Putting Consistency Back into Eventual Consistency

Valter Balegas Sérgio Duarte Carla Ferreira Rodrigo Rodrigues Nuno Preguiça
NOVA LINCS, FCT, Universidade Nova Lisboa
Mahsa Najafzadeh Marc Shapiro
Inria Paris-Rocquencourt & Sorbonne Universités, UPMC Univ Paris 06, LIP6

Abstract

Geo-replicated storage systems are at the core of current Internet services. The designers of the replication protocols used by these systems must choose between either supporting low-latency, eventually-consistent operations, or ensuring strong consistency to ease application correctness. We propose an alternative consistency model, *Explicit Consistency*, that strengthens eventual consistency with a guarantee to preserve specific invariants defined by the applications. Given these application-specific invariants, a system that supports Explicit Consistency identifies which operations would be unsafe under concurrent execution, and allows programmers to select either violation-avoidance or invariant-repair techniques. We show how to achieve the former, while allowing operations to complete locally in the common case, by relying on a *reservation* system that moves coordination off the critical path of operation execution. The latter, in turn, allows operations to execute without restriction, and restore invariants by applying a repair operation to the database state. We present the design and evaluation of Indigo, a middleware that provides Explicit Consistency on top of a causally-consistent data store. Indigo guarantees strong application invariants while providing similar latency to an eventually-consistent system in the common case.

1. Introduction

To improve user experience in services that operate on a global scale, from social networks and multi-player online games to e-commerce applications, the infrastructure that supports these services often resorts to geo-replication [9, 10, 12, 25, 27, 28, 41], i.e., maintains copies of application data and logic in multiple datacenters scattered across the globe. This ensures low latency, by routing requests to the closest datacenter, but only when the request does not require cross-datacenter synchronization. Executing update operations without cross-datacenter synchronization is normally achieved through weak consistency. The downside of weak consistency models is that applications have to deal with concurrent operations, which can lead to non-intuitive and undesirable semantics.

These semantic anomalies do not occur in systems that enforce strict serializability, i.e., serialize all operations in real-time order. Weaker models, such as serializability or snapshot isolation, relax synchronization, but still require frequent coordination among replicas, which increases latency and decreases availability. A promising alternative is to try to combine the strengths of both approaches by supporting both weak and strong consistency, depending on the operation [25, 41, 43]. In this approach, operations requiring strong consistency still incur high latency and are unavailable when the system partitions. Additionally, these systems make it harder to design applications, as operations need to be correctly classified to guarantee the correctness of the application.

In this paper, we propose *Explicit Consistency* as an alternative consistency model, in which an application specifies the invariants, or consistency rules, that the system must maintain. Unlike models defined in terms of execution orders, Explicit Consistency is defined in terms of application properties: the system is free to reorder execution of operations at different replicas, provided that the specified invariants are maintained.

In addition, we show that it is possible to implement explicit consistency while mostly avoiding cross-datacenter coordination, even for critical operations that could potentially break invariants. To this end, we propose a three-step methodology to derive a safe version of the application. First, we use static analysis to infer which operations can be safely executed without coordination. Second, for the remaining operations, we provide the programmer with a choice of invariant-repair [38] or violation-avoidance techniques. Finally, application code is instrumented with the appropriate calls to our middleware library.

Violation-avoidance extends escrow and reservation approaches [15, 17, 32, 35, 39]. The idea is that a replica coordinates in advance, to pre-allocate the permission to execute some collection of future updates, which (thanks to the reser-

EuroSys'15, April 21–24, 2015, Bordeaux, France.

http://dx.doi.org/10.1145/2741948.2741972

vation) will require no coordination. This amortizes the cost and moves it off the critical path.

Finally, we present the design of Indigo, a middleware for Explicit Consistency built on top of a geo-replicated key-value store. Indigo is designed in a way that is agnostic to the details of the underlying key-value store, only requiring it to ensure properties that are known to be efficient to implement, namely per-key, per-replica linearizability, causal consistency, and transactions with weak semantics [2, 27, 28].

In summary, we make the following contributions:

- Explicit Consistency, a new consistency model for application correctness, centered on the application semantics, and not on the order of operations.
- A methodology to derive an efficient reservation system for enforcing Explicit Consistency, based on the set of invariants associated with the application.
- Indigo, a middleware system implementing Explicit Consistency on top of a causally consistent geo-replicated key-value store.

The remainder of the paper is organized as follows: Section 2 introduces Explicit Consistency. Section 3 gives an overview of our approach. Section 4 presents the analysis for detecting unsafe concurrent operations. Section 5 details the techniques for handling these operations. Section 6 discusses the implementation of Indigo and Section 7 presents an evaluation of the system. Related work is discussed in Section 8. Finally, Section 9 concludes the paper.

2. Explicit Consistency

In this section we define precisely the consistency guarantees that Indigo provides. We start by defining the system model, and then how Explicit Consistency restricts the set of behaviors allowed by that model.

To illustrate the concepts, we use as running example the management of tournaments in a distributed multi-player game. The game maintains information about players and tournaments. Players can register and de-register from the game. Players compete in tournaments, for which they can enroll and disenroll. A set of matches occurs for each tournament. Each tournament has a maximum capacity. In some cases, e.g., when there are not enough participants, a tournament can be canceled before it starts. Otherwise a tournament's life cycle is creation, start, and end.

2.1 System Model and Definitions

We consider a database composed of a set of objects in a typical cloud deployment, where data is fully replicated in multiple datacenters, and partitioned inside each datacenter.

Applications access and modify the database by issuing high-level operations. These operations consist of a sequence of *read* and *write* operations enclosed in *transactions*. An application submits a transaction to a replica; its reads and writes execute on a private copy of the replica state. If the transaction commits, its writes are applied to the local replica (local transaction), and propagate asynchronously to remote replicas, where they are also applied (remote transaction). If the transaction aborts, it has no effect.

We denote by $t(S)$ the state after applying the write operations of committed transaction t to some state S. We define a database snapshot, S_n, as the state of the database after a sequence of committed transactions $t_1, \ldots, t_n$ from the initial database state, S_{init}, i.e., $S_n = t_n(\ldots(t_1(S_{init})))$. The state of a replica results from applying both local and remote transactions, in the order received.

The transaction set $T(S)$ of a database snapshot S is the set of transactions included in S, e.g., $T(S_n) = \{t_1, \ldots, t_n\}$. We say that a transaction t_a executing in a database snapshot S_a happened-before t_b executing in S_b, $t_a \prec t_b$, if $t_a \in T(S_b)$. Two transactions t_a and t_b are concurrent, $t_a \parallel t_b$, if $t_a \not\prec t_b \wedge t_b \not\prec t_a$ [24].

For a given set of transactions T, the happens-before relation defines a partial order among them, $O = (T, \prec)$. We say $O' = (T, <)$ is a valid serialization of $O = (T, \prec)$ if O' is a linear extension of O, i.e., $<$ is a total order compatible with $\prec$.

Transactions can execute concurrently, with each replica executing transactions according to a different valid serialization. We assume the system guarantees state convergence, i.e., all valid serializations of $(T, \prec)$ lead to the same database state. Different techniques can be used to this end, from a simple *last-writer-wins* strategy to more complex approaches based on conflict-free replicated data types (CRDTs) [38, 41].

2.2 Explicit Consistency

Explicit Consistency is a novel consistency semantics for replicated systems, where programmers define the application-specific correctness rules that should be met. These rules are expressed as invariants over the database state.

Even if each replica maintains some invariant locally, concurrent updates might still cause violation. Consider for instance a tournament with a maximum capacity, limiting the cardinality of the set of enrolled players. Two replicas could concurrently enroll players into the same tournament, each one respecting the capacity. However, if the merge function is the union of the two sets of players, the capacity might be exceeded nonetheless.

Our formal definition starts with the helper definition of an invariant I, as a logical condition over the state of the database. We say that state S is an *I-valid state* if I holds in S, i.e., if $I(S) = true$.

Definition 2.1 (I-valid serialization)**.** Given a set of transactions T and its associated happens-before partial order $\prec$, $O_i = (T, <)$ is an *I-valid serialization* of $O = (T, \prec)$ if O_i is a valid serialization of O, and I holds in every state that results from executing some prefix of O_i.

We can now formally define the conditions that a system must uphold to ensure Explicit Consistency.

Definition 2.2 (Explicit consistency)**.** A system provides Explicit Consistency if all serializations of $O = (T, \prec)$ are I-valid serializations, where T is the set of transactions executed in the system and $\prec$ their associated partial order.

This concept is related to the *I-confluence* of Bailis et al. [5]. I-confluence defines the conditions under which operations may execute concurrently, while still ensuring that the system converges to an I-valid state. The current work generalizes this to cases where coordination is needed, and furthermore proposes efficient solutions.

3. Overview

Given application invariants, our approach for Explicit Consistency has three steps: *(i)* Detect the sets of operations that may lead to invariant violation when executed concurrently, called *I-offender sets*. *(ii)* Select an efficient mechanism for handling I-offender sets. *(iii)* Instrument the application code to use the selected mechanism on top of a weakly consistent database system.

The first step consists of discovering I-offender sets. This analysis is based on a model of the effects of operations. This information is provided by the application programmer, as annotations specifying the changes performed by each operation. Using this information, combined with the application invariants, static analysis infers the sets of operation invocations that, when executed concurrently, may lead to invariant violation (I-offender sets). Conceptually, the analysis considers all reachable database states and, for each state, all sets of operation invocations that can execute in that state; it checks if executing these operations concurrently might cause an invariant violation. Obviously, it is not feasible to exhaustively consider all database states and operation sets; instead, a practical approach is to use static verification techniques, which are detailed in Section 4.

In the second, the developer decides which approach to use to handle the I-offender sets. There are two options. With the first, *invariant repair*, operations are allowed to execute concurrently, and the conflict resolution rules that merge their outputs should include code to restore the invariants. One example is a graph data structure that supports operations to add and remove vertices and edges; if one replica adds an edge while concurrently another replica removes one of the edge's vertices, the merged state might ignore the hanging edge to ensure the invariant that an edge connects two vertices [38]. A similar approach applies to trees [30].

The second option, *violation avoidance*, consists of restricting concurrency sufficiently to avoid the invariant violation. We propose a number of techniques to allow a replica to execute such operations safely, without coordinating frequently with the others. Consider for instance the enrollment invariant (if a player is enrolled in a tournament, both the player and the tournament must exist). Any replica is allowed to execute the *enrollTournament* operation without coordination, as long as all replicas are forbidden to run *removePlayer* and *removeTournament*. This *reservation* may apply to a particular subset of players and tournaments.

Our reservation mechanisms support such functionality with reservations tailored to the different types of invariants, as detailed in Section 5.

In the third step, the application code is instrumented to use the conflict-repair and conflict-avoidance mechanisms selected by the programmer. This involves extending operations to call the appropriate API functions supported by Indigo.

4. Determining I-offender sets

In this section we detail the first step of our approach.

4.1 Defining invariants and post-conditions

Defining application invariants An application invariant is described by a first-order logic formula. More formally, we assume the invariant is an universally quantified formula in prenex normal form[1]

$$\forall x_1, \cdots, x_n, \varphi(x_1, \cdots, x_n).$$

First-order logic formulas can express a wide variety of consistency constraints; we give some examples in Section 4.2.

An invariant can use predicates such as $player(P)$ or $enrolled(P, T)$. A user may interpret them to mean that P is a player and that P is enrolled in tournament T; but technically the system depends only on their truth values and on the formulas that relate them. The application developer needs only to specify the effects of operations on the truth values of the terms used in the invariant.

Similarly, numeric restrictions can be expressed through the use of functions. For example, we may use $nrPlayers(T)$ (the number of players in tournament T) to limit the size of a tournament: $\forall T, nrPlayers(T) \leq 5$.

An application must satisfy the conjunction of all invariants.

Defining operation postconditions To express the side effects of operations, postconditions state the properties that are ensured after the execution of an operation that modifies the database. There are two types of side effect clauses: predicate clauses, which describe a truth assignment for a predicate (stating whether the predicate is true or false after execution of the operation); and function clauses, which define the relation between the initial and final function values. To give some examples, operation $removePlayer(P)$, which removes player P, has a postcondition with predicate clause $\neg player(P)$, stating that predicate $player$ is false for player P. Operation $enrollTournament(P, T)$,

[1] Formula $\forall x, \varphi(x)$ is in prenex normal form if clause φ is quantifier-free. Every first-order logic formula has an equivalent prenex normal form.

which enrolls player P into tournament T, has a postcondition with two clauses, $enrolled(P,T)$ and $nrPlayers(T) = nrPlayers(T) + 1$. If the player is already enrolled, the operation produces no side effects.

The syntax for postconditions is given by the grammar:

$$\begin{array}{lll} post & ::= & clause_1 \wedge clause_2 \wedge \cdots \wedge clause_k \\ clause & ::= & pclause \mid fclause \\ pclause & ::= & p(o_1, o_2, \cdots, o_n) \mid \neg p(o_1, o_2, \cdots, o_n) \\ fclause & ::= & f(o_1, o_2, \cdots, o_n) = opr \mid opr \oplus opr \\ opr & ::= & n \mid f(o_1, o_2, \cdots, o_n) \\ \oplus & ::= & + \mid - \mid * \mid \ldots \end{array}$$

where p and f are predicates and functions respectively, over objects $o_1, o_2, \cdots, o_n$.

Although we impose that a postcondition is a conjunction, it is possible to deal with operations that have alternative side effects, by splitting the alternatives between multiple dummy operations. For example, an operation φ with postcondition $\varphi_1 \vee \varphi_2$ could be replaced by operations op_1 and op_2 with postconditions φ_1 and φ_2, respectively.

4.2 Expressiveness of Application Invariants

Despite the simplicity of our model, it can express significant classes of invariants, as discussed next.

4.2.1 Restrictions Over The State

An application can define the set of valid application states, using invariants that define conditions that must be satisfied in every database state. By combining user-defined predicates and functions, it is possible to address a wide range of application semantics.

Numeric constraints Numeric constraints refer to numeric properties of the application and set lower or upper bounds to data values. Often, they control the use or access to a limited resource. For example, to ensure that a player does not overspend her (virtual) budget: $\forall P, player(P) \Rightarrow budget(P) \geq 0$. Disallowing an experienced player from participating in a beginner tournament can be expressed as: $\forall T, P, enrolled(P,T) \wedge beginners(T) \Rightarrow score(P) \leq 30$. By using user-defined functions in the constraints, it is possible to express complex conditions over the database state. We have previously shown how to limit the number of enrolled players in a tournament by using a function that counts the enrolled players. The same approach can be used to limit the number of elements in the database that satisfy any generic condition.

Uniqueness, a common correctness property, may also be expressed using a counter function. For example, the formula $\forall P, player(P) \Rightarrow nrPlayerId(P) = 1$, states that P must have a unique player identifier. Whereas, the formula $\forall T, tournament(T) \Rightarrow nrLeaders(T) = 1$ states that a collection has exactly one leader.

Integrity constraints An integrity constraint specifies the relationships between different objects, such as the *foreign key constraint* in databases. A typical example is the one at the beginning of this section, stating that enrollment must refer to existing players and tournaments. If the tournament application had a score table for players, another integrity constraint might be that every table entry must belong to an existing player: $\forall P, hasScore(P) \Rightarrow player(P)$.

General constraints over the state An invariant may also capture general constraints. For example, consider an application to reserve meetings, where two meetings must not overlap in time. Using predicate $time(M,S,E)$ to mean that meeting M starts at time S and ends at time E, we could write this invariant as follows: $\forall M_1, M_2, S_1, S_2, E_1, E_2, time(M_1, S_1, E_1) \wedge time(M_2, S_2, E_2) \wedge M_1 \neq M_2 \Rightarrow E_2 \leq S_1 \vee S_2 \geq E_1$.

4.2.2 Restrictions Over State Transitions

In addition to conditions over database state, we support some forms of temporal specifications, i.e., restrictions over state transitions. Our approach is to turn this into an invariant over the state of the database, by augmenting the database with a so-called *history variable* that records its state in a given moment in the past [1, 33].

In our running example, we might want to state, for instance, that players may not enroll or drop from an active tournament, i.e., between the start and the end of the tournament. For this, when a tournament starts, the application stores the list of participants, which can later be checked against the list of enrollments. If $participant(P,T)$ asserts that player P participates in active tournament T, and $active(T)$ asserts that tournament T is active, the above rule can be specified as follows: $\forall P, T, active(T) \wedge enrolled(P,T) \Rightarrow participant(P,T)$.

An alternative is to use a logic with support for temporal logic expressions, which allow for writing expressions that specify rules over time [24, 34]. Such approach would require more complex specification for programmers and a more complex analysis. We decided to forgo temporal logic, since our experience showed that our simpler approach was sufficient for specifying common application invariants.

4.2.3 Existential quantifiers

Some properties require existential quantifiers, for instance to state that tournaments must have at least one player enrolled: $\forall T, tournament(T) \Rightarrow \exists P, enrolled(P,T)$. This can be easily handled, since the existential quantifier can be replaced by a function, using the technique called skolemization. For this example, we may use function $nrPlayers(T)$ as such: $\forall T, tournament(T) \Rightarrow nrPlayers(T) \geq 1$.

4.2.4 Uninterpreted predicates and functions

The fact that predicates and functions are uninterpreted imposes limitations to the invariants that can be expressed. It implies, for example, that it is not possible to express reachability properties or other properties over recursive data structures. To encode invariants that require such properties, the

```
@Invariant("forall(P : p, T : t) :- enrolled(p, t) =>
player(p) and tournament(t)")
@Invariant("forall(P : p) :- budget(p) >= 0")
@Invariant("forall(T : t) :- nrPlayers(t) <= Capacity")
@Invariant("forall(T : t) :- active(t)
=> nrPlayers(t) >= 1")
@Invariant("forall(T : t, P : p) :- active(t) and
enrolled(p,t) => participant(p, t)")
public interface ITournament {
 @True("player($0)")
 void addPlayer(P p);

 @False("player($0)")
 void removePlayer(P p);

 @True("tournament($0)")
 void addTournament(T t);

 @False("tournament($0)")
 void removeTournament(T t);

 @True("enrolled($0, $1)")
 @False("participant($0, $1)")
 @Increments("nrPlayers($1, 1)")
 @Decrements("budget($0, 1)")
 void enrollTournament(P p, T t);

 @False("enrolled($0, $1)")
 @Decrements("nrPlayers($1, 1)")
 void disenrollTournament(P p, T t);

 @True("active($0)")
 @True("participant(_, $0)")
 void beginTournament(T t);

 @False("active($0)")
 void endTournament(T t);

 @Increments("budget($0,$1)")
 void addFunds(P p, int amount);
}
```

Figure 1. Invariant specification for the tournament application in Java (excerpt)

programmer has to express predicates that encode coarser statements over the database, which lead to a conservative view of safe concurrency. For example, instead of specifying some property over a branch of a tree, the programmer can define the property over the whole tree.

4.2.5 Example

In Figure 1 shows how to express the invariants for the tournament application in our Java prototype. The invariants in the listing are a subset of the examples just discussed. Application invariants are entered as Java annotations to the application interface (or class), and operation side-effects as annotations to the corresponding methods. Our notation was defined to be simple to convert to the language of the Z3 theorem prover, used in our prototype.

4.3 Algorithm

To identify the sets of concurrent operations that may lead to an invariant violation, we perform static analysis of operation postconditions against invariants. This analysis focuses on the case where operations execute concurrently from the same state. Although we assume that in a sequential execution, the invariants hold[2] , nonetheless, concurrently executing operations at different replicas may cause an invariant violation, which we call a *conflict*.

First, we check whether concurrent operations may result in opposite postconditions (e.g., $p(x)$ and $\neg p(x)$), breaking the generic (implicit) invariant that a predicate cannot have two different values. For instance, consider operations $addPlayer(P)$ with effect $player(P)$, vs. $removePlayer(P)$ with effect $\neg player(P)$. These operations conflict, since executing them concurrently with the same parameter P leaves unclear whether player P exists or not in the database. The developer may address this convergence violation by using a conflict resolution policy such as *add-wins* or *remove-wins*.

The remainder of the analysis consists in checking the effect of executing pairs of operations concurrently on the invariant. Our approach is based on Hoare logic [18], where the triple $\{I \wedge P\}\ op\ \{I\}$ expresses that the execution of operation op, in a state where precondition P holds, preserves invariant I. To determine if a set of operations are safe, we substitute their effects on the invariant, obtaining I', and check that the formula I' is valid given that the preconditions to execute the operations hold.

Considering all pairs of operations is sufficient to detect all invariant violations. The intuition why this is correct is that the static analysis considers all possible initial states before executing each concurrent pair, and therefore adding a third concurrent operation is equivalent to modifying the initial state of the two other operations.

To illustrate this process, we consider our tournament application, with the following invariant I:

$$I = \forall P, T, enrolled(P,T) \Rightarrow player(P) \wedge tournament(T) \\ \wedge \\ nrPlayers(T) \leq 5$$

For simplicity of presentation, let us examine each of the conjuncts defined in invariant I separately. First, we consider the numeric restriction: $\forall T, nrPlayers(T) \leq 5$, to illustrate how to check if multiple instances of the same operation are self-conflicting. In this case, one of the operations we need to take into account is $enrollTournament(P,T)$ whose outcome affects $nrPlayers(T)$. This operation has precondition $nrPlayers(T) \leq 4$, the weakest precondition that ensures the sequential execution does not break the invariant (see Footnote 2). To determine if this may break the invariant, we substitute the effects of running the $enrollTournament$ operation twice into invariant I. We then check whether this results in a valid formula, when considering also the weakest precondition. In this example, this corresponds to the following derivation (where notation $I\langle f \rangle$ describes the application of effect f in invariant I):

$$\begin{aligned} & I\ \langle nrPlayers(T) \leftarrow nrPlayers(T)+1 \rangle \\ & \langle nrPlayers(T) \leftarrow nrPlayers(T)+1 \rangle \\ nrPlayers(T) \leq 5\ & \langle nrPlayers(T) \leftarrow nrPlayers(T)+1 \rangle \\ & \langle nrPlayers(T) \leftarrow nrPlayers(T)+1 \rangle \\ nrPlayers(T)+1 \leq 5\ & \langle nrPlayers(T) \leftarrow nrPlayers(T)+1 \rangle \\ & nrPlayers(T)+1+1 \leq 5 \end{aligned}$$

[2] This can be achieved by having a precondition such that an operation produces no side effects, if its sequential execution against a state that does not meet that precondition would violate invariants.

Algorithm 1 Algorithm for detecting unsafe operations.

Require: I : invariant; O : operations.
1: $C \leftarrow \emptyset$ {subsets of unsafe operations}
2: **for** $op \in O$ **do**
3: **if** *self-conflicting*$(I, \{op\})$ **then**
4: $C \leftarrow C \cup \{\{op\}\}$
5: **for** $op, op' \in O$ **do**
6: **if** *opposing*$(I, \{op, op'\})$ **then**
7: $C \leftarrow C \cup \{\{op, op'\}\}$
8: **for** $op, op' \in O : \{op, op'\} \notin C$ **do**
9: **if** *conflict*$(I, \{op, op'\})$ **then**
10: $C \leftarrow C \cup \{op, op'\}\}$
11: **return** C

The resulting assertion $I' = nrPlayers(T) + 1 + 1 \leq 5$ is not ensured when both the initial invariant and the weakest precondition $nrPlayers(T) \leq 4$ hold. This shows that concurrent executions of $enrollTournament(P, T)$ conflict and $enrollTournament$ is a self-conflicting operation.

The second clause of I is $\forall P, T, enrolled(P, T) \Rightarrow player(P) \wedge tournament(T)$. This case illustrates a conflict between different operations. In this case, we check whether concurrent $enrollTournament(P, T)$ and $removePlayer(P)$ may violate the invariant. Again, we substitute the effects of these operations into the invariant and check whether the resulting formula is valid, assuming that initially the invariant and the preconditions of the two operations hold.

$$I \ \langle enrolled(P,T) \leftarrow true \rangle \ \langle player(P) \leftarrow false \rangle$$
$$enrolled(P,T) \Rightarrow player(P) \wedge tournament(T) \ \langle enrolled(P,T) \leftarrow true \rangle \ \langle player(P) \leftarrow false \rangle$$
$$true \Rightarrow player(P) \wedge tournament(T) \ \langle player(P) \leftarrow false \rangle$$
$$true \Rightarrow false$$
$$false$$

As the resulting formula is not valid, another pair of I-offenders is identified: $\{enrollTournament, removePlayer\}$.

We now present the complete logic to detect I-offender sets in Algorithm 1. This algorithm statically determines the pairs of operation that are conflicting, which are defined as follows.

Definition 4.1 (Conflicting operations)**.** Operations op_1, op_2, $\cdots$, op_n *conflict* with respect to invariant I if, assuming that I is initially true and the preconditions for op_1 and op_2 to produce side effects are initially true, the result of substituting the postconditions of both operations into the invariant is not a valid formula.

The core of the algorithm is made of auxiliary functions, which use the satisfiability modulo theory (SMT) solver Z3 [11] to verify the validity of the logical formulas used in Definition 4.1. Function *self-conflicting*$(I, \{op\})$ determines whether *op* is self-conflicting, i.e., if concurrent executions of *op* with the same or different arguments may break the invariant. Function *opposing*$(I, \{op, op'\})$ determines whether *op* and *op*$'$ have opposing postconditions. Function *conflict*$(I, \{op, op'\})$ determines whether the pair of operations break invariant I, by making it false under concurrent execution. They use the solver to check the validity of a set of formulas, namely the invariant, the preconditions for producing effects, and the updated invariant after substituting the effects of both operations.

Algorithm 1 uses these functions for computing I-offender sets in three steps. The initial step (line 2) determines self-conflicting operations. The second step (line 5) determines opposing operations by detecting contradictory predicate assignments for any pair of operations. The last step (line 8) determines other I-offender sets by checking if combining the effects of any two distinct operations raises an invariant violation. If it leads to a conflict, it adds the pair to the set of I-offender sets.

The number of test cases generated is polynomial in the number of operations, $\mathcal{O}(|O|^2)$. However, the satisfiability problem to be solved in each auxiliary function is, in the general case, NP-complete [19]. Z3 relies on heuristics to analyze formulas efficiently, in most cases. The results presented in Section 7.1.1 suggest that it is fast enough to be practical.

5. Handling I-offender sets

The previous step identifies I-offender sets. These sets are reported to the programmer, who decides how each situation should be addressed. We now discuss the techniques that are available to the programmer in Indigo.

5.1 Invariant Repair

One approach is to allow the conflicting operations to execute concurrently, and to repair invariant violations after the fact. Indigo has only limited support for this approach, since it can only address invariants defined in the context of a single database object (even though the object can be complex, such as a tree or a graph). To this end, Indigo provides a library of objects that repair invariants automatically using techniques proposed in the literature, e.g., sets, maps, graphs, trees with different conflict resolution policies [30, 38].

Application developers may extend this library, in order to support additional invariants. For instance, the programmer might want to extend the unbounded set provided by the library, to implement a set with bounded capacity n. She could modify queries such that they ignore excess elements from the underlying unbounded set; however, she must take care to use a deterministic and monotonic algorithm to select the elements to ignore [31].

5.2 Invariant-Violation Avoidance

The alternative approach is to avoid the concurrent execution of operations that would lead to an invariant violation when combining their effects. Indigo provides a set of basic techniques for achieving this, which extend previous ideas from the literature [17, 32, 35, 39, 44]. In comparison to the previous work, we not only combine these ideas in the

same system, but we also propose a new implementation, which is optimized for a geo-replicated setting by requiring only peer-to-peer communication, and relying on CRDTs to manage information [38].

5.2.1 Reservations

We now discuss the high-level semantics of the techniques used to restrict the concurrent execution of updates. The next section discusses their implementation in weakly consistent stores.

UID generator: A very common invariant is uniqueness of identifiers [5, 25]. This problem can be easily solved, without coordination, by statically splitting the space of identifiers per replica. Indigo provides this service by appending a replica-specific suffix to a locally-unique identifier.

Multi-level lock reservation: The multi-level lock reservation (or simply multi-level lock) is our base mechanism to restrict the concurrent execution of operations that can break invariants. A multi-level lock can provide the following rights: *(i) shared forbid*, giving the shared right to forbid some action to occur; *(ii) shared allow*, giving the shared right to allow some action to occur; *(iii) exclusive allow*, giving the exclusive right to execute some action.

When a replica holds one of the above rights, no other replica holds rights of a different type. For instance, if a replica holds a *shared forbid*, no other replica has any form of *allow*. We now show how to use this knowledge to control the execution of I-offender sets.

In the tournament example, $\{enrollTournament(P,T)$, $removePlayer(P)\}$ is an I-offender set. To avoid the violation of invariants, we can associate an appropriate multi-level lock to each of the operations, for specific values of the parameters. For example, we can have a multi-level lock associated with $removePlayer(P)$, for each value of P. For executing $removePlayer(P)$, it is necessary to obtain the right *shared allow* on the reservation for $removePlayer(P)$. For executing $enrollTournament(P,T)$, it is necessary to obtain the *shared forbid* right on the reservation for $removePlayer(P)$. This guarantees that enrolling some player will not execute concurrently with deleting the same player. However, concurrent enrolls or concurrent removes are allowed. In particular, if all replicas hold the *shared forbid* right on removing players, the most frequent enroll operation can execute in any replica, without coordination with other replicas.

The *exclusive allow* right, in turn, is necessary when an operation is incompatible with itself, i.e., when executing concurrently the same operation may lead to an invariant violation.

Multi-level locks are a form of lock [17] that can be used to restrict the concurrent execution of operations in any I-offender sets. It would be possible to enforce any application invariants using only multi-level locks. However, in some cases it is possible to provide additional concurrency while enforcing invariants, by using the following reservations.

Multi-level mask reservation: For invariants of the form $P_1 \vee P_2 \vee \ldots \vee P_n$, the concurrent execution of any pair of operations that makes two different predicates false may lead to an invariant violation if all other predicates were originally false. In our analysis, each of these pairs is an I-offender set.

Using simple multi-level locks for every pair of operations is too restrictive, as getting a *shared allow* on one operation would prevent the execution of all operations that could make any of the other predicates false. The reason why this is overly pessimistic is that, in this case, for executing an operation that makes some predicate false it suffices to guarantee that some other predicate remains true, which can be done by only forbidding the operations that make it false.

To allow for this, Indigo includes a multi-level mask reservation that can be seen as a vector of multi-level locks. For the invariant $P_1 \vee P_2 \vee \ldots \vee P_n$, a multi-level mask with n entries is created, with entry i used to control operations that may make P_i false.

When a replica obtains a *shared allow* right in one entry, it must obtain a *shared forbid* right in some other entry. For example, an operation that may make P_i false needs to obtain the *shared allow* right on the i^{th} entry and a *shared forbid* right on an entry j for which the predicate is true. At runtime, to find an entry to forbid, it is only necessary to evaluate the current value of the predicate associated with each entry that can be locked.

Escrow reservation: For numeric invariants of the form $x \geq k$, we include an escrow reservation for allowing some decrements to execute without coordination [32]. Given an initial value for $x = x_0$, there are initially $x_0 - k$ rights to execute decrements. These rights can be split dynamically among replicas. For executing $x.decrement(n)$, the operation must acquire and consume n rights to decrement x in the replica it is submitted. If not enough rights exist in the replica, the system will try to obtain additional rights from other replicas. If this is not possible, the operation will fail. Executing $x.increment(n)$ creates n rights to decrement n, initially assigned to the replica in which the operation that executes the increment is submitted.

A similar approach is used for invariants of the form $x \leq k$, with increments consuming rights and decrements creating new rights. For invariants of the form $x+y+\ldots+z \geq k$, a single escrow reservation is used, with decrements to any of the involved variables consuming rights and increments creating rights. If a variable x is involved in more than one invariant, several escrow reservations will be affected by a single increment/decrement operation on x.

The variant called *escrow reservation for conditions* checks a count of elements against some condition; for instance, the number of participants in a tournament in the invariant $nrPlayers(T) < k$. In this case, if the same user

is enrolled twice concurrently, two rights are consumed, although the number of participants increases by only one. This is conservative, but "leaks" rights. However, if the same user is disenrolled twice concurrently, then the number of users increases by only one; creating two rights might later let the invariant be violated.

Our escrow reservation for conditions addresses this problem using the following approach (considering invariant $c \geq k$). A decrement operation requires rights, just as a normal escrow reservation. However, an increment operation does not create rights immediately, but instead tags the reservation to be reevaluated. One of the replicas, marked as the primary for the reservation, is entrusted with recreating rights. To do so, it evaluates the distance between the current state and the threshold, taking into account the aggregate number of outstanding rights. More precisely, given the current value for $c = c_1$ and the number k_1 of outstanding rights (i.e., rights assigned to a replica and still not used, as known by the primary replica), $c_1 - k - k_1$ rights are created and assigned initially to the primary replica. This can be done either when the reservation is marked for reevaluation, or when new rights are needed.

Partition lock reservation: For some invariants, it is desirable to have the ability to reserve part of a partitionable resource. For example, consider the invariant that forbids two tournaments to overlap in time. Two operations that schedule different tournaments will break the invariant if the time periods overlap. Using a multi-level lock, it would be necessary to obtain an *exclusive allow* for executing any operation to schedule a new tournament.

However, no invariant violation arises if the time periods of concurrent operations do not overlap. To address this case, we provide a partition lock that allows a replica to obtain an *exclusive lock* on an interval of real values.[3] Replicas can obtain locks on multiple intervals, given that no two intervals reserved by different replicas overlap.

In our example, time would be mapped to a real number. To execute the operation that schedules a tournament, a replica would have to obtain a lock on an interval that includes the time from the start to the end of the tournament.

5.2.2 Using Reservations

The analysis from Section 4 outputs I-offender sets and the corresponding invariant violated. A programmer, electing to use the conflict avoidance approach, must select the type of reservation to be used to avoid invariant violations. Figure 1 presents a default mapping between types of invariants and the corresponding reservations. Conservatively, it is always possible to resort to multi-level locks to enforce any invariant, at the expense of admissible concurrency, as discussed earlier.

[3] Partition locks are a simplified version of partitionable objects [44] and slot reservations [35].

Invariant type	Formula (example)	Reservation
Numeric	$x < K$	Escrow(x)
Referential	$p(x) \Rightarrow q(x)$	Multi-level lock
Disjunction	$p_1 \vee \ldots \vee p_n$	Multi-level mask
Overlapping	$t(s_1, e_1) \wedge t(s_2, e_2) \Rightarrow s_1 \geq e_2 \vee e_1 \leq s_2$	Partition lock
Default	—	Multi-level lock

Table 1. Default mapping from invariants to reservations.

When using multi-level locks to prevent the concurrent execution of I-offender sets, it is possible to use different sets of reservations. We call this a reservation system. For example, consider our tournament application with the following two I-offender sets, which follow from the integrity constraint associated with enrollment: $\{enrollTournament(P,T), removePlayer(P)\}$ and $\{enrollTournament(P,T), removeTournament(P)\}$.

Given these I-offender sets, two alternative reservation systems can be used. The first system includes a single multi-level lock associated with $enroll(P,T)$, where this operation would have to obtain a *shared allow* right to execute, while both $removePlayer(P)$ and $removeTournament(T)$ would have to obtain the *shared forbid* right to execute. The second system includes two multi-level locks associated with $removePlayer(P)$ and $removeTournament(T)$, where enroll would have to obtain the *shared forbid* right in both locks to execute.

A simple optimization process is used to decide which reservations to use. As generating all possible combinations of reservation types may take too long, this process starts by generating a small number of systems using the following heuristic algorithm: *(i)* select a random I-offender set; *(ii)* decide the reservation to control the concurrent execution of operations in the set, and associate the reservation with the operation: if a reservation already exists for some of the operations, use the same reservation; otherwise, generate a new reservation from the type previously selected by the user; *(iii)* select the remaining I-offender set, if any, that has the most operations controlled by existing reservations, and repeat the previous step.

For each generated combination of reservations, Indigo computes the expected frequency of reservation operations needed, using as input the expected frequency of operations. The optimization process tries to minimize this expected frequency of reservation operations.

After deciding which reservation system will be used, each operation is extended to acquire the appropriate rights before executing its code, and to release appropriate rights afterwards. For escrow locks, an operation that consumes rights will acquire rights before its execution (and these rights will not be released when the operation ends). Conversely, an operation that creates rights will create these rights after its execution. For multi-level masks, the pro-

grammer must provide the code that verifies the values of the predicate associated with each element of the disjunction.

6. Implementation

In this section, we discuss the implementation of Indigo as a middleware running on top of a causally consistent store. We first explain the implementation of reservations and how they are used to enforce Explicit Consistency. We conclude by explaining how Indigo is designed to use an existing geo-replicated store.

6.1 Reservations

Indigo maintains information about reservations as objects stored in the underlying causally consistent storage system. For each type of reservation, a specific object class exists. Each reservation instance maintains information about the rights assigned to each of the replicas; in Indigo, each datacenter is considered a single replica, as explained later.

The escrow lock object maintains the rights currently assigned to each replica, and the following operations modify its state: *escrow_consume* depletes rights assigned to the local replica; *escrow_generate* generates new rights assigned to the local replica; and *escrow_transfer* transfers rights from the local replica to some given replica. For example, for an invariant $x \geq K$, *escrow_consume* must be used by an operation that decrements x and *escrow_generate* by operations that increment x. For the escrow lock for conditions variant, a replica is tagged as the primary. The *escrow_generate* only creates rights in the primary.

When *escrow_consume* and *escrow_transfer* operations execute in a replica, if that replica has insufficient rights, the operation fails and it has no side effects. Otherwise, the state of the replica is updated accordingly and the side effects are asynchronously propagated to the other replicas, using the normal replication mechanisms of the underlying storage system. As operations only deplete rights of the replica where they are submitted, it is guaranteed that every replica has a conservative view of the rights assigned to it: all operations that have consumed rights are known, but operations that transferred new rights from some other replica may still have to be received. Given that the execution of operations is serialized by the replica, this approach guarantees the correctness of the system in the presence of any number of concurrent updates in different replicas and asynchronous replication, as no replica will ever consume more rights than those assigned to it.

The multi-level lock object maintains which right (exclusive allow, shared allow, shared forbid) is assigned to each replica, if any. Rights are obtained for executing operations with some given parameters. For instance, in the tournament example, for removing player P the replica needs a *shared allow* right for player P. Thus, a multi-level lock object manages the rights for the different parameters independently. Each replica can then hold a given right for a specific value of the parameters or a subset of the parameter values. For simplicity, in our description, we assume that a single parameter exists.

The following operations can be submitted to modify the state of the multi-level lock object: *mll_giveRight* gives a right to some other replica; a replica with a shared right can give the same right to some other replica; a replica that is the only one with some right can change the right type and give it to itself or to some other replica; *mll_freeRight* revokes a right assigned to the local replica. As a replica can have been given rights by multiple concurrent *mll_giveRight* operations executed in different replicas, *mll_freeRight* internally encodes which *mll_giveRight* operations are being revoked. This is necessary to guarantee that all replicas converge to the same state.

As with escrow lock objects, each replica has a conservative view of the rights assigned to it, as all operations that revoke the local rights are always executed initially in the local replica. Additionally, assuming causal consistency, if the local replica shows that it is the only replica with some right, that information is correct system-wide. This condition holds despite concurrent operations and the asynchronous propagation of updates, as any *mll_giveRight* executed in some replica is always propagated before a *mll_freeRight* in that replica. Thus, if the local replica shows that no other replica holds any right, that is because no *mll_giveRight* has been executed (without being revoked).

The multi-level mask object is implemented using a vector of multi-level lock objects, with operations specifying which multi-level lock must be modified.

The partition lock object maintains which replica owns each interval. When it is created, a single replica holds the complete interval of values. A single operation modifies the state of the object: *pol_giveRight*, which transfers part of the interval owned by the local replica to some other replica. Using the same reasoning as in the previous cases, it is clear that the local replica always has a conservative view of the intervals it owns.

6.2 Indigo Middleware

We have built a prototype of Indigo on top of a geo-replicated data store with the following properties: *(i)* causal consistency; *(ii)* support for transactions that access a database snapshot and merge concurrent updates using CRDTs [38]; *(iii)* linearizable execution of operations for each object in each datacenter. There are at least two systems that support all these functionalities: SwiftCloud [46] and Walter [41]. Given that SwiftCloud has a more extensive support for CRDTs, which are fundamental for invariant-repair, we decided to build the Indigo prototype on top of SwiftCloud.

Storing reservations Reservation objects are stored in the underlying storage system and they are replicated in all datacenters. Reservation rights are assigned to datacenters individually, which keeps the information small. As discussed

in the previous section, the execution of operations in reservation objects at a given datacenter must be linearizable (to guarantee that two concurrent transactions do not consume the same rights).

The execution of an operation in the replica where it is submitted has three phases: i) the reservation rights needed for executing the operation are obtained; if not all rights can be obtained, the operation fails; ii) the operation executes, reading and writing the objects of the database; iii) the used rights are released (except for escrow reservations, where the rights that are consumed are not released); new rights are created in this step. After the local execution, the side effects of the operation in the data and reservation objects are propagated and executed in other replicas asynchronously and atomically.

Note that reservations guarantee that operations that can lead to invariant violation do not execute concurrently, but they do not guarantee that the preconditions for the operation to generate side effects hold. For example, in the tournament, before removing a tournament it is necessary to disenroll all players, thus guaranteeing that no player in enrolled.

Reservations manager The reservations manager is a service that runs in each datacenter and is responsible for exchanging reservations between datacenters, tracking reservations in use by local clients, and providing clients the database snapshot information to access the underlying storage. For correctness, it is necessary to enforce that updates of an operation are atomic and that reads are causally consistent with the current rights at each replica. In Indigo, these properties are guaranteed directly by the underlying storage system.

An example shows why these properties are necessary. In our tournament application, to enroll a player it is necessary to obtain the right that allows the enroll (by forbidding the removal of both the player and the tournament). After the enroll completes, the right is released and can be obtained by an operation that wants to remove the tournament. The problem is that if the state observed by the remove tournament operation did not include the previous enrollment, the application could end up deleting the tournament without disenrolling the students, leading to an invariant violation.

Obtaining reservation rights The first and last phases of operation execution obtain and free the rights needed for operation execution. Indigo provides API functions for obtaining and releasing a list of rights. Indigo tries to obtain the necessary rights locally using ordered locking to avoid deadlocks. If other datacenters need to be contacted for obtaining some reservation rights, this process is executed before starting to obtain rights locally. Unlike the process for obtaining rights in the local datacenter, Indigo tries to obtain the needed rights from remote datacenters in parallel for minimizing latency. This approach is prone to deadlocks; therefore, if some remote right cannot be obtained, we use an exponential backoff approach that frees all rights and tries to obtain them again after an increasing amount of time.

When it is necessary to contact other datacenters to obtain some right, the latency of operation execution can be severely affected. Therefore, reservation rights are obtained proactively using the following strategy. Escrow lock rights are divided among datacenters, with a datacenter asking for additional rights to the datacenter it believes has more rights (based on local information). The primary of an escrow lock for conditions creates new rights by computing the number of missing rights whenever either it runs out of rights or the object is marked for reevaluation. Multi-level lock and multi-level mask rights are pre-allocated to allow executing the most common operations (based on the expected frequency of operations), with shared allow and forbid rights being shared among all datacenters. In the tournament example, *shared forbid* for removing tournaments and players can be owned in all datacenters, allowing the more frequent enroll operation to execute locally. Partition lock rights are initially assigned to a single replica, and transferred when needed.

The reservations manager maintains a cache of reservation objects and allows concurrent operations to use the same shared (allow or forbid) right. While some ongoing operation is using a shared or exclusive right, the right cannot be revoked. The information about ongoing operations is maintained in soft-state. If the machine where the reservations manager runs fails, the ongoing operation will fail when trying to release the obtained rights.

6.3 Fault tolerance

Indigo builds on the fault tolerance of the underlying storage system. In a typical geo-replicated store, data is replicated inside a datacenter using quorums or a state-machine replication algorithm. Thus, the failure of a machine inside a datacenter does not lead to any data loss. This also applies to the machine running the reservations manager: as explained before, ongoing transactions will fail in this case; committed changes to the reservation objects are stored in the underlying storage system.

If a datacenter (fails or) gets partitioned from other datacenters, it is impossible to transfer rights from and to the partitioned datacenter. In each partition, operations that only require rights available in the partition can execute normally. Operations requiring rights not available in the partition will fail. When the partition is repaired (or the datacenter recovers with its state intact), normal operation is resumed.

In the event that a datacenter fails losing its internal state, the rights held by that datacenter are lost. As reservation objects maintain the rights held by all replicas, the procedure to recover the rights lost by the datacenter failure is greatly simplified: it is only necessary to guarantee that recovery is executed only once with a state that reflects all updates received from the failed datacenter.

7. Evaluation

This section presents an evaluation of Indigo. The main question our evaluation tries to answer is how does Explicit Consistency compares against *causal consistency* and *strong consistency* in terms of latency and throughput with different workloads. Additionally, we try to answer the following questions:

- Can the algorithm for detecting I-offender sets be used with realistic applications?
- What is the impact of an increasing the amount of contention in objects and reservations?
- What is the impact of using an increasing number of reservations in each operation?
- What is the behavior when coordination is necessary for obtaining reservations?

7.1 Applications

To evaluate Indigo, we used the following two applications.

Ad counter The ad counter application models the information maintained by a system that manages ad impressions in online applications. This information needs to be geo-replicated for allowing the fast delivery of ads. For maximizing revenue, an ad should be impressed exactly the number of times the advertiser is willing to pay for. This invariant can be easily expressed as $nrImpressions(A_i) \leq K_i$, where K_i is the maximum number of times ad A_i should be impressed and the function $nrImpressions(A_i)$ returns the number of times it has been impressed.

Advertisers will typically require ads to be impressed a minimum number of times in some countries. For instance, ad A should be impressed exactly 10,000 times, with at least 4,000 impressions in the US and another 4,000 impressions in the EU. This example is modeled through the following invariants for specifying the limits on the number of impressions (where $nrImpressionsOther$ counts the sum of the number of impressions in datacenters other than those two with the impressions in excess of $4,000$ in the EU or the US):

$$nrImpressionsEU(A) \leq 4,000$$
$$nrImpressionsUS(A) \leq 4,000$$
$$nrImpressionsOther(A) \leq 2,000$$

We modeled this application by having one counter for each ad and region pair. Invariants were defined with the target limits stored in the database: $nrImpressions(R, A) \leq targetImpressions(R, A)$ A single update operation that increments the ad tally was defined, which increments the function $nrImpressions$. Our analysis shows that two increment operations for the same counter can lead to an invariant violation, but increments on different counters are independent. Invariants can be enforced by relying on escrow lock reservations for each ad.

Our experiments used workloads with a mix of: a read only operation that returns the value of a set of counters selected randomly; an operation that reads and increments a randomly selected counter. Our default workload included only increment operations.

Tournament management This is a version of the application for managing tournaments described in Section 2 (and used throughout the paper as our running example), extended with read operations for browsing tournaments. The operations defined in this application are similar to operations that one would find in other management applications such as courseware management.

As detailed throughout the paper, this application has a rich set of invariants, including uniqueness rules for assigning ids; generic referential integrity rules for enrollments; and numeric invariants for specifying the capacity of each tournament. This leads to a reservation system that uses both escrow lock for conditions and multi-level lock reservation objects. There are three operations that do not require any right to execute: add player, add tournament and disenroll tournament, although the latter accesses the escrow lock object associated with the capacity of the tournament. The other update operations involve acquiring rights before they can execute.

In our experiments we have run a workload with 82% of read operations (a value similar to the TPC-W shopping workload), 4% of update operations requiring no rights for executing, and 14% of update operations requiring rights (8% of the operations are enrollment and disenrolments).

7.1.1 Performance of the Analysis

We implemented in Java the algorithm described in Section 4 for detecting I-offender sets, relying on the satisfiability modulo theory (SMT) solver Z3 [11] for verifying invariants. As discussed in Section 4, our algorithm relies on the efficiency of Z3 to be able to analyze programs in reasonable time.

Our prototype was was able to find the existing I-offender sets in the applications we have implemented. The average running time of this process in a recent MacBook Pro laptop was 19 ms for the ad counter applications and 730 ms for the more complex tournament application.

For the evaluation of the analysis, we additionally modeled TPC-W, so that we get results for a standard benchmark application. This application has less invariants to check than our custom applications, but has more operations. The running time for detecting I-offender sets was in this case 320 ms. These results show that although the running time increases with the number of invariants and operations, our algorithm can process realistic applications in reasonable times.

7.2 Experimental Setup

We compare Indigo against three alternative approaches:

Causal Consistency (Causal) As our system was built on top of the causally consistent SwiftCloud system [46],

we have used unmodified SwiftCloud as representative of a system providing causal consistency. We note that this system cannot enforce invariants. This comparison allows us to measure the overhead introduced by Indigo.

Strong Consistency (Strong) We have emulated a strongly consistent system by running Indigo in a single DC and forwarding all operations to that DC. We note that this approach allows more concurrency than a typical strong consistency system as it allows updates on the same objects to proceed concurrently and be merged if they do not violate invariants.

RedBlue consistency (RedBlue) We have emulated a system with RedBlue consistency [25] by running Indigo in all DCs and having red operations (those that may violate invariants and require reservations) execute in a master DC, while blue operations execute in the closest DC, while respecting causal dependencies.

Our experiments comprised 3 Amazon EC2 datacenters, US-East, US-West and EU, with inter-datacenter latency presented in Table 2. In each DC, Indigo servers run in a single m3.xlarge virtual machine with 4 vCPUs and 8 ECUs of computational power, and 15GB of memory available. Clients that issue transactions run in up to three m3.xlarge machines. Where appropriate, we placed the master DC in the US-East datacenter to minimize the overall communication latency and this way optimize the performance of that configuration.

RTT (ms)	US-E	US-W
US-West	81	–
EU	93	161

Table 2. RTT Latency among datacenters in Amazon EC2

7.3 Latency and Throughput

We start by comparing the latency and throughput of Indigo with alternative deployments for both applications.

We ran the ad counter application with 1000 ads and a single invariant for each ad. The maximum number of impressions was set sufficiently high to guarantee that the limit is not reached. The workload included only update operations for incrementing the counter. This allowed us to measure the peak throughput when operations were able to obtain reservations in advance. The results are presented in Figure 2, and show that Indigo achieves throughput and latency similar to a causally consistent system. Strong and RedBlue results are similar to each other, as all update operations are red and execute in the master DC in both configurations.

Figure 3 presents the results when running the tournament application with the default workload. As before, results show that Indigo achieves throughput and latency similar to a causally consistent system. In this case, as most operations are either read-only or otherwise can be classified as blue and thus execute in the local datacenter, the throughput of RedBlue is only slightly worse than that of Indigo.

Figure 4 details these results, presenting the latency per operation type (for selected operations) in a run with throughput close to the peak value. The results show that Indigo exhibits lower latency than RedBlue for red operations. These operations can execute in the local DC in Indigo, as they require either no reservation or reservations that can be shared and are typically locally available.

Two other results deserve some discussion: *Remove tournament* requires canceling shared forbid rights acquired by other DCs before being able to acquire the shared allow right for removing the tournament, which explain the high latency. Sometimes latency is very high (as shown by the line with the maximum value). This is a result of the asynchronous algorithms implemented and the approach for requesting remote DCs to cancel their rights, which can fail when a right is being used.

Add player has a surprisingly high latency in all configurations. Analyzing the situation, we found out that the reason for this lies in the fact that this operation manipulates very large objects used to maintain indexes, causing all configurations to have a fixed overhead.

7.4 Micro-benchmarks

Next, we examine the impact of key parameters.

Increasing contention Figure 5(a) shows the throughput of the system with increasing contention in the ad counter application, by varying the number of counters in the experiment. As expected, the throughput of Indigo decreases when contention increases as several steps require executing operations sequentially. Furthermore, the results reflect the fact that our middleware introduces an additional level of contention, because operations have to contact the reservation manager.

Increasing number of invariants Figure 5(b) presents the results of the ad counter application with an increasing number of invariants involved in each operation: the operation reads 5 counters (R5) and updates one to three counters (W1 to W3). In this case, the results show that the peak throughput for Indigo decreases while latency keeps constant. The reason for this is that for escrow locks, each invariant has an associated reservation object. Thus, when increasing the number of invariants, the number of updated objects also increases, with an impact on the operations that each datacenter needs to execute. To verify our explanation, we ran a workload with operations that access the same number of counters in the weak consistency configuration. The presented results show the same pattern of decreased throughput.

Impact when transferring reservations Figure 5(c) shows the latency of individual operations executed in the US-W datacenter in the ad counter application, for a workload where increments reach the invariant limit for multiple counters and where the rights were initially assigned to a single

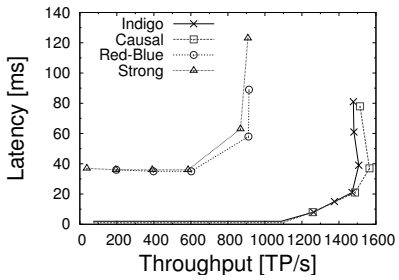


Figure 2. Peak throughput (ad counter application).

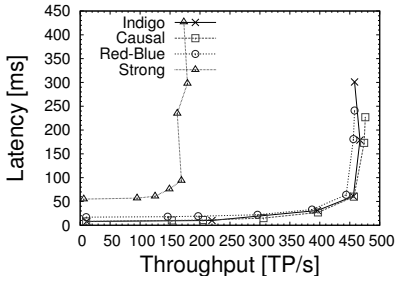


Figure 3. Peak throughput (tournament application).

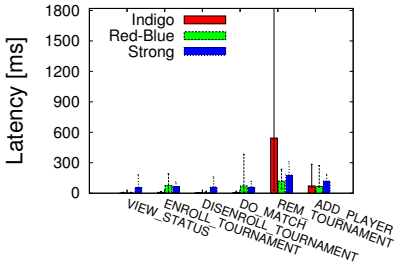
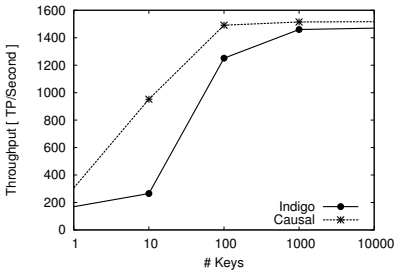
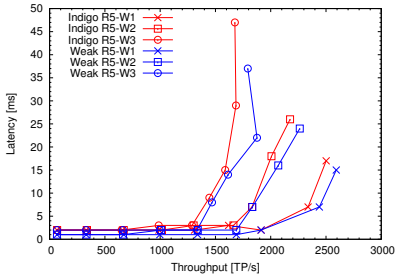


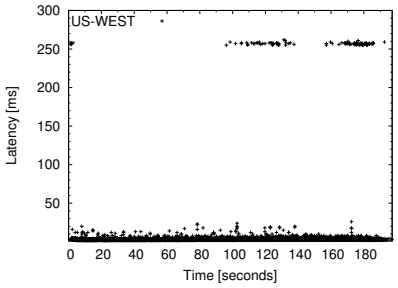
Figure 4. Average latency per op. type - Indigo (tournament app.).



(a) Peak throughput with increasing contention (ad counter application).



(b) Peak throughput with an increasing number of invariants (ad counter application).



(c) Latency of individual operations of US-W datacenter (ad counter application).

Figure 5. Micro-benchmarks.

datacenter. When rights do not exist locally, Indigo cannot mask the latency imposed by coordination, in this case, for obtaining additional rights from the remote datacenters. This explains the high latency operations close to the start of the experiment. As a bulk of rights is obtained, the following operations execute with low latency until it is necessary to obtain additional rights. When a replica believes that no other replica has available rights in an escrow lock object, it does not contact replicas. Instead, the operation fail locally, leading to low latency.

In Figure 4, we showed the impact of obtaining a multi-level lock shared right that requires revoking rights present in all other replicas. We have discussed this problem and a possible solution in Section 7.3. Nevertheless, it is important to note that such impact in latency is only experienced when it is necessary to revoke shared forbid rights in all replicas before acquiring the needed shared allow right. The positive consequence of this approach is that enroll operations requiring the shared forbid right that was shared by all replicas can execute with latency close to zero. The maximum latency line in enroll operation shows the maximum latency experienced when a replica acquires a shared forbid right from a replica already holding such right.

8. Related Work

Geo-replicated storage systems Many cloud storage systems supporting geo-replication emerged in recent years. Some offer variants of eventual consistency, where operations return right after being executed in a single datacenter, usually the closest one, so that end-user response times are improved [2, 12, 23, 27, 28]. These variants target different requirements, such as: reading a causally consistent view of the database (causal consistency) [2, 3, 14, 27]; supporting limited transactions where a set of updates are made visible atomically [4, 28]; supporting application-specific or type-specific reconciliation with no lost updates [7, 12, 27, 41], etc. Indigo is built on top of a geo-replicated store supporting causal consistency, a restricted form of transactions and automatic reconciliation; it extends those properties by enforcing application invariants.

Eventual consistency is insufficient for some applications that require (some operations to execute under) strong consistency for correctness. Spanner provides strong consistency for the whole database, at the cost of incurring coordination overhead for all updates [10]. Transaction chains support transaction serializability with latency proportional to the latency to the first replica that is accessed [47]. MDCC [22] and Replicated Commit [29] propose optimized approaches for executing transactions but still incur in inter-datacenter latency for committing transactions.

Some systems combine the benefits of weak and strong consistency models by supporting both. In Walter [41] and Gemini [25], transactions that can execute under weak consistency run fast, without needing to coordinate with other datacenters. Bayou [42] and Pileus [43] allow operations to read data with different consistency levels, from strong to eventual consistency. PNUTS [9] and DynamoDB [40] also combine weak consistency with per-object strong consistency relying on conditional writes, where a write fails in the presence of concurrent writes. Indigo enforces Explicit Consistency rules, exploring application semantics to let (most) operations execute in a single datacenter.

Exploring application semantics Several works have explored the semantics of applications (and data types) for improving concurrent execution. Semantic types [16] have been used for building non serializable schedules that preserve consistency in distributed databases. Conflict-free replicated data types [38] explore commutativity for enabling the automatic merge of concurrent updates, which Walter [41], Gemini [25] and SwiftCloud [46] use as the basis for providing eventual consistency. Indigo goes further by exploring application semantics to enforce application invariants.

Escrow transactions [32] 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 [35, 39, 44]. The demarcation protocol [6] is aimed 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 [15] provide time-limited assertions over the database state, which can improve latency of read operations in cloud storages.

Indigo builds on these works, but it is the first 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.

Other related work Bailis et al. [5] studied the possibility of avoiding coordination in database systems and still maintain application invariants. Our work complements that, addressing the cases that cannot entirely avoid coordination, yet allow operations to execute immediately by obtaining the required reservations in bulk and in anticipation.

Others have tried to reduce the need for coordination by bounding the degree of divergence among replicas. Epsilon-serializability [36] and TACT [45] use deterministic algorithms for bounding the amount of divergence observed by an application using different metrics: numerical error, order error and staleness. Consistency rationing [21] uses a statistical model to predict the evolution of replica state and allows applications to switch from weak to strong consistency upon the likelihood of invariant violation. In contrast to these works, Indigo focuses on enforcing invariants efficiently.

The static analysis of code is a standard technique used extensively for various purposes, including in a context similar to ours [8, 13, 20]. Sieve [26] combines static and dynamic analysis to infer which operations should use strong consistency and which operations should use weak consistency in a RedBlue system [25]. Roy et al. [37] 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. The latter work also proposes an algorithm to allow replicas to execute transactions independently by defining conditions that must be met in each replica. Whenever an operation cannot commit locally, a new set of conditions is computed and installed in all replicas using two-phase commit. In Indigo, replicas can exchange rights in a peer-to-peer manner.

9. Conclusions

This paper proposes an application-centric consistency model for geo-replicated services, Explicit Consistency, where programmers specify the consistency rules that the system must maintain as a set of invariants. We describe a methodology that helps programmers decide which invariant-repair and violation-avoidance techniques to use to enforce Explicit Consistency, extending existing applications. We also present the design of Indigo, a middleware that can enforce Explicit Consistency on top of a causally consistent store. The results show that the modified applications have performance similar to weak consistency for most operations, while being able to enforce application invariants. Some rare operations that require intricate rights transfers exhibit high latency. As future work, we intend to improve the algorithms for exchanging reservation rights on those situations.

Acknowledgments

We would like to thank our shepherd Gustavo Alonso, the anonymous reviewers and Alexey Gotsman for their helpful comments on a previous version of this work. This research is supported in part by EU FP7 SyncFree project (609551), FCT/MCT SFRH/BD/87540/2012, PTDC/ EEI-SCR/ 1837/ 2012 and PEst-OE/ EEI/ UI0527/ 2014. The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC Starting Grant.

References

[1] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991.

[2] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Com-*

puter Systems, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[3] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[4] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 27–38, New York, NY, USA, 2014. ACM.

[5] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow. (to appear)*, 2015.

[6] 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.

[7] Basho. Riak. `http://basho.com/riak/`, 2014. Accessed Oct/2014.

[8] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[9] 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.

[10] 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 *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[11] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[13] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 12 1998.

[14] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[15] ed Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for Faster Strong Consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, Berkeley, CA, USA, 2014. USENIX Association.

[16] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.

[17] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Readings in Database Systems. chapter Granularity of Locks and Degrees of Consistency in a Shared Data Base, pages 94–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[19] H. B. Hunt and D. J. Rosenkrantz. The Complexity of Testing Predicate Locks. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 127–133, New York, NY, USA, 1979. ACM.

[20] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55. Springer-Verlag, Berlin, Heidelberg, 2011.

[21] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay Only when It Matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009.

[22] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

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

[24] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[25] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[26] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[27] 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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Princi-*

ples, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[28] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[29] 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, July 2013.

[30] S. Martin, M. Ahmed-Nacer, and P. Urso. Abstract unordered and ordered trees CRDT. Research Report RR-7825, INRIA, Dec. 2011.

[31] S. Martin, M. Ahmed-Nacer, and P. Urso. Controlled conflict resolution for replicated document. In *Proceedings of the 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 471–480. IEEE, Oct 2012.

[32] P. E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.

[33] S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, Ithaca, NY, USA, 1975.

[34] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[35] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for Conflict Avoidance in a Mobile Database System. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM.

[36] K. Ramamritham and C. Pu. A Formal Characterization of Epsilon Serializability. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):997–1007, Dec. 1995.

[37] S. Roy, L. Kot, N. Foster, J. Gehrke, H. Hojjat, and C. Koch. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (to appear)*, SIGMOD '15. ACM, May-June 2015.

[38] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[39] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 42–61, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[40] S. Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[41] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[42] 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 *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[43] 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 *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[44] G. D. Walborn and P. K. Chrysanthis. Supporting Semantics-based Transaction Processing in Mobile Database Applications. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, SRDS '95, pages 31–40, Washington, DC, USA, 1995. IEEE Computer Society.

[45] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.

[46] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. Research Report RR-8347, INRIA, Oct. 2013.

[47] 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 *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.