Fine-grained consistency for geo-replicated systems

abstract

To deliver fast responses to worldwide users, the major Internet providers deploy geo-replication techniques to serve as many requests at data centers in close proximity to users as possible. However, 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 tradeoff 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

Users are expecting fast responses from services they are interacting with [36]. To cope with this unprecedented demand from a large user base, many Internet service providers such as Google [6], Microsoft [7], Facebook [11] and Amazon [3] not only replicate data in a single data center (region), but also across multiple geographically dispersed data centers [38, 20, 19, 2]. Unfortunately, geo-replication also leads to an inherent tension between achieving high-performance and ensuring application-specific 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 invariants to be preserved). Given that latency and connectivity among data centers are so constrained, the negative effects imposed by maintaining strong consistency on user observed latency and overall system scalability become worse [21, 41, 29, 16, 15].

To relieve the above fundamental tension in geo-distributed scenarios, there are some proposals for 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 [21, 42, 18] or causal consistency [31]; some approaches allow multiple consistency levels to coexist [29, 16, 10, 4] in a single system. One of the representatives of the latter case is our prior proposal—RedBlue consistency [29], under which some operations must be executed under strong consistency (and therefore incur in a high performance penalty) while other operations can be executed under weaker consistency (namely causal consistency [31]). 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 one can be weakly consistent.

This binary classification methodology is effective for many web 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 [33] and *token* by Gotsman et al. [22] allow for finer-grained coordination of operations, these 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 this limitation, 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 in the context of PoR consistency is achieved by imposing fewer or more restrictions on pairs of operations.

The key to making a geo-replicated setting of a given application fast as possible under PoR consistency 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. Nevertheless, this is challenging because missing required restrictions will lead 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 restrictions while avoiding unnecessary ones.

Furthermore, from a protocol implementation perspective, we observe that given a set of restrictions over pairs of operations, there is no one-fit-all coordination policy to enforce them. In fact, there exist several coordination techniques/protocols that can be used for enforcing a given restriction, such as Paxos, distributed locking, or global barriers. However, depending on the frequency over time in which the system receives operations confined by a restriction, different coordination approaches lead to different performance tradeoffs. For instance, geo-replicating an auction site, to preserve an appliation specific invariant that the declared winner when closing an auction must be the bidder with highest accepted bid, the developer has to place a restriction between two operations `close_acution` and `place_bid`. However, as the frequencies of these two operations are not balanced, i.e., `close_acution` is rare and `place_bid` is dominating, asking them to pay the same amount of coordination cost is not ideal. 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 deployment characteristics at runtime.

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 this web application with Olisipo, and compared against the results we obtained from the RedBlue consistent version. Our experimental results show that PoR consistency requires fewer restrictions than RedBlue consistency, and the usage of PoR consistency 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, i.e., a function from the previous state to the next state, i.e.:

$$u \in \mathscr{U} : \mathscr{S} \rightarrow (\mathscr{S} \rightarrow \mathscr{S})$$

To simplify the notation, 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 be first executed in a *sandbox* against the current state of the replica at the user's primary site, without inteference from other concurrent operations. In this phase, the execution only identifies what changes u would introduce to the system against state S that is observed by u and will not commit these changes. At the end of executing h_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 a **state convergence** property. 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 work RedBlue consistency [29] is based on an explicit division of shadow operations into blue operations whose order of execution can vary from site to site, and red operations that must be executed in the same relative order at all sites. For guiding developers in making use of RedBlue consistency, this work also identified that

```
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.

```
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.

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

(b) Shadow placeBid' operation.

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

(d) Shadow closeAuction' operation.

Figure 1: Pseudocode for the original and shadow operations of the `placeBid` and `closeAuction` transactions in an auction site example.

a sufficient condition for ensuring the state convergence property is that a shadow operation must be labeled red if it is not globally commutative. In addition to state convergence, a second sufficient condition was identified, for ensuring that invariants are maintained, stating the following: the replicas will only transition between valid states if we label as red all shadow operations that may violate an invariant when applied against a different state from the one they were generated from. For the remaining shadow operations, which have passed the two condition checks, we can safely label them blue.

3 Partial Order-Restriction 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 for an item, declaring 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 shadow operations of the two prior operations, respectively, guaranteeing that these shadow operations apply changes in a commutative fashion regardless of execution order.

When applying RedBlue consistency to replicate such an auction service, we note that the concurrent execution under weak consistency of a `placeBid` operation with a bid that is higher than all accepted bids and a `closeAuction` operation 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 in 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 classification 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

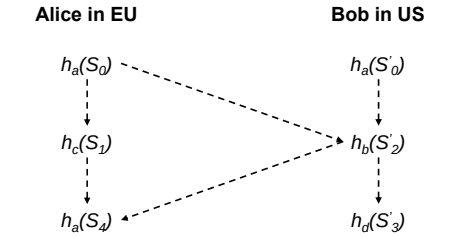
The definition of PoR consistency includes three important components: (1) a set of restrictions, which specify the visibility relations between pairs of operations; (2) a restricted partial order (or short, R-order), which establishes a (global) partial order of operations respecting operation visibility 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 $U \times U$.*

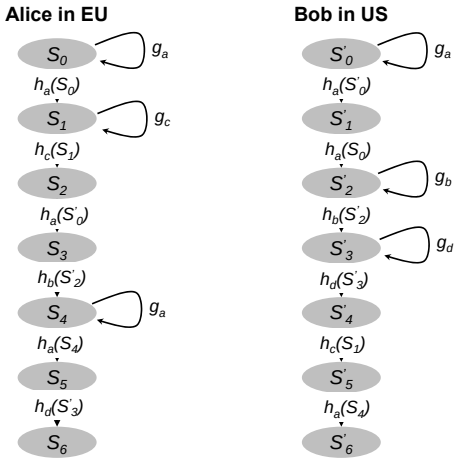
For any two operations u and v in U, if there exists a restriction relation between them, we denote this relation as $r(u,v)$.

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 also 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



(a) Restricted partial order O of shadow operations



(b) Causal serializations of O

Figure 2: Restricted partial order and 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 represent generator operations.

each individual replica at every site will behave according to this global view. To introduce these local views, we have to recall the system model our consistency definition is built on. 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. 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 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 legal serialization of O;*
- *For any $h_v(S) \in U$ generated by $g_v \in V_i$, S is the state obtained after applying the sequence of shadow operations preceding h_v in O_i;*
- *For any $g_v \in V_i$ and $h_u(S) \in U$, $h_u(S) <_i g_v$ 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. Since 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, h_b, it realizes that the shadow operation it would generate may need to be restricted w.r.t a concurrent shadow operation initially triggered at the EU site. As a result, h_b at the US site must wait until the respective concurrent shadow operation $h_a(S_0)$ gets propagated from the EU site to Bob's site. Then h_b will read the state introduced by locally applying $h_a(S_0)$ from Alice, and produce a shadow operation $h_a(S'_2)$. Note that this production will establish a dependency between $h_a(S_0)$ and $h_a(S'_2)$ (as shown in Fig. 2a), thus enforcing that they cannot be applied in different relative orders in different 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 in RedBlue consistency 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 small set of restrictions for making the replicated service eventually converge and never violate invariants so that the amount of required coordination is minimal. With regard to state convergence, we take a similar methodology adopted in prior research [37, 29, 28], which is to check operation commutativity. However, 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.

To always 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 isolate the operations that exclusively contribute to an invariant violation from the rest. To do so, we introduce a new concept, called an `I-conflict set`, which defines 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 by removing any shadow operation from such a set, the violation will no longer persist. To identify a minimal set of restrictions, we first perform an analysis over any subsets of the shadow operation set to discover all `I-conflict` sets. Then, for any such set, 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 1 *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]

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. We call this set an `invariant-conflict operation set`, or short, `I-conflict set`. After these are identified, 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 5 (Invariant-conflict operation set) *A set of shadow operations G is an* invariant-conflict operation set *(or short,* 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., $P = (G \setminus \{u\}, <)$, and $\exists$ 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 necessarily to be included in G, as even after excluding the request from either *Bob* or *Alice*, the violation still remains. According to Definition 5, $\{placeBid', closeAuction'\}$ is an `I-conflict set`, while $\{placeBid', placeBid', closeAuction'\}$ is not. Intuitively, avoiding invariant violations is to prevent all operations from the corresponding `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 then can formulate the invariant preservation property into the following theorem.

Theorem 2 *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 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*

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

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)}
       return R
```

- *for any pair of shadow operations u and v, $r(u,v)$ in $R_{\mathscr{S}}$ if u and v don't commute.*

4.3 Identifying restrictions

As discussed in the previous subsection, the key to making a replicated system adopt PoR consistency and strike an appropriate balance between performance and consistency semantics is to identify a finest set of restrictions, which ensure both state convergence and invariant preservation. With regard to the former property, we design a state convergence restrictions discovery method (Algorithm 1), which performs an operation commutativity analysis between pairs of operations. If two operations do not commute, then a restriction between them is added to the returning result restriction set.

For discovering the required restrictions for invariant preservation, we have to exhaustively explore all `I-conflict` sets that trigger violations. However, it is very challenging to achieve this since there might exist infinite number of violating executions containing at least one I-conflict set. Therefore, the exploration may not guarantee to terminate. To solve this problem, we decide to take a more efficient approach, in which we collapse many similar executions of a replicated system into a single execution class. To do so, we use programming language techniques such as weakest precondition and postcondition analysis. For every operation u, we denote $u.wpre$ as its weakest precondition, which is a condition on the initial state and the parameter values ensuring that u always preserves invariants. We also denote $u.post$ as the postcondition summarizing the final state after the execution of u against all possible valid state. We flag a set of operations T as `I-conflict` 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 $t.post$; and (b) $|T| > 1$, any subset of T is not `I-conflict` (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 operations in $T \setminus \{u\}$. (This procedure is implemented by Algorithm 2.)

To find a restriction set, for each identified `I-conflict` set T, we add a restriction between any pair of operations from T if no pairs of operations from that set is ever restricted. Otherwise, T will be skipped. This is because the relevant violating executions, where all shadow operations from T are not restricted, have been already eliminated, and hence there is no need to analyze T. (This procedure is implemented by Algorithm 2.)

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 |T'| == 1 then
7:                 R ← R ∪ {r(Q'_0, Q'_0)}
8:             else if ∀u, v ∈ T', r(u, v) ∉ R then
9:                 R ← R ∪ {r(T'_i, T'_j)}, where i ≠ j and T'_i, T'_j ∈ T'
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
```

5 Design and Implementation of Olisipo

We implemented a prototype system to offer PoR consistent replication. This prototype consists of two our prior work SIEVE and Gemini which is a commutative operations transformer and a causally consistent storage, respectively, and Olisipo, which adapts applications to run with SIEVE and Gemini under PoR consistency. In this section we provide a detailed explanation of the design and implementation of Olisipo, and refer readers to check the materials about the other two components in [29] and [28].

5.1 Design rationale

We observed that there exist several coordination techniques/protocols that can be used for enforcing a given restriction, such as Paxos, distributed locking, or escrow

techniques. However, depending on the frequency at runtime in which the system receives operations confined by a restriction, different coordination approaches lead to different performance tradeoffs. Therefore, the question we need to answer is: how to choose the cheapest protocol for enforcing a given restriction?

Consider the previously mentioned RUBiS example. In this example, maintaining the invariant that winners always match highest successful bidders requires a restriction between any pair of `placeBid'` and `closeAuction'` operations. The simplest coordination scheme would be forcing the two types of shadow operations to pay the same coordination cost for figuring out the existence of concurrent counterparts. However, this solution yields a very poor performance due to the imbalanced workload between the two types of shadow operations, i.e., `placeBid'` is more prevalent than `closeAuction'`. As a result, reducing the latency for `placeBid'` while maintaining the corresponding ordering constraint will comprise a better user experience.

In summary, we propose to build a coordination service called Olisipo offering coordination policies, each of which presents a tradeoff 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 that has the lowest cost.

5.2 Coordination protocols

In this subsection, we present the two coordination techniques that we currently support in Olisipo and concrete scenarios where these mechanisms are more adequate. The two protocols we implemented are `symmetry (Sym)` and `asymmetry (Asym)`. Given a restriction $r(u,v)$ between two operations u and v, the `symmetry` protocol requires both u and v to coordinate with each other for establishing an order between them. In contrast, the `asymmetry` protocol provides different treatment for u and v by only requiring u (or v) to inform the counterpart operation in the restriction v (or u) about its existence, while allowing v (or u) to be executed fast without coordination if no u (or v) operations are running simultaneously. We further detail the two protocols as follows:

Sym. This protocol requires us to set up a logically centralized counter service, which maintains a counter for every shadow operation type present in a restriction $r(u,v)$, which we will refer to as c_u and c_v, and serializes reads and writes to these counters. Every such counter represents the total number of the corresponding operations that have been accepted by the underlying system. Additionally, every replica at different data centers maintains a local copy of these counters, each of which represents the number of corresponding operations that have been observed by that replica. Initially, all local

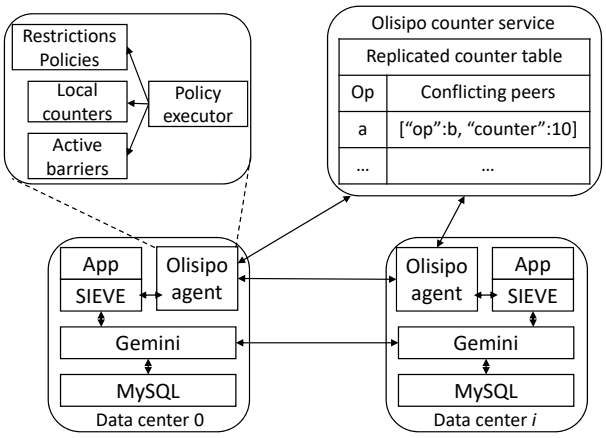


Figure 3: Olisipo architecture

copies, as well as the global counters, have all values set to zero. Whenever an operation of type u is received by a replica, that replica contacts the counter service to increase the corresponding counter c_u and get a fresh copy of the counter maintained for v. Upon receiving the reply from the counter service, that replica can then compare the value of c_v with its local copy. If they are the same, then the replica can execute u without waiting. If the value is greater than the local copy, the local execution can only take place when all missing operations of type v have been locally replicated. Conversely, the same procedure is also applied to v. After replicating operations, the local copy of the counters will be brought to be up-to-date. To make the counter service fault tolerant, we leverage a Paxos-like state machine replication library (BFT-SMART [17]) to replicate counters across geo-locations.

Asym. Unlike the above centralized solution, the asymmetry protocol implements distributed barrier in a decentralized manner as follows. Assume, for simplicity 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 the effects of the operation u it has just executed). Such a coordination strategy might incur in a high overhead; however, it might be interesting when one of the two operations in the restriction is rarely submitted to the system. For instance, in the auction example, `closeAuction'` is a candidate for being used as barrier, since `placeBid'` dominates the operation space.

5.3 Architecture overview

All design choices and details presented above lead to the high level system architecture depicted in Fig. 3. The

Olisipo architecture consists of a counter service replicated across data centers and a local agent deployed in every data center. While the counter service is required by executing the `Sym` protocol for keeping track of the number of different operations that have been accepted by the system, the local agent is responsible for placing coordination only when the corresponding operation is confined by restrictions. Every local agent keeps a restriction table, which defines all identified restrictions between pairs of operations and the corresponding coordination policy. In addition, every agent also stores some meta data required for different protocols: With regard to 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.

5.4 Implementation

We implemented Olisipo using Java (2.8*k* lines of code)[2], and BFT-SMART [40] for replicating the state of the centralized counter service, MySQL as the backend storage, and Netty as the communication library [8].We integrated Olisipo with Gemini and SIEVE so that Gemini serves as the underlying causally consistent replication tier while SIEVE is used to produce commutative shadow operations at runtime. The source code of Olisipo is available at [9].

Workflow. A user issues her request to an application server located at the closest data center, which runs an instance of SIEVE (introduced in [28]) and a local agent shown in Fig. 3. SIEVE intercepts the communication between the app server and the backend MySQL database and executes the corresponding generator operation. When the execution ends, SIEVE produces a commutative shadow operation that accumulates side effects of that request, and then asks the local Olisipo agent for placing coordination if needed before committing and replicating that shadow operation. To do so, the Olisipo agent looks up the restriction table to determine if that operation is confined by any restriction. If so, then the `policy executor` of Olisipo orders that operation with respect to all its conflicting operations that are running concurrently at other data centers. This is achieved by executing different protocols according to the lookup result. When conflicting operations are serialized, SIEVE sends these operations to Gemini for replicating them across all data centers while respecting the established order.

[2]The number of lines of code is measured by `cloc` [5].

6 Evaluation

Concerning the evaluation of Olisipo, we focus on two main aspects. First, we want 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. Second, we explore the impacts on user observed latency and system throughput introduced by three factors: adopting PoR consistent replication, using different protocols, and adding more restrictions.

6.1 Case study

Here we report our experience on discovering restrictions in RUBiS[3].

State convergence. As we deploy RUBiS alongside SIEVE, all shadow operations generated at runtime commute w.r.t each other and there is no need to restrict any pair of shadow operations. The final output of the state convergence restriction discovery method (Algorithm 1) is an empty restriction set.

Invariant preservation. 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 drawing the weakest preconditions and postconditions of all RUBiS shadow operations. Those conditions are summarized in Table 1 and used by the `I-conflict set` analysis (Algorithm 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 contrast, 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;
- $\{storeBuyNow', storeBuyNow'\}$. Invariant (c) would be violated if both operations simultaneously deducted a positive number from *stock* while *stock* was not enough;
- $\{placeBid', closeAuction'\}$. 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

[3]The original RUBiS is not complete since it does not include a closeAuction operation that declares the winners for auctions. As a result, we extended the original RUBiS by adding it a closing auction functionality.

$placeBid'$ $(itId, cId, bid)$	wp	$\exists u \in item\_table.\ u.id = itId \wedge u.status = open$	valid auction
	post	$bidTable = bidTable \cup \{< itId, bid, cId >\}$	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	$winnerTable = winnerTable \cup \{< wId, itId >\}$	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 over pairs of shadow operations that are required when replicating the extended RUBiS under RedBlue or PoR consistency

	US-East	US-West	EU-FRA
US-East	0.299 ± 0.042 ms 1052.0 ± 0.0 Mbps	71.200 ± 0.021 ms 47.4 ± 1.6 Mbps	88.742 ± 1.856 ms 29.6 ± 5.6 Mbps
US-West	66.365 ± 0.006 ms 47.4 ± 1.6 Mbps	0.238 ± 0.003 ms 1050.7 ± 4.1 Mbps	162.156 ± 0.179 ms 17.4 ± 1.7 Mbps
EU-FRA	88.168 ± 0.035 ms 36.2 ± 0.1 Mbps	162.163 ± 0.157 ms 20.1 ± 0.1 Mbps	0.226 ± 0.003 ms 1052.0 ± 0.0 Mbps

Table 3: Average round trip latency and bandwidth between Amazon datacenters

restrictions, namely $r(registerUser', registerUser')$, $r(storeBuyNow', storeBuyNow')$ and $r(placeBid', closeAuction')$, which are summarized in Table 2. Compared to the PoR consistency solution,however, replicating RUBiS via RedBlue consistency 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 pair-wise fashion.

6.2 Experimental setup

Deployment parameters. 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.

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 Olisipo, SIEVE, and Gemini. In addition, 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 (shown in Table 2) we identified in the context of RedBlue consistency, as baselines. We refer to the 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.

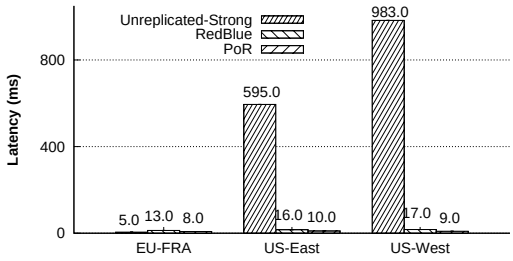
We choose to run the bidding mix workload of RUBiS, where 15% of user interactions are updates. To allow the client emulator to issue the newly introduced `closeAuction` requests, we have to slightly change the transition table equipped with the original RUBiS code by assigning a positive probability value for this request. The new transition table can be found here [12]. For all experiments we vary the workload by increasing the number of concurrent client threads in every client emulator. We also disable the `thinking time` option for issuing requests so that there is no waiting time between two contiguous requests from the same client thread. With regard to the data set, we populate it via the following parameters: the RUBiS database contains 33,000 items for sale, 1 million users, and 500,000 old items.

6.3 Experimental results

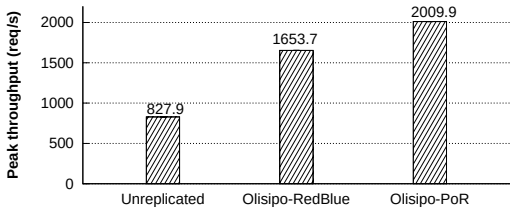
6.3.1 Average user observed latency

The major concern of adopting PoR consistent replication with Olisipo is to reduce the user perceived latency. To understand this improvement, first, we analyze the average latency for users at each data center. In the following experiments, each user issues a single request at a time in a close loop.

As shown in Fig. 4a, all users except those in EU-FRA observe notably lower latency in the *Olisipo-PoR* and *RedBlue* experiments, compared to the users from the same locations in the *Unreplicated-Strong* experiment. This improvement is because, under both PoR and RedBlue consistency, most of the requests are handled locally, while in the unreplicated RUBiS, requests from users at the two US data centers have to be redirected to EU-FRA, which incurs expensive inter-datacenter communication. *Olisipo-PoR* improves the average latency for users at the three sites by 38.5%, 37.5% and 47.1%,



(a) Overall average latency



(b) Peak throughput

Figure 4: Performance comparison between three system configurations

respectively, in comparison to *RedBlue*. 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 experiment, the absolute number (9 ms) is reasonably low.

6.3.2 Peak throughput

We now focus on the improvement on scalability by PoR consistency. Fig. 4b shows peak throughput achieved by the three configurations, which is measured when the corresponding system is saturated. The speedup of the *Olisipo-PoR* deployment is 1.43*x* against the *Unreplicated-Strong*. The 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 RedBlue consistent RUBiS, RedBlue consistent RUBiS achieves better scalability, namely a 21.5% increase in peak throughput. This improvement is because PoR consistency avoids the cost for coordinating unnecessary restrictions required by RedBlue consistency when replicating RUBiS, as shown in Table 2.

6.3.3 Per request latency

We explore the per request latency for RUBiS requests which produces shadow operations that may or may not need coordination. For this round of experiements, each site runs a single user issuing a request per time to the closest site.

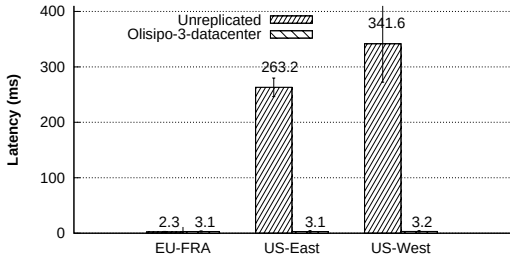


Figure 5: Average latency bar graph of a RUBiS request `storeComment` for users located at three sites. In the context of PoR consistency, this request is non-conflicting and hence does not require coordination.

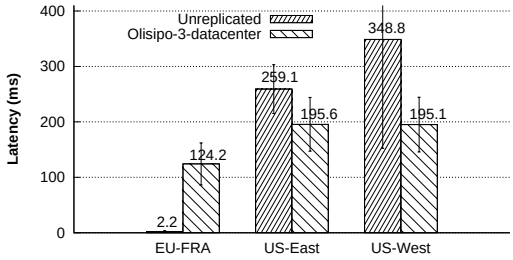


Figure 6: Average latency bar graph of a RUBiS request `storeBuyNow` for users located at three sites. In the context of PoR consistency, `storeBuyNow` conflicts w.r.t itself and is regulated by the `Sym` protocol when being replicated.

Latency of non-conflicting requests. Among all non-conflicting requests in RUBiS, we chose one representative request called `storeComment` as the illustrating example, which places a comment on a user profile. As depicted in Fig. 5, the conclusion we can draw from this graph is consistent with the one regarding Fig. 4a. However, the major difference between these two figures is that users from EU-FRA in both experiments have almost identical latency. This is because the `storeComment` request requires no coordination and the cost of generating and applying the corresponding shadow operation is modest.

Latency of conflicting requests. Then, 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 is conflicting with respect to itself. As shown in Fig. 6, the user observed latency of the `storeBuyNow` request at all three sites is significantly higher than the latency of `storeComment` (shown in Fig. 5), which is a non-conflicting request. This is because most of the lifecycle of these requests was spent asking the centralized counter service for granting permissions, which consists of 3 replicas spanning three sites and executing a Paxos-like consensus protocol. Additionally, user observed latency at EU-FRA is lower

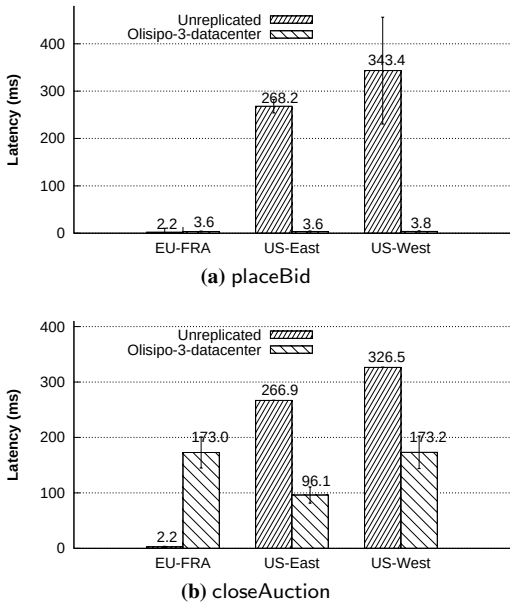


Figure 7: Average latency bar graph of a `placeBid` and `closeAuction` request for users locating in three sites. They produce two conflicting shadow operations, which are regulated by the `Asym` protocol.

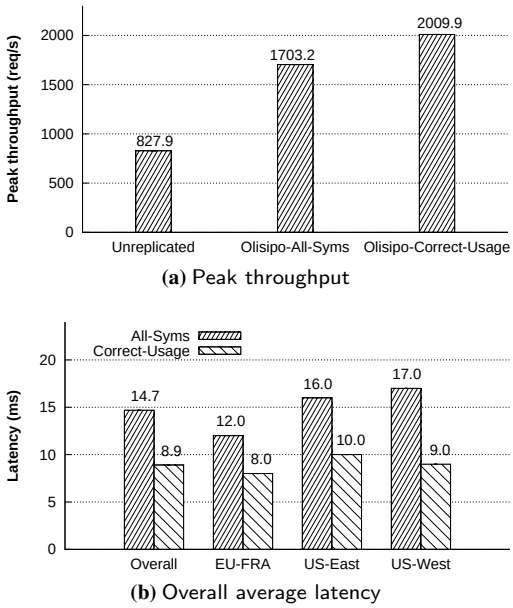


Figure 8: Peak throughput and overall average latency bar graphs of systems using different protocols.

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, namely one acts as a distributed barrier and the other proceeds if no active barriers are running. In the case study section (Section 6.1), we assign the `Asym` protocol to regulate the $r(placeBid', closeAuction')$ restriction, while selecting the less frequent shadow operation $closeAuction'$ to work as a barrier. As shown in Fig. 7a, the average latency measured for the *placeBid* request, which produces $placeBid'$, looks very similar to the results obtained for non-conflicting requests shown in Fig. 5. This is because the ratio of *closeAuction* to *placeBid* is very low (2.7%) 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, compared to *placeBid*, the average latency of *closeAuction* is remarkably 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. As shown in Fig. 7b, users issuing *closeAuction* observed a latency slightly higher than the maximal RTT between their primary data center and the remaining data centers. For example, as shown in Table 3, the maximal RTT on average for US-East users 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

As motivated in the design of Olisipo, 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. 8 summarizes the comparison of peak throughput and average latency among three experiments, namely `Unreplicated`, `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 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 re-

quests provided that `closeAuction` requests sparsely arrive in the system.

7 Related work

Consistency models. In the past decades, many consistency proposals have been focusing on the reduction in coordination among concurrent operations to improve scalability in replicated systems [24, 41, 29, 28, 13, 14, 43]. 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 tradeoff between performance and consistency using the visibility restrictions between pairs of operations to express consistency semantics. In addition, most previous proposals [13, 24, 41, 14] only take into account operation commutativity to determine the need for coordination, instead of invariant preservation, which is analyzed in our solution.

In the family of consistency proposals concerning application-specific invariants, Bailis et al. [15] proposed I-confluence to avoid coordination by determining if a set of transactions are I-confluent w.r.t database integrities, i.e., integrity constraints might be violated if they were executing without coordination. Indigo [16] 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 the top of eventual consistency. Similar to Indigo, warranties [30] 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 [34]. In contrast to these approaches, PoR consistency takes an alternative approach by modeling consistency as restrictions over operations.

There also exist a few proposals which map consistency semantics to the ordering constraints defined over pairs of operations. For example, Generic Broadcast defines conflict relations between messages for fast message delivery, which are analogous to visibility restrictions used in our solution [33]. Most recently, a concurrent work proposed by Gotsman et al. encoded the `conflict relation` concept into a proof system, which enables to analyze if consistency choices expressed into conflict relations meet the target properties [22]. Our approach differs from all these consistency proposals in the following aspects. First, we provide programmers with the ability to infer a minimal set of (fine-grained) restrictions to achieve state convergence and invariant preservation. Second, we explore the possibility of using different coordination protocols to enforcing restrictions efficiently.

Paxos and its variants. State machine replication [35] is a standard technique to make a set of servers behave like a single machine. Paxos [25], one of the classic algorithms that implement state machines, forces every replica to process a set of requests in the same sequential order. In order to reduce the number of message exchanges for achieving distributed consensus, several variants of Paxos have been proposed. Fast Paxos [27] aims at improving latency by allowing every replica to propose values but suffers from high latency when concurrent proposals occur. To avoid the penalty introduced by collisions, some other variants of Paxos explore operation semantics to take into account a weaker guarantee that not all operations are needed to be totally ordered [26, 23, 32]. Generalized Paxos (GPaxos) allows replicas to execute a set of operations in different orders as long as operations commute w.r.t each other; however, it still has to resort to the classic Paxos algorithm [25] when the leader notices two concurrent non-commuting requests [26]. Kraska et al. design an optimistic commit protocol called MDCC, which embodies GPaxos and explores operation commutativity for making geo-replicated transactions fast [23]. EPaxos takes as input a set of pre-defined constraints, each of which defines a dependency between a pair of operations, and enables each replica to order two concurrent conflicting requests according to their apriori dependency relation [32].

Our work and these Paxos variants significantly differ 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, all these protocols only reduce the number of communication steps, but still require to talk to a large quorum of replicas. In contrast, 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.

Efficient transaction processing. Some other work aim to reduce coordination in transaction processing. For example, transaction chopping [39] and Lynx [44] suggest that breaking large transactions into smaller pieces can improve performance, and they design analysis algorithms for chopping transactions without sacrificing serializability. While this work has been done merely by checking conflicts in read/write sets between pairs of transaction pieces, we design a comprehensive and fine-grained analysis concerning commutativity and invariant preservation for avoiding coordination when possible. These techniques are also orthogonal to our proposal so that we can apply them to prune out the non-critical code sections prior to running our analysis.

Geo-replicated systems. Azure Cosmos DB from Microsoft is a globally distributed database enabling users to specify various consistency guarantees for reads, which are mapped to desirable service level

agreements [10]. Similiarly, Google Data Store offers strong and eventual consistency options for read-only queries [4]. Unlike this approach, we focus on understanding the impact of concurrent updates on system properties and performance, and explore these findings to reduce coordination of updates that cannot be executed in parallel at different sites.

8 Conclusion

In this paper, we proposed a research direction for building fast and consistent geo-replicated systems that employ a minimal amount of coordination in order to achieve both invariant preservation and state convergence. To this end, we first defined a new generic consistency model called PoR consistency, which maps consistency requirements to fine-grained restrictions over pairs of operations. Second, we developed a static analysis to infer, for a given application, a minimal set of restrictions for ensuring the two previously mentioned properties, in which no restrictions can be removed and no new restrictions need to be added. Third, we built an efficient coordination service called Olisipo for coordinating conflicting operations. Our evaluation of running RUBiS with different setups shows that the joint work of PoR consistency and Olisipo significantly improves the system performance of geo-replicated systems.

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 Jul-2017].

[5] Count Lines of Code. http://cloc.sourceforge.net/. [Online; accessed Jan-2018].

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

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

[8] Netty IO. http://netty.io/. [Online; accessed Jan-2018].

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

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

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

[12] Modified RUBiS Transition Table. http://www.mpi-sws.org/~chengli/olisipofiles/workload/vasco_transitions_3.xls, 2015. [Online; accessed Jan-2018].

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] LAMPORT, L. Fast Paxos. *Distributed Computing 19*, 2 (October 2006), 79–103.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] SCHNEIDER, F. B. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[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] SHASHA, D., LLIRBAT, F., SIMON, E., AND VALDURIEZ, P. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst. 20*, 3 (Sept. 1995), 325–363.

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

[41] 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.

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

[43] 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.

[44] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 276–291.