SO-1SR: Towards a self-optimizing One-Copy Serializability Protocol for Data Management in the Cloud

Ilir Fetai Heiko Schuldt
Databases and Information Systems Group
University of Basel, Switzerland
{ilir.fetai|heiko.schuldt}@unibas.ch

ABSTRACT

Clouds are very attractive environments for deploying different types of applications due to their pay-as-you-go cost model and their highly available and scalable infrastructure. Data management is an integral part of the applications deployed in the Cloud. Thus, it is of outmost importance to provide highly available and scalable data management solutions tailored to the needs of the Cloud. Data availability can be increased by using well-known replication techniques. Data replication also increases scalability in case of read-only transactions, but generates a considerable overhead for keeping the replicas consistent in case of update transactions. In order to meet the scalability demands of their customers, current Cloud providers use DBMSs that only support weak data consistency. While weak consistency is considered to be sufficient for many of the currently deployed applications in the Cloud, more and more applications with strong consistency guarantees, like traditional online stores, are moved to the Cloud. In the presence of replicated data, these applications require one-copy serializability (1SR). Hence, in order to exploit the advantages of the Cloud also for these applications, it is essential to provide scalable, available, low-cost, and strongly consistent data management, which is able to adapt dynamically based on application and system conditions. In this paper, we present SO-1SR (self-optimizing 1SR), a novel customizable load balancing approach to transaction execution on top of replicated data in the Cloud which is able to efficiently use existing resources and to optimize transaction execution in an adaptive and dynamic manner without a dedicated load balancing component. The evaluation of SO-1SR on the basis of the TPC-C benchmark in the AWS Cloud environment has shown that the SO-1SR load balancer is much more efficient compared to existing load balancing techniques.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management Systems—*Transaction processing; Concurrency*

CloudDB'13, October 28, 2013, San Francisco, CA, USA.

ACM 978-1-4503-2416-8/13/10
http://dx.doi.org/10.1145/2516588.2516590 ...$15.00.

Keywords

Data Replication, Data Consistency, Transaction Management, Load Balancing

1. INTRODUCTION

Data replication is a mechanism used to increase the availability of data by storing redundant copies (*replicas*) at usually geographically distributed sites. Especially in the Cloud, data replication is intensively used as it is essential for providing a high degree of availability. In case of read-only transactions, data replication can increase system scalability by using the additional processing capacities of the hosts where replicas reside to balance the load. For many applications that are newly to be deployed in the Cloud, one-copy serializability (1SR) is the desired level of data consistency for replicated systems. It guarantees *serializable execution of concurrent transactions* and a *one-copy* view on the data. The most common approaches to implement 1SR use lock-based protocols such as strict two-phase locking (S2PL) for providing serializable transaction execution, and two-phase commit (2PC) for synchronously updating all replicas. However, in case of update transactions, 1SR together with replicated data generates a considerable overhead and thus decreases the overall scalability of the system. According to Brewer's CAP theorem [6] it is impossible to jointly provide consistency, availability and partition tolerance in a distributed system. As partition tolerance is a mandatory requirement in distributed systems, this implies a trade-off between availability and consistency. The stronger the consistency level, the lower are availability and scalability (and vice-versa).

Due to their pay-as-you-go model, in Cloud environments also the *costs* that incur for guaranteeing a certain consistency level on top of replicated data have to be considered. Strong consistency is costly to enforce from both a performance and monetary point of view. Weak consistency, on the other hand, is cheaper but may lead to high inconsistency costs for compensating the effects of possible anomalies and access to stale data.

In an attempt to provide maximum scalability and availability, a first generation of Cloud DBMSs, such as NoSQL systems of key-value stores [8, 9], provide only weak consistency. This has been sufficient for satisfying the consistency requirements of simple Cloud applications. However, more sophisticated applications, like traditional online stores, governmental services, etc., are increasingly discovering the Cloud and its advantages (scalability, availability and the pay-as-you-go cost model) and thus are to be moved into

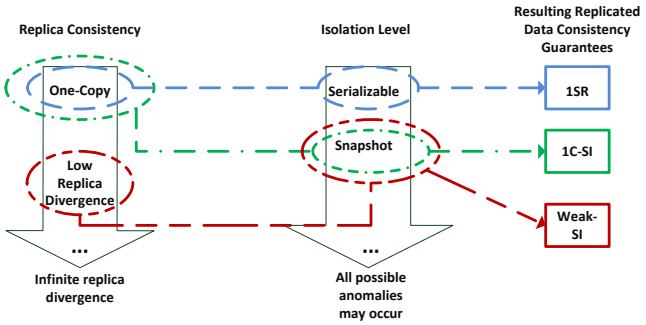


Figure 1: Replica Consistency and Concurrency Control

the Cloud. These applications essentially demand strong consistency guarantees and providing these guarantees on top of weakly consistent DBMS is very challenging [2]. At the same time, existing commercial and open source DBMSs which provide strong consistency cannot exploit the availability and scalability advantages of the Cloud.

As depicted in Figure 1, data consistency in replicated system has two aspects: first, *concurrency control (transaction isolation)* and second, *replica consistency*. For concurrency control, different isolation levels which define the possible anomalies and concrete protocols which implement a certain isolation level are available. Replica consistency controls the allowed divergence of replicas in comparison to the most recent data (i.e., the site where the most recent update operation has been executed). Combining both aspects leads to a broad spectrum of approaches that differ in terms of consistency, availability, scalability, and, especially in a Cloud context, the costs that incur.

As part of a long-term project, we are currently developing a comprehensive CloudDBMS that can be easily tailored to the particular needs of a Cloud application and that will allow to dynamically optimize transaction execution by selecting the best (w.r.t. cost and performance) combination of concurrency control and replica consistency. The core of this novel CouldDBMS is *SO-1SR* (self-optimizing 1SR), a novel protocol that allows to dynamically optimize all phases of the execution of transactions on replicated data in a Cloud (i.e., the selection of the best replica(s), the selection of replicas for eager commit, the point in time for lazily propagating changes to replicas) separately, albeit not independently. To the best of our knowledge, this is the first approach that holistically addresses optimization of transaction execution on top of replicated data in Cloud environments.

In this paper, we focus on the first phase of transaction execution, the selection of the best replicas in the Cloud, on the basis of the current load of the different replica sites. The contribution of the paper is threefold: first, we present the SO-1SR approach to optimize transaction execution without a dedicated load balancer. Second, we show how this approach can by dynamically adapted to changing requirements and/or systems. Third, we present the result of detailed evaluations based on the TPC-C benchmark in a typical Cloud setting using AWS resources.

The paper is structured as follows: Section 2 discusses related work. The system model of our CloudDBMS is presented in Section 3 and the SO-1SR protocol is introduced in Section 4. The evaluation results of SO-1SR's load balancer are presented in Section 5. Section 6 concludes.

2. RELATED WORK

In the last decade, data replication has attracted quite some attention in the research community. The main objective of the work is to design data replication mechanisms that are highly scalable on one hand, and on the other hand provide dedicated correctness guarantees, ideally 1SR [5]. The necessity for providing databases with support for strong consistency by still retaining the availability and scalability properties of NoSQL databases is best documented by recent initiatives such as Google's Spanner, a highly scalable, globally-distributed and consistent database [10].

In centralized (non-replicated) DBMS, the data consistency level is defined by the isolation level of transactions (cf. Figure 1). The focus of research in this area is to achieve serializable transaction execution at low overhead [7]. In replicated systems, replica consistency is the second pillar that has to be considered to determine the overall consistency guarantees. Current approaches usually provide a one-copy view by updating replicas eagerly, which generates a considerable overhead and leads to a decreased level of scalability [12]. Other approaches relax the replica consistency by updating replicas in a lazy manner. While this leads to an increase in scalability, it also implies inconsistencies that may be costly to resolve [15]. In [23], a fully decentralized approach is introduced which guarantees 1SR for replicated data. This has been extended by a refresh mechanism that allows replicas to refresh their data if transactions request data with a higher freshness than locally available [24]. However, both works rely on a tree based replication structure which distinguishes between read-only and updatable nodes, with updatable nodes being eagerly updated. The updates to the read-only nodes are propagated in a lazy fashion. [19] proposes an approach for increasing the throughput of OLAP queries by trading freshness for performance. It is also based on a replication scheme which distinguishes between read-only and update nodes and does not always provide freshest data. In contrast to [24, 19], our SO-1SR does not rely on a specific replication scheme.

A family of 1SR protocols avoids the use of 2PC and relies on efficient group communication [14]. Although this leads to decreased transaction response times, it does not provide a one-copy view on the data [17, 20]. In contrast to these approaches, our SO-1SR aims at holistically optimizing the different transaction phases separately, yet not independently in order to take advantage of the scalability provided by the Cloud and its cost model (cost-awareness). Existing data consistency protocols like [15, 11] are towards the first to incorporate the cost factor into the design. However, the cost is just one of the parameters in the optimization space; other important parameters like distance between replicas, replica type and capacity, data importance, data popularity, etc., which also strongly influence the overall performance are not considered. To the best of our knowledge, none of the existing approaches works has incorporated the optimization of transaction execution phase (load balancing) as part of the consistency protocol, although it has a huge impact on the overall performance (e.g., resources are blocked for a shorter time).

3. SYSTEM MODEL

SO-1SR assumes that transactional applications are built on top of a Cloud data environment which replicates data

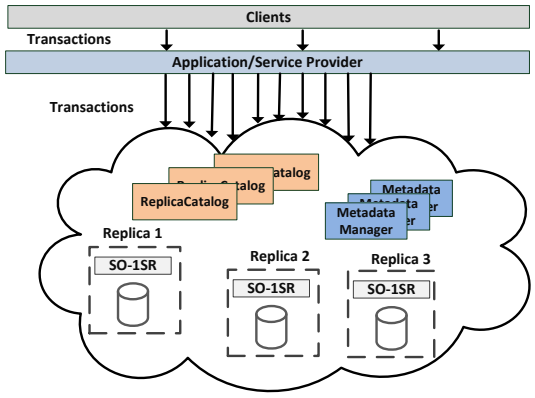


Figure 2: System Model

for availability and scalability inside the same or across different data centers. Further, we assume the SO-1SR middleware to be present at each replica node. Clients submit transaction to the application servers, which forward them to the SO-1SR middleware for optimal execution. SO-1SR is based on a fully replicated system with an update anywhere approach (multi-master replication) and a flat transaction model (cf. [1]). However, in order to provide strong consistency guarantees, protocols like 2PC or Paxos are needed. The goal of our SO-1SR is not to invent new protocols for replica synchronization that replace 2PC or Paxos, but to decrease latency by using clever optimization techniques at different phases of transaction life-cycle.

A transaction consists of a set of operations accessing objects uniquely identified by an *objectId* in read or write mode. There are two type of transactions, *read-only* and *update*; the latter contains at least one write operation. If a transaction T_i is assigned to replica R_k for execution, then R_k is called a *local replica*. All other replicas are *remote* to T_i. Each replica can serve read-only and update transactions. The underlying replication mechanism (static or dynamic) is not relevant to SO-1SR, however it relies on a *replica catalog* which manages information about the available replicas. The metadata needed by SO-1SR to achieve its goal is managed by a clustered MetadataManager. Figure 2 provides an overview of the SO-1SR system model.

3.1 User-Defined and System Parameters

Two different types of parameters influence the optimization behavior of SO-1SR: *user-defined parameters* and *system parameters*. The user specifying relevant parameters can be the application developer or architect who designs the transactional application. System parameters are dynamically collected by SO-1SR and analyzed for taking appropriate optimization decisions. In the current version of SO-1SR, the priority of transactions is a user-defined parameter; all others are system parameters.

Transaction Priority: Usually, application transactions have different priorities. If we take for instance an order-entry application as defined by the TPC-C benchmark [22], the transactions for ordering an item are more important than administrative transactions for checking the stock level of products, i.e., users are less tolerant to delays with increasing transaction importance. Hence, transactions may have different priorities and should be treated based on their priority. The priority can be specified at transaction level explicitely or implicitely by specifying the profit generated by the transaction or its response time requirements.

Replica Load: The load of a replica is defined by the number of requests being executed or waiting for execution in a queue. By properly balancing the load between replicas, the overall system performance can be increased. Advanced models are needed for the prediction of load and response time by taking into account transaction priorities. A load balancing mechanism should then balance the load so that the overall response time is minimized (and the profit maximized).

Replica Type: In a replicated data Cloud environment, different types of replicas may coexist. They may have different different processing capabilities and thus strongly influence the response time of transactions (as, for instance, different AWS instance types[1]).

Replica Distance: Requests (transaction execution, commit, etc.) need to be forwarded from one replica to another. Replicas may have different distances between each other and in combination with dynamic replication, replicas may be created/destroyed dynamically, which again influences the network distance between the replicas. Additionally, different bandwidth capabilities can be assigned to replicas.

3.2 Management of Metadata

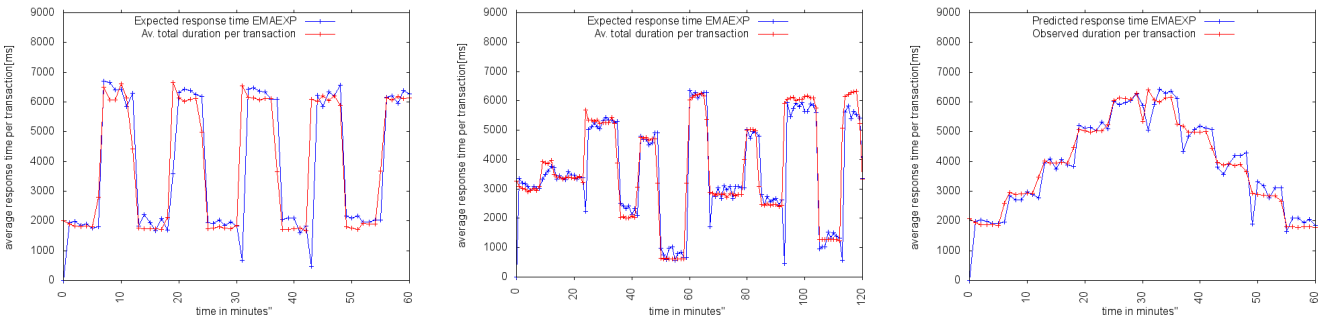
All system metadata need to be collected at runtime. It is used by each replica for optimizing transaction execution (e.g., load balancing) in an *autonomous* and *dynamic* manner. SO-1SR manages metadata separately from the user data by using a clustered MetadataManager based on Apache ZooKeeper[2]. The following design decisions leads to a very scalable metadata management. *Publish only when necessary:* Each replica collects data locally and publishes it periodically to the MetdataManager whenever the difference from the previously published metadata is significant enough. *Publish/Subscribe:* Each replica registers its interest for metadata at the MetadataManager which, in turn, immediately distributes it whenever it is updated. *Caching:* Each replica caches metadata (its own the medata from other replicas). Replicas are responsible for updating their cache once new metadata has been published.

4. SO-1SR FOR OPTIMIZING TRANSACTION EXECUTION IN THE CLOUD

Cloud environments provide virtually infinite capacity. In order to exploit this capacity, it is crucial to provide infrastructures that are able to scale at each layer of the entire stack – in particular, this should also include the data management layer. However, even though scalable data management is essential, most existing approaches only address load balancing at the application server layer. In addition, current load balancing approaches mostly require a dedicated component, which is in sharp contrast to the Cloud paradigm. Moreover, Cloud environments should support the specification and/or dynamic collection of the parameters introduced in Section 3 to optimize transaction execution by load balancing.

In this Section, we describe in detail the SO-1SR load balancing approach. The SO-1SR system model requires the SO-1SR middleware to be present at each replica. The

[1]http://aws.amazon.com/de/ec2/instance-types/
[2]http://zookeeper.apache.org/



(a) Alternate fix workload type. (b) Alternate random workload type. (c) Staircase workload type.

Figure 3: Response time prediction for the different workload types

SO-1SR execution engine is based on a queue model with a certain thread pool size. Each request (transaction, commit, etc.) is put into the queue and once a thread is available, it will dequeue a request and execute it. The response time of a request is influenced by the load of the replica, which is the sum of requests waiting in the queue and being processed. In what follows we will describe the details of how SO-1SR optimizes response time of transaction execution by effectively balancing the load between available replicas without the need of a dedicated load balancer and by taking transaction priorities and overhead into account.

In SO-1SR, the time is divided in periods of length p_{length}. Each replica samples its load with a sampling rate of s_f. At the end of the current period (p_i), the replicas will predict their load for the next period (p_{i+1}) by using the Exponential Moving Average (EMA) [18] and publish it to the MetadataManager. EMA is the weighted mean of n measures, where the weights decrease exponentially. Other prediction models, which are able to predict workload bursts as described in [21] can be implemented as modules and integrated into our SO-1SR framework by providing a simple API $getExpectedResponseTime(T_{type})$. Let S_t be the value of EMA at any time period t and s_t be the resource measure sampled at time period t. Further, let W be the sample period. Then, EMA can be calculated as follows:

$$\begin{aligned} EMA(S_t) &= \alpha(t_n - t_{n-1}) \cdot s_t \\ &\quad + (1 - \alpha(t_n - t_{n-1})) \cdot EMA(S_{t-1}) \\ \alpha(t_n - t_{n-1}) &= 1 - \exp(\frac{(t_n - t_{n-1})}{W}) \end{aligned} \quad (1)$$

While EMA allows to predict the load, SO-1SR is actually more interested in reducing the response time of transactions. In order to predict the response time, each SO-1SR replica stores for each executed transaction the current load and response time. By using a linear regression [13], a relationship between load and response time is established. With the linear regression, each replica is able to predict its expected response time for the next period, which is then published to the MetadataManager. As it can be seen in Figure 3, the combination of EMA for load prediction with the linear regression worked quite well for the workload types we have evaluated as part of this work: *alternatefix*, *alternate random* and *staircase*. Similar results were reported in [3]. In case of the alternatefix workload type (Figure 3a), the workload will alternate between a lower and an upper range based on a specific period duration. The alternate random workload (Figure 3b) type will also alternate the workload, however by choosing the lower range and upper range randomly. The period duration is also chosen randomly based on a specified range. The staircase workload type (Figure 3c) generates a step-wise symmetric increasing and decreasing workload.

All replicas are notified by the MetadataManager once new metadata is published. Based on the metadata each replica receives, a decision has to be taken for the best load balancing strategy with the goal of reducing the overall response time of transactions, min $Resp(T)$, with $Resp(T)$ being the average response time of transactions (T). There are different possible strategies to load balancing. In the simplest approach, each replica would have only a local view on the strategy and would simply decide on per transaction basis whether to execute it locally or forward it to the least loaded replica. Although this would reduce response time of transactions being forwarded, it would not minimize the overall average response time. Even worse, it may lead to an overall increase of response time and thus a penalty, since the approach does not take into account the overhead generated at the remote replica. Other possible strategies are to randomly forward transactions to any replica, or based on a Round Robin strategy. Even the strategy based on the least loaded replica is not well suited since it does not take the replica capacity into account.

SO-1SR follows a different approach which is depicted in Figure 4. The main idea is that each replica which suffers from a high load (in Figure 4, these are the replicas R_4 and R_1) needs to calculate the *optimal* rate at which it should forward transactions to the least loaded replicas (replicas R_3 and R_2 in Figure 4). The optimal rate leads to a reduction of the response time at the heavy loaded replicas by still avoiding the overloading of the least loaded replicas and minimizing the network overhead. It is important to mention that each replica will decide locally on the best load balancing strategy. However, since each replica runs the same algorithm and has a consistent view on the metadata (e.g., load, expected response time, etc.), the decision will be globally consistent. In the first step, the overloaded and underloaded replicas are determined. Since all replicas have a consistent view on the load of all other replicas, the decision will be globally consistent. In the second step, based on the load balancing approach described above, each replica will calculate the forward rate of transactions from the overloaded to the underloaded replicas. Again, since all replicas run the same algorithm based on the same metadata, all replicas will get consistent results for the forward rate from the overloaded to underloaded replicas. This means that no additional synchronization is necessary between the replicas

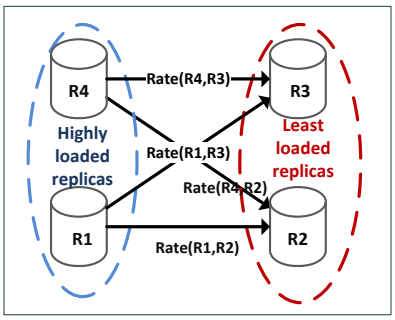


Figure 4: Optimal load balancing with SO-1SR

apart from the sharing of metadata. In Figure 4, both replicas R_1 and R_4 will in the first step calculate how they will distribute the load (independently of each other) and then in the second step calculate the best possible forward rate to each of the least loaded replicas (R_2 and R_3). The entire load balancing process to minimize the overall response time is only executed if there is a change in the load of any replica in the system. In that case all replicas are notified by the MetadataManager to initiate the process. As already mentioned, SO-1SR does not predict the load on the basis of single measures, but by using averaging methods (EMA). Hence, in a stable system, there is only a marginal overhead

In what follows we will explain the different steps of the load balancing process on the basis of a sample system consisting of two replicas, R_1 and R_2, both having the same processing capabilities. Let $E[Load(R_1)] > E[Load(R_2)]$ with $E[Load(R_1)]$ be the expected load of R_1 and $E[Load(R_2)]$ that of R_2. The expected overall load at any point in time of the SO-1SR system is defined as the sum of $E[Load(R_1)]$ and $E[Load(R_2)]$: $E[TotalLoad] = E[Load(R_1)] + E[Load(R_2)]$. A perfect balancer would divide the load so that each replica would get $0.5 \cdot E[TotalLoad]$. However, balancing the load means that transactions have to be forwarded from one replica to the other, in our case from R_1 to R_2. This generates additional overhead (e.g., network overhead) which has also to be taken into account. Additionally, the more transactions are forwarded from R_1 to R_2 the higher the impact on the response time of existing transactions at R_2. The send rate for achieving a certain load can be calculated using Little's law [16]: $Rate(X,Y) = L(Y) \cdot (ResidenceTime(Y))^{-1}$, with $L(Y)$ being the load of replica Y ($\sum_{r\ inR} L(r) = 1$) and $ResidenceTime(Y)$ the average time a request spends at replica Y. In our case, since transactions need to be forwarded from R_1 to R_2, $L(R_2)$ is the load we need to achieve at R_2 and $ResidenceTime(R_2)$ is the average time a requests spends at R_2. Now the goal is to find the value of $Rate(R_1, R_2)$ so that min $Resp(T)$ can be achieved. In our example, the starting point for finding the optimal rate at which transactions should be forwarded from R_1 to R_2 ($Rate(R_1, R_2)$) is the evenly distributed load.

$$E[NewLoad(R_1)] = E[NewLoad(R_2)] = 0.5 \cdot E[TotalLoad]$$

To characterize the new load at replica R_2, the transactions forwarded from R_1 to R_2 at rate $Rate(R_1, R_2)$ have to be considered. If $Rate(R_1, R_2)$ is known, we can then calculate the overall response time by also taking the network overhead into account. The rate defines actually the number of transactions to be forwarded per time unit. Now, starting with the initial $Rate(R_1, R_2)$ (evenly distributed load), SO-1SR will iterate through different possible load balancing values to find the optimal $Rate(R_1, R_2)$ by decrementing $E[NewLoad(R_2)]$ in each iteration step. At each iteration, the new $Rate(R_1, R_2)$ and new overall response time is calculated until the new overall response time is greater than the one calculated from the previous iteration. The value of the previous iteration is the optimal forward rate. Otherwise, the procedure is repeated until the point is reached in which no transactions are forwarded at all ($E[NewLoad(R_2)] = E[Load(R_2)]$).

From an application point of view, different transactions generate different profit and thus have different priorities. In the context of a TPC-C application, the transactions for ordering an item and executing a payment are more important than transactions for checking the stock level. In its current version, SO-1SR supports only binary priorities. Additional transaction semantics can be introduced in form of an upper bound of expected response time ($Resp_U(T_{P1})$) to be guaranteed up to a certain system load. If specified, the upper bound of the response time must be guaranteed – otherwise, the DBMS provider (Cloud Provider) is penalized according to a specific penalty model (part of an SLA agreement). Transactions of priority 0 do not generate any profit. The only requirement with regard to priority 0 transactions is to avoid starvation. Thus, SO-1SR focuses on the optimization of transactions with priority 1 and the avoidance of starvation for transactions with priority 0. This means that the goal is to minimize the overall average response time of transactions with priority 1, i.e., min $Resp(T_{P1})$ with T_{P1} being all priority 1 transactions.

With the introduction of an upper bound and a penalty for the response time of transactions with priority 1, the optimization problem can be re-formulated as:

$$\min\ u(Resp(T_{P1}) - Resp_U(T_{P1})) \cdot Pen(T_{P1}) \qquad (2)$$

The goal of Equation (2) is to keep the average response time of transactions below the agreed upper bound $Resp_U(T_{P1})$. $Pen(T_{P1})$ defines the penalty that applies per time unit if the average response time is above the specified upper bound. In Equation (2), $u(t)$ is a unit step function with the following property: if $t \leq 0$ then $u(t) = 0$ else $u(t) = t$.

Minimizing the overall penalty of transactions with priority 1 is however not sufficient for the following reason. Let us assume that the application provider has defined a high value for $Resp_U(T_{Prio1})$, i.e., it is quite tolerant to delays. This case might even result in a situation where the load is not balanced at all, since executing each transaction at the local replica may still satisfy Equation (2). However, that does not maximize profit, since the throughput has also to be taken into account. The more transactions of priority 1 are executed, the higher the profit.

SO-1SR is able to avoid starvation by reserving a certain time period for the execution of transactions of priority 0 waiting for execution in the execution queue. Based on the the expected response time and the number of transactions waiting in the queue, SO-1SR is able to accurately predict the time it needs to execute all transactions with priority 0. If the execution is faster than the reserved time slot, than SO-1SR will immediately start executing transactions with priority 1 to avoid that resources are wasted. The reserved time defines an upper bound and it is not expanded if the number of transactions with priority 0 waiting for execution is too high. However, other models such as like aging are also possible.

Server	Environment	Description
Server Machines	CPU	5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each)
	Memory	1.7 GiB of memory
	DBMS	Derby 10.9.1.0
	WebService Container	Apache Tomcat 7.0.32
Client Machines	CPU	2 EC2 Compute Unit (1 virtual core with 2 EC2 Compute Unit)
All	OS	Linux Ubuntu 10.4
	Java	Sun Java 1.6.0_35-b10

Table 1: Setup of experimental system

Finally, a load balancing scheme has also to be able to timely react to load bursts. There are prediction models such as [21] that are able to predict burst by incorporating application knowledge into the prediction model. It is well known that averages and linear models (linear regression) adapt more slowly to sudden load changes. For applications that are subject to such burst load situations, other prediction can be used. However, the algorithm for distributing the load among replicas will remain the same. It is important to note that SO-1SR follows a modular approach. It is even possible to combine different modules for load prediction with other modules for response time prediction based on the load.

5. EVALUATION

SO-1SR has been evaluated based on a transactional online store as defined by the TPC-C benchmark in an AWS EC2 Cloud environment. As baseline, we have used a traditional 1SR (T-1SR) implementation as described in [4]. Hence, the difference between our SO-1SR implementation and T-1SR lies in the load balancing strategy. SO-1SR uses the load balancing approach described in Section 4, whereas T-1SR is uses a random balancing strategy and a strategy based on the least loaded replica and does not handle transaction priorities. It means, T-1SR treats all transactions as having the same priority. In the case of the random load balancer (T-1SR-R), each replica, when receiving a transaction for execution, will assign the transactions to a replica by choosing randomly from the available replicas (including itself). Only the initial replica can forward a transaction. The replica receiving a forwarded transaction must execute it (the initial replica forwards transaction T by calling the $forceExecute(T)$ operation on the second replica). The approach based on the least load (T-1SR-LL), will assign a transaction to the least loaded replica based on the load published at the MetadataManager. In order to avoid the overload of the least loaded replicas, T-1SR-LL uses a hysteresis-based approach, which may publish the new expected load out of schedule, if there is a deviation by a certain factor from the last published load.

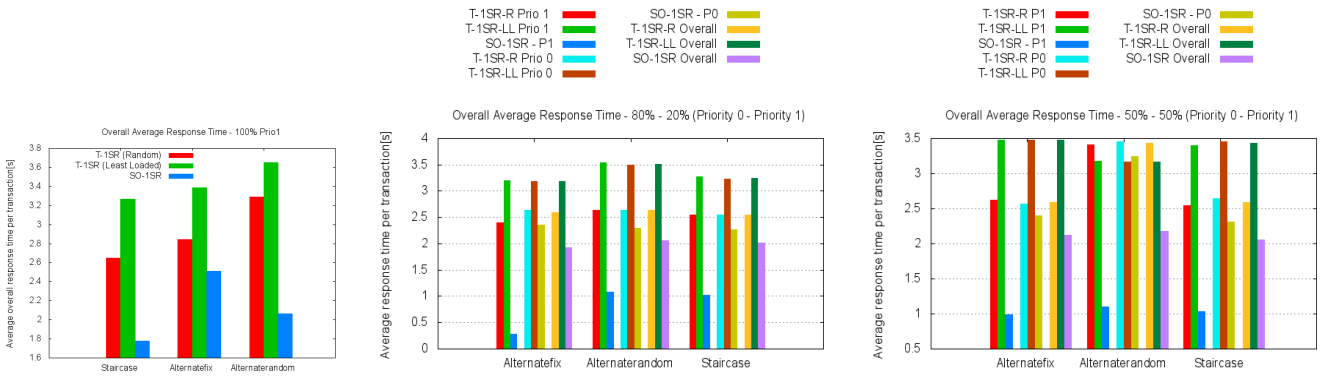
The transaction managers for both SO-1SR and T-1SR have been tested using the different workload types as described in Section 4. The goal of the experiments is to show that the SO-1SR load balancer is able to execute transactions so that the overall response time is minimized and that SO-1SR is able to adapt to changing system conditions. The basic evaluation setup is as follows: Each experiment is run based on the specified parameters listed in Table 2 using the T-1SR TransactionManager. Then the same experiment is repeated using the SO-1SR TransactionManager. A number of replicas and clients is started as defined by the corresponding experiment. Each client is attached to one of the replicas and will generate a number of Emulated Browsers (EBs). The number of EBs is determined on the basis of the specified $EBRange$ and the specified workload type. If for example, the $EBRange = 10 \dots 50$, then in the case of the alternate fix workload type, the workload will alternate between 10 and 50 EBs based on the $PeriodDuration$. In case of the alternate random workload type, the number of EBs will be chosen randomly from the alternating ranges $[10 \dots 20]$ and $[21 \dots 50]$ (more general: $[Min \dots \frac{1}{2}(Max - Min)]$ and $[\frac{1}{2}(Max - Min) + 1 \dots Max]$). The number of EBs for the staircase workload type will be increased stepwise based on the $PeriodDuration$ starting with 10 (Min) EBs until the maximal load of 50 (Max) EBs is reached. Then it will start decreasing until the number of 10 (Min) EBs is reached. Each EB will generate a transaction and once it gets a response will start the next one. Each experiment is repeated ten times. The depicted results present the averaged results over all runs.

Table 1 lists the characteristics of the different components used for the evaluations. The different components of the system are implemented as Web Services, which allows different deployment strategies. Apache Tomcat is used as a Web Service container. Each request (transaction, commit, etc.) is forwarded to the Apache Tomcat container running the TransactionManager. The container will then forward the request to the internal queue of the TransactionManager. In the evaluation results we neglect the time it takes to forward a request from the container to the TransactionManager, as it is implementation specific and thus difficult to model. The reported results do however take the network overhead into account.

Different priority ratios. The goal of this experiment is to compare the response time of transactions when executed with different load balancing strategies T-1SR-R, T-1SR-LL and SO-1SR for different transaction priority ratios: 100% priority 1, 80% priority 0 and 20% priority 1, 50% priority

Parameter	Description
Workload Type	staircase, alternate fix or alternate random.
Number of Replicas	Number of replicas involved in the experiment.
Number of Clients	Number of clients involved in the experiment.
EB Range	$Min \dots Max$. Defines the range of EBs based on the specified workload type.
Period Duration	Defines the adaption steps for a workload type.
Total Duration	Total duration of an experiment.

Table 2: Evaluation parameters



(a) 100% Priority 1 (b) 80% Priority 0 - 20% Priority 1 (c) 50% Priority 0 - 50% Priority 1

Figure 5: SO-1SR vs T-1SR-R and T-1SR-LL for different priority ratios

0 and 50% priority 1. Priority 1 transactions correspond to the *NewOrder* transaction type as defined by TPC-C. All other TPC-C transactions are of priority 0. In this experiment, we have used two replicas and two clients to mimic a Cloud environment with two geographically distributed data centers. The clients will generate transactions based on the specified workload type. The first client will work with an EB range of [20 . . . 100], whereas the second one with a range of [5 . . . 20], i.e., the first replica will have a higher load than the second one. The clients are attached to one of the replicas and will always send their transactions to the same replica. Load balancing is executed inside the replicated system, i.e., each replica will decide on the best replica for execution based on its load balancing strategy. The evaluation results for the different priority ratios are depicted in Figure 5. As it can be seen, the SO-1SR load balancing always outperforms T-1SR-R and T-1SR-LL. The results for priority 1 ratio of 100% (Figure 5a) show that SO-1SR is able to balance the load so that the overall response time is minimized. The lower response times of SO-1SR compared to T-1SR-R and T-1SR-LL are based on the fact that SO-1SR balances the load by taking also the forward overhead (network) and replica capacities (based on their response times) into account. The experiments with the 80%-20% and 50%-50% priority ratios show that SO-1SR is able to considerably reduce the response time of priority 1 transactions by giving them precedence over transactions with priority 0 (see Figure 5b and 5c).

Mixed workload types. The goal of this experiment is to show that SO-1SR is able to optimally balance the load and outperform other load balancing strategies even in an environment with mixed workload types. For this experiment, we have used four replicas and four clients (i.e., a Cloud environment that guarantees an even higher degree of availability compared to the previous setup). Each client is attached to one of the replicas and will send transactions to that replica. The workload is generated according to the following specification. The first client generates an alternate fix workload with EB range [50 . . . 100]. The second and the third client generate also alternate fix workload, but with EB range of [20 . . . 50] and 5 . . . 10], whereas the fourth one generates transactions according to the staircase workload with EB range [5 . . . 10]. Like in the previous experiment, this evaluation has also been conducted for different priority ratios.

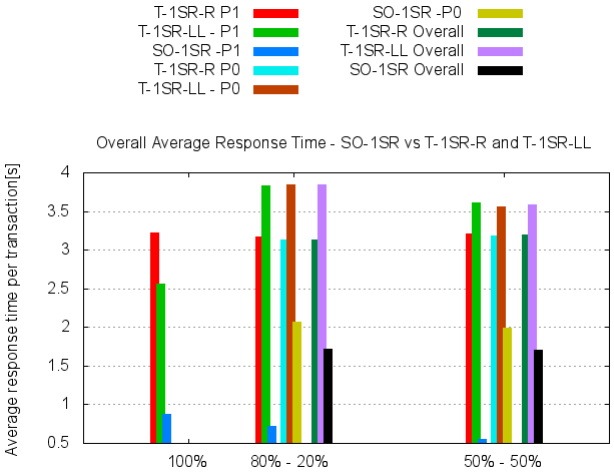


Figure 6: Mixed Workload: SO-1SR vs. T-1SR-R and T-1SR-LL for different priority ratios (100% Prio1, 80% Prio 0 - 20% Prio1, 50% Prio0 - 50% Prio 1)

The results are depicted in Figure 6. The results show that in case of all transactions having priority 1, the SO-1SR TransactionManager is able to considerably reduce the response time by effectively balancing the load between the available replicas. In the case of mixed transaction priorities, SO-1SR will reduce the response time by balancing the load by giving precedence to the transactions with priority 1. In the case of 100% priority 1 ratio, SO-1SR reduces the average response by 60% to 70% compared to the T-1SR load balancing approaches. In case of 80%-20% ratio, the average response time of priority 1 transactions is reduced by 77%-80%, whereas for the 50%-50% ratio we have achieved a reduction of 80%-84%. The overall reduction by also taking priority 0 transactions into account is at about 50%. Executing this same scenario with a higher number of clients (e.g, 8 or 16) would lead to an increased load at replicas. Even in that case, SO-1SR would be able to balance the load and decrease response time of transactions, even though these numbers are beyond the number of data centers of current Cloud providers.

6. CONCLUSION AND OUTLOOK

Users deploy their applications in the Cloud in order to take advantage of its highly available and scalable infras-

tructure. Additionally, the Cloud provides a very attractive pay-as-you-go cost model. Databases are an important part of the applications deployed in the Cloud and significantly impact the overall system scalability in the presence of replication (which, in turn, is necessary for reasons of availability). Current DBMSs provided by Cloud providers (e.g., simpleDB, S3, etc.) provide only relaxed consistency guarantees and thus increase the complexity of the application design. Other existing solutions providing strong consistency guarantees use strategies that work well for one application type, but fail to satisfy others. With our SO-1SR approach, we follow the goal towards incorporating different user-defined and system parameter into an optimization model and thus providing a highly scalable, dynamic and adaptive framework that is able to guarantee 1SR. In this paper, we have identified important parameters which influence the transaction execution and we have presented a dynamic and adaptive load balancing schema.

In our future work, we will address the optimization of the other phases of transaction execution in a replicated Cloud environment and the interdependencies between these phases. Our goal is to identify the parameters which influence the (eager) transaction commit (e.g., data popularity, replica popularity) and the refresh / propagation of changes for lazily updated replicas. Again, the idea is not to assign a fixed strategy but to decide dynamically, based on the extended optimization model. First ideas go into the direction of building consistency views. Inside the same consistency view, all replicas are updated in synchronous manner, whereas the update between the views is done asynchronously. This leads to the situation in which replicas have different consistency states. However, since our SO-1SR protocol should guarantee 1SR, each transaction must be executed on an update replica. Thus, the load balancer may be restricted to balance the load inside a consistency view, which may not be optimal, or use any replica for balancing the load. In the second case, an outdated replica must actively refresh its data (additional overhead). The main challenge is to define a model which finds the optimal size and optimal members (replicas) of the consistency views so that, at the same time, the commit costs and load balancing are optimized.

Acknowledgement

This work has been supported by the Swiss National Science Foundation, project GriDMan (contract no. 200021_132201).

7. REFERENCES

[1] Transactions across datacenters (and other weekend projects). `http://www.google.com/intl/de/events/io/2009/sessions/TransactionsAcrossDatacenters.html`, 2009. [Online; accessed 20-June-2013].

[2] D. Agrawal et al. Managing geo-replicated data in multi-datacenters. In *Databases in Networked Information Systems*, volume 7813 of *Lecture Notes in Computer Science*, pages 23–43. Springer Berlin Heidelberg, 2013.

[3] M. Andreolini and S. Casolari. Load Prediction Models in Web-based Systems. In Proc. VALUETOOLS, 2006.

[4] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst.*, 9(4):596–615, Dec. 1984.

[5] P. A. Bernstein and N. Goodman. Serializability Theory for Replicated Databases. *Journal of Computer and System Sciences*, 1985.

[6] E. A. Brewer. Towards Robust Distributed Systems. In Proc. PODC, Portland, Oregon, USA, 2000.

[7] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable Isolation for Snapshot Databases. In Proc. SIGMOD, Vancouver, Canada, 2008.

[8] F. Chang et al. Bigtable: a distributed storage system for structured data. In *Proc. OSDI*, 2006.

[9] B. F. Cooper et al. Pnuts: Yahoo!'s hosted data serving platform. *In Proc. VLDB*, 2008.

[10] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *Proc. OSDI*, Hollywood, CA, USA, 2012.

[11] I. Fetai and H. Schuldt. Cost-based data consistency in a data-as-a-service cloud environment. In Proc. CLOUD, Honolulu, HI, USA, 2012.

[12] J. Gray et al. The dangers of replication and a solution. In Proc. SIGMOD, 1996.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning; Data Mining, Inference, and Prediction*. Springer, 2013.

[14] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In Proc. VLDB, 2000.

[15] T. Kraska et al. Consistency rationing in the cloud: pay only when it matters. In Proc. VLDB, 2009.

[16] J. D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):pp. 383–387, 1961.

[17] F. D. Muñoz Escoí et al. Revising 1-copy equivalence in replicated databases with snapshot isolation. In Proc. CoopIS, DOA, IS, and ODBASE: Part I, 2009.

[18] B. G. Robert. *Smoothing, Forecasting and Prediction of Discrete Time Serie*. Prentice-Hall, 1963.

[19] U. Röhm et al. FAS—a Freshness-Sensitive Coordination Middleware. In Proc. VLDB, 2002.

[20] M. Ruiz-Fuertes et al. 1-copy equivalence: Atomic commit versus atomic broadcast in termination protocols. 2010.

[21] M. Sladescu et al. Event aware workload prediction: A study using auction events. In Proc. WISE, 2012.

[22] TPC-C Benchmark. `http://www.tpc.org/tpcc/`. Accessed: 04/2013.

[23] L. C. Voicu et al. Re: GRIDiT - Coordinating distributed update transactions on replicated data in the Grid. In Proc. GRID, 2009.

[24] L. C. Voicu et al. Flexible data access in a cloud based on freshness requirements. In Proc. CLOUD, 2010.