Fine-grained consistency for geo-replicated systems

Cheng Li[♭], Nuno Preguiça[‡], Rodrigo Rodrigues[*]
[♭]*University of Science and Technology of China,* [‡]*NOVA LINCS & FCT, Univ. NOVA de Lisboa,*
[*]*INESC-ID & Instituto Superior Técnico, Universidade de Lisboa*

Abstract

To deliver fast responses to users worldwide, major Internet providers rely on geo-replication to serve requests at data centers close to users. This deployment leads to a fundamental tension between improving system performance and reducing costly cross-site coordination for maintaining service properties such as state convergence and invariant preservation. Previous proposals for managing this trade-off resorted to coarse-grained operations labeling or coordination strategies that were oblivious to the frequency of operations. In this paper, we present a novel fine-grained consistency definition, Partial Order-Restrictions consistency (or short, PoR consistency), generalizing the trade-off between performance and the amount of coordination paid to restrict the ordering of certain operations. To offer efficient PoR consistent replication, we implement Olisipo, a coordination service assigning different coordination policies to various restrictions by taking into account the relative frequency of the confined operations. Our experimental results show that PoR consistency significantly outperforms a state-of-the-art solution (RedBlue consistency) on a 3-data center RUBiS benchmark.

1 Introduction

To cope with the demand for fast response times [36] from an increasingly large user base, many Internet service providers such as Google [8], Microsoft [9], Facebook [13] or Amazon [3] replicate data across multiple geographically dispersed data centers [38, 21, 20, 2]. However, geo-replication also leads to an inherent tension between achieving high performance and ensuring properties such as state convergence (i.e., all replicas eventually reach the same final state) and invariant preservation (i.e., the behavior of the system obeys its specification, which can be defined as a set of application-specific invariants to be preserved) [22, 40, 30, 17, 16].

Some proposals address this fundamental tension in geo-replication by weakening strong consistency to different extents: some researchers suggest to completely drop strong consistency and instead adopt some form of weaker consistency such as eventual consistency [22, 41, 19] or causal consistency [32]; other approaches allow multiple consistency levels to coexist in a single system [30, 17, 12, 4]. As an example of the latter group, our prior proposal on RedBlue consistency [30], allows some operations to execute under strong consistency (and therefore incur a high performance penalty) while other operations can execute under weaker consistency (namely causal consistency). The core of this solution is a labeling methodology for guiding the programmer to assign consistency levels to operations. The labeling process works as follows: operations that either do not commute w.r.t. all others or potentially violate invariants must be strongly consistent, while the remaining ones can be weakly consistent.

This binary classification methodology is effective for many applications, but it can also lead to unnecessary coordination in some cases. In particular, as we will later illustrate, there are cases where it is important to synchronize the execution of two specific operations, but those operations do not need to be synchronized with any other operation in the system (and this synchronization would happen across all strongly consistent operations in the previous scheme). Furthermore, while concepts such as *conflict relation* in generic broadcast [34] and *token* by Gotsman et al. [24] allow for a finer-grained coordination of operations, these either lack a precise method for identifying a set of restrictions to ensure safety or an implementation that achieves efficient coordination by adapting to the observed workload.

To overcome these limitations, in this paper, we propose a novel generic consistency definition, *Partial Order-Restrictions consistency* (or short, *PoR consistency)*, which takes a set of restrictions as input and forces these restrictions to be met in all partial orders. This creates the opportunity for defining many consistency guarantees within a single replication framework by expressing consistency levels in terms of visibility restrictions on pairs of operations. Weakening or strengthening the consistency semantics is achieved by imposing fewer or more restrictions.

Under PoR consistency, the key to making a geo-replicated deployment of a given application perform well is to identify a set of restrictions over pairs of its operations so that state convergence and invariant preservation are ensured if these restrictions are enforced throughout all executions of the system. However, this is challenging because missing required restrictions may cause applications to diverge state or violate invariants, while placing unnecessary restrictions will lead to a performance penalty due to the additional coordination. To

this end, we design principles guiding programmers to identify the important restrictions while avoiding unnecessary ones.

Furthermore, from a protocol implementation perspective, given a set of restrictions over pairs of operations, there exist several coordination protocols that can be used for enforcing a given restriction, such as Paxos, distributed locking, or escrow techniques. However, depending on the frequency over time with which the system receives operations confined by a restriction, different coordination approaches lead to different performance trade-offs. Therefore, to minimize the runtime coordination overhead, we also propose an efficient coordination service called Olisipo that helps replicated services use the most efficient protocol by taking into account the system workload.

To demonstrate the power of PoR consistency, we extended RUBiS to incorporate a closing auction functionality, determined how to best run it under PoR consistency, replicated it with Olisipo, and compared its performance against a RedBlue consistent version. Our experimental results show that PoR consistency requires fewer restrictions and offers a significantly better performance than RedBlue consistency.

2 Preliminaries

2.1 System model

We assume a geo-distributed system with state fully replicated across k sites denoted by $site_0 \dots site_{k-1}$, where each site hosts a replica, and each replica runs as a deterministic state machine. In the rest of the document, the terms "site" and "replica" are interchangeable.

The system defines a set of operations $\mathscr{U}$ manipulating a set of reachable states $\mathscr{S}$. Each operation u is initially submitted by a user at one site which we call u's *primary site* and denote $site(u)$. An operation is defined mathematically as a function that receives the current state of the system and returns another function corresponding to its side effects. We refer to the former function as the *generator* function, denoted by g_u; this generator function, when applied to a given state $S \in \mathscr{S}$, returns a *shadow* function or *shadow operation*, denoted $h_u(S)$.

Implementation-wise, the generator function will first execute in a *sandbox* against the current state of the replica at the primary site, without interference from other concurrent operations. In this phase, the execution only identifies what changes u would introduce to state S that is observed by u and will not commit these changes. At the end of executing g_u, the identified side-effect or shadow operation $h_u(S)$ will be sent and applied across all replicas including the primary site.

A desirable property is that all replicas that have applied the same set of shadow operations are in the same state, i.e., the underlying system offers **state convergence**. In addition, the system maintains a set of application-specific **invariants**. For instance, an online shopping service cannot sell more items than those available in stock. To capture this notion, we define the function $valid(S)$ to be *true* if state S satisfies all these invariants and *false* otherwise.

2.2 RedBlue consistency

Our prior proposal called RedBlue consistency [30] is based on a division of shadow operations into blue operations, whose order of execution can vary from site to site, and red operations that must execute in the same relative order at all sites. For guiding developers in making use of RedBlue consistency, this work identified that a condition for ensuring state convergence is that a shadow operation must be labeled red if it is not globally commutative. For ensuring that invariants are maintained, a sufficient condition was identified, stating that all shadow operations that may violate an invariant when being applied against a different state from the one they were generated must be labeled red. For the remaining shadow operations, which have passed the two condition checks, we can safely label them blue.

3 Partial Order-Restrictions consistency

3.1 Motivating example

We illustrate the limitations of coarse-grained labeling schemes like RedBlue consistency through an eBay-like auction service in Fig.1, where an operation `placeBid` (Fig.1(a)) creates a new bid for an item if the corresponding auction is still open, and an operation `closeAuction` (Fig.1(c)) closes an auction and declares a single winner. In this example, the application-specific invariant is that the winner must be associated with the highest bid across all accepted bids. The other two subfigures (Fig.1(b) and Fig.1(d)) depict the commutative shadow operations of these two operations.

When applying RedBlue consistency to replicate such an auction service, we note that the concurrent execution under weak consistency of a `placeBid` with a bid higher than all accepted bids and a `closeAuction` can lead to the violation of the application invariant. This happens because the generation of `closeAuction'` will ignore the highest bid created by the concurrent shadow `placeBid'`. Unfortunately, the only way to address this issue in RedBlue consistency is to label both shadow operations as strongly consistent, i.e., all shadow operations of either type will be totally ordered w.r.t each other, which will incur a high overhead in geo-distributed settings. Intuitively, however, there is no need to order pairs of `placeBid'` shadow operations, since a bid coming before or after another does not affect the winner selection. This highlights that a coarse-grained operation clas-

```
boolean placeBid(int itemId, int clientId, int bid){
 boolean result = false;
 beginTxn();
 if(open(itemId)){
   createShadowOp(placeBid', itemId, clientId, bid);
   result = true;
 }
 commitTxn();
 return result;
}
```

(a) Original placeBid operation.

```
placeBid'(int itemId, int clientId, int bid){
  exec(INSERT INTO bidTable VALUES (bid, clientId, itemId));
}
```

(b) Shadow placeBid' operation.

```
int closeAuction(int itemId){
 int winner = -1;
 beginTxn();
 close(itemId);
 winner = exec(SELECT userId FROM bidTable WHERE iId = itemId
       ORDER BY bid DESC limit 1);
 createShadowOp(closeAuction', itemId, winner);
 commitTxn();
 return winner;
}
```

(c) Original closeAuction operation.

```
closeAuction'(int itemId, int winner){
 close(itemId);
 exec(INSERT INTO winnerTable VALUES (itemId, winner));
}
```

(d) Shadow closeAuction' operation.

Figure 1: Pseudocode for the `placeBid` and `closeAuction` operations of an auction site

sification into two levels of consistency can be conservative, and some services could benefit from additional flexibility in terms of the level of coordination.

To overcome these limitations of RedBlue consistency, we next propose Partial Order-Restrictions consistency (or short, PoR consistency), a novel consistency model that allows the developer to reason about various fine-grained consistency requirements in a single system. The key intuition behind our proposal is that this model is generic and can be perceived as a set of restrictions imposed over admissible partial orders across the operations of a replicated system.

3.2 Defining PoR consistency

The definition of PoR consistency includes three important components: (1) a set of restrictions, which specify the visibility relations between pairs of operations; (2) a restricted partial order (or short, R-order), which establishes a (global) partial order of operations respecting operation visibility relations; and (3) a set of site-specific causal serializations, which correspond to total orders in which the operations are locally applied. We define these components formally as follows:

Definition 1 (Restriction)**.** *Given a set of operations U, a* restriction *is a symmetric binary relation on $U \times U$.*

For any two operations u and v in U, if there exists a restriction relation between them, we denote this relation as $r(u,v)$.

Definition 2 (Restricted partial order)**.** *Given a set of operations U, and a set of restrictions R over U, a* restricted partial order *(or short,* R-order*) is a partial order $O = (U, \prec)$ with the following constraint: $\forall u, v \in U$, $r(u,v) \in R \implies u \prec v \vee v \prec u$.*

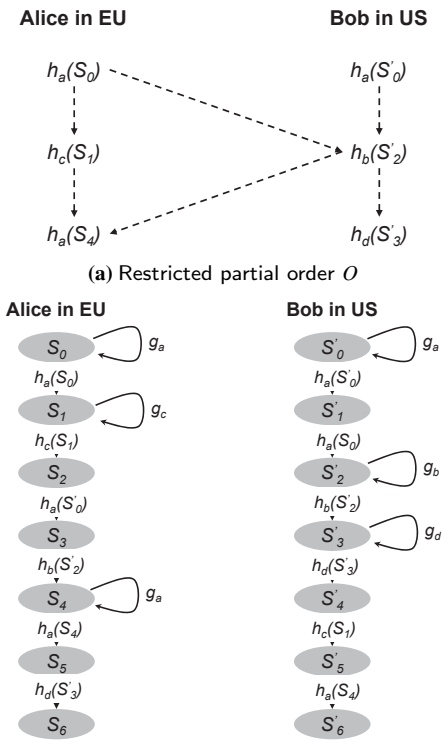
We say that the restrictions in R are met in the corresponding R-order if this order satisfies the above definition. This definition places constraints on a global view of a replicated system; however, it fails to explain how each individual replica at every site will behave according to this global view. When user requests are accepted by any site, that site executes their generator operations and creates corresponding shadow operations which will be replicated across all sites. In addition, every site not only commits shadow operations created by itself, but also applies remote ones shipped from all other sites against its local state. We denote U as the set of shadow operations produced across all sites, while for a site i, we denote V_i as its generator operation set. The following definition models the execution of each site as a growing linear extension of the global R-order, which incorporates a notion of causality, due to the fact that the visibility dependencies that are established when shadow operations are initially generated, are then preserved while the corresponding shadow operations are replicated.

Definition 3 (Causal legal serialization)**.** *Given a site i, an R-order $O = (U, \prec)$ and the set of generator operations V_i received at site i, we say that $O_i = (U \cup V_i, <_i)$ is an* i-causal legal serialization *(or short, a* causal serialization*) of O if*

- *O_i is a total order;*
- *$(U, <_i)$ is a linear extension of O;*
- *For any $h_v(S) \in U$ generated by $g_v \in V_i$, (1) S is the state obtained after applying the sequence of shadow operations preceding g_v in O_i; (2) For any $h_u(S') \in U$, $h_u(S') <_i g_v$ in O_i iff $h_u(S') \prec h_v(S)$ in O.*

Definition 4 (Partial Order-Restrictions consistency)**.** *A replicated system $\mathscr{S}$ spanning k sites with a set of restrictions R is* Partial Order-Restrictions consistent *(or short,* PoR consistent*) if each site i applies shadow operations according to an i-causal serialization of R-order O.*

Fig.2 shows a restricted partial order and its causal legal serializations executed at two sites, namely EU and US, where we restrict pairs of shadow operations where one corresponds to a and the other to b. When the US site executes a generator of b, g_b, it realizes that the shadow operation it would generate may need to be re-



(a) Restricted partial order O

(b) Causal legal serializations of O

Figure 2: Restricted partial order of shadow operations and its causal legal serializations for a system spanning two sites. There exists a restriction $r(h_a(S), h_b(S))$ for all valid S. Dotted arrows in Fig.2a indicate dependencies between shadow operations. Loops in Fig.2b represent generator operations.

stricted w.r.t a concurrent shadow operation initially triggered at the EU site. As a result, g_b at the US site must wait until the respective concurrent shadow operation $h_a(S_0)$ gets propagated from Alice's site to Bob's site. Then g_b will read the state introduced by locally applying $h_a(S_0)$ from Alice, and produce a shadow operation $h_b(S'_2)$. Note that this production will establish a dependency between $h_a(S_0)$ and $h_b(S'_2)$ (as shown in Fig.2a), thus enforcing that they cannot be applied in different relative orders in all causal legal serializations (as shown in Fig.2b). Unlike these two shadow operations, we do not restrict any pair of shadow operations of a; as such, the first operations issued by both Alice and Bob will be concurrently executed without being aware of each other. This example indicates the flexibility and performance benefits of having PoR consistency, compared with RedBlue consistency, since under the latter model all shadow operations of a and b would be serialized w.r.t each other.

4 Restriction inference

When replicating a service under PoR consistency, the first step is to infer restrictions to ensure two important system properties, namely state convergence and invariant preservation. The major challenge we face is to identify a minimal set of restrictions for making the replicated service converge and not violate invariants. With regard to state convergence, we take a similar methodology adopted in prior research [37, 30, 29], which is to check operation commutativity.

To preserve application-specific invariants, instead of totally ordering all *non-invariant safe* shadow operations, i.e., those that potentially transition from a valid state to an invalid one, we try to identify a minimal set of shadow operations that lead to an invariant violation when they are running concurrently in a coordination-free manner. By minimal, we mean that removing any operation from that set would no longer meet that goal. Once this set is identified, adding a restriction between any pair of its operations is sufficient to eliminate the problematic executions.

4.1 State convergence

A PoR consistent replicated system is state convergent if all its replicas reach the same final state when the system becomes quiescent, i.e., for any pair of causal legal serializations of any R-order, L_1 and L_2, we have $S_0(L_1) = S_0(L_2)$, where S_0 is a valid initial state. We state a necessary and sufficient condition to achieve this in the following theorem.

Theorem 5. *A PoR consistent system $\mathscr{S}$ with a set of restrictions R is **convergent**, if and only if, for any pair of its shadow operations u and v, $r(u,v) \in R$ if u and v don't commute.*[1]

Unlike RedBlue consistency, under which all operations that are not globally commutative must be totally ordered, PoR consistency only requires that an operation must be ordered w.r.t another one if they do not commute.

4.2 Invariant preservation

In RedBlue consistency, the methodology for identifying restrictions imposed on RedBlue orders for maintaining invariants is to check if a shadow operation is *invariant safe* or not (meaning whether it can potentially violate invariants when executed against a different state from the one that it was generated from). If not, to avoid invariant violations, the generation and replication of all non-invariant safe shadow operations must be coordinated. However, we observed that for some non-invariant safe shadow operations u, the corresponding violation only happens when a particular subset of non-invariant safe shadow operations (including u) are not partially ordered. Therefore, to eliminate all invariant violating executions with a minimal amount of coordination, we need to precisely define, for each violation, the minimal set of non-invariant safe shadow operations that are involved.

[1] All proofs are in a separate technical report [7].

We call this set an `invariant-conflict operation set`, or short, `I-conflict set`. Preserving invariants only requires adding a single restriction over any two shadow operations from each `I-conflict set` so that the concurrent violating executions will be eliminated from all admissible partial orders. We formally define `I-conflict sets` as follows.

Definition 6 (Invariant-conflict operation set)**.** *A set of shadow operations G is an* invariant-conflict operation set *(or* I-conflict set*) if the following conditions are met:*

- $\forall u \in G$, *u is non-invariant safe;*
- $|G| > 1$*;*
- $\forall u \in G$, $\forall$ *sequence P consisting of all shadow operations in G except u, i.e., $P = (G \setminus \{u\}, <)$, $\exists$ a reachable and valid state S, s.t. $S(P)$ is valid, and $S(P+u)$ is invalid.*

In the above definition, the last point asserts that G is minimal, i.e., removing one shadow operation from it will no longer lead to invariant violations. We will use the following example to illustrate the importance of minimality. Imagine that we have an auction on an item i being replicated across three sites such as US, UK and DE, and having initially a 5 dollar bid from *Charlie*. Suppose also that three shadow operations, namely, $placeBid'(i, Bob, 10)$, $placeBid'(i, Alice, 15)$, and $closeAuction'(i)$ are accepted concurrently at the three locations, respectively. After applying all of them against the same initial state at every site, we end up with an invalid state, where *Charlie* rather than *Bob* and *Alice* won the auction. This invariant violating execution involves three concurrent shadow operations, but one of the two bid placing shadow operations is not necessary to be included in G, as even after excluding the request from either *Bob* or *Alice*, the violation still remains. This is reflected in Definition 6, according to which $\{placeBid', closeAuction'\}$ is an `I-conflict set`, while $\{placeBid', placeBid', closeAuction'\}$ is not. Intuitively, avoiding invariant violations requires preventing all operations from the `I-conflict` set from running in a coordination-free manner. The minimality property enforced in the `I-conflict set` definition allows us to avoid adding unnecessary restrictions.

Based on the above definition, we formulate the invariant preservation property into the following theorem.

Theorem 7. *Given a PoR consistent system $\mathscr{S}$ with a set of restrictions $R_{\mathscr{S}}$, for any execution of $\mathscr{S}$ that starts from a valid state, no site is ever in an invalid state, if the following conditions are met:*

- *for any of its `I-conflict set` G, there exists a restriction $r(u,v)$ in $R_{\mathscr{S}}$, for at least one pair of shadow operations $u, v \in G$; and*
- *for any pair of shadow operations u and v, $r(u,v)$ in $R_{\mathscr{S}}$ if u and v do not commute.*

Algorithm 1 Find state convergence restrictions

```
1:  function SCRDISCOVER(T)                    ▷ T: the set of shadow operations of the target system
2:      R ← {}                                 ▷ R: the restriction set
3:      for i ← 0 to |T| − 1 do
4:          for j ← i to |T| − 1 do
5:              if T_i do not commute with T_j then
6:                  R ← R ∪ {r(T_i, T_j)}
7:      return R
```

Algorithm 2 Find invariant preserving restrictions

```
1:  function IPRDISCOVER(T)
2:      R ← {}                                 ▷ R: the restriction set
3:      Q ← power set of T
4:      for all Q' ∈ Q do
5:          if ICONFLICTCHECK(Q') then
6:              if |Q'| == 1 then
7:                  R ← R ∪ {r(Q'_0, Q'_0)}
8:              else if ∀u, v ∈ Q', r(u,v) ∉ R then
9:                  R ← R ∪ {r(u,v)}, for an arbitrary choice of u, v ∈ Q'
10:     return R
11: function ICONFLICTCHECK(T)
12:     if |T| == 1 then
13:         if ¬(T_0.post ⟹ T_0.wpre) then
14:             return true
15:     if |T| > 1 then
16:         subset_iconflict ← false
17:         for i ← 2 to |T| − 1 do
18:             for all R s.t. |R| == i and R ⊂ T do
19:                 if ICONFLICTCHECK(R) then
20:                     subset_iconflict ← true
21:                     break
22:         if !subset_iconflict then
23:             for all t ∈ T do
24:                 post ← ∧_{x∈T\{t}} x.post
25:                 if ¬(post ⟹ t.wpre) then
26:                     return true
27:     return false
```

4.3 Identifying restrictions

The key to striking a sensible balance between performance and consistency semantics is to identify a minimal set of restrictions that ensure both state convergence and invariant preservation. With regard to the former property, inspired by Theorem 5, we design a discovery method for finding restrictions to ensure state convergence (Alg. 1). This method systematically performs an operation commutativity analysis between pairs of shadow operations: if two shadow operations do not commute, then a restriction between them is added to the returning restriction set (line 5-6).

To discover restrictions for preserving invariants, we could exhaustively explore all `I-conflict` sets consisting of concurrent shadow operations that trigger violations. However, it is very challenging to achieve this

```
//each permission consists of a set of operations
Permission p;

//receive a set of operations that need to be monitored
Permission getPermission(TxnId tid, String opName);

//wait until the set of operations in p have been applied
void waitForBeingExecuted(TxnId tid, Permission p);

//clean up all required resources occupied
void cleanUp(TxnId tid);
```

Figure 3: Olisipo coordination policy interface

since there might exist a large number of violating executions containing at least one I-conflict set. To make this exploration more efficient, we first collapse many similar executions of a replicated system into a single execution class, and then perform a weakest precondition and postcondition analysis over these classes [23].

In particular, for every shadow operation u, we denote $u.wpre$ as its weakest precondition, a condition on the initial state ensuring that u always preserves invariants. We also denote $u.post$ as the postcondition that captures the side effects of operations through a condition that always holds after the operation is executed. In Alg. 2, we flag a set of shadow operations T as `I-conflicting` if either of the following two conditions is met: (a) T contains a single operation t and t is self-conflicting, i.e., $t.wpre$ is invalidated by $t.post$ (line 12-14); or (b) $|T| > 1$, any subset of T is not `I-conflicting` (but can be self-conflicting) and there exists an operation u from T such that $u.wpre$ can be invalidated by the compound postcondition of all the operations in $T \setminus \{u\}$ (line 16-26).

Once these `I-conflict` sets are determined, then for each such set T, we add a restriction between an arbitrary pair of shadow operations from T if no pair of operations from that set was previously restricted (line 8-9). Otherwise, T will be skipped since the preexisting restriction suffices to preserve invariants. In addition, for shadow operations that are self-conflicting, we have to place a restriction between pairs of shadow operations of that type (line 6-7).

5 Design and Implementation of Olisipo

Several coordination protocols can be used for enforcing a given restriction, such as Paxos, distributed locking, or escrow techniques. However, depending on the observed runtime frequency of operations confined by a restriction, different approaches lead to different performance.

In the previously mentioned auction example, maintaining the invariant that winners always match highest successful bidders requires a restriction between any pair of `placeBid'` and `closeAuction'` operations. A simple scheme would be forcing instances of either operation to pay the same coordination cost. However, since `placeBid'` is likely to be more prevalent than `closeAuction'`, reducing the latency for `placeBid'` and penalizing `closeAuction'` is likely to lead to better performance.

To address this, we propose a coordination service called Olisipo offering a range of coordination policies, each of which presents a trade-off between the cost of each operation and the overall cost. This service allows us to use runtime information about the relative frequency of operations to select an efficient coordination mechanism for a given restriction.

5.1 Coordination protocols

Olisipo supports two built-in protocols, namely `symmetric (Sym)` and `asymmetric (Asym)`, but can be extended with customized coordination policies, which need to be compatible with our interface (Fig. 3). The difference between the two protocols is that, in the case of `Sym`, given a restriction $r(u,v)$ between two operations u and v, the protocol requires both u and v to coordinate with each other for establishing an order between them, whereas the `Asym` protocol allows one of them to proceed by default, while requiring the other to obtain permission before proceeding.

Sym. This protocol requires to set up a logically centralized counter service, which maintains, for each restriction $r(u,v)$, two counters c_u and c_v. Each one represents the total number of operations of the corresponding type that have been accepted by the underlying system. Additionally, every replica at different data centers maintains a local copy of these counters, representing the number of operations of each type that have been executed by that replica. Initially, all local copies, as well as the global counters, have all values set to zero. Whenever an operation is received by a replica, that replica contacts the counter service to increase the corresponding centralized counter and get a fresh copy of the counter maintained for both types of operations. Upon receiving the reply from the counter service, that replica compares the received values with its local copy. If they are the same, then the replica can execute the operation without waiting. Otherwise, the local execution can only take place when all missing operations have been locally replicated. To make the counter service fault tolerant, we leverage a Paxos-like state machine replication library (BFT-SMART [18]) to replicate counters across geo-locations.

Asym. Unlike the above centralized solution, the asymmetric protocol implements distributed barrier in a decentralized manner. Assume, for instance, that u is the barrier. In this case, whenever a replica r receives an operation u it would have to enter the barrier, and contact all other replicas to request participation. This requires all replicas in the system to stop processing operations of type v and enter the barrier. After receiving an acknowledgment of the barrier entrance from all replicas, r can execute the operation, and then notify all replicas that it has left the barrier (while at the same time propagating

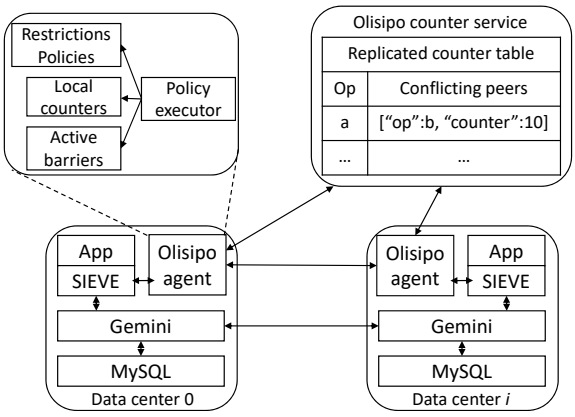


Figure 4: Olisipo architecture

the effects of the operation u it has just executed). Such a coordination strategy might incur a high overhead; however, this is beneficial when one of the two operations in the restriction is rarely submitted to the system.

5.2 Implementation details

As depicted in Fig. 4, the Olisipo architecture consists of a counter service replicated across data centers and a local agent deployed in each data center. The counter service is required only for the `Sym` protocol, whereas the local agent enforces the coordination that is needed by both protocols. To this end, every agent also stores some meta data required for different protocols: for the `Sym` protocol, it maintains a local copy of the replicated counter service, which is used for learning if the local counters lag behind the global counters, which means the corresponding data centers have to wait until all missing operations have been locally incorporated. For the `Asym` protocol, every agent maintains a list of active barriers, which are used for locally deciding if relevant operations blocked on such barriers can proceed. Note that these protocols are not optimized for performance, but nonetheless suffice to demonstrate the benefits of using PoR consistency and enforcing it in a way that takes into account frequency of different operation types.

We implemented Olisipo using Java (2.8k lines of code), linked with BFT-SMART [39] for replicating the centralized counter service and MySQL as the backend storage. We integrated Olisipo with our prior prototypes for Gemini [5] and SIEVE [6], so that Gemini serves as the underlying causally consistent replication tier while SIEVE is used to produce commutative shadow operations at runtime. The code of Olisipo is available at [11].

Workflow. User requests are directed to an application server running at the local data center, which executes the corresponding generator operation. The result is a commutative shadow operation, which is then forwarded to the local Olisipo agent for placing coordination if needed before committing; if the coordination allows for that serialization (which is determined according to the specific protocol for enforcing it) then the shadow operation is sent to Gemini for replicating it across all data centers; otherwise the generator operation must be retried in a new serial order.

6 Evaluation

In our experimental evaluation, we first try to understand if the methodology for inferring restrictions presented in Sec. 4 is effective when applied to real world applications, i.e., it finds a minimal set of restrictions. Finally, we try to assess the impact on latency and system throughput introduced by three factors: adopting PoR consistent replication, using different protocols, and adding more restrictions.

6.1 Case study

Next, we report our experience on discovering restrictions in RUBiS. RUBiS is a fairly simple auction-like benchmark. The original benchmark we used as a starting point contained only 16 transactions and did not include an operation to declare the winner of an auction, so we added a close auction functionality. In the future, we intend to explore other benchmarks with more complex OLTP or OLAP queries.

State convergence. Given that we deploy RUBiS with SIEVE, all shadow operations generated at runtime commute w.r.t. each other by construction, and there is no need modify the application nor restrict any pair of shadow operations for state convergence purposes. All that is required in SIEVE is to specify the desired conflict resolving semantics by choosing from a set of built-in solutions [29].

Invariant preservation. We manually perform the procedure of identifying restrictions to make a geo-replicated RUBiS deployment invariant preserving, as previously presented in Section 4.3. (We leave the automation of this step as future work.) In particular, we determined four invariants of RUBiS, namely (a) identifiers assigned by the system are unique; (b) nicknames chosen by users are unique; (c) item stock must be non-negative; and (d) the auction winner must be associated with the highest bid across all accepted bids. We continued by manually determining the weakest preconditions and postconditions of all RUBiS shadow operations. Those conditions are summarized in Tab. 1 and used by the `I-conflict set` analysis (Alg. 2). With regard to the first invariant, since we take advantage of the coordination-free unique identifier generation method offered by SIEVE, no `I-conflict sets` were found for violating it. In turn, for the remaining three invariants, we identified the following `I-conflict` sets:

- $\{registerUser', registerUser'\}$. Invariant (b) would be violated if the two operations proposed the same *nickname* and were submitted to different sites simultaneously;

$placeBid'$ $(itId, cId, bid)$	wp	$\exists u \in item\_table.\ u.id = itId \wedge u.status = open$	valid auction
	post	$bidTable = bidTable \cup \{< itId, cId, bid >\}$	new bid placed
$closeAuction'$ $(itId, wId)$	wp	$\exists w \in bidTable.\ w.cId = wId \wedge \forall v \in bidTable \setminus \{w\}.\ w.bid > v.bid$	highest accepted bid
	post	$winnerTable = winnerTable \cup \{< itId, wId >\}$	winner declared
$registerUser'$ $(uId, username)$	wp	$\forall u \in user\_table.\ u.name <> username$	username not seen before
	post	$user\_table = user\_table \cup \{< uId, username >\}$	new user added
$storeBuyNow'$ $(itId, delta)$	wp	$\exists u \in item\_table.\ u.id = itId \wedge u.stock >= delta$	enough stock left
	post	$u.stock -= delta$	delta applied

Table 1: Weakest preconditions and postconditions of selected shadow operations of RUBiS

RedBlue consistency	PoR consistency
$r(registerUser', registerUser')$	$r(registerUser', registerUser')$
$r(storeBuyNow', storeBuyNow')$	$r(storeBuyNow', storeBuyNow')$
$r(placeBid', placeBid')$	$r(placeBid', closeAuction')$
$r(closeAuction', closeAuction')$	
$r(placeBid', closeAuction')$	
$r(registerUser', storeBuyNow')$	
$r(registerUser', placeBid')$	
$r(registerUser', closeAuction')$	
$r(storeBuyNow', placeBid')$	
$r(storeBuyNow', closeAuction')$	

Table 2: Restrictions required when replicating the extended RUBiS under RedBlue or PoR consistency

- $\{storeBuyNow', storeBuyNow'\}$. Invariant (c) would be violated if both operations simultaneously subtracted some number of items from *stock*, and the sum of the purchases exceeded the previous *stock* value;
- $\{placeBid', closeAuction'\}$. Invariant (d) would be violated if both operations were submitted at the same time to different sites, and $placeBid'$ carried a higher bid than all accepted bids.

Each `I-conflict set` above covers a class of violating executions of the respective invariant. To eliminate the corresponding violations, we added three restrictions, namely $r(registerUser', registerUser')$, $r(storeBuyNow', storeBuyNow')$ and $r(placeBid', closeAuction')$. In Tab.2 we compare to the PoR consistency solution with using RedBlue consistency. The latter solution would require more restrictions, since the definition states that all non-invariant safe shadow operations must be strongly consistent, i.e., the four shadow operations presented in the above list must be restricted in a pairwise fashion.

We assign the `Sym` protocol to coordinate shadow operations confined by all these restrictions except $r(placeBid', closeAuction')$. This is because $placeBid'$ is significantly more prevalent than $closeAuction'$ in RUBiS, e.g., in a bidding mix workload, the ratio of the number of $closeAuction'$ to the number of $placeBid'$ is only 2.7%. Therefore, we assign the `Asym` protocol to coordinate this restriction and additionally make $closeAuction'$ act as the barrier.

6.2 Experimental setup

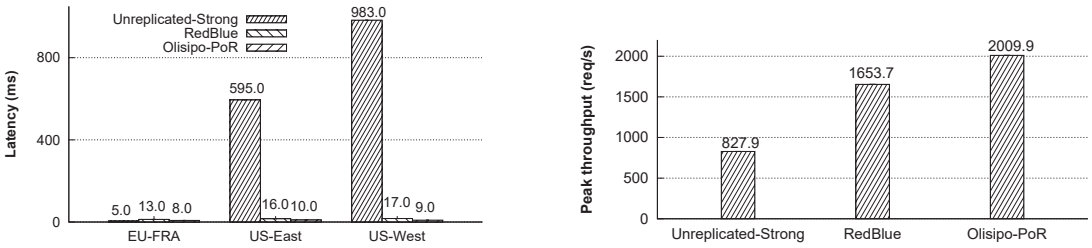
We run experiments on Amazon EC2 [1] using `m4.2xlarge` virtual machine instances located in three sites: US Virginia (US-East), US California (US-West) and EU Frankfurt (EU-FRA). Table 3 shows the average round trip latency and observed bandwidth between every pair of sites. Each VM has 8 virtual cores and 32GB of RAM. VMs run Debian 8 (Jessie) 64 bit, MySQL 5.5.18, Tomcat 6.0.35, and OpenJDK 8 software.

	US-East	US-West	EU-FRA
US-East	0.299 ms 1052.0 Mbps	71.200ms 47.4 Mbps	88.742 ms 29.6 Mbps
US-West	66.365 ms 47.4 Mbps	0.238 ms 1050.7 Mbps	162.156 ms 17.4 Mbps
EU-FRA	88.168 ms 36.2 Mbps	162.163 ms 20.1 Mbps	0.226 ms 1052.0 Mbps

Table 3: Average round trip latency and bandwidth between Amazon data centers

Configuration and workloads. Unless stated otherwise, in all experiments, we deploy the BFT-SMART library under the crash-fault-tolerance model (CFT) with 3 replicas across three sites, and assign the replica at EU-FRA to act as the leader of the consensus protocol. We replicate RUBiS under PoR consistency across three sites using the previously mentioned combination of Olisipo, SIEVE, and Gemini. As additional baselines, we run an unreplicated strongly consistent RUBiS in the EU-FRA site, and a 3 site RedBlue consistency deployment, in which we replicate RUBiS via the PoR consistency framework but with the set of restrictions required by RedBlue consistency (shown in Tab.2). We refer to these three setups as "*Olisipo-PoR*", "*Unreplicated-Strong*", and "*RedBlue*", respectively. For all experiments, emulated clients are equally distributed across three sites and connect to their closest data center according to physical proximity.

We choose to run the bidding mix workload of RUBiS, where 15% of user interactions are updates. To allow the client emulator to issue the newly introduced `closeAuction` requests, we have to slightly change the transition table in the original RUBiS code by assigning a positive probability value for this request. The new transition table can be found here [10]. For all experiments we vary the workload by increasing the number of concurrent client threads in every client emulator, and disable the `thinking time` option so that there is no waiting time between two contiguous requests from the same client thread. We populate the data set via the following parameters: the RUBiS database contains 33,000 items for sale, 1 million users, and 500,000 old items.



(a) Overall average latency **(b)** Peak throughput

Figure 5: Performance comparison between three system configurations

6.3 Results

6.3.1 Average user observed latency

The main advantage of adopting PoR consistent replication with Olisipo is to reduce user-perceived latency. To assess this improvement, we start by analyzing the average latency for users at each data center. In this set of experiments, each user issues a single request at a time in a closed loop.

As shown in Fig.5(a), all users except those in EU-FRA observe a lower latency in the *Olisipo-PoR* and *RedBlue* configurations, compared to the users from the same locations in the *Unreplicated-Strong* configurations. This improvement is because, under both PoR and RedBlue consistency, most requests are handled locally within a data center, whereas in the unreplicated setting, requests from users at the two US data centers have to communicate with EU-FRA, which incurs an expensive inter-datacenter communication. At a more detailed level, in the *Unreplicated-Strong* experiment, the raw latency values perceived by users at both US-East and US-West are higher than the round-trip time from the user to the server site (EU-FRA) because processing each request involves sending one or more images to the user.

Compared to *RedBlue*, *Olisipo-PoR* improves the average latency for users at the three sites by 38.5%, 37.5% and 47.1%, respectively. We further observed that users at EU-FRA in the replicated experiments experience a higher latency than users from the same region accessing an unreplicated RUBiS. This is due to the additional work required for incorporating remote shadow operations into the local causal serialization and placing coordination when needed for serializing conflicting requests. Note that although the user observed latency for *Olisipo-PoR* at EU-FRA is almost twice as large as the latency of the unreplicated setup, the absolute number (9 ms) is reasonably low.

6.3.2 Peak throughput

We now focus on the improvement in scalability with the client load achieved by PoR consistency. Fig.5(b) shows the peak throughput achieved by the three configurations, which is measured when the corresponding system is saturated. The improvement of the *Olisipo-PoR* deployment is 1.43*X* when compared to the *Unreplicated-Strong* setup. This increase in throughput is because PoR consistency offers fine-grained consistency so that only a minority of requests need to pay the coordination cost, while the remaining can be processed locally. Compared to a RedBlue consistent RUBiS, the PoR consistent version increases peak throughput by 21.5%, since PoR consistency avoids the cost for coordinating several restrictions required by RedBlue consistency (shown in Tab.2).

6.3.3 Per request latency

Next, we evaluate the per request latency of RUBiS requests. For this round of experiments, each site runs a single user issuing a request at a time.

Latency of non-conflicting requests. Among all RUBiS non-conflicting requests, we chose one representative request called `storeComment`, which places a comment on a user profile, as the illustrating example. Fig.6(a) shows that PoR consistent RUBiS makes users across the three sites observe evenly low latency, and the speedup in the user observed latency for the remote users located at US-East and US-West is 84.9*x* and 106.8*x*, respectively, compared to the *unreplicated* strongly consistent deployment. These performance gains happen because, under PoR consistency, the `storeComment` request requires no coordination and can be processed locally. In contrast, in the *unreplicated* experiment, users at the two US sites have to contact the server at EU-FRA and thus perceive a higher latency. We also notice that users from EU-FRA in both experiments have almost identical latency, which is different from the results in Fig.5(a), since the cost of generating and applying the shadow operations of the `storeComment` request is modest.

Latency of conflicting requests. Next, we shift our attention from non-conflicting requests to conflicting ones. As introduced before, Olisipo uses two different protocols (`Sym` and `Asym`) to coordinate conflicting requests. We start by analyzing the latency of requests handled by the `Sym` protocol. The illustrative example we selected is `storeBuyNow`, which produces self-conflicting shadow operations. As shown in Fig.6(b), the user observed latency of the `storeBuyNow` request at all three sites is significantly higher than the latency of `storeComment` (shown in Fig.6(a)), which is a non-conflicting request. This is because most of the lifecycle of these requests was spent asking permission to the centralized counter

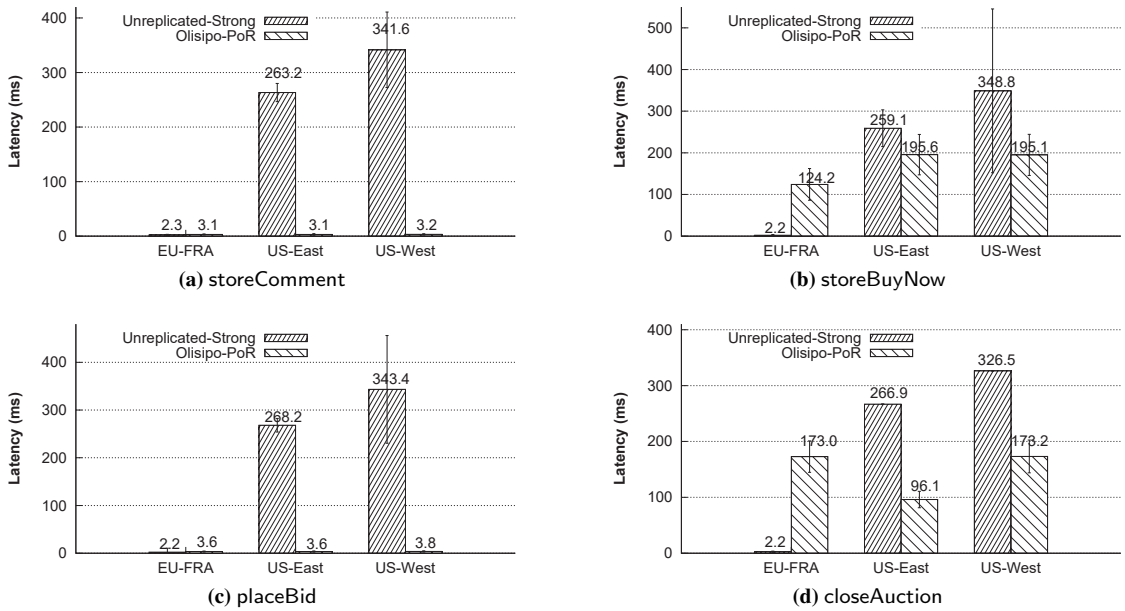


Figure 6: Average latency bar graph of four requests for users at three sites. `storeComment` produces non-conflicting shadow operations, while the ones of `storeBuyNow` conflict w.r.t themselves and are regulated by the `Sym` protocol. `placeBid` and `closeAuction` produce two conflicting shadow operations regulated by the `Asym` protocol.

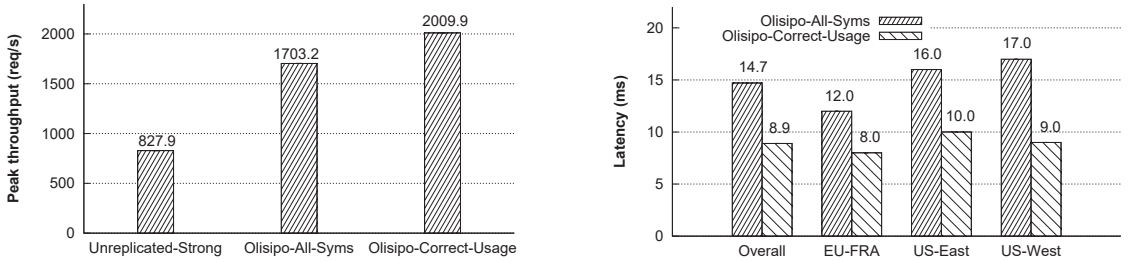
service, which consists of 3 replicas spanning three sites and executing a Paxos-like consensus protocol. Additionally, user observed latency at EU-FRA is lower than the remaining two sites, since the leader of the consensus protocol is co-located with EU-FRA users.

We continue by analyzing the average latency of requests that are coordinated by the `Asym` protocol. Unlike the `Sym` protocol, any pair of operations confined in a restriction will be treated differently by the `Asym` protocol, since one acts as a distributed barrier and the other proceeds if no active barriers are running. In Sect. 6.1, we assigned the `Asym` protocol to regulate the $r(placeBid', closeAuction')$ restriction, while selecting the less frequent shadow operation $closeAuction'$ as a barrier. As shown in Fig.6(c), the average latency measured for the *placeBid* request, which produces $placeBid'$, is very similar to the results obtained for non-conflicting requests shown in Fig.6(a). This is because the ratio of *closeAuction* to *placeBid* is very low and most of the time the *placeBid* request commits immediately without waiting for joining or leaving barriers.

Next, we consider the barrier request *closeAuction* handled by the `Asym` protocol. As expected, Fig.6(d) shows that, compared to *placeBid*, the average latency of *closeAuction* is noticeably higher due to the coordination across sites, through which this request forces all sites not to process incoming *placeBid* requests and collects results of all relevant completed *placeBid* requests. We also notice that users issuing *closeAuction* observed a latency that is slightly higher than the maximum RTT between their primary site and the remaining sites. For example, as shown in Tab.3, the maximum RTT for US-East users to the other two sites is 88.7 ms, while the average latency of *closeAuction* observed by the same group of users is 96.1 ms.

6.3.4 Impact of different protocols

The purpose of offering different coordination protocols is to improve runtime performance by taking into account the workload characteristics. To validate this, we first deploy an experiment denoted by `Olisipo-Correct-Usage`, in which we take into account the runtime information that $closeAuction'$ occurs sparsely and assign the `Asym` protocol to regulate the restriction $r(placeBid', closeAuction')$. We then deploy another experiment denoted by `Olisipo-All-Syms`, in which the restriction $r(placeBid', closeAuction')$ is handled by the `Sym` protocol. Fig. 7 summarizes the comparison of peak throughput and average latency among three experiments, namely `Unreplicated-Strong`, `Olisipo-All-Syms` and `Olisipo-Correct-Usage`. The `Olisipo-All-Syms` setup improves the peak throughput of the unreplicated RUBiS system by 105.7%, because of the coordination-free execution of non-conflicting requests. However, compared to `Olisipo-Correct-Usage`, the performance of `Olisipo-All-Syms` degrades in two dimensions, namely a 15.3% decrease in peak throughput and a 65.2%, 50.0%, 60.0%, 88.9% increase in request latency for all, EU-FRA, US-East, US-West users, respectively. The reason for this performance loss is as follows: every `placeBid'` shadow operation in `Olisipo-All-Syms` requires a communication step between its



(a) Peak throughput **(b)** Overall average latency

Figure 7: Peak throughput and overall average latency bar graphs of systems using different protocols.

primary site and the centralized counter service for being coordinated, while most of time `placeBid'` shadow operations in `Olisipo-Correct-Usage` work as non-conflicting requests provided that `closeAuction` requests sparsely arrive in the system.

7 Related work

In the past decades, many consistency proposals focused on reducing coordination among concurrent operations to improve scalability in geo-replicated systems [25, 40, 30, 29, 14, 15, 42, 12, 4, 32]. However, they only allow the programmer to choose from a limited number of consistency levels that they support, such as strong, causal or eventual consistency. Unlike these approaches, PoR consistency offers a fine-grained tunable trade-off between performance and consistency using the visibility restrictions between pairs of operations to express consistency semantics. Some of these proposals for consistency models with reduced coordination also analyzed or even enforced conditions for ensuring state convergence despite the lack of coordination [14, 25, 40, 15, 32, 40]. In addition to state convergence, our solution also analyzes invariant preservation.

In the space of consistency proposals that looked into how to enforce application-specific invariants, Bailis et al. [16] proposed I-confluence, which avoids coordination by determining if a set of transactions are I-confluent, i.e., if the integrity constraints might be violated when they are executing without coordination. Indigo [17] defines consistency as a set of invariants that must hold at any time, and presents a set of mechanisms to enforce these invariants efficiently on top of eventual consistency. Similar to Indigo, warranties [31] map consistency requirements to a set of assertions that must hold in a given period of time, but it needs to periodically invalidate assertions when updates arrive. Roy et al. additionally propose a program analysis against transaction code for producing warranties [35]. In contrast, PoR consistency takes an alternative approach by modeling consistency as restrictions over operations.

A few proposals map consistency semantics to the ordering constraints defined over pairs of operations. Generic Broadcast defines conflict relations between messages for fast message delivery, which are analogous to visibility restrictions used in our solution [34]. However, they do not analyze how to determine the conditions for ensuring invariant preservation. The recent work of Gotsman et al. [24] encodes the concept of a conflict relation into a proof system, which allows for analyzing if consistency choices expressed as conflict relations is sufficient for enforcing application invariants. In comparison, our work makes three contributions. First, our methods allow to find a minimal set of restrictions to be used. Second, we propose a set of coordination methods that adapt to the workloads in order to be more efficient. Third, we present the design and implementation of a complete system that offers PoR consistency.

Some variants of Paxos [26] have explored operation semantics to relax the need to process all operations in the same sequential order. Generalized Paxos allows replicas to execute commutative operations in different orders [27]. EPaxos uses dependencies between pairs of operations to order concurrent conflicting requests [33]. Our work differs from these Paxos variants in that we develop an analysis to extract pairs of conflicting operations by considering the impact of concurrent executions on achieving state convergence and invariant preservation. Furthermore, unlike these protocols, in our work, operations that are not confined by conflicting relations can be first accepted in a single replica and later asynchronously replicated to other replicas.

Finally, our own previous workshop paper described the motivation and a high-level overview of a solution to this problem [28].

8 Conclusion

In this paper, we proposed a technique for achieving convergence and invariant-preservation in geo-replicated systems with a minimal amount of coordination. This combines a new generic consistency model called PoR consistency, an analysis for determining a minimal set of restrictions, and a coordination service called Olisipo for efficiently serializing pairs of operations. Our evaluation of running RUBiS with different setups shows that the joint work of PoR consistency and Olisipo significantly improves the performance of geo-replicated systems.

Acknowledgments

We sincerely thank Robbert van Renesse, Jiawei Wang, Xinyu Feng, and the anonymous reviewers for their insightful comments and suggestions. The research of C. Li is supported by the Fundamental Research Funds for the Central Universities (Grant no. WK2150110011). This work is supported by the Portuguese Fundação para a Ciência e a Tecnologia through projects UID/CEC/50021/2013 and UID/CEC/04516/2013 and by the EU H2020 LightKone project (732505).

References

[1] Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2/. [Online; accessed Jan-2018].

[2] Amazon S3 Introduces Cross-Region Replication. https://aws.amazon.com/about-aws/whats-new/2015/03/amazon-s3-introduces-cross-region-replication/. [Online; accessed Jan-2018].

[3] Amazon Web Services (AWS) - Cloud Computing Services. http://aws.amazon.com/. [Online; accessed Jan-2018].

[4] Balancing Strong and Eventual Consistency with Google Cloud Datastore. https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore/. [Online; accessed Jan-2018].

[5] Gemini Code Repository. https://github.com/pandaworrior/RedBlue_consistency. [Online; accessed Jan-2018].

[6] SIEVE Code Repository. https://github.com/pandaworrior/SIEVE. [Online; accessed Jan-2018].

[7] Extended version with proofs. https://github.com/mr-cheng-li/por_tr. [Online; accessed Feb-2018].

[8] Google Webpage. www.google.com. [Online; accessed Jan-2018].

[9] Microsoft US — Devices and Services. www.microsoft.com/. [Online; accessed Jan-2018].

[10] Modified RUBiS Transition Table. https://github.com/pandaworrior/VascoRepo/blob/master/vasco/config/vasco_transitions_3.xls. [Online; accessed Jan-2018].

[11] Olisipo code repository. https://github.com/pandaworrior/VascoRepo. [Online; accessed Jan-2018].

[12] Welcome to Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction. [Online; accessed Jan-2018].

[13] Welcome to Facebook - Log In, Sign Up or Learn More. https://www.facebook.com/. [Online; accessed Jan-2018].

[14] Alvaro, P., Conway, N., Hellerstein, J. M., and Maier, D. Blazes: Coordination Analysis for Distributed Programs. In *Proceedings of the IEEE 30th International Conference on Data Engineering* (2014), ICDE'14.

[15] Alvaro, P., Conway, N., Hellerstein, J. M., and Marczak, W. R. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research* (2011), CIDR'11.

[16] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Coordination Avoidance in Database Systems. *Proc. VLDB Endow. 8*, 3 (Nov. 2014), 185–196.

[17] Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N., Najafzadeh, M., and Shapiro, M. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 6:1–6:16.

[18] Bessani, A., Sousa, J. a., and Alchieri, E. E. P. State Machine Replication for the Masses with BFT-SMART. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2014), DSN '14, IEEE Computer Society, pp. 355–362.

[19] Burckhardt, S., Gotsman, A., and Yang, H. Understanding Eventual Consistency. Tech. Rep. MSR-TR-2013-39, March 2013.

[20] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., Haq, M. F. u., Haq, M. I. u., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., and Rigas, L. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 143–157.

[21] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[22] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[23] Dijkstra, E. W. *A Discipline of Programming*, 1st ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[24] Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., and Shapiro, M. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2015), POPL'15, ACM.

[25] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. Providing High Availability Using Lazy Replication. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 360–391.

[26] Lamport, L. The Part-time Parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[27] Lamport, L. Generalized Consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2005.

[28] Li, C., Leitão, J. a., Clement, A., Preguiça, N., and Rodrigues, R. Minimizing Coordination in Replicated Systems. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data* (New York, NY, USA, 2015), PaPoC '15, ACM, pp. 8:1–8:4.

[29] Li, C., Leitão, J. a., Clement, A., Preguiça, N., Rodrigues, R., and Vafeiadis, V. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the*

2014 USENIX Conference on USENIX Annual Technical Conference (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 281–292.

[30] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N., AND RODRIGUES, R. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 265–278.

[31] LIU, J., MAGRINO, T., ARDEN, O., GEORGE, M. D., AND MYERS, A. C. Warranties for Faster Strong Consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 503–517.

[32] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. 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 Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[33] MORARU, I., ANDERSEN, D. G., AND KAMINSKY, M. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 358–372.

[34] PEDONE, F., AND SCHIPER, A. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing* (1999), DISC '99.

[35] ROY, S., KOT, L., BENDER, G., DING, B., HOJJAT, H., KOCH, C., FOSTER, N., AND GEHRKE, J. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1311–1326.

[36] SCHURMAN, E., AND BRUTLAG, J. Performance Related Changes and their User Impact. http://slideplayer.com/slide/1402419/, 2009. Presented at *Velocity Web Performance and Operations Conference*. [Online; accessed Jan-2018].

[37] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Tech. Rep. 7506, INRIA, Jan. 2011.

[38] SHARMA, Y., AJOUX, P., ANG, P., CALLIES, D., CHOUDHARY, A., DEMAILLY, L., FERSCH, T., GUZ, L. A., KOTULSKI, A., KULKARNI, S., KUMAR, S., LI, H., LI, J., MAKEEV, E., PRAKASAM, K., VAN RENESSE, R., ROY, S., SETH, P., SONG, Y. J., VEERARAGHAVAN, K., WESTER, B., AND XIE, P. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 351–366.

[39] SOUSA, J., ALCHIERI, E., AND BESSANI, A. BFT-SMART Code Repository. https://github.com/bft-smart/library. [Online; accessed Jan-2018].

[40] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 385–400.

[41] VOGELS, W. Eventually Consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44.

[42] ZHANG, I., SHARMA, N. K., SZEKERES, A., KRISHNAMURTHY, A., AND PORTS, D. R. K. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 263–278.