Geo-Replication: Fast If Possible, Consistent If Necessary*

Valter Balegas[1], Cheng Li[2], Mahsa Najafzadeh[4], Daniel Porto[3], Allen Clement[2,5], Srgio Duarte[1], Carla Ferreira[1], Johannes Gehrke[6], João Leitão[1], Nuno Preguia[1], Rodrigo Rodrigues[3], Marc Shapiro[4], Viktor Vafeiadis[2]

[1]NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa [2]Max Planck Institute for Software Systems (MPI-SWS)

[3]INESC-ID / IST, University of Lisbon [4]Inria & Sorbonne Universits, UPMC Univ Paris 06, LIP6

[5]Currently at Google [6]Microsoft and Cornell University

Abstract

Geo-replicated storage systems are at the core of current Internet services. Unfortunately, there exists a fundamental tension between consistency and performance for offering scalable geo-replication. Weakening consistency semantics leads to less coordination and consequently a good user experience, but it may introduce anomalies such as state divergence and invariant violation. In contrast, maintaining stronger consistency precludes anomalies but requires more coordination. This paper discusses two main contributions to address this tension. First, RedBlue Consistency enables blue operations to be fast (and weakly consistent) while the remaining red operations are strongly consistent (and slow). We identify sufficient conditions for determining when operations can be blue or must be red. Second, Explicit Consistency further increases the space of operations that can be fast by restricting the concurrent execution of only the operations that can break application-defined invariants. We further show how to allow operations to complete locally in the common case, by relying on a reservation system that moves coordination off the critical path of operation execution.

1 Introduction

A geo-replicated system maintains copies of the service state across geographically dispersed locations. Geo-replication is not only employed today by virtually all the providers of major Internet services, who typically manage several data centers spread across the globe, but is also accessible to anyone outsourcing their computational needs to cloud providers, since cloud services allow computations or VMs to be instantiated in different data centers.

There are two main reasons for deploying geo-replicated systems. The first reason is disaster tolerance, i.e., the ability to tolerate the unplanned outage of an entire data center, due to catastrophic events such as natural disasters [1]. The second reason is to reduce the latency between the users and the machines that provide the service. The importance of this aspect is demonstrated by several studies that point out an inverse correlation between response times and user satisfaction for important Internet services such as search [30].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Student author are followed by faculty author names, both in alphabetical order. Cheng Li and Valter Balegas are the lead authors of the work.

However, there is a fundamental tension between latency and consistency: intuitively, ensuring strong consistency requires coordination between replicas before returning a reply to the user, while, alternatively, a fast response can be given without replica coordination, but only ensuring weak consistency guarantees. This tension has led the providers of global-scale Internet services to choose, for some parts of their services, storage systems offering weak consistency guarantees such as eventual consistency [13], and, for other components, systems with strong consistency such as serializability [10].

This paper revisits in a unified way two of our recent results in trying to achieve a balance between performance and consistency, by devising methods to build geo-replicated systems that introduce a small amount of coordination between replicas to achieve the desired semantics, i.e., systems that are fast when possible and consistent when necessary [21, 7]. In the first result, we improve the performance of geo-replicated systems by (1) allowing different operations to execute in either a weakly consistent (fast) or strongly consistent (slow) manner; and (2) identifying a set of principles for making safe use and increasing the space of fast operations. In the second result, we further increase the space of operations that can execute fast by (1) identifying the operations that can break application invariants when executing concurrently; and (2) deploying concurrency control mechanisms that remove coordination from the critical path of operation execution, while preserving invariants.

We start the presentation by laying out our terminology and system model in Section 2. We present an initial approach based on a coarse-grained classification into strong and weak consistency in Section 3. One key aspect of this approach is operation commutativity, and we explain how to achieve it using CRDTs in Section 4. Then we present an approach that makes use of fine-grained coordination between pairs of operations in Section 5. We discuss related work in section 6 and conclude in Section 7.

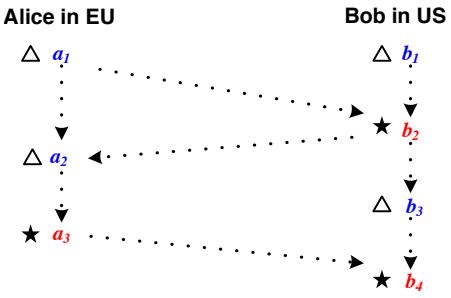
2 System model

Our system model is that of a fully replicated distributed system, where replicas are located in different data centers. Each replica follows a deterministic state machine: there is a set of operations $\mathcal{U}$, which manipulate a set of reachable system states $\mathcal{S}$. Each operation u is initially submitted at a given replica (preferably in the closest data center), which we call the *origin replica* of u. When the remaining replicas receive a request to replicate this operation, they will apply the operation against their local state.

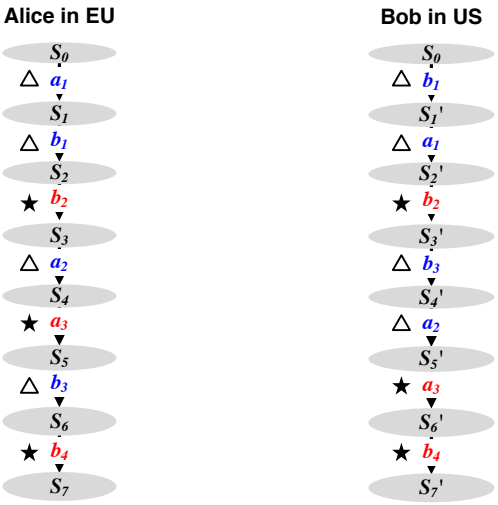
Throughout our explanation we will highlight two important properties that the replicated system should obey. First, there is the **state convergence** property, which says that all the sites that have executed the same set of operations against the same initial state are in the same final state. This is important to prevent a situation where the system quiesces (no more updates are received) and read-only queries return different results depending on which sites the users are connected to. The second property is to **preserve application-specific invariants**, which comprise a specification for the behavior of the system. To define these, we introduce the primitive *valid*(S) to be *true* if state S obeys these invariants and *false* otherwise.

3 Mixing consistency levels in RedBlue consistency

In this section, we present a hybrid consistency model called RedBlue consistency, where weakly consistent operations, labeled blue, can be executed at a single replica and propagated in the background, with mostly no coordination with concurrent actions at other replicas, while others, labeled red, require a stronger consistency level and thus require cross-replica coordination. RedBlue consistency is one of several systems that propose labeling operations according to their consistency levels [18, 33, 21, 36], but improves on these systems by offering a precise method for labeling operations.



(a) RedBlue order O of operations



(b) Serializations of O

Figure 1: RedBlue order and serializations for a system spanning two sites. Operations marked with ★ are red, and operations marked with △ are blue. Dotted arrows in (a) indicate the partial ordering of operations.

3.1 Defining RedBlue consistency

RedBlue consistency relies on three components: (1) a partitioning of operations into weakly consistent blue operations whose order of execution can vary from site to site, and red operations that must be executed in the same order at all sites, (2) a RedBlue order, which defines a partial order of operations where red operations have to be ordered with respect to each other, and (3) a set of site-specific serializations (i.e., total orders) in which the operations are locally applied. More precisely:

Definition 1 (RedBlue consistency): A replicated system is *RedBlue consistent* if each site i applies operations according to a linear extension of a RedBlue order O of the operations that were invoked, where O is a partial order among those operations with the requirement that red operations are totally ordered in O.

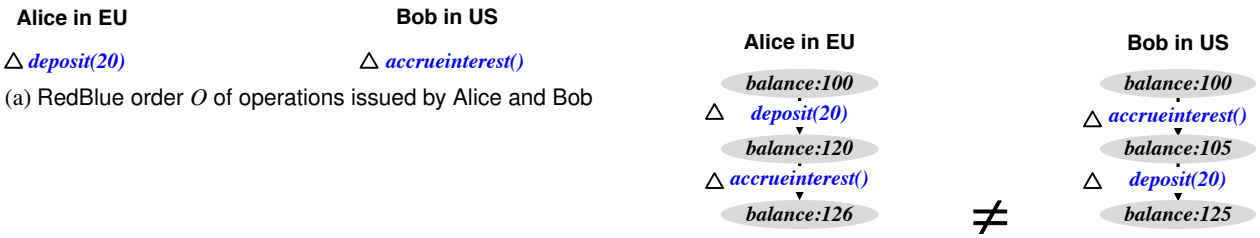
Figure 1 shows a RedBlue order and two serializations, i.e., the linear extensions of that order in which operations are applied at two different sites. In systems where every operation is labeled red, RedBlue consistency is equivalent to serializability [10]; in systems where every operation is labeled blue, RedBlue consistency becomes a form of causal consistency [37, 23, 25], since the partial order conveys the necessary causality between operations.

When applying RedBlue consistency to an application, we would like to label all operations blue to obtain best performance. However, this could lead to state divergence and invariant violation, when operations are not commutative. We describe a set of sufficient conditions to guide the classification of operations in order to safely use weak consistency when possible.

3.2 Ensuring state convergence

In the context of RedBlue consistency, we can formalize state convergence as follows:

Definition 2 (State convergence): A RedBlue consistent system is state convergent if all serializations of the underlying RedBlue order O reach the same state S w.r.t. any initial state S_0.



(a) RedBlue order O of operations issued by Alice and Bob

(b) Serializations of O leading to diverged state

Figure 2: A RedBlue consistent account with initial balance of 100 and final diverged state

.

To find a correct labeling for maintaining state convergence while providing low latency access, we describe a simple banking example, in which users may share an account that is modified via three operations, namely `deposit`, `withdraw` and `accrueinterest`[1]. Keeping in mind that one of the goals of RedBlue consistency is to make the target service as fast as possible, we tentatively label all these operations blue. According to this labeling result, we construct a RedBlue order of deposits and interest accruals made by two users Alice and Bob and two possible serializations applied at both branches of the bank, as shown in Figure 2. This example shows that the labeling of these operations as described is not state convergent. This is because RedBlue consistency allows the two sites to execute blue operations in a different order, but two of the blue operations in the example are non-commutative, namely `deposit` and `accrueinterest`. To prevent this situation, a sufficient condition to guarantee state convergence in a system supporting RedBlue consistency is that every blue operation commutes with all other operations, blue or red.

However, when applying this condition to the banking example, it implies that we need to label all three operations red (`deposit`, `withdraw` and `accrueinterest`). This is equivalent to running the system under serializability, which requires coordination across replicas for executing all these operations. To address the problem that it is difficult to find operations that commute with all other operations in the system, we observe that, in many cases, while operations may not be commutative, we can make the changes they induce on the system state to commute. In the banking example, we can engineer `accrueinterest` commute with the remaining two operations by first computing the amount of interested accrued at the primary replica and then treating that value as a deposit.

To exploit this observation and increase operation commutativity, we propose a change to our original system model, where we split each original application operation u into two components: a *generator operation* g_u with no side-effects, which is executed only at the primary site against some system state S and produces a *shadow operation* $h_u(S)$, which is executed at every site (including the primary site). The generator operation decides which state transitions should be made while the shadow operation applies the transitions in a state-independent manner.

3.3 Preserving invariants

Although the concept of shadow operation helps produce more commutative operations, labeling too many shadow operations as blue may introduce the problem of breaking application invariants. In the banking example, assuming that the shared bank account has an initial balance of 100, if both Alice and Bob withdraw 70 and 60 respectively, the final balance would be -30. This violates the invariant that a bank balance should never be negative. To determine which shadow operations can be safely labeled blue, we begin by defining that a shadow operation is invariant safe if, when applied to a valid state, it always transitions the system into another valid state. This allows us to define the following sufficient condition: a RedBlue consistent system preserves

[1] `accrueinterest` computes a new balance by multiplying the old balance value and (1 + `interest rate`).

invariants (meaning that all its sites are always in valid states) if all shadow operations that are not invariant safe are labeled red (i.e., strongly consistent).

3.4 What can be blue? What must be red?

In summary, the two conditions above lead to the following procedure for deciding which shadow operations can be blue or must be red if a RedBlue consistent system is to provide both state convergence and invariant preservation:

1. For any pair of non-commutative shadow operations h_u and h_v, label both h_u and h_v red.
2. For any shadow operation h_u that is not invariant safe, label h_u red.
3. Label all remaining shadow operations blue.

4 State convergence

In the previous section we discussed how RedBlue consistency achieves state convergence by relying on shadow operations that commute with each other. With this approach, defining a new operation also implies writing one or more commutative shadow operations, each of which corresponds to a distinct side effect. The major challenge of doing this manual work is that, in an application with a large number of operations, this process may be complex and error-prone.

We now discuss an alternative principled approach to create commutative operations by design. Our approach builds on conflict-free replicated data types (CRDTs) [31], which are specially-designed data structures that can be replicated and modified concurrently, and include mechanisms to merge concurrent updates in a deterministic way. Application operations consist of updates to these elementary data types, thus guaranteeing state convergence.

4.1 CRDTs

A CRDT is a data type that can be replicated at multiple replicas. As such, it defines an interface with a set of operations to read and to modify its state. A CRDT replica can be modified by locally executing an update operation. When different replicas of the same object are modified concurrently, they temporarily diverge. CRDTs have built-in support for achieving strong eventual consistency [31], in which all replicas will eventually reach the same (equivalent) state after applying the same set of updates, without relying on a distributed conflict arbitration process.

Two main flavors of CRDTs have been studied in the literature: operation-based CRDTs and state-based CRDTs. For each of these, sufficient conditions for achieving strong eventual consistency have been established.

In operation-based CRDTs (or commutative replicated data types), updates are propagated by broadcasting operations to every replica in causal order. Interestingly, this proposal matches the operation execution decomposition presented in RedBlue consistency (Section 3), where operations are divided in two components, a *generator* operation that executes in the local replica, has no side effect and produces a *shadow* operation, which is propagated and executed in all replicas. The two types of operations are analogous to *prepare* and *downstream* operations in the context of operation-based CRDTs, respectively, with the main difference that shadow operations are assigned a consistency level in RedBlue consistency. Similarly to the consequence of commutative shadow operations, the replicas of an operation-based CRDT converge to the same state after executing the same set of updates (in any order that respects causality) if the execution of any two concurrent downstream operations commutes [31].

SQL type	CRDT	Description
FIELD*	LWW	Use last-writer-wins to solve concurrent updates
	NUMDELTA	Add a delta to the numeric value
TABLE	AOSET, UOSET, AUSET, ARSET	Sets with restricted operations (add, update, and/or remove). Conflicting operations are logically executed by timestamp order.

Table 1: Commutative replicated data types (CRDTs) for relational data. * FIELD covers primitive types such as integer, float, double, datetime and string.

A state-based CRDT (or convergent replicated data type) defines, in addition to the operations to read and update its state, an operation to merge the state of two replicas. Replicas synchronize by exchanging the full replica states: when a new state is received, the new updates are incorporated in the local replica by executing the merge function. It has been shown that the replicas of a state-based CRDT converge to the same state after all replicas synchronize (directly or indirectly) if: (1) all the possible states of an object are partially ordered, forming a join-semilattice; (2) the merge operation between two states is the semilattice join; and (3) an update monotonically increases the state according to the defined partial order [31].

4.2 Examples

CRDTs have been used in a number of research systems, such as Walter [35] and SwiftCloud [39], and commercial systems, such as Riak [2] and SoundCloud [3]. These systems include CRDTs that implement several data types, such as registers, counters, sets, maps, and flags. For each such data type, it is possible to define and implement different semantics to handle concurrent updates, leading to different CRDTs. These semantics define which is the final state of a CRDT when concurrent updates occur. For example, for sets, it is possible to define an *add-wins* semantics, where, in the presence of a concurrent add and remove of some element *e*, the final state will contain *e* (or, more precisely, there exists an add of *e* that does not happen before a remove of *e*). It is also possible to define a *remove-wins* semantics, where the remove will win over a concurrent add. Other semantics can also be implemented, such as a *last-writer-wins* strategy where an element will belong to the set or not depending on which was the last operation executed, according to the order among operations.

When creating an application, an application developer must select the CRDT with the most appropriate semantics for its goal. For example, in the bank account example, the balance of an account can be modeled as a counter and the set of accounts of a client can be maintained in an add-win set or map CRDT.

In general, an application operation will manipulate multiple data objects. When using CRDTs, it is possible to maintain replicas of these objects in multiple nodes. An operation can execute by accessing a single replica of each object it accesses. These updates can later be propagated to other nodes, with CRDT rules guaranteeing that the replicas of each object will converge to the same state. By propagating the updates to all objects modified in an operation atomically, it is possible to guarantee that all effects of an operation are observed at the same time.

CRDTs for relational databases In relational databases, it is also possible to model data using CRDTs. Table 1 presents the mapping proposed in SIEVE [20]. Regarding table fields, we defined only two CRDTs. The *LWW* CRDT can be used with any field type and implements a *last-writer-wins* strategy for defining the final value of a field. The *NUMDELTA* CRDT can be used with numeric fields, and transforms each update operation in a downstream operation that adds or subtracts a constant to the value of the field. This can be used to support account balances, counters, etc.

A database table can be seen as a set of tuples. In the general case, and following the semantics of the *ARSET* CRDT, when concurrent *insert*, *update* and *delete* operations occur, the following rules can be used: (1) concurrent inserts of tuples with the same key are treated as an insert followed by a sequence of updates;

(2) for concurrent updates, the rules defined for fields are used to deterministically define the final value; (3) a delete will only take effect if no concurrent update or insert was executed.

While using CRDTs guarantees that all replicas converge to the same state, it does not guarantee that the convergence rules executed independently by different CRDTs maintain application invariants. Next, we show how we can address this problem by restricting the concurrent execution of operations that can break application invariants.

5 Preserving invariants with minimal coordination

As mentioned before, in the banking example, the `withdraw` operation, despite being commutative, cannot execute under weak consistency, as the concurrent execution of multiple withdrawals can break the invariant that the account balance cannot be negative. To avoid the possibility of breaking the invariant, RedBlue consistency would label all withdrawals as red, requiring replicas to coordinate the execution of every withdraw operation. In practice, however, only in a few cases the cumulative effects of all concurrent withdrawals will surpass the actual balance of the account.

To relieve the strong constraint imposed by RedBlue consistency, we propose a more efficient coordination plan: given some account balance, replicas can coordinate beforehand to split the balance among them. Until a replica consumes its allocated share of the balance, it can execute operations locally, without coordination with other replicas, with the guarantee that the balance will not become negative, i.e., the application invariant will not be broken.

The above idea has been previously explored in the context of escrow transactions [9, 27]. We revisit and generalize the concept of escrow transactions, to allow replicas to assess the safety of operations without coordination when executing operations. In our generalization, when replicas cannot ensure an operation is safe by reading local state, they contact remote peers to update their vision of the database to decide the fate of the operation. In addition, we discuss how we avoid the coordination across sites for all red operations, which is required for totally ordering them. Instead, we identify a small set of coordination requirements between operations, and show how to enforce those rules at runtime.

5.1 Explicit Consistency in a nutshell

We present a new consistency model, called Explicit Consistency, that extends RedBlue consistency to avoid the coordination of red operations when possible. The idea is that instead of labeling shadow operations as red or blue, programmers specify the application invariants. The system must execute operations while guaranteeing that these invariants are not broken.

To this end, we propose the following methodology for creating applications that adhere to Explicit Consistency. First, programmers must specify the application invariants and operation effects. Second, we provide a tool to analyze the specification of the application and identify the pairs of conflicting shadow operations. Non-conflicting shadow operations execute without any restrictions, as blue operations. We include a library of CRDTs to help programmers define commutative operations. Third, for each pair of conflicting shadow operations, the programmer can use a specialized concurrency control mechanism that restricts the concurrent execution of these operations. This mechanism executes coordination outside of the critical path of operation execution, allowing these operations to execute locally without the need to coordinate with other replicas.

The following sections provide additional details on these steps to use the Explicit Consistency model.

5.2 Application specification

Programmers specify application invariants and the post-conditions of shadow operations as first order logic expressions. Invariants must be written as universally quantified formulas in prenex normal form, while the

```
01: //Invariant declaration
02: @Invariant( "forall(aId : Aid) :- balance(aId) >= 0" )
03: @Invariant( "forall(cId : Cid, aId : Aid) :- userAccount(cId, aId) =>
04:                                              registeredUser(cId)")
05: public interface Bank{
06:
07:   @decrement(balance(accountId), amount)
08:   boolean withdraw( CId clientId, Aid accountId, Int amount);
09:
10:   @true(userAccount(clientId, accountId))
11:   boolean assignAccount( CId clientId, AId accountId);
12:
13:   @false(userAccount(clientId, accountId))
14:   boolean closeAccount( CId clientId, AId accountId);
16:
17:   @false(registeredUser(clientId))
18:   boolean endContract( CId clientId);
19:
20: }
```

(a) Specification written with Java Annotations.

Operations X *Operations*		*Conflict*
withdraw(cId, aId, amount)	withdraw(cId, aId, amount)	**Non-idempotent**
...		
assignAccount(cId,aId)	closeAccount(cId, aId)	**Opposing**
...		
assignAccount(cId,aId)	endContract(cId)	**Conflict**
...		

(b) Conflicting pairs of operations for the Bank example.

Figure 3: Bank application specification and analysis results.

grammar for specifying applications post-conditions is restricted to predicate assignments, that assert the truth value of some predicate, and function clauses, which define the relation between the value of some predicate before and after the execution of the operation.

The code snippet in figure 3(a) shows the specification of the banking application. We extended this example to illustrate different invariant violations. In the extended version, clients must have a valid contract with the bank to be able to access an account. Clients might have multiple accounts and must close all of them before finishing the contract. In Line 2, the invariant guarantees that an account balance is never negative. In line 3, the invariant states that, for every open account, the account holder must be registered with the bank.

5.3 Analysis

The analysis checks which are the shadow operations whose concurrent execution might produce a database state that is invalid with respect to the declared invariants. Conceptually, for each pair of operations and for every valid state where these operation can execute, the algorithm verifies if the execution of both operations will lead to a state that is not valid according to the invariants of the application. Obviously, checking every pair of operations in every valid state exhaustively is unfeasible. Instead, our algorithm relies in the Z3 satisfiability modulo theory (SMT) solver to perform this verification efficiently. A full description of the algorithm is given in our prior publication [7].

Figure 3(b) summarizes the conflicts in the example of Figure 3(a): two concurrent successful withdrawals might make the balance negative (non-idempotence); assigning and removing an account concurrently for the same user might leave the system in an inconsistent state, because each shadow operation writes different values for the predicate *userAccount*(*cId*, *aId*) (opposing post-conditions); and finally, the pair *createAccount*(*cId*, *aId*) and *endContract*(*cId*) might violate the integrity constraint of line 3, because a new account is being added to a user that is ending a contract with the bank.

5.4 Code instrumentation

After identifying which operations can lead to conflicts, the programmer must instrument the application to avoid them.

Some conflicts can be handled by simply relying on CRDTs to automatically solve them. For example,

our analysis can report that operations have opposing post-conditions: e.g., operations `assignAccout` and `remAccount` assign the value *true* and *false* to predicate $userAccount(cId, aId)$. In this situation, the programmer can choose a preferred value for the predicate and use a CRDT that automatically implements the selected decision[2].

Other conflicts must be handled by restricting the concurrent execution of operations that can cause invariants to be broken. To this end, we provide a set of specialized reservation-based concurrency control mechanisms.

For conflicts on numeric invariants, like the one that withdraw causes, we support an escrow reservation for allowing some decrements of numeric values to execute without coordination. In an escrow reservation, each replica is assigned a budget of decrements, based on the initial value of the data. In our example, when a replica receives a withdraw request, if the local budget is sufficient, the generator operation executes immediately without coordination, generating a shadow operation that decrements the balance. This local execution is safe, guaranteeing that the invariant still holds after executing all concurrent operations, because the sum of the budgets of all replicas is equal to the value of the initial value. If the local budget is not enough to satisfy the request, the replica needs to contact remote replicas to increase its budget, until it can satisfy the request. If that is not possible, because there are not enough resources globally, then the generator operation fails, generating no shadow operation.

For conflicts on generic invariants, we include a multi-value lock reservation. This lock can be in one of the following three states: (1) shared forbid, giving the shared right to forbid some action to occur; (2) shared allow, giving the shared right to allow some action to occur; (3) exclusive allow, giving the exclusive right to execute some action. The idea is that, for a conflicting pair of operations, (o_1, o_2), the lock will be associated with the execution of one of the operations, say o_1. To execute o_1, a replica must hold the lock in the shared allow mode. This right can be shared by multiple replicas. To execute o_2, a replica must hold the lock in the shared forbid mode. As before, when executing the generator operation, if the replica already holds the necessary locks (in the required mode to execute the operation), it can execute locally and generate the corresponding shadow operation. If not, it must contact other replicas to obtain the necessary locks.

Besides these two locks, we also proposed other locks that can efficiently restrict the concurrent execution of operations that conflict in other types of invariants, including conditions on the number of elements that satisfy a given condition and disjunctions. In a related work, Gotsman et. al. [16] have shown how to prove that a given set of locks is sufficient for maintaining invariants.

6 Related work

Many cloud storage systems supporting geo-replication have emerged in recent years. Some of these systems offer variants of eventual consistency, where operations produce responses right after being executed in a single data center (usually the closest one) and are replicated in the background, so that user observed latency is improved [13, 23, 24, 4, 19]. These variants target different requirements, such as: reading a causally consistent view of the database (causal consistency) [23, 4, 14, 6]; supporting limited transactions where a set of updates are made visible atomically [24, 5]; supporting application-specific or type-specific reconciliation with no lost updates [13, 23, 35, 2], etc.

While some systems implement eventual consistency by relying on a simple last-writer-wins strategy, others have explored the semantics of applications (and data types). Semantic types [15] have been used for building non-serializable schedules that preserve consistency in distributed databases. Conflict-free replicated data types [31] explore commutativity for enabling the automatic merge of concurrent updates to the same data types.

Eventual consistency is insufficient for some applications that require some operations to execute under strong consistency for correctness. To this end, several systems support strong consistency. Spanner provides

[2]In our experience, boolean predicates can be implemented using Set CRDTs with add-wins and remove-wins policies to enforce that the corresponding predicate becomes true or false respectively.

strong consistency for the whole database, at the cost of incurring coordination overhead for all updates [12]. Transaction chains support transaction serializability with latency proportional to the latency to the first replica that the corresponding transaction accesses [40]. MDCC [17] and Replicated Commit [26] propose optimized approaches for executing transactions but still incur inter-data center latency for committing transactions.

Some systems combine the benefits of weak and strong consistency models by allowing both levels to coexist. In Walter [35], transactions that can execute under weak consistency run fast, without needing to coordinate with other datacenters. Bayou [37] and Pileus [36] allow operations to read data with different consistency levels, from strong to eventual consistency. PNUTS [11] and DynamoDB [34] also combine weak consistency with per-object strong consistency relying on conditional writes, where a write fails in the presence of concurrent writes. RedBlue consistency also combines weak and strong consistency in the same system. Unlike other systems, RedBlue consistency splits operations into generator and shadow parts to allow more operations to commute, and define a procedure to help programmers labeling shadow operations as weak or strong.

Escrow transactions [27] offer a mechanism for enforcing numeric invariants under concurrent execution of transactions. By enforcing local invariants in each transaction, they can guarantee that a global invariant is not broken. This idea can be applied to other data types, and it has been explored for supporting disconnected operation in mobile computing [38, 28, 32]. Balegas et al. [8] proposed the bounded counter CRDT that can be used to enforce numeric invariants in weakly consistent cloud databases. The demarcation protocol [9] aims at maintaining invariants in distributed databases. Although its underlying protocols are similar to escrow-based approaches, it focuses on maintaining invariants across different objects. Warranties [22] provide time-limited assertions over the database state, which can improve latency of read operations in cloud storages. Indigo builds on similar ideas for enforcing application invariants, but it is the first piece of work to provide an approach that, starting from application invariants expressed in first-order logic, leads to the deployment of the appropriate techniques for enforcing such invariants in a geo-replicated weakly consistent data store. Gotsman et. al. [16] propose a proof rule for establishing that the use of a given set of techniques is sufficient to ensure the preservation of invariants.

The static analysis of code is a standard technique used extensively for various purposes, including in a context similar to ours. SIEVE [20] combines static and dynamic analysis to infer which operations should use strong consistency and which operations should use weak consistency in a RedBlue system [21]. Roy et al. [29] present an analysis algorithm that describes the semantics of transactions. These works are complementary to ours, since the proposed techniques could be used to automatically infer application side effects.

7 Conclusion

In this paper we summarized two of our recent results in addressing the fundamental tension between latency and consistency in geo-replicated systems. First, RedBlue consistency [21] offers fast geo-replication by presenting sufficient conditions that allow programmers to safely separate weakly consistent (fast) operations from strongly consistent (slow) ones in a coarse-grained manner. To increase the space of potential fast operations and simplify the programmer's task of defining commutative operations, we propose the use of conflict-free replicated data types. Second, Explicit Consistency [7] enables programmers to make fine-grained decisions on consistency level assignments by connecting application invariants to ordering conflicts between pairs of operations, and explores efficient reservation techniques for coordinating conflicting operations with low cost.

Acknowledgments

The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC Starting Grant. This research was also supported in part by EU FP7 SyncFree project (609551), FCT/MCT SFRH/BD/87540/2012, PEst-OE/ EEI/ UI0527/ 2014, NOVA LINCS (UID/CEC/04516/2013), and INESC-ID (UID/CEC/50021/2013).

References

[1] 7 Data Center Disasters You'll Never See Coming. http://www.informationweek.com/cloud/7-data-center-disasters-youll-never-see-coming/d/d-id/1320702. Accessed Feb-2016.

[2] Using data types – riak documentation. http://docs.basho.com/riak/latest/dev/using/data-types/. Accessed Feb-2016.

[3] Consistency without Consensus: CRDTs in Production at SoundCloud. http://www.slideshare.net/InfoQ/consistency-without-consensus-crdts-in-production-at-soundcloud Accessed Feb-2016.

[4] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *EuroSys '13*, 85–98, 2013. ACM.

[5] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable Atomic Visibility with RAMP Transactions. In *SIGMOD '14*, 27–38, 2014. ACM.

[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *SIGMOD '13*, 761–772, 2013. ACM.

[7] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *EuroSys '15*, 6:1–6:16, 2015. ACM.

[8] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguica. Extending eventually consistent cloud databases for enforcing numeric invariants. In *SRDS '15*, 31–36, Sept 2015.

[9] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-distributed Database. In *OSDI '12*, 251–264, 2012. USENIX Association.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP '07*, 205–220, 2007. ACM.

[14] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *SOCC '13*, 11:1–11:14, 2013. ACM.

[15] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.

[16] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *POPL 2016*, 371–384, 2016. ACM.

[17] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data Center Consistency. In *EuroSys '13*, 113–126, 2013. ACM.

[18] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[19] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. In *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[20] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *ATC '14*, 281–292, 2014. USENIX Association.

[21] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *OSDI '12*, 265–278, 2012. USENIX Association.

[22] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for faster strong consistency. In *NSDI '14*, 2014. USENIX Association.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *SOSP '11* , 401–416, 2011. ACM.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *NSDI '13*, 313–328, 2013. USENIX Association.

[25] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: cloud storage with minimal trust. In *OSDI*, 2010.

[26] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.*, 6(9):661–672, 2013.

[27] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.

[28] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for Conflict Avoidance in a Mobile Database System. In *MobiSys '03*, 43–56, 2003. ACM.

[29] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD '15*, 1311–1326, 2015.

[30] E. Schurman and J. Brutlag. Performance related changes and their user impact. Presented at velocity web performance and operations conference. `http://slideplayer.com/slide/1402419/`, 2009.

[31] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *SSS '11*, 386–400, 2011. Springer-Verlag.

[32] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers. In *Middleware '08*, 42–61, 2008. Springer-Verlag New York, Inc.

[33] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually Consistent Byzantine-Fault Tolerance. In *NSDI'09*, 169–184, 2009.

[34] S. Sivasubramanian. Amazon DynamoDB: A Seamlessly Scalable Non-relational Database Service. In *SIGMOD '12*, 729–730, 2012. ACM.

[35] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional Storage for Geo-replicated Systems. In *SOSP '11*, 385–400, 2011. ACM.

[36] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *SOSP '13*, 309–324, 2013. ACM.

[37] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP '95*, 172–182, 1995. ACM.

[38] G. D. Walborn and P. K. Chrysanthis. Supporting Semantics-based Transaction Processing in Mobile Database Applications. In *SRDS '95*, 31–40, 1995. IEEE Computer Society.

[39] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Middleware '15*, 75–87, 2015. ACM.

[40] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *SOSP '13*, 276–291, 2013. ACM.