Cost-Based Data Consistency in a Data-as-a-Service Cloud Environment

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

***Abstract*—Clouds are becoming the preferred platforms for large-scale applications. Currently, Cloud environments focus on high scalability and availability by relaxing consistency. Weak consistency is considered to be sufficient for most of the currently deployed applications in the Cloud. However, the Cloud is increasingly being promoted as environment for running a wide range of different types of applications on top of replicated data — of which not all will be satisfied with weak consistency. Strong consistency, even though demanded by applications, decreases availability and is costly to enforce from both a performance and monetary point of view. On the other hand, weak consistency may generate high costs due to the access to inconsistent data. In this paper, we present a novel approach, called *cost-based concurrency control (C3)*, that allows to dynamically and adaptively switch at runtime between different consistency levels of transactions. C3 has been implemented in a Data-as-a-Service Cloud environment and considers all costs that incur during execution. These costs are determined by infrastructure costs for running a transaction in a certain consistency level (called *consistency costs*) and, optionally, by additional application-specific costs for compensating the effects of accessing inconsistent data (called *inconsistency costs*). C3 considers transaction mixes running different consistency levels at the same time while enforcing the inherent consistency guarantees of each of these protocols. The main contribution of this paper is threefold. First, it thoroughly analyzes the consistency costs of the most common concurrency control protocols; second, it specifies a set of rules that allow to dynamically select the most appropriate consistency level with the goal of minimizing the overall costs (consistency and inconsistency costs); third, it provides a protocol that guarantees that anomalies in the transaction mixes supported by C3 are avoided and that enforces the correct execution of all transaction in a transaction mix. We have evaluated C3 on the basis of real infrastructure costs, derived from Amazon's EC2. The results demonstrate the feasibility of the cost model and show that C3 leads to a reduction of the overall costs of transactions compared to a fixed consistency level.**

***Keywords*-Cloud Computing, Data-as-a-Service, Transaction Management, Concurrency Control, Data Consistency**

I. Introduction

In general, the goal of Cloud computing is to provide different types of services (IaaS, PaaS and SaaS) at low cost. It promises infinite scalability and high availability [1]. It is the responsibility of the Cloud provider to guarantee that data is highly available and that the infrastructure will elastically scale in order to handle heavy loads. This frees clients from the burden of managing their own infrastructure, so that they can concentrate on their core business. The business model in the Cloud is *pay-per-use*. That means, users can avoid investments and pay only what they consume. Data management is a central aspect of applications deployed in the Cloud [2], [3]. Usually, application servers and web servers can be easily scaled out by adding new server instances. However, replicating only the servers does not guarantee high availability of the data. Moreover, from a performance point of view, the database usually becomes the bottleneck [4], [5]. Data replication is a mechanism used to increase availability and scalability for read-only transactions. But it also increases the complexity when it comes to data consistency in the presence of update transactions. As a consequence of the CAP conjecture [6], most applications in the Cloud run with relaxed consistency. While this is sufficient for many of the current Cloud applications, strong consistency is crucial for all types of applications which require access to consistent data. The main goal of the applications deployed in the Cloud is to generate high profit at low operational costs. Different consistency levels generate different operational costs: the stronger the consistency level, the higher the costs for enforcing it, and the lower the degree of scalability. Weak consistency is cheaper but may result in inconsistency costs [7]. The operational costs are generated by the Cloud resources which need to be used for achieving a certain consistency level. The inconsistency costs are application-specific costs and reflect the additional work which needs to be done in order to compensate the effects of inconsistent data (e.g., compensating oversold books or tickets).

There is a range of concurrency control protocols (CCPs) leading to different consistency guarantees: from One-Copy Serializability (1SR), providing the strongest consistency guarantees, Snapshot Isolation (SI), providing weaker consistency guarantees than 1SR, to Session Consistency (SC) which just provides read-your-writes (RYW) inside a session. In the ideal case, a system should run transactions with weak consistency as long as possible in order to save operational costs and automatically switch to a stronger consistency level as soon as the expected inconsistency costs are too high. Such desired behavior, which is not supported yet by any existing system or Cloud provider, will be provided by our *cost-based concurrency control* (C^3), the novel approach to concurrency control in Data-as-a-Service (DaaS) Cloud environments we propose in this paper. With C^3, the overall cost of the application will be minimized.

The necessity for adaptive consistency control was already identified in [8], [9] where the authors define different application scenarios ranging from simple online bookstores to complex collaborative systems. However, both approaches do not treat the problem of costs (balance of consistency and inconsistency costs) related to data consistency in the Cloud. Our C^3 approach is based on a generic cost model for data consistency and an intuitive yet powerful programming API which allows users to specify cost parameters at transaction level. Based on these parameters C^3 will adjust the consistency level of the transaction so that the overall costs are minimized.

The contributions of this paper are as follows: **(1)** We introduce C^3 which is based on a generic cost model for adaptive concurrency control as described in [10]. **(2)** We analyze the possible anomalies in case of consistency mixes and provide a mechanism for avoiding these anomalies. Moreover, we have implemented and evaluated C^3 on top of a range of CCPs (1SR, Strong Snapshot Isolation – SSI, and SC) using resources of a concrete Cloud provider and in the context of the transactional web e-Commerce benchmark TPC-W.

This paper is organized as follows: Section II presents related work. In Section III we provide an overview of replication and concurrency control in replicated systems. In Section IV we introduce C^3 and present its architecture. The core of this Section however is the analysis of the anomalies in case of transaction mixes and the mechanism for avoiding these anomalies. Section V presents the details of the C^3 protocol. The evaluation of C^3 on top of Amazon EC2 is provided in Section VI. Section VII concludes.

II. Related Work

Data replication has attracted many researchers from the database and distributed systems community. The main challenge is to provide replication solutions, which are scalable and at the same time provide strong consistency guarantees. Usual solutions in this trade-off between performance and consistency would either relax consistency guarantees in order to increase performance or vice-versa. The traditional CCPs providing strong consistency guarantees, like 1SR and SSI, are not applicable for large-scale replicated data environments like the Cloud. Currently, Cloud data environments are oriented towards scalability by providing replication solutions with weak consistency guarantees. However, the Cloud is more and more becoming a platform for deploying business applications, most of which require strong transactional guarantees.

Kemme and Alonso [11], [12] present one of the first approaches to replication management which provides strong consistency guarantees while being scalable. They avoid costly protocols, like 2PC, by exploiting efficient group communication.

In addition to performance, the costs are also an important factor in the Cloud. In general, strong consistency generates high costs for enforcing it. Weak consistency on the other hand, can lead to high operative losses due to, for example, oversells [7]. The cost parameter increases the complexity of designing efficient CCPs. In contrast to the approach in [11], our work does not focus on the design of new efficient CCPs, but is rather based on the idea of adaptive cost-based concurrency control: use existing CCPs and adjust the consistency level at runtime according to cost constraints, i.e., find the right balance between consistency and inconsistency costs. In [7], different types of applications are described for which the consistency level might have to be changed at runtime. The basic assumption is that not all data need the same consistency guarantees. The approach introduces a consistency rationing model, which categorizes data in three different categories. Adaptive consistency means that the consistency level changes between 1SR and Session Consistency at runtime depending on the specific policy defined at data level (annotation of the schema). However, in our opinion the approach presented in [7] is limited to simple cases in which data can be easily categorized. If we take more complex data structures or heterogeneous data sources, like stream data, XML, RDF, then the categorization is not that simple anymore if at all possible. Another drawback of the approach presented in [7] is that it is not possible to have different applications work on shared data and still satisfy their possibly diverging consistency requirements. If the consistency level is specified per data object and if it turns out that this consistency level is inappropriate for another application/scenario, then the consistency specification at data level has to be changed. This may lead to unexpected behavior of existing applications and makes application behavior data-dependent.
A first approach to cost-based concurrency control that considers the cost of single operations (and their undo operations for recovery purposes) in order to select and influence CCPs can be found in transactional processes [13]. However, this does not consider any infrastructure-related costs for enforcing a selected CCP.

III. Concurrency Control in Replicated Systems

A replicated system is a distributed system in which multiple copies of the same data are stored at multiple sites. Replication increases availability and scalability. However, it also increases the complexity of data consistency as correct concurrent access requires *serializable histories*. A history is serial if for any pair of transactions T_i and T_j, either T_i is executed before T_j, or vice versa, i.e. the transactions are executed serially, one after the other. A serializable history is a history in which transactions are allowed to be executed in parallel, yet with the result being the same as in a serial history. In what follows, we briefly introduce the most prominent concurrency control protocols, namely One-Copy Serializability, Snapshot Isolation and Session Consistency.

One-Copy Serializability (1SR): In a system providing 1SR guarantees, the user is basically unaware that the data is replicated as all the replicas are always consistent. The implication of 1SR is that each write transaction (i.e. a transaction which writes a data object) must update all copies. Depending on the number of replicas, this may considerably increase the response time of update transactions.

Snapshot Isolation (SI): The idea of SI is to increase concurrency compared to 1SR. Its advantage arises from the fact that reads are never blocked. SI avoids many of the possible inconsistencies. However the *write-skew* anomaly is possible [14]. Different variants of SI exist for replicated systems which lead to different consistency guarantees. As part of this work we have considered *Strong SI (SSI)*. The SSI protocol requires that a transaction T_k, which starts after a committed transaction T_i must see a database state including the effects of T_i [15]. In centralized systems it is easy to provide the latest snapshot, whereas in distributed systems, this leads to delays of transaction starts. 2PC is usually used for updating all replicas in the context of the transaction. Other possible SI variants, which relax the consistency requirements for replicated systems are described in [14], [15].

Session Consistency (SC): SC is a variation of the eventual consistency model[1]. In this model, data is accessed in the context of a session. Inside the session, the system guarantees read-your-writes consistency. These guarantees do not span different sessions. Transactions are guaranteed not to see values older than their writes. However, they may see newer values which have been written by concurrent transactions. Regarding the conflict resolution between transactions, for non-commutative updates (e.g., value overrides) the last commit wins, for the commutative ones (e.g., incrementing or decrementing a numerical value) the updates are executed one after the other. From an isolation point of view, different anomalies between SC transactions are possible. As data is usually propagated in a lazy way in SC, inconsistencies may also occur due to the access to stale data.

IV. C^3 Architecture

The ideal CCP provides strong guarantees and is cheap to enforce. So, one of the possible approaches could be to try and design such an ideal cost-effective and correct concurrency protocol. However, there is no "silver bullet" and each CCP finds its own balance between consistency and inconsistency costs. In contrast, our new C^3 approach is based on existing CCPs and introduces an adaptive layer, which is able to dynamically switch between the different protocols at runtime in order to save costs (consistency and inconsistency costs) and decrease response time of transactions while at the same time providing the best possible correctness guarantees.

A. System Model

Our system model considers full replication and an update-anywhere approach. 1SR and SSI transactions update all replicas inside the transaction context by using 2PC to ensure atomic replica commitment, whereas SC transactions commit only at the local replica. Replica reconciliation is done on a periodic basis. The system model does not constrain the replication strategy, i.e., the creation and destruction of replicas. However, we assume the existence of a replica catalog. This means that any replica in the system is aware of all other replicas. Care must be taken in case of dynamic replication, so that transaction commits are not interfered by the replica deployment process. Regarding the positioning in the Cloud, our solution does not provide simple datastore functionality (it is even independent of specific datastore), but a DaaS with C^3 semantics with the goal of minimizing application costs. The C^3 middleware consists of the following components. **TransactionManager:** Is responsible for managing all transactions and implements the C^3 protocol. **SiteManager:** Provides an abstract layer for managing local data access (insert, update, delete, read). **TimestampManager:** Manages timestamps of transactions and guarantees that transactions will get timestamps according to their arrival order. **LockManager:** Provides lock management functionality. **ReplicaManager:** Manages the available replicas. **FreshnessManager:** Is responsible for the management of freshness information (Section IV-D). All components are implemented as Web services and can be deployed in different possible configurations. Regarding the logical architecture, a replica site consists of a TransactionManager and SiteManager, i.e., when talking about replicas, both components are expected to be present at that site. Both components further consist of a middleware layer present at each replica and are equipped with a local datastore; the SiteManager uses the datastore for handling the "real" data, whereas the TransactionManager stores data related to its functionality. From the transaction perspective there are two types of replicas: *local* and *remote*. If a transaction T_i is assigned to replica R^j for execution, then R^j is called a *local replica* for T_i. All others are *remote replicas* for transaction T_i.

B. Transaction model

A transaction consists of a set of operations accessing objects, uniquely identified by an *objectId*, in read or write mode. A *read-only* transaction consists of read operations only, whereas *update* transactions contain at least one update operation. In our model transactions are assigned unique start and commit timestamps that reflect the start and the

[1]In a system providing weak consistency, it is not guaranteed that reads that follow writes will see the updated value. The period until all replicas are up-to-date is called the *inconsistency window*. Eventual consistency is a form of weak consistency. The system guarantees that if no new updates are made to the objects eventually all accesses will return the last updated value

commit order of transactions: the most recently started transaction gets the highest start timestamp and the most recent commit gets the highest commit timestamp. We further assume that the write- and read-sets of transactions are available. These are necessary for avoiding anomalies in case of consistency mixes.

C. Avoiding Anomalies of Consistency Mixes

In C^3, where a range of different CCPs is provided, it is possible that the same data object is accessed by different transactions with different consistency levels for the following reasons. First, the application developer has designed the application in such a way that the same data objects can be accessed by transactions with different consistency levels. Second, different applications work on the same data and may have different consistency requirements. Third, the different replicas may decide to execute adaptive transactions accessing the same data objects based on a cost model with different consistency levels depending on the locally collected statistical data. In more detail, the possible inconsistencies are as follows: **(1)** Inconsistencies due to the isolation level between transactions running the same CCP (e.g., write-skew between SSI transactions). **(2)** Inconsistencies due to isolation level between transaction running different CCPs (e.g., anomalies between 1SR–SSI, 1SR–SC and SSI–SC). **(3)** Inconsistencies due to data staleness. If a transaction works on old data, it may generate inconsistencies, e.g., decrement a value based on an old value. The mechanism we provide targets the inconsistencies **(2)**. Inconsistencies **(3)** are handled by the replica synchronization mechanism described in Section IV-E which guarantees transactions requiring strong consistency (1SR and SSI) to always work on the most recent data. The goal of our mechanism is not to avoid inconsistencies between transactions of the same consistency level. Cahill et al. [16] have already provided a mechanism which avoids inconsistencies between SI transactions, in order to make SI serializable. Another important fact is that our approach guarantees the chosen consistency level (e.g., 1SR or SSI) based on the latest globally commited data. If the data was corrupted, i.e., an inconsistency was introduced by transactions running a lower consistency level (e.g., SC), that inconsistency will not be corrected. The guarantee our mechanism provides is that for example an 1SR transaction will work on the most recent data and will not be disturbed by transactions running a lower consistency level.

The detailed analysis of the possible inconsistencies in case of transaction mixes and the algorithms for avoiding these inconsistencies is provided in [10].

D. FreshnessManagement

The FreshnessManager provides freshness information for each of the data objects. This information is used mainly for the replica synchronization mechanism described in Section IV-E. We distinguish two types of freshness information: last committed timestamp for non-commutative updates (e.g., strings) and a range of timestamps for commutative updates (e.g., integers). The commit timestamp range is sorted in ascending order. For this purpose, the FreshnessManager provides an API allowing the replicas to add freshness information for both types of updates. One important aspect is that the access to the FreshnessManager must be synchronized. The reading of freshness information for a set of data objects should not be disturbed by concurrent writes.

E. Replica synchronization

In a system supporting lazy replication, it might happen that a transaction requiring strong consistency guarantees (1SR or SSI) is executed on a replica which contains stale data. In order to provide the desired guarantees, the local replica must first synchronize with the replicas containing the most recent data. The mechanism is based on the functionality provided by the FreshnessManager and additionally requires that for the commutative updates the SiteManagers are able to provide the update logs for specific objects and commit timestamps. The synchronization consists of the following steps, which are executed before the transaction is effectively started, but after the transactions has successfully passed the startup checks of the mechanism for avoiding anomalies of transaction mixes and has acquired the locks [10]. **(1)** Check if all data objects to be accessed by the transaction are up-to-date. The check is done by comparing the local commit timestamps with the commit timestamps at the FreshnessManager. In case of the commutative updates, it must be checked if there are gaps in the update range. **(2)** If the most recent commit timestamp is not satisfied or there are gaps for commutative updates, than the local replica synchronizes with the replicas which can provide the most recent data or the missing updates in case of commutative updates.

V. COST-BASED CONCURRENCY CONTROL

The goal of our C^3 approach is to dynamically adjust the consistency level of transactions at runtime according to specific constraints. In the Cloud the constraints are the costs: the stronger the consistency level the higher the costs. A weak consistency level does not automatically mean less costs, since the inconsistency costs have to be taken into account. The system adjusts at runtime the consistency level in order to minimize the total costs (the sum of consistency and inconsistency costs). Based on the cost model described in [10] we have defined a set of rules to achieve C^3, with the goal of providing the best possible consistency guarantees while minimizing application costs. The focus is set on minimizing the costs of the services delivered by the end-service providers. The end-service provider specifies the cost parameters for its (transactional) services. In doing so, it

Parameter	Definition
CB_{Trx}	Consistency (operational) budget for a transaction.
C_{Inc}	Inconsistency costs for a transaction.
$Coststrategy$	[Optimal | Minimal]. Default is Minimal. The semantics of this parameter is to use up the specified consistency budget CB_{Trx} (Optimal) even if that does not minimize the overall costs or minimize overall costs $E(C_{OverallConsL})$ (Minimal).
$E(C_{ConsCCP})$	Expected consistency (operational) costs of a specific CCP, where CCP=1SR|SSI|SC.
$E(C_{OverallCCP})$	Expected overall costs of a specific CCP.

Table I
C^3 PARAMETERS

effectively determines the consistency guarantees that are provided by the services. In order to show the operating principles of C^3, in what follows we provide a detailed example. Let us assume that the end-service provider has specified the consistency budget and inconsistency costs for a specific transaction. This case corresponds to the Rules 4 and 5 (Section V-A). The cost parameters are always specified per transaction execution, and should not be seen as a kind of budget for the entire system. The default behavior in this case is that the system will execute the transaction with the consistency level having the lowest expected overall costs. By doing this the system tries to minimize the costs. However, the end-service provider may have provided enough budget for running the transaction with 1SR and the transaction might be forced to use up the budget. In that case, the transaction would run with 1SR even if that does not minimize the costs. A thorough analysis of the cost model for C^3 is provided in our Technical Report [10]. Below we summarize the most important aspects of C^3.

A. C^3 Rules

Table I summarizes the parameters used in the C^3 rules which are specified at transaction level. At least one of the cost parameters, C_{Inc} or CB_{Trx}, must be specified. Moreover, it is possible to specify both parameters, the system will then decide how to behave depending on the specified values (Rules 4 and 5). Each of the rules is only applied if its preconditions are true. A '*' as a value means that the parameter can take any value ≥ 0 or even unspecified, '$\perp$' means unspecified, whereas 'Number' means any number ≥ 0.

Rule 1: Preconditions: Transaction has infinite consistency budget available ($CB_{Trx} = \infty$ & $C_{Inc} = *$).
The desired consistency level is enforced independently of the specified inconsistency costs. The important aspect of this rule is that it allows the system to be operated as a traditional database. If for example all transactions are forced (provided with infinite budget) to run with 1SR, then the system "switches off" the adaptive component and behaves as a traditional distributed database by enforcing 1SR. Consistency mixes are still possible, if for example some transactions are forced to run 1SR, whereas some others run SSI or SC.

Rule 2: Preconditions: The consistency budget is specified, whereas the inconsistency costs are left unspecified ($CB_{Trx} = Number$ & $C_{Inc} = \perp$).
The system will check if there is enough budget for the transaction to be executed with a strong consistency (1SR, SSI). If not, it will execute it with SC.

$$CL = \begin{cases} 1SR & \text{if } E(C_{Cons1SR}) \leq CB_{Trx} \\ SSI & \text{if } E(C_{ConsSSI}) \leq CB_{Trx} \\ SC & \text{if } E(C_{ConsSC}) \leq CB_{Trx} \\ Abort & \text{else} \end{cases}$$

Rule 3: Preconditions: The inconsistency costs are specified, whereas the consistency budget is set to 0 or left unspecified ($CB_{Trx} = \perp$ & $C_{Inc} = Number$).
The system will check to find the consistency level with the lowest expected overall (sum of consistency and inconsistency) costs, in order to minimize overall costs.

$$CL = \begin{cases} minimum(E(C_{Overall1SR}), E(C_{OverallSSI}), \\ E(C_{OverallSC})) \end{cases}$$

Rule 4: Preconditions: Both the consistency budget and inconsistency costs are specified. $Coststrategy = Minimal$.
Similarly to rule 3, also in this case the system tries to minimize the overall costs.

Rule 5: Preconditions: Similarly to rule 4, the consistency budget and inconsistency costs are specified. $Coststrategy = Optimal$.
The C^3 system will execute the transaction with the consistency level having the lowest expected cost or which does not exceed the provided consistency budget. According to this rule it might be that a transaction is executed with 1SR if the expected costs do not exceed the specified budget even if that does not minimize the overall costs. It is important to notice that the checks are executed hierarchically starting with 1SR down to SC and will stop as soon as one of the consistency levels satisfies the conditions.

$$CL = \begin{cases} 1SR & \text{if } isLowest(E(C_{Overall1SR})) \\ & \quad || \ (E(C_{Cons1SR}) \leq CB_{Trx}) \\ & \\ SSI & \text{if } isLowest(E(C_{OverallSSI})) \\ & \quad || \ (E(C_{ConsSSI}) \leq CB_{Trx}) \\ & \\ SC & else \end{cases}$$

The rules specified above are used by the C^3 to decide on the consistency level of a transaction before the transaction is effectively executed. In the current work, we have taken the most prominent CCPs. The architecture of C^3 is component-based, i.e. the CCPs are implemented as components consisting of the functionality and the cost model. This means that, the C^3 framework can be easily extended to take into account additional CCPs by just introducing the corresponding CCP components into the framework. Even different CCP implementations providing same consistency level (e.g., 1SR) can be plugged into the framework as dedicated components. Our C^3 provides the possibility to activate/deactivate specific components. If different CCP components providing the same consistency level are active then C^3 will iterate through the components and use the CCP component having the lowest costs for the decision making based on the C^3 rules. Let us assume that in our system there are two active CCP components providing 1SR: one based on 2PC for replica commitment and the second one based on group communication protocols. C^3 will calculate the expected costs for both CCP components based on their cost models and will take the one having the lowest cost as a basis for executing the C^3 rules. It means, the costs of a specific consistency level in the C^3 rules correspond to the CCP component of that consistency level having the lowest expected cost.

VI. Evaluation of C^3

C^3 adds additional complexity to the design of transactional systems. The additional parameters like consistency budget and inconsistency costs require careful analysis of the transaction design. Additional actors have to be involved in the design process in order to provide precise values for the additional parameters. The inconsistency cost is the critical parameter: if the value is too low, then too many transactions are executed with weak consistency, which may generate high penalty costs. On the other side, if the value is too high, it may lead to high operational costs, since many transactions are executed with strong consistency. An additional complexity comes from the fact that, for adaptive transactions, the system will decide on the CCP and by that on the consistency level, based on the cost threshold provided to it and collected statistical data at runtime. This increases the complexity to argue about correctness of transactions and debugging in case of errors.

In what follows, we will describe the experiments done on top of an AWS EC2 to evaluate C^3.

A. Setup

Application Scenario: For the evaluation, we have implemented an application scenario as specified in the TPC-W benchmark, which models an online bookstore. TPC-W emulates users that browse and order books from the online shop. It defines 14 different web interactions and three different interaction mixes. From the possible mixes, the Ordering Mix is the most update-intensive mix, which specifies that 10% of actions are book purchases. We have used the Ordering Mix for the evaluation of C^3.

Infrastructure & Deployment: In the evaluations of C^3 we have used the Apache Tomcat Webservice container for deploying and running the implemented Web services. We have used two different machine types. *EC2 client machines*, which host the clients, are equipped with 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit); *EC2 server machines*, which host the services and are equipped with 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each). Both machine types contain 1.7GB of RAM and use Ubuntu 10.94 32-bit as operating system.

The TransactionManager and SiteManager are deployed to different WebService containers on the same EC2 server machine, each having their own local datastore, whereas all the centralized components/services (see Section IV-A) are deployed to dedicated WebService containers and machines.

Costs: The runtime costs of a transaction consists of the transaction execution, data storage and retrieval, consistency (additional services) and eventually inconsistency or penalty costs. The costs for the execution and data access are usually defined by the Cloud provider. The additional services we have implemented are used for achieving a certain consistency level and we have defined the costs for these services. However, we have used the pricing schema of the AWS SQS and the charges are based on the costs for a request to a specific service and the data transfer. We have used a standard price for all services, i.e. the costs for a request or a reply (lock messages, 2PC messages and freshness messages) is the same independently of the service type, namely \$0.001. In addition, costs are generated depending on the data transferred into or out of a service. In our experiments we have charged \$0.00001 per byte.

Experiment Parameters: In our experiments we have used four replicas and two clients. The clients start transactions which are assigned randomly to one of the replicas. Each experiment is repeated 10 times. The number of book types in the system was set to 1'000 each with an instance chosen randomly between 10-100. Each transaction buys 10 different book types and a randomly chosen number of instances of a single book type between 1-10. The transaction size is set to 50, i.e. each transaction works on 50 objects (reads and writes) according to the Ordering Mix specified by the TPC-W.

B. Costs per Transaction

The goal of this experiment was to show the advantages of the transactions with adaptive behavior of C^3 with regard to the overall costs compared to transactions with fixed consistency level. During this experiment transactions are provided with the inconsistency costs and the default cost strategy (see Section V). This means that adaptive transactions are executed based on Rule 3 (Section V). The parameters as specified in VI-A are also used in this experiment; additionally we have set the inconsistency costs to \$1, based on an empirical analysis. The experiment consists of single tests, where each test generates during a period of 300 seconds a system load of 500, 1'000, 1'500, 5'000, 6'000 and 10'000 transactions. Each test is executed as follows. First, the system and the data are initialized. Afterwards, independently 1SR, SSI, SC, and adaptive transactions are generated and executed by the system. It is important to notice that during test execution, before transactions of a specific consistency level are executed, the system and data are re-initialized. This is done in order to ensure fairness between transactions running different consistency levels.

The consistency costs of the different CCPs are depicted in Figure 1. When the number of transactions increases, the consistency costs of 1SR and SSI are getting higher, which is a consequence of the increasing lock conflict rate. SSI transactions have lower consistency costs than 1SR, since no read-locks are used. This leads to a decreased conflict rate compared to 1SR. SC transactions have constant consistency costs, which are generated by the startup and commit checks (Section IV-C). The interesting aspect of this evaluation is the behavior of the adaptive transactions. These will be executed mainly with SC if the inconsistency probability is low, which is the case with low numbers of transactions (500, 1'000, 1'500). As the number of transactions increases, the inconsistency probability will also go up. This explains the increase in the consistency costs depicted in Figure 1. On the other hand, transactions running with fixed SC will observe an increasing number of inconsistencies and consequently, higher inconsistency costs as the number of transactions is getting higher. Adaptive transactions, in contrast, will find the right balance between inconsistency and consistency costs. This explains the difference in the overall costs between SC and adaptive transactions (see Figure 2). Figure 3 summarizes the overall costs (sum of consistency and inconsistency costs) over all tests, which clearly shows the advantage of adaptive transactions. During our evaluations, we have used rather low inconsistency costs. Real applications have much higher inconsistency costs, which would be even more in favor of the adaptive transaction model. In the online bookstore scenario, the adaptive transactions will lead to a switch between SC and SSI, since the expected write-skew costs are very low. In other scenarios the situation may look different, especially if the expected write-skew costs are high. In that case adaptive transactions will finally also switch to 1SR.

C. Vary Inconsistency Costs

This experiment shows the impact of the inconsistency costs on the behavior of the C^3 system. The setup is as follows. The number of executed transactions remains the same, namely 5'000. The inconsistency costs are varied between 0.01 and 0.25. The results depicted in Figure 4 show the behavior of adaptive transactions with increasing inconsistency costs. As long as the inconsistency costs are low enough, adaptive transactions will be executed mainly

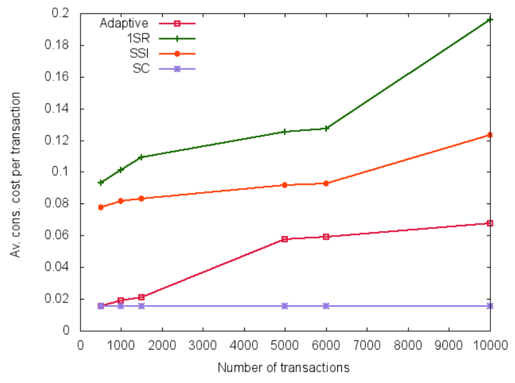

Figure 1. Consistency Costs of the Different CCPs

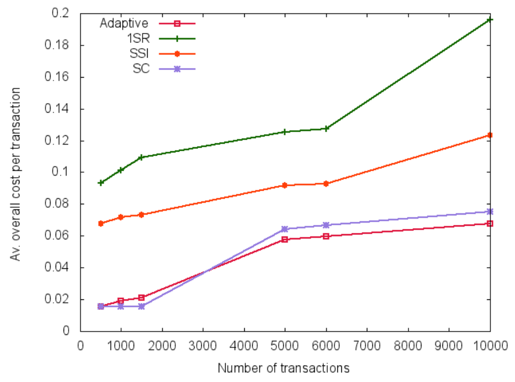

Figure 2. Overall Costs (Consistency and Inconsistency Costs)

with SC. As the inconsistency costs increase more adaptive transactions will switch to SSI. As a consequence, the consistency costs of adaptive transactions will also increase. However, this are much lower than SSI. In Figure 5 it can be seen that in addition to the consistency costs, also the response time of adaptive transactions will increase with increasing inconsistency costs.

VII. Conclusions and Outlook

Cloud providers have to offer a flexible yet powerful transactional middleware in order to support a wide range of CCPs which are needed to accommodate the needs of applications of different types. One of the main reason for

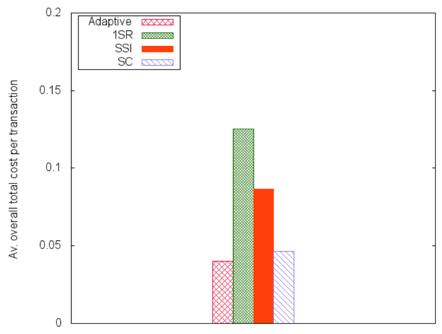

Figure 3. Total Average Costs over all Tests

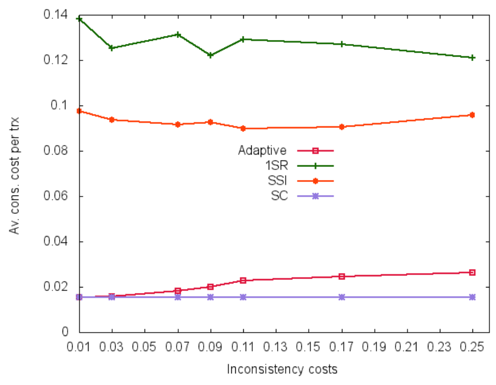

Figure 4. Consistency Costs of the Different CCPs

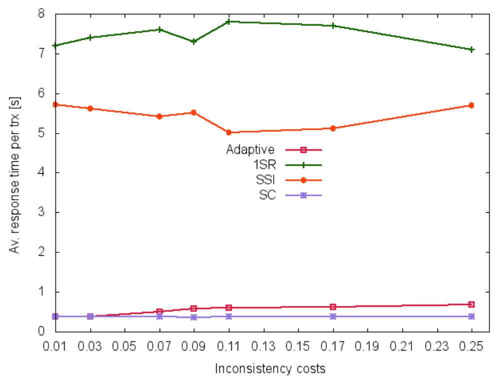

Figure 5. Response Time of the Different CCPs

customers deploying their applications in the Cloud is its *pay-per-use* business model. The Customers do not have to do any upfront investments to build scalable systems. They can deploy their applications in the Cloud and pay for what they consume. The main goal is actually to generate profit from their business at low operational costs. However, different CCPs generate different costs, as the higher the consistency guarantees it provides, the higher the costs and the lower the scalability that can be provided. Weak consistency generates less operational costs, but may generate high penalty costs, due to, for example, oversells, customer disappointment, etc. In this paper, we have presented our C^3 which is independent of the infrastructure of a specific Cloud provider. The main features of C^3 are its adaptive behavior (a CCP does not have to be specified at build-time but is dynamically selected at run-time), and its special treatment for consistency mixes having conflicting data access. Since it is costly for customers to execute transactions with strong consistency, they need the guarantee that either the system can enforce the consistency (the users get what they have paid for), or the system will not waste the user's budget if it cannot provide such guarantees. The evaluation of C^3 shows that transactions with adaptive behavior outperform the transactions which have fixed behavior, not only from the monetary costs point of view, but also from the performance point of view.

In future work we plan to extend the cost model for non-uniform data access [17] where some data objects might be accessed more frequently than others. We will extend the model to take into account hotspots. Moreover, in general, the architects/developers of transactional systems are faced with the task of selecting the most appropriate consistency level for their transactions. For C^3, this task is even more difficult, since additional parameters come into play. Therefore, we will develop a catalog with generic criteria for deciding on the consistency level of transactions.

Acknowledgment

This work was supported by the Swiss National Science Foundation (project GriDMan) No. 200021_132201 / 1.

References

[1] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a Database on S3," in *Proc. 2008 SIGMOD*. Vancouver, Canada: ACM, 2008.
[2] S. Das, D. Agrawal, and A. El Abbadi, "G-Store: a Scalable Data Store for Transactional Multi Key Access in the Cloud," in *Proc. SoCC 2010*, Indianapolis, IN, USA, 2010.
[3] D. Kossmann and T. Kraska, "Data Management in the Cloud: Promises, State-of-the-art, and Open Questions," *Datenbank-Spektrum*, 2010.
[4] R. Rawson and J. Gray, "HBase at Hadoop World NYC." http://www.docstoc.com/docs/12426408/HBase-at-Hadoop-World-NYC/, 2009.
[5] F. Yang, J. Shanmugasundaram, and R. Yerneni, "A Scalable Data Platform for a Large Number of Small Applications," in *Proc. CIDR'09*, Asilomar, CA, USA, 2009.
[6] E. A. Brewer, "Towards Robust Distributed Systems (abstract)," in *Proc. PODC'2000*, Portland, OR, USA, 2000.
[7] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency Rationing in the Cloud: Pay only when it Matters," *PVLDB*, 2009.
[8] Y. Lu, Y. Lu, and H. Jiang, "Adaptive Consistency Guarantees for Large-Scale Replicated Services," in *Proc. NAS 2008*, 2008.
[9] Y. Yang and D. Li, "Separating data and control: support for adaptable consistency protocols in collaborative systems." in *CSCW'04*, 2004.
[10] I. Fetai and H. Schuldt, "Cost-based data consistency in a data-as-a-service cloud environment," University of Basel, Switzerland, CS Technical Report CS-2012-001, Feb. 2012, available at http://informatik.unibas.ch/research/publications_tec_report.html.
[11] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-r, a new way to implement database replication," in *Proc. VLDB'2000*. Cairo, Egypt: Morgan Kaufmann, 2000.
[12] ——, "Database Replication: a Tale of Research across Communities," *PVLDB*, 2010.
[13] H. Schuldt, "Process Locking: A Protocol based on Ordered Shared Locks for the Execution of Transactional Processes," in *Proc. PODS'01*. Santa Barbara, USA: ACM Press, 2001.
[14] S. Elnikety, "Database Replication using Generalized Snapshot Isolation," in *Proc. SRDS'05*, Orlando, FL, USA, 2005.
[15] K. Daudjee and K. Salem, "Lazy Database Replication with Snapshot Isolation," in *Proc. VLDB'06*, Seoul, Korea, 2006.
[16] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," in *Proc. SIGMOD 2008*, Vancouver, Canada, 2008.
[17] S. Banerjee and V. Li, "A General Model for Non-Uniform Data Access in a Database System," in *Proc. COMPSAC*, 1991.