performance that they can deliver to tenants and the operating costs. In such database systems, a key functionality for service providers is database replication, which is useful for availability, performance, elasticity, and quality of service. This paper presents RepliC, an approach to database replication in the cloud with quality of service, elasticity, and support to multi-tenancy. In order to evaluate RepliC, some experiments that measure the quality of service and elasticity are presented. Our experiment results confirm that RepliC ensures the quality of service with small SLA violations, while using resources efficiently.**

***Keywords*-Cloud Computing, Database, Replication, Elastic, Quality of Service.**

I. Introduction and Motivation

Cloud computing is an extremely successful paradigm of service-oriented computing and has revolutionized the way computing infrastructure is abstracted and used. Scalability, elasticity, pay-per-use pricing, and economies of scale are the major reasons for the successful and widespread adoption of cloud infrastructures. Since the majority of cloud applications are data-driven, database management systems (DBMSs) powering these applications are critical components in the cloud software stack [1].

Many companies expect cloud providers to guarantee quality of service (QoS) using service level agreement (SLAs) based on performance aspects. Nevertheless, in general, providers base their SLAs only on the availability of services. Therefore, it is crucial that providers offer SLAs based on performance for their customers [2]. For many systems, most of the time consumed by requests is related to the DBMS (instead of the web server or application server). Thus, it is important that quality is applied to the DBMS to control the consumed time [3].

One major benefit claimed for cloud computing is elasticity, that adjusts the system's capacity at runtime by adding and removing resources without service interruption in order to handle the workload variation [4]. Stateless systems, such as web and application servers, can be scaled elastically by simply running more instances on additional physical nodes provisioned on demand. On the other hand, stateful systems, such as DBMSs, are hard to scale elastically because of the requirement of maintaining consistency of the database that they manage.

To increase resource utilization in the presence of smaller customers, providers employ multi-tenancy, in which multiple users or applications (server "tenants") are collocated on a single server while enabling effective resource sharing [5]. Sharing of resources at different levels of abstraction and distinct isolation levels results in various multi-tenancy models; the shared machine, shared process, and shared table models are well known [6]. This resource sharing improves providers' profits. Ideally, each tenant on a multi-tenant server is both unaware and unaffected by other tenants operating on the machine, and is afforded the same high performance it would receive on a dedicated server. However, dealing with unpredicted load patterns and elasticity is critical to ensure that the tenants' SLAs are met [1].

Database replication techniques have been used to improve availability, performance, and scalability in different environments [4]. Aspects related to multi-tenant database and elasticity to guarantee QoS have received little attention. These issues are important in cloud environments; the providers need to add replicas according to the workload to avoid SLA violation. Furthermore, they need to remove replicas when the workoad decreases and also to consolidate tenants.

Some solutions to database replication in the cloud do not address performance issues [7]. In [8] [9] it is not detailed the strategy of replication and the works [10] [11] do not implement elasticity, multi-tenancy or QoS. Other solutions do not address multi-tenant database [12] [13] [14] [15] [16] or they address only the shared machine multi-tenant model [17].

To overcome these limitations, this paper presents RepliC, an approach to database replication in the cloud with quality of service, elasticity, and support to multi-tenancy. In RepliC, elasticity adjusts the system's capacity at runtime by adding and removing replicas without service interruption in order to handle the workload variation. To the best of our knowledge, this is the first paper to formulate and explore the problem of how to work the replication in multi-tenant database environments with the goal of providing performance-based SLAs.

The major contributions of this paper are as follows:

978-0-7695-4862-3/12 $26.00 © 2012 IEEE
DOI 10.1109/UCC.2012.36

- We present RepliC, an elastic multi-tenant approach to database replication in the cloud. While other approaches for database replication have been proposed, our approach is unique in that it operates with different multi-tenant databases in the same machine and guarantees quality of service.
- We present an elastic database strategy for multi-tenant environments. Our strategy adds/removes replicas quickly according to the current workload. We demonstrate that this strategy is effective in managing replicas in a wide variety of scenarios.
- We present a full prototype implementation and experimental evaluation of our end-to-end system. We demonstrate that our approach guarantees quality of service with small SLA violations.

Problem Definition

In this work, we consider a set of database services and virtual machines (VMs) in the cloud. Each service is composed of set of replicas (i.e. tenants) of the same database and each VM has one or more replicas. The target workload is given by the rate of transactions. The SLA (i.e. response time) gives the objectives of each database's services. A penalty is applied to the transactions that exceed the SLA's value. The main problem is how to ensure the SLA for a set of database services according to the current workload while using resources efficiently with small SLA violations.

Organization: This paper is organized as follows: Section II explains RepliC's theoretical aspects. The implementation of the solution is described in Section III. The evaluation of RepliC is presented in Section IV. Section V surveys related work and, finally, Section VI presents the conclusions.

II. RepliC

RepliC is an approach to database replication in the cloud with quality of service, elasticity, and support to multi-tenancy. The elasticity adjusts the system's capacity by adding and removing replicas according to current workload. Monitoring techniques are used to check the status of the system and to make modifications in the replication configuration to ensure quality of service. RepliC uses information from databases and from the provider's infrastructure to provide resources.

Key-Value systems are extremely successful, but they have some limitations: simplified data model, lack of transactional support, and lack of attribute-based accesses can result in considerable overhead in re-architecting legacy applications which are predominantly based on relational DBMS technology [8]. Additionally, an application with smaller storage requirements (tens of MBs to a few GBs) would not take advantage of Key-Value stores. RepliC uses the relational data model and works with full replication.

Solutions for quality of service in the cloud may be cloud provider centric or customer-centric [3]. Cloud provider centric approaches attempt to maximize resource utilization while meeting an application's SLA in the face of fluctuating workloads. A customer-centric approach attempts to optimize the capacity needed by choosing the best server configuration that matches their needs. In this work, we focus on the cloud provider centric approach.

A. Multi-Tenant Database

There are many ways to deploy a database as a service on a cluster with multi-tenancy [18]: (1) all tenant data are stored together within the same database and the same tables with extra annotation such as "TenantID" to differentiate the records from different tenants [6]; (2) tenants are housed within a single database, but with separate schemas to differentiate their tables and provide better schema-level security; (3) each tenant is housed in a separate database within the same DBMS instance (for even greater security); (4) each tenant has a separate Virtual Machine (VM) with an OS and DBMS, which allows for resource control via VM management [9]. We use option 3 to implement multi-tenancy, since this option provides a good trade-off between wasted resources due to extra OSs in the VM method (option 4), and the complex manageability and security issues associated with options 1 and 2 [18].

With this multi-tenant strategy, interference may occur between tenants running in the same instance. This is related to the DBMS type used and to the workload [19]. Some works use analytical models or evaluate resource allocations by running actual experiments [18] [20]. However, both approaches incur a significant overhead once the workload changes. The former needs to recalibrate and re-validate models, whereas the latter has to run a new set of experiments to select a new resource allocation. During the adaptation period, the system may run with an inefficient configuration [21].

These strategies consider prior knowledge of the workload. However, cloud computing environments are dynamic and information about the workload is obtained at runtime. To address this problem, we use a strategy based on DejaVu [21], a framework for learning and reusing optimized resource allocations. RepliC monitors the resources, stores information about the tenants, their workloads and the limits within which they can be executed without SLA violation. Interference between two replicas P and Q happens in case of SLA violation due to workload increase on any of them. We model the interference among replicas with a set of rules. A rule of interference between P and Q, $P \rightarrow Q$, defines the workload limit within which SLA violation does not occur, as shown in the following rule: $P \rightarrow Q$ *if* $workload\_current_{P,Q} > workload\_SLA_{P,Q}$.

This definition can be applied to k services (e.g. $P \rightarrow Q, R, k$). In case of SLA violation with a value less than that of the current workload, a new rule with the former value is added to RepliC. When the workload changes, RepliC checks these rules, limits the transactions sent to the replicas with interference. To ensure quality of service, the workload excess is sent to a replica of the system with available resources. Otherwise, a new replica is started to handle this increased workload. These rules are reused in order to avoid executions similar to the ones that previously caused SLA violation.

B. Quality of Service for DBMS

In this paper, we extend the definitions of QoS for DBMS proposed in our previous work [3]. We use the SLA metric of response time: The value of maximum time response expected for processing the input workload. RepliC uses quantile-based SLAs and each service has a SLA associated with it (e.g. $Wiki_{SLA}$= 0.5s).

RepliC uses an efficient strategy to calculate collected data and combines different monitoring techniques to treat the variability of metrics. The collect process is performed six times with an interval of 10 seconds. For each collect process, RepliC calculates the median and standard deviation. Two medians with lower deviation are selected as the final values to be stored. To these values, it is applied an exponentially weighted moving average X'_t = α X_t + (1 - α) X'_{t-1}. If the monitored value is not in accordance with the defined SLA, more resources are added. On the other hand, if the SLA is satisfied over time, resources are removed. Thus, we can define a satisfaction function for the SLA (FSS). The function is satisfied if the SLA S_i is satisfied; otherwise, it is not.

SLA violation is computed as a penalty. Penalty is per unit of time that the transaction exceeds the SLA's value. Let *p* be the monetary value per unit of time that the transaction exceeds the SLA's value. Let TR_t denote the time needed to process a transaction *t* and SLA_t the agreed time to process a given transaction *t*. Penalty is defined as the sum of all transactions that did not meet SLA multiplied by the monetary value, according to the formula $Penalties = \sum_{i=1}^{n} \max(TR_t - SLA_t, 0) * p$.

C. Elastic Database Replication

Elasticity is the ability to grow and shrink the number of replicas to adapt to the incoming load [22]. Figure 1 shows the diagram to add and remove replicas. The process decision is based on the satisfaction function for the SLA (FSS), described in the previous subsection. The RepliC approach makes the process of removing replicas easy. When a replica needs to be removed from the system, RepliC selects the replica with the lowest workload and this replica simply does not receive any further requests. Thus, new clients will not be assigned to it. When all current clients on the replica have disconnected, the replica can leave the system. If the removal needs to be quick, then the replica can inform all connected clients that they should reconnect to another replica.

However, growing the number of replicas is a more complex issue. A simple approach would rely in stopping request processing, and then transfer the database state to the newly provisioned replica. But this would result in a loss of availability during the provisioning of the new replica. Elasticity requires a non-intrusive way of provisioning additional replicas. That is, we have to perform the state transfer to the new replica while at the same time providing the service without significant performance loss [22]. When a new replica needs to be provisioned, it has to receive the database state from a working replica. In our solution, the provisioning process is started when a replica is added in the system. After addition, RepliC

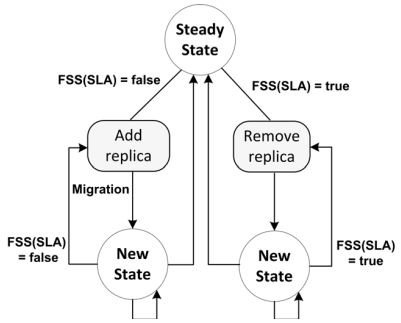

Fig. 1. Elasticity through adding and removing replicas.

uses a database copy (i.e. snapshot) and the log stores since the last backup. The new replicas are added and updated through data migration. Data migration involves applying snapshot and logs of missing updates to the new replica to bring it up-to-date.

D. Replica Consistency Management

In cloud storages, we have to provide high data availability. Therefore, updating all copies synchronously is not suitable due to longer response time, especially when there are storage node failures or when storage nodes are located in distributed clouds [23]. Our approach combines synchronous and asynchronous protocols. The use of this approach allows us to increase both responsiveness and availability. In this work, we focus on strong consistency and full replication. The *one-copy serializability* criterion is used in this work as its model of correctness.

RepliC uses virtual groups to manage the replicas [22]. We use a strategy based in our previous work [24] and in the work proposed by [23]. This strategy divides the replicas of each database into two groups: a read group, which executes read-only transactions, and an update group, which executes update transactions. The update group has two types of replicas: primary and secondary replicas. The replicas of the read group are called slaves. Update transactions are performed synchronously in the update group (i.e. in particular, the primary copy is always updated immediately). Thus, they are sent asynchronously to the read group. This replication strategy prevents a single point of failure and provides data availability and load balancing.

The replica of the update group that received the transaction is called primary replica, and is responsible for verifying conflicts with the other transactions that might be executing locally, sending a multicast with a total ordering property to the other replicas in this group. These replicas are called secondary replicas to the primary replica that sent the multicast, and perform a certification test, which verifies if a local transaction in the primary is in conflict with any other transaction executing in the secondary. The transactions that are executed by the update group receive a unique identifier, allowing RepliC to identify them. Modifications in the update group are serialized and sent continuously by the primary

to the read group through a multicast with a FIFO ordering property.

Figure 2 shows an example of the update group. In VM_1, the replica of database DB_1 is the primary to the client c_x, and the replica of database DB_1 in VM_2 and VM_N are the secondaries. On the other hand, the replica of database DB_1 in VM_2 is the primary to client c_y, and the other replicas of this database are secondaries. In this example, DB_1 in VM_M is always a secondary replica.

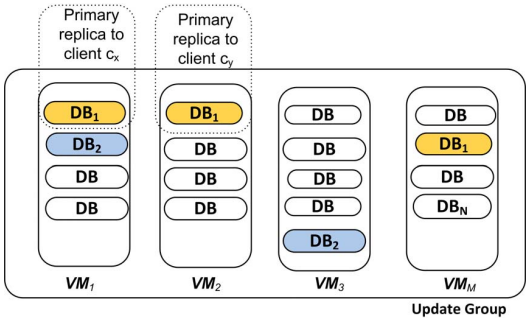

Fig. 2. Multi-tenant Update Group.

These modifications are pushed into local queues in each slave replica in the read group and executed in the same sequence as in the update group. The read group executes two types of transaction: propagation and refresh. Propagation transactions are executed while the replica is idle, when no read or refresh transactions are being executed, with the purpose of carrying out the updates. Refresh transactions are applied to execute the transactions in the local queue of a slave replica to bring it up-to-date in strong consistency. Figure 3 depicts an example of a read group. Each slave replica has a queue with the transactions to be executed. Another replica type (i.e. primary or secondary) has been allocated in the same VM, (e.g. DB_1). DB_1 has already executed the transactions in the queue, and it is completely up-to-date. DB_2, DB_3 and DB_4 need to execute refresh transactions.

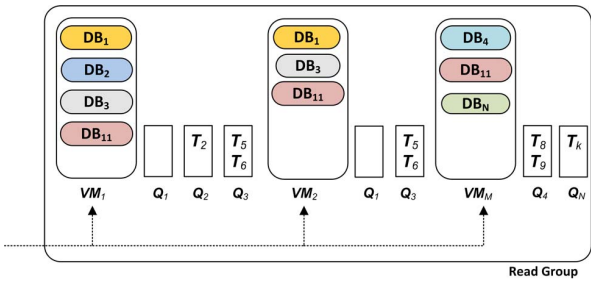

Fig. 3. Multi-tenant Read Group.

E. Database Fault Tolerance

There are different types of faults (e.g. hardware, networking, storage, operating system, and the DBMS instance). Infrastructure as a Service providers primarily apply redundant hardware to address the majority of these failures. In this work, to improve database service availability, each database has at least two replicas, and the replicas of the database service are not allocated in the same virtual machine.

In RepliC, we adopt the write-ahead logging scheme. Read-only transactions access the consistent snapshots of the database, and hence they do not need to perform the commit process. For update transactions, during the commit process, the log and commit records are stored in a distributed storage service (e.g. Amazon S3) for durability. RepliC uses an agent to check the DBMS state and its hosting virtual machine in order to detect failures. When a transaction is committed and persisted in the storage service, data can be retrieved through this service in case of failure.

Similar to [11], if the primary replica fails, the transaction can be retrieved through the log file, which stores the transactions to be executed. Replicas of the read and update group are allocated logically in a ring network topology. Thus, failures can be detected by the adjacent replicas. In case of failure in a secondary or slave replica, transactions are not sent to these replicas, and they are removed from the system. In the future, if such replica becomes operational, a process checks its state before adding it. Otherwise, a new secondary or slave replica is created. In this work, the replica recovery process is not addressed. However, the recovery strategy proposed by [22] can be used with some changes.

III. System Architecture and Implementation

The RepliC approach was implemented in our QoSDBC system [3]. The architecture of the QoSDBC is divided into three parts: *QoSDBDriver*, *QoSDBCoordinator* and *Agent*. *QosDBCDriver* is the component that provides access to the system. This driver replaces the database-specific JDBC driver but offers the same interface, without modification to the database engine. The *QoSDBCCoordinator* consists of a set of services that address the management of replicas. The *Agent* is a component added to each VM which is responsible for interacting with the VM and the DBMS. Specifically, this agent monitors and interacts with the DBMSs, while checking the state of monitored resources. An overview of the architecture of the QoSDBC is shown in Figure 4.

The Monitoring Service is responsible for managing the information collected by the agent about the state of the VMs and about the DBMS. The SLA Service manages agreements between costumers and service provider. A catalog stores the collected data and information about the resources. The Balancing Service distributes the requests to the replicas, and the Provisioning Service defines the resources required to ensure quality of service. Finally, the Scheduling Service directs requests to the replicas and keeps a log of the last transactions submitted to the system. The data is persisted in a distributed storage service. These RepliC services can be replicated to improve availability.

We have implemented a prototype of RepliC in Java with support to Amazon's EC2 public cloud. RepliC can track the database-workload and system resources. The agent collects

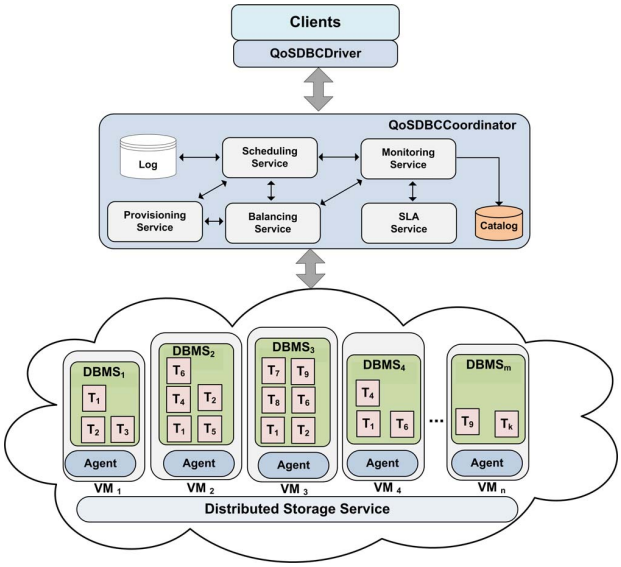

Fig. 4. QoSDBC architecture.

information about the database. Resource metrics can be obtained directly by query through the Amazon EC2 CloudWatch APIs. The monitored data is stored in an Amazon SimpleDB database. We use the Java APIs provided by Amazon EC2 to deploy, terminate, or reconfigure servers allocated to a database service. We created an Amazon Machine Image (AMI) with the DBMS and the agent, that allows starting a new VM quickly. The Amazon Elastic Block Store (EBS) is used for storing and sharing the snapshots of the databases.

For the data migration process, we are using the high performance backup tool Xtrabackup [25] which allows hot backup. XtraBackup produces a consistent-in-time snapshot of the database without interrupting transaction processing [5]. RepliC leverages this hot backup function to obtain a consistent snapshot for use in starting a new replica instance. To avoid the interference between the DBMS and the backup/restore process, we use Linux utility *pv*, which allows for limiting the amount of data passing through a Unix pipe. This strategy effectively limits the resource usage of both CPU and I/O.

IV. Experimental Evaluation

In this section, we describe the overall setup of our experimentation effort and the results we have obtained from it. These experiments are realized on cloud provider perspective.

A. Benchmark

There are some benchmarks for evaluating DBMSs in the cloud [26]. However, the development of standard cloud database services benchmarks is a challenge, because cloud systems have different characteristics (e.g. data model, levels of consistency, query language). For multi-tenant environments, it is of vital importance to use an appropriate benchmark to evaluate multi-tenant database systems. According to [27], there is no standard benchmark for this task. That work [27] proposes a strategy to generate a configurable database base schema. However, this strategy focuses on the table model and does not allow working with different multi-tenant models. Ahmad and Bowman [20] use multiple instances TPC-C and TPC-H to simulate a multi-tenant environment. We use a new strategy with different databases in our evaluation. Thus, we provide a full multi-tenant environment. These databases were provided by OLTPBenchmark framework [19]. This framework provides different benchmarks, such as TPC-C, Wikipedia, Twitter, YCSB and AuctionMark. OLTPBenchmark is designed to be able to produce variable rate, variable mixture load against any JDBC-enabled relational database.

B. Environment

We deploy a prototype of RepliC and conduct experiments on a cluster of commodity machines on Amazon EC2 in the same availability zone (us-west-1c). Each machine in our system runs on a small instance of EC2. This instance is a virtual machine with a 1.7 GHz Xeon processor, 1.7 GB memory and 160 GB disk capacity. We use the Ubuntu 12.04 operating system, MySQL 5.5 database with InnoDB engine and 128 MB buffer.

We define three services, where each service has a database (replicas are added/removed according to the workload) and a SLA (i.e. response time) associated to it. According to OLTPBenchmark, the following databases were generated: AuctionMark (450 MB), Wikipedia (600 MB), and YCSB (600 MB). For the databases, we used all transactions with default weights defined by OLTPBenchmark. The following values of response time were defined for each service: AM_{SLA} = 1s, $Wiki_{SLA}$ = 0.1s , $YCSB_{SLA}$ = 0.2s, and response time percentile of 95%.

C. Experiments

We conducted a detailed evaluation of our solution to verify the quality of service and elasticity. We evaluate RepliC's quality of service by comparing it with a reactive provisioning strategy such as Amazon Auto Scaling [28]. The behavior of an auto scaling engine follows a set of rules that must be defined. An auto scaling rule has the form of a pair consisting of an antecedent and a consequent. The rule was defined as follows: the number of VMs increases by one when average CPU utilization exceeds 80% for a continuous period of 2 minutes, and one VM is removed when CPU utilization drops below 20% with the same period [29].

The reactive provisioning implements primary copy protocol. For the RepliC and reactive provisioning strategy, two replicas were used initially and then their amounts change according to workload. In the reactive provisioning configuration, all primary replicas of each service were allocated in the same machine. In RepliC's configuration, the allocation was the following: VM_1=$\{AM_{Primary}$,$Wiki_{Slave}$,$YCSB_{Slave}\}$ and VM_2=$\{AM_{Slave}$,$Wiki_{Primary}$,$YCSB_{Primary}\}$.

D. Experimental Results

1) Quality of Service

In the first QoS experiment, we increased the YCSB service workload (i.e. rate) and fixed the workloads of the other services. The experiment consisted of increasing the system workload (i.e. rate) every 20 minutes. The interval to change the workload is similar to [13]. For the AuctionMark (AM) and Wiki, the rate was fixed at 1000. All services were performed with 100 clients/terminals.

Figure 5(a) shows the variation of the SLA response time metric with reactive provisioning/primary copy when the YCSB workload was increased. We can observe that the YCSB response time increased when the rate was increased. The reactive strategy took a long time to identify the increase in response time. This happened because the reactive strategy is resource-oriented and the threshold for making decision was not reached. Only after the rate reached to 3000, a new VM with a slave replica of YCSB was added to the system.

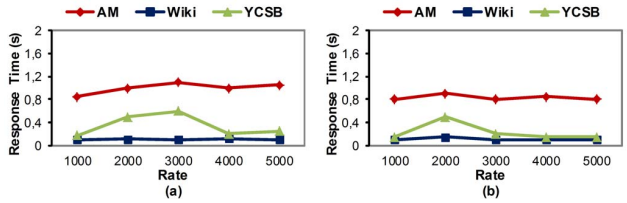

Fig. 5. Increase the rate of YCSB service - (a) Response Time with Reactive Provisioning - (b) Response Time with RepliC

Figure 5(b) presents the result with RepliC. With an increasing YCSB rate, RepliC's monitoring strategy identified when the workload increased (i.e. rate 2000) and added a new VM with a slave replica of YCSB. In this experiment, interference between the YCSB and other databases did not occur.

In the second QoS experiment, we increased the rate of all services simultaneously. Figure 6(a) shows the variation of SLA response time metric with reactive provisioning when the AM, Wiki, and YCSB workloads were increased. In this case, the response time of all services increases quickly. Two VMs with slave replicas of all databases were added when the rate increased to 2000. The primary copy used for reactive provisioning had a limit on the amount of transactions it could manage. Thus, the system became overloaded and the primary copy had problems processing and sending updates to the slave replicas. In addition, there was a lot of interference among the databases running on the same VM (i.e. primary copy), thus increasing the response time and violating the SLA.

RepliC's results are shown in Figure 6(b). The response time increased, but RepliC added two VMs with primary and slave replicas for each service. With the strategy of monitoring the environment and checking the interference among the replicas, RepliC identified the conditions under which the replicas can be executed with less SLA violation. RepliC was able to improve the response time by distributing the update transactions among the update replicas. In addition,

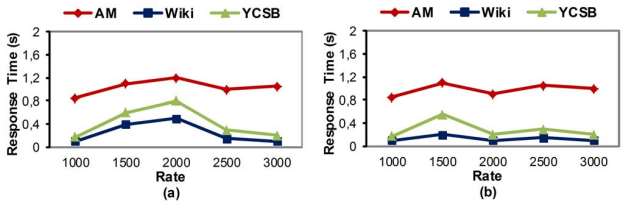

Fig. 6. Increase the rate of all services - (a) Response Time with Reactive Provisioning - (b) Response Time with RepliC

the efficient update replica allocation was the key to improve the response time since these replicas were distributed among the VMs. In this experiment, RepliC stored rules about the interference among the databases, mainly between AM and YCSB.

2) Elasticity

To analyze elasticity, an experiment was conducted varying the rate over a period of time. In the first elasticity experiment, we increased/decreased the YCSB service workload and fixed the workloads of the other services. The YCSB rates were set according to the following ordered pairs, which represent (time in minute, rate): (20, 1000), (40, 5000), (60, 1000), (80, 1000), and (100, 3000). The same values of AuctionMark and Wiki services in the first QoS experiment were used here.

Figure 7(a) shows the variation of SLA response time metric with reactive provisioning. With the increase in the YCSB workload, a new VM with a slave replica of YCSB was added, which received part of the transactions, and then the YCSB response time decreased. However, in time 60, with the decrease in workload, the CPU threshold was not reached (i.e. 20%) and, as such, the VM was not removed. The reactive provisioning uses a resource-based strategy and there is not a direct or linear relation between the resources allocated and the SLA. Therefore, this strategy does not work well with elasticity for stateful systems (e.g. DBMS).

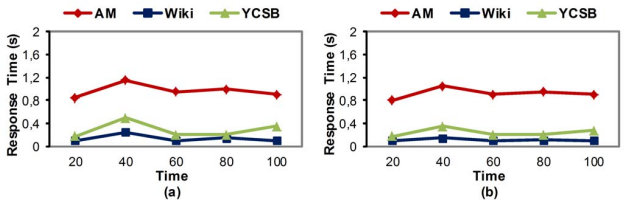

Fig. 7. Elasticity of YCSB service - (a) Response Time with Reactive Provisioning - (b) Response Time with RepliC

RepliC's results in this experiment are presented in Figure 7(b). With the increase in the YCSB workload, a new VM with a slave replica of YCSB was added and the SLA was guaranteed. With the decrease in the workload, a replica was removed, even the DBMS keeping current resources allocated, since the response time improved. This is related to RepliC's monitoring strategy, which identifies variation in

both workload and response time. Thus, RepliC changes the amount of replicas quickly.

In the second elasticity experiment, we evaluated the elasticity of all services simultaneously. Table I shows the rate of each service during the experiment.

	Rate		
Time	AM	Wiki	YCSB
20	1000	1000	1000
40	5000	5000	2000
60	5000	1000	1000
80	1000	5000	2000
100	1000	1000	1000

TABLE I
RATE OF EACH SERVICE DURING THE EXPERIMENT.

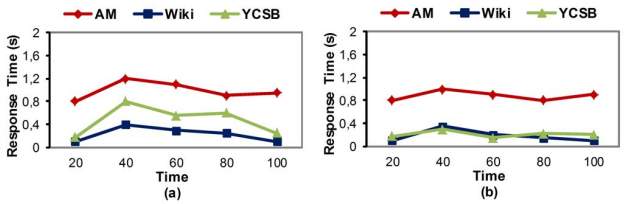

Fig. 8. Elasticity of all services - (a) Response Time with Reactive Provisioning - (b) Response Time with RepliC

Figure 8(a) shows the variation of SLA response time metric with reactive provisioning according to Table I. In time 40, with the increase in the rate of AM and Wiki, the response time increased to all services. Thus, a new replica of those databases is added. In time 60, with the decrease rate of Wiki, the response time remained high. The response time improved at the end of the experiment, when the workload of all services decreased. The reason for this behavior is that reactive provisioning has difficulty managing the increase and decrease of the workload, mainly due to interference among the databases running on the same machine and the overload of the primary copy.

Figure 8(b) presents RepliC's results. With the increase in workload, RepliC added two new VMs with primary/slave replicas of the AM and Wiki services. Thus, the response time decreases. In time 60, the workload of Wiki decreases and RepliC removes one slave replica of this service. In time 80, RepliC adds one new slave replica of Wiki. RepliC reused the rules stored about the interference between the AM and the YCSB in the second QoS experiment, thus reducing the SLA violations.

RepliC's synchronization strategy, which initially updates the replica from the update group, does not have to update every single replica every time a database is updated, thus improving performance. The efficient allocation of the primary replicas were the key to improve RepliC's response time since theses replicas were distributed among the machines. In addition, this approach reduces the interference among the slave and primary replicas, which are responsible for sending updates to slaves replicas. During the execution of these experiments, we observed a small amount (less than 3%) of aborts in the update group, due to the certification test. The average time to add a new VM was 55 seconds using the AMI created. In these experiments, RepliC used only one VM more than the reactive provisioning strategy, but guaranteed QoS, which shows that our solution uses resources efficiently.

3) SLA Violation

Table II shows the SLA violation for the previous experiments. In the first QoS experiment (Figure 5), the reactive provisioning presents 39% SLA violation in the YCSB service. RepliC ensured better quality. One of the reasons for it is that the transactions were distributed among the update and read groups, improving processing efficiency. In the QoS experiment (Figure 6), the reactive strategy presented an inferior result compared to RepliC. However, RepliC's values increased considerably. This is due to the fact that more messages were being sent from the update group to the read group. In these QoS experiments, RepliC had less SLA violation than the reactive strategy.

	Reactive provisioning			RepliC		
Experiment	AM	Wiki	YCSB	AM	Wiki	YCSB
QoS-Figure5	12%	7%	39%	1%	3%	14%
QoS-Figure6	20%	35%	47%	14%	10%	20%
Elastic-Figure7	14%	16%	27%	6%	5%	15%
Elastic-Figure8	26%	25%	51%	12%	14%	16%

TABLE II
PERCENTAGES OF SLA VIOLATIONS WITHOUT CONSIDERING OUR PROPOSED PENALTY MODEL.

In the elasticity experiment (Figure 7), RepliC presented less SLA violation, especially for the service that was changing the workload (i.e. YCSB service). For the experiment (Figure 8), the reactive strategy had a high value of violation for all services. Meanwhile, RepliC managed the workload variation and interference among databases better. Results of RepliC considering our proposed penalty model were even more significative. However, due to space limitations, we presented in Table II only the percentages of SLA violation without taking it into account.

V. RELATED WORK

SQL Azure [7] implements the primary copy protocol with strong consistency. SQL Azure also implements high availability, fault tolerance, and multi-tenancy. However, it does not address performance issues in quality of service. [10] presents the design and implementation of a data-management platform that can scale to a large number of small applications. This work implements various techniques for database deployment and SLA management that ensure high throughput and high availability. However, this platform has no elasticity and does not present different QoS metrics, such as response time. Elmore et al. [8] presents an elastic and autonomic multi-tenant database, but this work is in development and does not have implementation and evaluation.

Carlo et al. [9] introduces Relational Cloud, a scalable relational database-as-a-service for cloud computing environment. Relational Cloud focuses on efficient multi-tenancy, elastic scalability and database privacy. Relational Cloud does not detail the replication strategy nor does it address quality of service. In [17], SmartSLA is proposed, a cost-aware resource management system. SmartSLA uses machine learning techniques to learn a system performance model through a data-driven approach. SmartSLA implements the primary copy protocol asynchronously and only addresses the question of the number of replicas needed to maintain the SLA. In addition, SmartSLA only addresses the shared machine multi-tenant model.

Re:FRESHiT [30] combines eager and lazy replication protocols that take into account different levels of freshness. Re:FRESHiT uses a novel protocol capable of handling the propagation of updates to read-only nodes and the freshness-aware routing of requests in data clouds. The work [11] focuses on data replication in a virtualized environment and uses primary copy protocol. However, this work does not address elasticity, multi-tenancy or QoS. The CloudDB AutoAdmin project [16] presents an SLA definition for database and uses replication to ensure QoS. However, it does not address multi-tenancy issues.

There are some works that use virtual machine replication, such as Amazon RDS [15], RemusDB [12], Dolly [13], and FlurryDB [14]. In these works, each database replica runs in a separate virtual machine and the entire virtual machine is cloned. They use a strategy based on the two-phase commit protocol. So, these works do not address multi-tenancy and techniques for data replication, and only Dolly addresses quality of service.

VI. Conclusion and Future Work

This work presented RepliC, an approach to database replication in the cloud. RepliC intends to improve quality of service and addresses elasticity and multi-tenant model. The elasticity adjusts the system's capacity by adding and removing replicas according to the current workload. We evaluated the RepliC approach considering quality of service characteristics, including SLA violation issues. According to the analysis of the obtained results, RepliC improves the QoS for multi-tenant database environments. As future work, we intend to conduct experiments of availability and improve the process to add/remove replicas using Markov Decision Process. Other important issues to be addressed are related to cost and different multi-tenant models. Finally, we intend to conduct a study with techniques of machine learning to improve resource prediction.

Acknowledgment

This work is partly supported by the Amazon AWS in Education research grant award.

References

[1] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in *SIGMOD '11*, 2011, pp. 301–312.
[2] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *PVLDB*, vol. 3, no. 1, pp. 460–471, 2010.
[3] F. R. C. Sousa, L. O. Moreira, G. A. C. Santos, and J. C. Machado, "Quality of service for database in the cloud," in *CLOSER '12*, 2012, pp. 595–601.
[4] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez, *Database Replication*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.
[5] H. J. M. H. K. Sean Barker, Yun Chi and P. Shenoy, "Cut me some slack: Latency-aware live migration for databases," in *EDBT '12*, 2012, pp. 432–443.
[6] T. Kiefer and W. Lehner, "Private table database virtualization for dbaas," in *UCC '11*, 2011, pp. 328–329.
[7] Microsoft, "*Microsoft Azure*," 2012, http://www.microsoft.com/azure/.
[8] A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi, "Towards an elastic and autonomic multitenant database," in *NetDB '11*, 2011.
[9] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational cloud: a database service for the cloud," in *CIDR*, 2011, pp. 235–240.
[10] F. Yang, J. Shanmugasundaram, and R. Yerneni, "A scalable data platform for a large number of small applications," in *CIDR*, 2009, pp. 1–10.
[11] S. Savinov and K. Daudjee, "Dynamic database replica provisioning through virtualization," in *CloudDB '10*, 2010, pp. 41–46.
[12] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield, "Remusdb: Transparent high availability for database systems," *PVLDB*, vol. 4, no. 11, pp. 738–748, 2011.
[13] E. Cecchet, R. Singh, U. Sharma, and P. Shenoy, "Dolly: virtualization-driven database provisioning for the cloud," in *VEE '11*, 2011, pp. 51–62.
[14] M. J. Mior and E. de Lara, "Flurrydb: a dynamically scalable relational database with virtual machine cloning," in *SYSTOR '11*, 2011, pp. 1–9.
[15] Amazon, "*Amazon RDS*," 2012, http://aws.amazon.com/rds/.
[16] S. Sakr, L. Zhao, H. Wada, and A. Liu, "Clouddb autoadmin: Towards a truly elastic cloud-based data store," *IEEE ICWS*, pp. 732–733, 2011.
[17] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigümüs, "Intelligent management of virtualized resources for database systems in cloud environment," in *ICDE '11*, 2011, pp. 87–98.
[18] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan, "Towards multi-tenant performance slos," in *ICDE '12*, 2012, pp. 702–713.
[19] OLTPBenchmark, "*OLTPBenchmark*," 2012, http://www.oltpbenchmark.com.
[20] M. Ahmad and I. T. Bowman, "Predicting system performance for multi-tenant database workloads," in *DBTest '11*, 2011, pp. 1–6.
[21] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "Dejavu: accelerating resource allocation in virtualized environments," in *ASPLOS '12*, 2012, pp. 423–436.
[22] F. Perez-Sorrosal, M. Patio-Martinez, R. Jimenez-Peris, and B. Kemme, "Elastic si-cache: consistent and scalable caching in multi-tier architectures," *The VLDB Journal*, pp. 1–25, 2011.
[23] H. T. Vo, C. Chen, and B. C. Ooi, "Towards elastic transactional cloud storage with range query support," *PVLDB*, vol. 3, no. 1, pp. 506–517, 2010.
[24] F. R. C. Sousa, H. J. A. C. Filho, and J. C. Machado, "A new approach to replication of xml data," in *DEXA*, 2007, pp. 141–150.
[25] Percona, "*XtraBackup*," 2012, http://www.percona.com/software/percona-xtrabackup.
[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *ACM SoCC*, 2010, pp. 143–154.
[27] M. Hui, D. Jiang, G. Li, and Y. Zhou, "Supporting database applications as a service," in *ICDE*, 2009, pp. 832–843.
[28] Amazon, "*Amazon Auto Scaling*," 2012, http://aws.amazon.com/autoscaling/.
[29] S. Islam, K. Lee, A. Fekete, and A. Liu, "How a consumer can measure elasticity for cloud platforms," in *ICPE '12*, 2012, pp. 85–96.
[30] L. C. Voicu, H. Schuldt, Y. Breitbart, and H.-J. Schek, "Flexible data access in a cloud based on freshness requirements," *IEEE CLOUD*, pp. 180–187, 2010.