Cost-Based Adaptive Concurrency Control in the Cloud

Ilir Fetai Heiko Schuldt

Technical Report CS-2012-001
University of Basel
Email: {ilir.fetai|heiko.schuldt}@unibas.ch

Abstract

The recent advent of Cloud computing has strongly influenced the deployment of large-scale applications. Currently, Cloud environments mainly focus on high scalability and availability of these applications. Consistency, in contrast, is relaxed and weak consistency is commonly considered to be sufficient. However, an increasing number of applications are no longer satisfied with weak consistency. Strong consistency, in turn, decreases availability and is costly to enforce from both a performance and infrastructure point of view. On the other hand, weak consistency may lead to high costs due to the access to inconsistent data. In this technical report, we introduce a novel approach called *cost-based concurrency control (C^3)*. Essentially, C^3 allows to dynamically and adaptively switch at runtime between different consistency levels of transactions in a Cloud environment based on the costs incurring 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*). C^3 jointly supports transactions of different consistency levels and enforces the inherent consistency guarantees of each protocol. We first analyze the consistency costs of concurrency control protocols; second, we specify a set of rules that allow to dynamically select the best consistency level with the goal of minimizing the overall costs; third, we provide a protocol that enforces the correct execution of all transactions in a transaction mix. We have evaluated C^3 on top of amazon's EC2. The results show that C^3 leads to a reduction of the overall transaction costs compared to a fixed consistency level.

Keywords: Cloud Computing, Data-as-a-Service, Transaction Management, Concurrency Control, Data Consistency, Cost Model

Department of Mathematics and Computer Science
Bernoullistrasse 16
Phone: +41 61 267 05 55 Fax: +41 61 267 05 59

Databases and Information Systems
CH – 4056 Basel, Switzerland.
URL: http://dbis.cs.unibas.ch/

1 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 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 for what they consume.

Data management is a central aspect of applications deployed in the Cloud [2, 3]. Usually, application and web servers can be easily scaled out by adding new server instances. However, by replicating only the servers, high availability of the data cannot be guaranteed and 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 generated by the additional work which needs to be done in order to compensate the effects of data inconsistency (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, will be provided by *cost-based concurrency control* (C^3), the novel approach to concurrency control in Cloud environments we propose in this technical report. With C^3, the overall cost of the application will be minimized. If we take for example an online book store application, the transactions for modifying credit card information should probably run under strong consistency guarantees (1SR or SI). On the other hand, transactions for buying books may run with SC as long as the expected oversell costs (inconsistency costs) are low enough. This will save operational costs. However, the system should switch to a stronger consistency level as soon as the expected oversell costs in SC are higher than the expected costs of the stronger consistency level (SI or 1SR), in order to avoid penalty costs.

We argue that Cloud providers should offer a transactional middleware implementing a wide range of CCPs. Moreover, the should allow application developers to specify the desired consistency at transaction level. The middleware should also support C^3 semantics, with the goal of finding the right balance between consistency and inconsistency costs. A precondition for such a middleware is the analysis and development of a cost model for the different CCPs. This, together

with a concrete prizing model of a Cloud provider, is needed as a basis for the implementation of C^3. As part of this work we have implemented a middleware which provides a range of CCPs and C^3 semantics.

The contributions of this technical report are as follows:

1. We provide a step-by-step analysis of the cost components of the different CCPs. This is particularly interesting, as in the context of Cloud computing, providers publish detailed prizing schemes that allow to quantify the costs incurring for using compute resources and thus to derive the consistency costs of different CCPs.

2. We introduce C^3 which is based on a generic cost model for adaptive concurrency control.

3. 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 report is organized as follows: Section 2 presents related work. In Section 3 we provide an overview of replication and concurrency control in replicated systems. Section 4 introduces C^3 and presents the C^3 system model and its components. The core of this section is however the analysis of the anomalies in case of transaction mixes and the mechanism for avoiding these anomalies. Section 5 provides a step-by-step analysis of the cost components of the different CCPs and Section 6 presents the details of the C^3 protocol. The evaluation of C^3 on top of amazon EC2 is provided in Section 7. Section 8 concludes.

2 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, which require strong transactional guarantees. In [3] the authors mention the problem of consistency vs. availability as one of the main open challenges for the data management in the Cloud. This issue is not specific to the Cloud, even though in the Cloud there is an additional level complexity due to the presence of the cost parameters.

Kemme and Alonso [8, 9] present one of the first approaches to directly address the issue of replication management that provides strong consistency guarantees and that is 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 operational losses due to, for example, oversells [7]. The cost parameter increases the complexity of designing efficient CCPs. In contrast to the approach in [8], our work is not focused

on designing new efficient CCPs, but is rather based on the idea of adaptive concurrency control on top of different CCPs as presented in [10]: we use existing CCPs and adjust the consistency level at runtime according to specific constraints. In [7], different types of applications are described which require changes of the consistency level 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. The A category contains data which require 1SR, the C category encompasses data that are handled with session consistency, whereas category B data is handled with adaptive consistency. Adaptive consistency means that the consistency level is changed between 1SR and Session Consistency at runtime depending on the specific policy defined at data level (annotation of the schema). Different types of policies have been implemented which can cope with different types of applications. This limits the approach presented in [7] 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 [11]. However, this does not consider any infrastructure-related costs for enforcing a selected CCP.

3 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, Strong 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 [12]. The implication of 1SR is that each transaction, which writes to a data object, must update all copies of that data object. Hence, all replica updates are done in the context of one single transaction. Depending on the number of replicas, this may considerably increase the response time of update transactions. 1SR guarantees always consistent data, but has the disadvantage of decreasing availability and scalability.

Snapshot Isolation (SI) The idea of SI is to achieve increased concurrency compared to 1SR [13]. The advantage of SI arises from the fact that reads are never blocked. SI avoids many of the possible inconsistencies, however the *write-skew* [13] anomaly is possible. Write-skew anomalies may occur if two transactions T_i and T_k concurrently read an overlapping data set and concurrently make disjoint updates. 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)* [14]. 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 [14]. 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 [15]. Other possible SI variants, which relax the consistency requirements for replicated systems are described in [14, 16].

Session Consistency (SC) SC is a variation of the eventual consistency model[1] [17]. 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. It is important to notice that 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 the 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.

4 C^3 Overview

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" as such a protocol would be strongly dependent on the application for which it is used. Each CCP finds its own balance between consistency guarantees it provides and costs for enforcing it. In contrast, the 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 and decrease response time of transactions while at the same time providing the best possible correctness guarantees.

4.1 C^3 System Model

The C^3 system model is based on a full replication and 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 C^3 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

[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.

of dynamic replication, so that transaction commits are not interfered by the replica deployment process.

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 4.3).

All components are implemented as Web services and can be deployed in different possible configurations. Regarding the logical architecture, a replica consists of a TransactionManager and a SiteManager, i.e., when talking about replicas, we refer to both components. 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.

4.2 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 only read operations, 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 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 for avoiding anomalies in case of consistency mixes (see Section 4.4).

4.3 FreshnessManagement

The FreshnessManager manages freshness information for each of the data objects. This information is used mainly for the replica synchronization mechanism described in Section 4.5. We distinguish two types of freshness information: last committed timestamp for non-commutative updates (e.g., strings), see Table 1; and a range of timestamps for commutative updates (e.g., integers). The commit timestamp range is sorted in ascending order, see Table 2. 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.

OID	commitTimestamp	replicaId
objectA	1317728037	131.152.55.100:8080

Table 1: Last committed timestamp – *non-commutative* updates

OID	commitTimestamp	replicaId
objectB	1317728037	131.152.55.91:8080
	1317728237	131.152.55.91:8080
	1317739437	131.152.55.100:8080

Table 2: Range of committed timestamps – *commutative* updates

4.4 Avoiding Anomalies in 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 concurrent 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. This is one of the major differences between our and [7], as C^3 allows different applications to work on the same data and still satisfy their different consistency requirements while in [7], the consistency level is fixed and attached to a data object. 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 types of 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 CCP (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 second type of inconsistencies 2. Inconsistencies of type 3 are handled by the replica synchronization mechanism described in Section 4.5 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 1. Cahill et al. [18] have already provided a mechanism which avoids inconsistencies between SI transactions, i.e., which makes SI serializable. Another important fact is that if for example two SSI transactions have introduced an inconsistency (e.g., write-skew), our mechanism will just make sure that a transaction (1SR or SSI) requiring latest data will get them. But, the existing 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.

1SR-SSI In the case of 1SR and SSI transactions accessing the same data objects, only the 1SR transactions may observe an unrepeatable read or a phantom read in case they read data which

is written by SSI transactions. This may happen since SSI transactions do not check for or use read-locks. The simple solution is to extend the SSI lock mechanism to check if a read-lock of a 1SR transaction is set before acquiring a write-lock. Note that SSI transactions are not affected by 1SR concurrent writes since they read from their own snapshot.

1SR–SC and SSI–SC If any transaction running with strong consistency (1SR or SSI) accesses the same data objects that are also accessed in write-mode by an SC transaction, all of them may observe different anomalies. Since SC is the lowest consistency level, inconsistencies are expected. It means, the mechanism described here does not try to avoid problems on the SC side, but on the 1SR and SSI side. These are the costly transactions and the user expects them to behave properly. The anomalies we are trying to avoid are the following:

1. 1SR–SC: w/w and r/w conflicts, i.e., lost update and inconsistent reads on the 1SR side. In case of r/r no anomalies are possible, whereas w/r conflicts may only lead to anomalies on SC side.

2. SSI–SC: w/w conflicts, i.e., lost-update on the SSI side. In this case no r/w conflicts are possible, since SSI transactions read from their own snapshot.

The mechanism for avoiding inconsistencies consists of two steps, and it is based on a centralized LockManager and FreshnessManager. The fully decentralized approach is foreseen for future extension of C^3.

1. Checks at transaction startup.

2. Checks at commit time.

Startup checks Before an 1SR or SSI transaction starts executing, i.e., as the very first step, it checks *only* in the local replica whether there are conflicting accesses with the currently running SC transactions. The rest of the necessary checks for avoiding system-wide conflicts is done at commit time. If there are no conflicts, then all locks can be set and execution can start. Additionally, SSI transactions must check for read-locks before acquiring a write-lock in order to avoid conflicts/anomalies between 1SR and SSI. If conflicts are detected, than either the transaction is aborted or its consistency level is changed to SC. SC costs are lower and the user may want to accept inconsistencies. It would have been possible to do all the checks at commit time and avoid the checks at startup. However, by doing some checks at startup we lower the probability of an 1SR or SSI transaction being aborted after its execution and thus wasting a large amount of work and generating high costs. SC transactions must also check at start time for conflicts with running 1SR and SSI transactions, in order to avoid disturbing running 1SR and SSI transactions.

Commit-time checks As already mentioned, conflict checks at startup are done only at the local node. It means, running SC transactions at other replicas are not taken into account. This mechanism minimizes the performance impact of the system. However, it means that remote conflicts are possible. In particular, there might be overlapping commits between 1SR, SSI, and SC transactions. In order to avoid inconsistencies, SC transactions have to check whether there is an overlapping commit by 1SR or SSI transactions and vice versa, which can be easily done using the information provided by the FreshnessManager. No check has to be done between 1SR and SSI

since both set the write-locks at the central LockManager and thus possible conflicts are avoided at startup. An overlapping commit is defined as follows: $commit(T_k) > start(T_i)$ and T_k has updated at least one data object read/written by T_i. In this case T_i must abort.

4.5 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. Figures 1 and 2 depict the execution order of each of the steps for 1SR, SSI, and SC update-transactions.

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.

As already mentioned, the FreshnessManager is a centralized component. Although it might be clustered in order to increase its performance and availability, it can not achieve the availability of a fully decentralized solution. A failure of the FreshnessManager would have a huge impact especially for transactions requiring the most recent data: 1SR and SSI. In case of failure and if the most recent data is not available in the local replica, 1SR and SSI transactions must either be aborted or the user has specified that the transaction's consistency level should be decreased, i.e., be executed with a low consistency level (SC). Transactions with strong consistency should never be executed based on stale data, since strong consistency is costly and the probability of inconsistencies is high if transactions are executed based on stale data. This might have as consequence high transaction costs (sum of consistency and inconsistency costs).

5 A Cost Model for Concurrency Control

In this section we present a step-by-step analysis of the costs of different CCPs. The cost model is based on the following implementation assumptions.

1. 1SR uses strong strict two-phase locking (SS2PL) for ensuring serializability.

2. All SSI write-locks are acquired at the beginning of a transaction (pessimistic approach) and released after the transaction's commit.

3. 2PC is used for the eager replication (1SR and SSI).

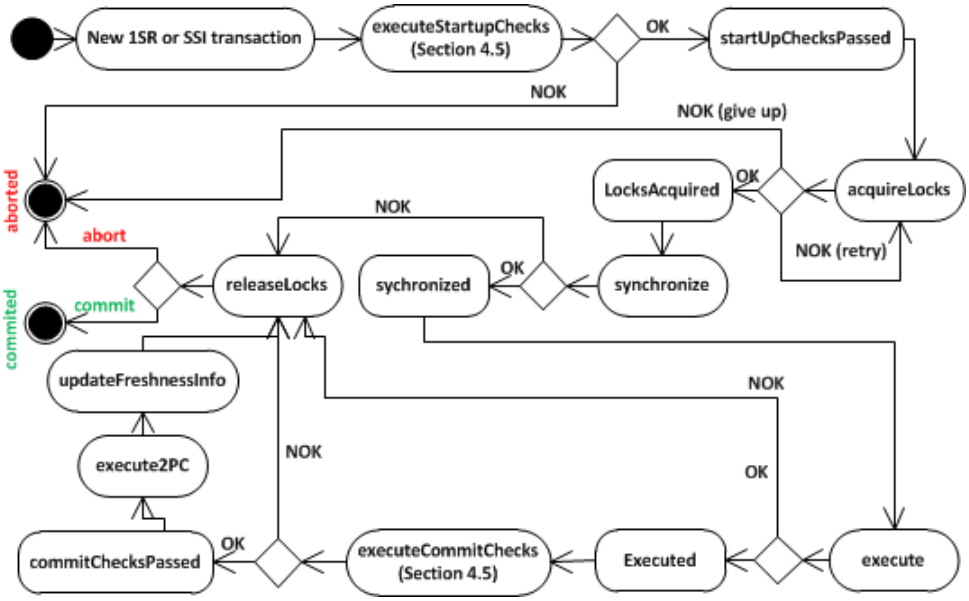

Figure 1: Execution of 1SR and SSI update-transactions

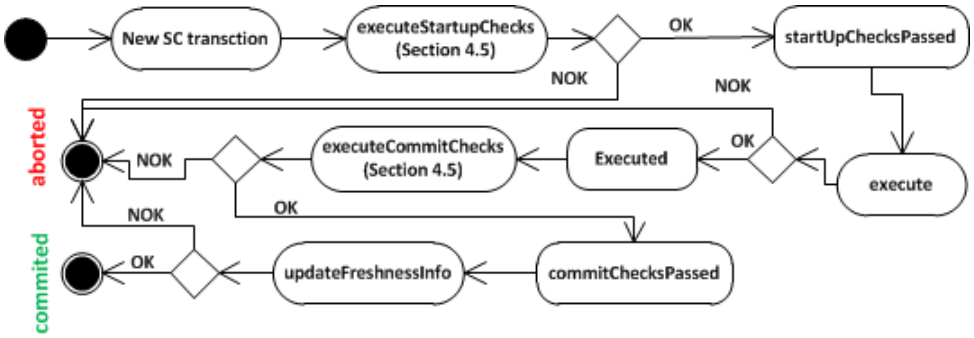

Figure 2: Execution of SC update-transactions

Note that these assumptions are not intrinsic to C^3. Slightly different implementation assumptions would lead to minor changes in the cost model but would not affect the behavior of adaptive concurrency control in C^3.

5.1 Cloud Cost Models

All Cloud providers offer the following basic resources: CPU, Storage and Network Bandwidth. This is commonly referred to as *Infrastructure as a Service (IaaS)*. Based on IaaS, it is possible to provide a *Platform as a Service (PaaS)*, for developing and deploying applications. The next step in the service stack is *Software as a Service (SaaS)*. For IaaS, the providers usually charge for the components used (machine, network bandwidth, etc.). For PaaS and SaaS, the cost model is usually based on aggregated costs, i.e., the costs of IaaS are included and additional costs might also be taken into account (e.g., licensing costs), but the accounting is based for example on the

number of requests made to a specific service. If we take for example the AWS SQS[2] customers are charged based on the number of read/write requests and the amount of data read from and written to SQS. All cloud pricing models of the currently available providers can be classified as follows.

1. Pay-per-use: there are no fixed costs, and the provider charges only for the effectively used resources (e.g., Google AE).

2. Fixed costs: the provider charges for example a monthly fee, which is independent of the used resources and no further costs are charged (e.g., Microsoft Azure).

3. Fixed and pay-per-use: The provider charges a fixed fee, but the pay-per-use prices are reduced (e.g., AWS).

The C^3 cost model is based on service (aggregated) costs rather than costs for the IaaS, analogous to the AWS SQS pricing model. We are interested in the costs for using the necessary services to implement a specific CCP. Services in our case are the different components as described in Section 4.1.

5.2 Transaction Consistency Cost Components

The following consistency cost components incur depending on the provided consistency guarantee by a specific CCP:

1. **Startup Check Costs:** These costs incur due to the mechanism for avoiding anomalies in case of consistency mixes as described in Section 4.4.

2. **Lock Costs:** Depending on the implementation of different CCPs, additional lock costs are generated. 1SR uses read and write locks, whereas SSI only uses write locks. SC does not use any locks at all.

3. **Replica Synchronization Costs:** Depending on a transaction's consistency requirements, an (outdated) replica might have to be synchronized before it can be effectively executed.

4. **Commit Check Costs:** In addition to the startup checks, each transaction must execute additional checks before commiting, to ensure that no data from its read or write set has been updated after its start (Section 4.4).

5. **2PC Costs:** We assume that 2PC is used in order to ensure atomic commitment of replica updates. Hence, for synchronous updates, additional costs for the execution of 2PC incur.

6. **Freshness Update Costs:** In our model each transaction is responsible for updating the freshness information at the FreshnessManager, which generates additional costs (Section 4.3).

5.3 Lock Costs

Lock costs consist of the aggregated service costs for using the lock service.

[2]SQS is the Amazon's Queueing Service.

Parameter	Definition
C_{LR}	Costs of a lock request
C_{LA}	Costs of a lock acknowledgement
C_{ULR}	Costs of an unlock request
C_{ULA}	Costs of an unlock acknowledgement
N_{LM}	Number of lock managers. Since we are using a centralized lock manager, $N_{LM} = 1$

Table 3: Lock Cost Parameters

Basic Assumptions **(1)** Lock management is based on S2PL and works in non-blocking mode, i.e., if the lock for any object at any lock manager could not be acquired, all successfully acquired locks are released. By this we avoid deadlock situations. **(2)** Lock requests are bundled in one message.

Lock Cost Analysis Table 3 lists the parameters used in the calculation of the lock costs.

Cost model The costs for getting the locks consist of lock requests, unlock requests and the corresponding acknowledgements.

$$\begin{aligned} C_{LockSucc} &= C_{LR} + C_{LA} + C_{ULR} + C_{ULA} \\ C_{LockFail} &= C_{LR} + C_{LA} \end{aligned} \tag{1}$$

The total lock costs by taking into account failures and number of attempts is given below:

$$C_{L_{Total}} = (\sum_{failedAttempt=0}^{N_{LFailedAttempts}} C_{LockFail}) + C_{LockSucc} \tag{2}$$

5.4 Expected Lock Costs

Basic assumptions All transactions are considered to be of the same size (N) and lock access is uniformly distributed.

Expected Lock Cost Analysis In Table 4 we have listed the parameters used in the calculation of the expected lock costs.
1SR transactions can observe r/w, w/r and w/w lock conflicts, since read- and write-locks are used. In case of SSI, r/w lock conflicts are not possible, due to the missing read-locks. SC transactions do not use any locks, so no lock conflicts can be observed by SC transactions.

The probability of the i-th lock request failing is given with:

$$\begin{aligned} P_{c,i} &= \frac{TotalOccupied - i}{DB_{Size} - i} \\ P_{AllLocksSucceed} &= (1 - P_{c,0}) \cdot \cdots \cdot (1 - P_{c,NLocks-1}) \\ P_{AllLocksSucceed} &= \prod_{i=0}^{NLocks-1} \frac{DB_{Size} - TotalOccupied}{DB_{Size} - i} \end{aligned} \tag{3}$$

Parameter	Definition
N_R	Total number of objects accessed in read mode by a transaction
N_W	Total number of objects accessed in write mode by a transaction
N	Total number of objects accessed by a transaction $N = N_R + N_W$
$NLocks$	Total number of locks, for 1SR transaction $NLocks = N$ for SSI transaction $NLocks = W$
DB_{Size}	Total number of distinct objects in the system

Table 4: Expected Lock Cost Parameters

Parameter	Definition
$C_{2PCPrepare}$	Costs of a 2PC prepare message
$C_{2PCPrepareACK}$	Costs of a 2PC prepare ACK
$C_{2PCDecision}$	Costs of a 2PC decision message
$C_{2PCDecisionACK}$	Costs of a 2PC decision ACK
R	Number of replicas

Table 5: 2PC Cost Parameters

$TotalOccupied$: In case of 1SR transactions, for the probability calculation of an r/w conflict, $TotalOccupied$ corresponds to the total number of currently occupied write-locks. For the w/r and w/w conflicts all occupied locks have to be taken into account, i.e., read- and write-locks. The same applies also in case of SSI transactions, although per definition SSI transactions do not use read-locks. However, in order to avoid inconsistencies between 1SR and SSI transactions, as defined in the mechanism for avoiding inconsistencies in case of transaction mixes (Section 4.4), SSI transactions must check for read-locks before acquiring a write-lock. The expected number of retries until a transaction gets all locks is $E(N_{LRetries}) = \frac{1}{P(AllLocksSucceed)}$.

Finally, the total expected lock costs $E(C_{Locking})$ are:

$$\begin{aligned} E(C_{Locking}) = E(N_{LRetries}) \cdot C_{LockFail} \\ + C_{LockSucc} \end{aligned} \tag{4}$$

5.5 2PC Costs

As in the case of lock costs we are only interested in the service (aggregated) costs for executing 2PC, i.e., costs for exchanging 2PC control messages. The costs for each of the 2PC messages is calculated based on a service price model, i.e., number of requests and size of data exchanged. If we take for the propagation of update logs, which is done as part of the prepare message, $C_{2PCPrepare}$ is calculated by also taking into account the size of the update logs.

$$\begin{aligned} C_{2PC} = (R-1) \\ \cdot (C_{2PCPrepare} + C_{2PCPrepareACK} \\ + C_{2PCDecision} + C_{2PCDecisionACK}) \end{aligned} \tag{5}$$

5.6 Expected 2PC Costs

The handling of the different possible failures during 2PC (of the coordinator or of a normal node, called 'agent) generates additional costs. The following failures may occur during execution of 2PC.

1. **Coordinator fails before writing commit log record:** The coordinator repeats sending of "prepare" message. In the worst case, additional costs of $2 \cdot (R-1) \cdot C_{2PCMsg}$ incur.

2. **Coordinator fails while one or more agents are uncertain and agent timeout expires:** If the coordinator remains unavailable for a longer period of time, the agents remain blocked waiting for the decision. By the use of the *cooperative termination protocol* the agents can reach a decision even if the coordinator is unavailable. However, the precondition is that at least one of the agents received the decision. If all of them are uncertain, then the cooperative termination protocol causes considerable extra costs, without being able to reach a decision. A detailed description of the cooperative termination protocol can be found in [12]. In the worst case, i.e., all replicas are uncertain, costs of $\left(\frac{3 \cdot R^2 - 5 \cdot R + 2}{2}\right) \cdot C_{2PCMsg}$ incur.

3. **Coordinator fails while one or more agents are uncertain, coordinator recovers before timeout expires:** In this case, the coordinator will resend the decision to all agents. The agents will respond again with ACK. The number of additional messages is $2 \cdot (R-1) \cdot C_{2PCMsg}$.

4. **One or more agents fail before writing the prepared log record:** Let R_{Failed} be the number of failed agents before writing the prepared log record. The coordinator will keep resending "prepare" message until it receives a vote. The additional costs generated are $(N_{Retries} \cdot R_{Failed} + R_{Failed}) \cdot C_{2PCMsg}$.

5. **One or more agents fail before writing the decision log record:** Let R_{Failed} be the number of failed agents before writing the decision log record. The coordinator will keep sending the decision. After the agents have recovered they will respond with ACK. The additional costs generated are $(N_{Retries} \cdot R_{Failed} + R_{Failed}) \cdot C_{2PCMsg}$.

Basic assumptions Either the coordinator has failed or one or several agents.

Expected 2PC Cost Analysis In Table 6 we have listed the parameters used in the calculation of the expected 2PC costs.
Equation (6) defines the probabilities for the possible coordinator failures during 2PC. Equations (7) and (8) define the probabilities of agent failures during the different phases of 2PC execution, whereas Equation (9) defines the probability of agents being uncertain during coordinator failure. In Equation (10) we specify the expected 2PC costs by taking into account the different possible failures.

$$\begin{aligned} P(CooFBLD) &= P(F_{Coo})[T_{F_{Coo}} < T_{CooCLR}] \\ P(CooFBEL) &= P(F_{Coo})[T_{F_{Coo}} < T_{CooELR}] \end{aligned} \tag{6}$$

Parameter	Definition
F_{Coo}	Coordinator failure event
$T_{F_{Coo}}$	Indicates the time when the coordinator fails
$P(F_{Coo})$	Probability of coordinator failure event
T_{CooCLR}	Indicates the time when the coordinator writes commit/abort log record
T_{CooELR}	Indicates the time when the coordinator writes end of transaction log record
$P(CooFBLD)$	Probability that the coordinator fails before logging the commit/abort decision
$P(CooFBEL)$	Probability that the coordinator fails before logging the end of transaction
F_{Ag_i}	Agent *i* failure event
$T_{F_{Ag_i}}$	Indicates the time when agent *i* fails
$P(F_{Ag_i})$	Probability of agent *i* failure event
T_{AgP_i}	Indicates the time when agent *i* sends prepared message to coordinator
T_{AgRC_i}	Indicates the time when agent *i* receives the commit message from the coordinator
T_{AgPLR_i}	Indicates the time when agent *i* writes prepared log record
T_{AgCLR_i}	Indicates the time when agent *i* writes commit/abort log record
$P(AgFBPL_i)$	Probability that agent *i* fails before logging the prepared decision
$P(AgFBPL)$	Probability that 1..* agents fail before logging the prepared decision
$P(AgFBCL_i)$	Probability that agent *i* fails before logging the commit/abort decision
$P(AgFBCL)$	Probability that 1..* agents fail before logging the commit/abort decision
$P(AgU_i)$	Probability that agent *i* is uncertain when the coordinator has failed
$P(AgU)$	Probability that 1..* agents are uncertain when the coordinator has failed

Table 6: 2PC Expected Cost Parameters

$$\begin{aligned} P(AgFBPL_i) &= P(F_{Ag_i})[T_{F_{Ag_i}} < T_{AgPLR_i}] \\ P(AgFBPL) &= (P(AgFBPL_1) \cup \cdots \cup P(AgFBPL_{R-1})) \end{aligned} \tag{7}$$

$$\begin{aligned} P(AgFBCL_i) &= P(F_{Ag_i})[T_{F_{Ag_i}} < T_{AgCLR_i}] \\ P(AgFBCL) &= (P(AgFBCL_1) \cup \cdots \cup P(AgFBCL_{R-1})) \end{aligned} \tag{8}$$

$$\begin{aligned} P(AgU_i) &= P(F_{Coo})[T_{AgP_i} \leq T_{F_{Coo}} < T_{AgRC_i}] \\ P(AgU) &= P(AgU_1) \cup P(AgU_2) \cup \cdots \cup P(AgU_{R-1}) \end{aligned} \tag{9}$$

Parameter	Definition
$C_{FreshnessReq}$	Costs of a freshness request
$C_{FreshnessRepl}$	Costs of a freshness reply
$C_{SynchReq}$	Costs of a synchronisation request
$C_{SynchRepl}$	Costs of a synchronisation reply
$E(N_{Synchs})$	Expected number of necessary synchronization interactions. Includes bulking possibility.
$E(C_{TotalSynch})$	Expected total costs of synchronisation

Table 7: Replica Synchronization Cost Parameters

$$\begin{aligned} E(C_{2PC}) = C_{2PC} &+ (P(CooFBLD) + P(CooFBEL)) \\ &\cdot 2 \cdot (R-1) \cdot C_{2PCMsg} \\ &+ P(AgU) \cdot C_{2PCTerminationProtocol} \\ &+ (P(AgFBPL) + P(AgFBCL)) \\ &\cdot (N_{Retries} \cdot R_{Failed} + R_{Failed}) \cdot C_{2PCMsg} \end{aligned} \tag{10}$$

5.7 Expected Replica Synchronization Costs

Table 7 lists the parameters used in the calculation of the expected costs of the replica synchronization mechanism, which is described in Section 4.5. Again, the cost parameters follow the model of service costs and are calculated by also taking into account the size of the request. For example $C_{FreshnessReq}$ is calculated on the basis of the number of objects for which the freshness information should be provided.

$$\begin{aligned} E(C_{TotalSynch}) = &(C_{FreshnessReq} + C_{FreshnessRepl}) \\ &+ E(N_{Synchs}) \cdot (C_{SynchReq} + C_{SynchRepl}) \end{aligned} \tag{11}$$

5.8 Expected Inconsistency Costs

Inconsistencies can occur in the following cases:

1. Anomalies due to the isolation level of the transactions running the same consistency level: Anomalies between SC transactions and the write-skew anomaly between SSI transactions.

2. Inconsistencies resulting from the access to stale data, which may be observed by SC transactions due to the lazy propagation. This may happen if an SC transaction T_i updates data at replica R_n and if another SC transaction T_k accesses that data at replica R_m during the inconsistency window of R_m. 1SR and SSI transactions avoid this problem by synchronizing the replicas before the transaction operations are executed. Inconsistencies due to consistency mixes are prevented from occurring by the mechanism defined in Section 4.4.

Basic assumptions The access probability is the same for all objects. All transactions are of size N.

Parameter	Definition
C_{Inc}	Inconsistency costs
$N_{SCAccObjR}$	Number of currently accessed distinct objects by SC transactions in read mode
$N_{SCAccObjW}$	Number of currently accessed distinct objects by SC transactions in write mode
X_{SC}	Random variable which models the number of conflicting accesses for the different possible conflicts
$P(X_{SC_{rw}} = 0)$	Probability that no conflicting r/w access occurs between SC transactions
$P(SC_{Crw})$	Probability that at least one conflicting w/r access occurs ($P(X_{SC_{rw}} \geq 1)$)
$P(X_{SC_{wr}} = 0)$	Probability that no conflicting r/w access occurs between SC transactions
$P(SC_{Cwr})$	Probability that at least one conflicting w/r access occurs ($P(X_{SC_{wr}} \geq 1)$)
$P(X_{SC_{ww}} = 0)$	Probability that no conflicting w/w access occurs between SC transactions
$P(SC_{Cww})$	Probability that at least one conflicting w/w access occurs ($P(X_{SC_{ww}} \geq 1)$)
$P(SCInc)$	Probability of SC inconsistency
$E(C_{SCInc})$	Expected SC inconsistency cost

Table 8: Expected SC Inconsistency Cost Parameters

Expected SC Inconsistency Costs Table 8 lists the parameters used in the calculation of the expected SC inconsistency costs. The probability of an r/w conflict is $P(SCConflict_{rw}) = \frac{N_{SCAccObjW}}{DB_{Size}}$, the probability of w/r conflict $P(SCConflict_{wr}) = \frac{N_{SCAccObjR}}{DB_{Size}}$, and the probability of a w/w conflict is given by $P(SCConflict_{ww}) = \frac{N_{SCAccObjW}}{DB_{Size}}$. The Equation below defines the probability of an anomaly between SC transactions.

$$
\begin{aligned}
P(SC_{Crw}) &= 1 - (1 - P(SCConflict_{rw}))^{N_R} \\
P(SC_{Cwr}) &= 1 - (1 - P(SCConflict_{wr}))^{N_W} \\
P(SC_{Cww}) &= 1 - (1 - P(SCConflict_{ww}))^{N_W} \\
P(SCAnomaly) &= P(SC_{Crw}) + P(SC_{Cwr}) + P(SC_{Cww}) \\
&\quad - P(SC_{Crw}) \cdot P(SC_{Cwr}) \\
&\quad - P(SC_{Crw}) \cdot P(SC_{Cww}) \\
&\quad - P(SC_{Cwr}) \cdot P(SC_{Cww}) \\
&\quad + P(SC_{Crw}) \cdot P(SC_{Cwr}) \cdot P(SC_{Cww})
\end{aligned}
\tag{12}
$$

In the case of SC, expected inconsistency costs (both isolation anomalies and data staleness) which are not mutually exclusive have to be taken into account. For the calculation of the probability of an SC transaction accessing stale data, we assume that the system periodically reconciliates. During the reconciliation phase no new updates are accepted. The probability of an SC transaction accessing stale data is thus calculated based on the number of SC update transactions executed after the last and before the new reconciliation. Let the $N_{SCUpdate}$ be the total number of SC update transactions executed between two reconciliations, $N_{SCUpdatedObj}$ the number of distinct

Parameter	Definition
$N_{SSIAccObjW}$	Number of currently accessed distinct objects by SSI transactions in write mode
$N_{SSIAccObjR}$	Number of currently accessed distinct objects by SSI transactions in read mode
$X_{SSI_{rw}}$	Random variable which models the number of r/w conflicts between SSI transactions
$P(X_{SSI_{rw}} = 0)$	Probability that no r/w conflict between SSI transactions occurs
$P(X_{SSI_{rw}} \geq 1)$	Probability that at least one r/w conflict between SSI transactions occurs
$X_{SSI_{wr}}$	Random variable which models the number of w/r conflicts between SSI transactions
$P(X_{SSI_{wr}} = 0)$	Probability that no w/r conflict between SSI transactions occurs
$P(X_{SSI_{wr}} \geq 1)$	Probability that at least one w/r conflict between SSI transactions occurs

Table 9: Expected Write-Skew Cost Parameters

objects updated by the $N_{SCUpdate}$ transactions, t the total number of executed SC transactions, and R the total number of replicas. The probability of an SC transaction accessing stale data is give by the Equation (13).

$$P(SCStale) = \overbrace{(1 - \sum_{t=0}^{N_{SCUpdate}} (\frac{1}{R})^t)}^{i} \cdot \overbrace{\frac{N_{SCUpdatedObj}}{DB_Size}}^{ii} \tag{13}$$

Part i of the Equation (13) defines the probability that not all transactions have been executed by the local node, whereas part ii denotes the probability of conflicts with the executed SC update transactions. The expected SC inconsistency costs are given by the Equation (14).

$$\begin{aligned} E(C_{IncSC}) = C_{Inc} \cdot (&P(SCAnomaly) \\ &+ P(SCStale) \\ &- P(SCAnomaly) \cdot P(SCStale)) \end{aligned} \tag{14}$$

Expected SSI Inconsistency Costs In case of SSI, only the write-skew inconsistency is possible. In Table 9 we have summarized the parameters used in the calculation of the expected write-skew costs. For two concurrent SSI transactions T_i and T_k a write-skew anomaly can occur if the following holds:
$readset_i \cap writeset_k \neq \emptyset \;\; \& \;\; writeset_i \cap readset_k \neq \emptyset$. The probability of a r/w conflict is $P(SSIConflict_{rw}) = \frac{N_{SSIAccObjW}}{DB_{Size}}$, whereas the probability of a w/r conflict is $P(SSIConflict_{wr}) = \frac{N_{SSIAccObjR}}{DB_{Size}}$. The probability that at least one r/w conflict occurs is given in Equation (15). Equation (16) then defines the probability that at least one w/r conflict occurs.

$$P(X_{SSI_{rw}} \geq 1) = 1 - (1 - P(SSIConflict_{rw}))^{N_R} \tag{15}$$

$$P(X_{SSI_{wr}} \geq 1) = 1 - (1 - P(SSIConflict_{wr}))^{N_W} \tag{16}$$

Since r/w and w/r conflict events are statistically independent, the probability of a write-skew anomaly occurring is given by:
$P(WriteSkew) = P(X_{SSI_{rw}} \geq 1) \cdot P(X_{SSI_{wr}} \geq 1)$. The expected write-skew costs are specified in Equation (17), whereas the total expected costs are given by the Equation (22).

$$E(C_{IncSSI}) = E(C_{WriteSkew}) = C_{Inc} \cdot P(WriteSkew) \tag{17}$$

5.9 Expected Consistency Costs

The Equations (18), (19) and (20) define the expected consistency costs of read and update SC, SSI, and 1SR transactions.

$$\begin{aligned} E(C_{SCConsRead}) &= 0 \\ E(C_{SCConsUpdate}) &= C_{StartupChecks} + C_{CommitChecks} \\ &\quad + C_{FreshnessUpdate} \end{aligned} \tag{18}$$

$$\begin{aligned} E(C_{SSIConsRead}) &= E(C_{TotalSynch}) \\ E(C_{SSIConsUpdate}) &= C_{StartupChecks} + E(C_{Locking}) \\ &\quad + E(C_{TotalSynch}) + C_{CommitChecks} \\ &\quad + E(C_{2PC}) + C_{FreshnessUpdate} \end{aligned} \tag{19}$$

$$\begin{aligned} E(C_{1SRConsRead}) &= C_{StartupChecks} + E(C_{Locking}) \\ &\quad + E(C_{TotalSynch}) \\ E(C_{1SRConsUpdate}) &= C_{StartupChecks} + E(C_{Locking}) \\ &\quad + E(C_{TotalSynch}) + C_{CommitChecks} \\ &\quad + E(C_{2PC}) + C_{FreshnessUpdate} \end{aligned} \tag{20}$$

In case of SC read-only transactions no consistency costs are generated. For SSI only the expected synchronization costs contribute to the total expected consistency costs, whereas for 1SR the costs for the startup checks and read-locks are also taken into account.

For the calculation of the expected consistency costs in case of update transactions, additional cost components must be taken into account. These cost components can be easily derived from Figures 1 and 2.

$C_{StartupChecks}$, $C_{CommitChecks}$ and $C_{FreshnessUpdate}$ are calculated using the usual pricing schema by charging for the service call and the data transferred into or out of the service. Usually $C_{StartupChecks}$ is lower than $C_{CommitChecks}$ since startup checks are only done on the local node.

5.10 Expected Overall Costs

In the Equations (21), (22) and (23) the overall costs of read and update SC, SSI and 1SR transactions are specified. The overall costs consist of the consisteny and inconsistency costs.

$$\begin{aligned} E(C_{SCOverallRead}) &= E(C_{IncSC}) \\ E(C_{SCOverallUpdate}) &= E(C_{SCConsUpdate}) + E(C_{IncSC}) \end{aligned} \tag{21}$$

$$E(C_{SSIOverallRead}) = E(C_{SSIConsRead})$$
$$E(C_{SSIOverallUpdate}) = E(C_{SSIConsUpdate}) + E(C_{WriteSkew}) \quad (22)$$

$$E(C_{1SROverallRead}) = E(C_{1SRConsRead})$$
$$E(C_{1SROverallUpdate}) = E(C_{1SRConsUpdate}) \quad (23)$$

The expected overall costs for SC read transactions are the same as the expected inconsistency costs. For SC and SSI update transactions in addition to the consistency costs the inconsistency costs have to be taken into account. For 1SR transactions the expected overall costs remain the same as the expected consistency costs, since no inconsistencies can be observed or generated by 1SR transactions.

Inconsistency costs are not related to the Cloud resources consumed, but are application-specific costs for compensating the effects of an inconsistency.

6 Cost-based Concurrency Control

As already stated, the goal of our C^3 approach is to dynamically adjust the consistency level 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. In the ideal case, 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 introduced in Section 5 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 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 6.1). The cost parameters are always specified per transaction execution, and should not be seen as a kind of a 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. The architecture of C^3 is component-based, i.e., all CCPs are implemented as components consisting of the functionality and the cost model. It means, the framework can be easily extended to take into account additional CCPs by just introducing the corresponding CCP components into C^3. Even different CCPs 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/deactive 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 rules specified below. Let us assume that in our system there are two active CCP components providing 1SR: one based on 2PC for replica commitment and

Parameter	Definition
CB_{Trx}	Consistency budget for a transaction execution
C_{Inc}	Inconsistency costs
$Coststrategy$	[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).

Table 10: C^3 Parameters

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 below. 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.

6.1 C^3 Rules

Table 10 summarizes the parameters used in the rules defining C^3. 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 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 0 or 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, than the system "switches off" the adaptive component and behaves as a traditional database by enforcing 1SR. The same applies for SSI and SC. Consistency mixes are still possible, if for example some transactions are forced to run 1SR, whereas some others SSI or SC.

Rule 2: Preconditions: The consistency budget is specified, whereas the inconsistency costs are set to 0 or 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 consistency 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.

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

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.

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

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.

7 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.

7.1 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 one EC2 Compute Unit (one virtual core with one EC2 Compute Unit); *EC2 server machines*, which host the services and are equipped with five EC2 Compute Units (two 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 container on the same EC2 server machine, each having its own local datastore, whereas all the centralized components/services (see Section 4.1) 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.

7.2 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 6). It means that adaptive transactions are executed based on Rule 3 (Section 6). The parameters as specified in Section 7.1 are also used in this experiment; additionally we have set the inconsistency costs to 1. 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 7.2. 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 4.4). 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 7.2. On the other hand, transactions running with fixed SC will observe an increasing number of inconsistencies 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 7.2). Figure 5 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.

7.3 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 6 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 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 7 it can be seen that in addition to the consistency costs, also the response time of adaptive transactions will increase with increasing inconsistency costs.

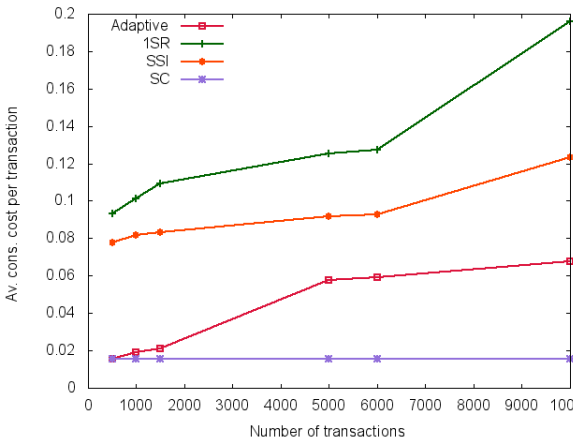

Figure 3: Consistency costs of the different CCPs

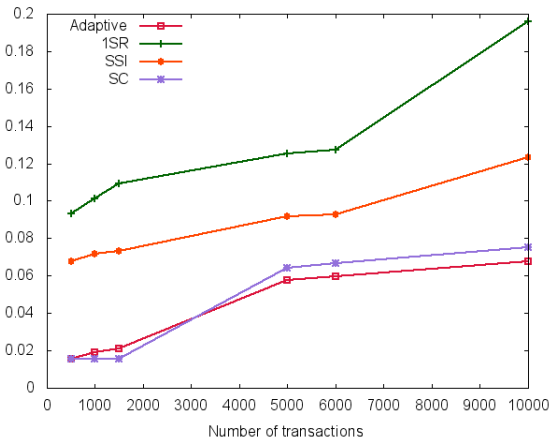

Figure 4: Overall costs

8 Conclusion

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. The main reason for 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 an exact model for the calculation of the costs incurring for different CCPs, which is independent of a specific Cloud provider. In order to save costs, a Cloud transactional middleware should provide C^3 semantics. 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 payed 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.

Acknowledgment

This work was supported by the Swiss National Science Foundation (SNF) in the context of the project GriDMan, contract No. 200021_132201 / 1.

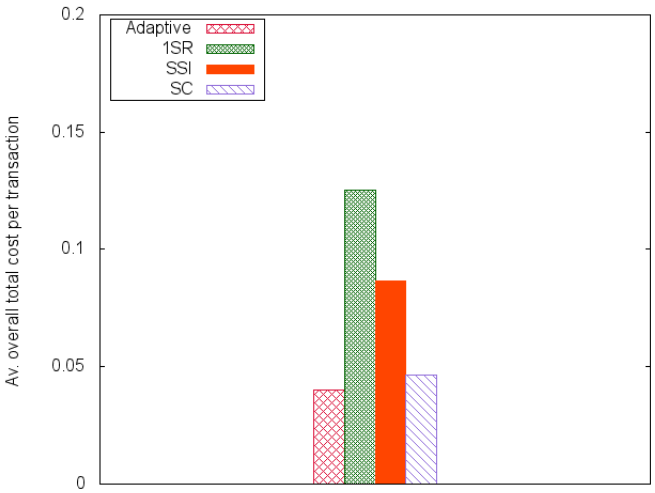

Figure 5: Total average costs over all tests

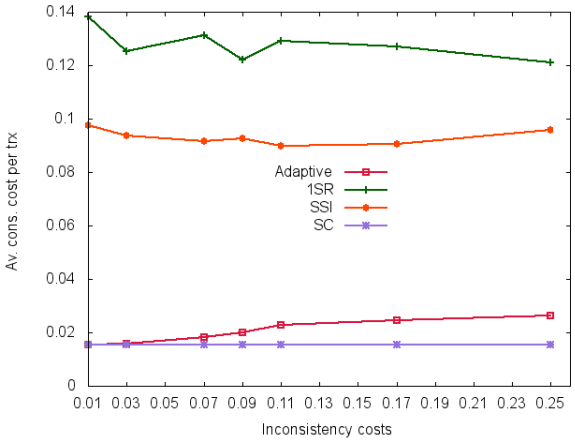

Figure 6: Consistency costs of the different CCPs

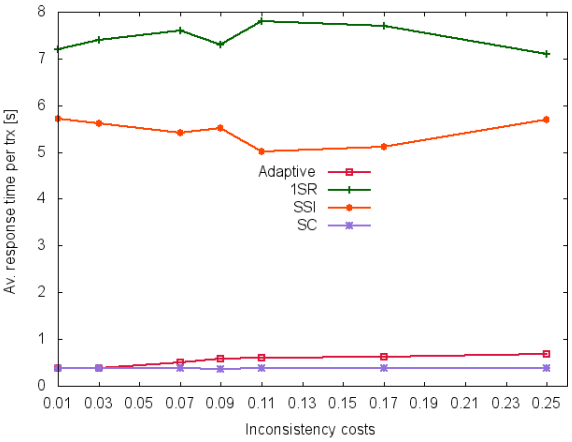

Figure 7: Response time of the different CCPs

References

[1] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a Database on S3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 251–264, Vancouver, Canada, 2008. ACM.

[2] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: a Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*, pages 163–174, Indianapolis, IN, USA, 2010. ACM.

[3] Donald Kossmann and Tim 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] Fan Yang, Jayavel Shanmugasundaram, and Ramana Yerneni. A Scalable Data Platform for a Large Number of Small Applications. In *Proc. CIDR'09*, Asilomar, CA, USA, 2009.

[6] Eric A. Brewer. Towards Robust Distributed Systems (abstract). In *Proc. PODC'2000)*, page 7, Portland, OR, USA, July 2000.

[7] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the Cloud: Pay only when it Matters. *PVLDB*, 2:253–264, August 2009.

[8] Bettina Kemme and Gustavo Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Proc. VLDB'2000*, pages 134–143, Cairo, Egypt, 2000. Morgan Kaufmann.

[9] Bettina Kemme and Gustavo Alonso. Database Replication: a Tale of Research across Communities. *PVLDB*, 3(1):5–12, 2010.

[10] Yijun Lu, Ying Lu, and Hong Jiang. Adaptive Consistency Guarantees for Large-Scale Replicated Services. In *Proceedings of the International Conference on Networking, Architecture, and Storage (NAS 2008)*, pages 89 –96, June 2008.

[11] Heiko Schuldt. Process Locking: A Protocol based on Ordered Shared Locks for the Execution of Transactional Processes. In *Proc. PODS'01*, pages 289–300, Santa Barbara, USA, 2001. ACM Press.

[12] Philip A. Bernstein, Vassos Hadzilakos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL Isolation Levels. *SIGMOD Record*, 24:1–10, May 1995.

[14] Khuzaima Daudjee and Kenneth Salem. Lazy Database Replication with Snapshot Isolation. In *Proc. VLDB'06*, pages 715–726, Seoul, Korea, 2006.

[15] Philip Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan-Kaufmann, 2009.

[16] Sameh Elnikety. Database Replication using Generalized Snapshot Isolation. In *Proc. SRDS'05*, pages 73–84, Orlando, FL, USA, 2005.

[17] Werner Vogels. Eventually consistent. `http://www.allthingsdistributed.com/2007/12/eventually_consistent.html/`, 2007.

[18] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, New York, NY, USA, 2008. ACM.