Minimizing Coordination in Replicated Systems

Cheng Li[†] João Leitão[‡] Allen Clement[†] * Nuno Preguiça[‡] Rodrigo Rodrigues[‡]
[†]MPI-SWS [‡]NOVA LINCS / NOVA Univ. Lisbon

Abstract

Replication has been widely adopted to build highly scalable services, but this goal is often compromised by the coordination required to ensure application-specific properties such as state convergence and invariant preservation. In this paper, we propose a principled mechanism to minimize coordination in replicated systems via the following components: a) a notion of restriction over pairs of operations, which captures the fact that the two operations must be ordered w.r.t. each other in any partial order; b) a generic consistency model which, given a set of restrictions, requires those restrictions to be met in all admissible partial orders; c) principles for identifying a minimal set of restrictions to ensure the above properties; and d) a coordination service that dynamically maps restrictions to the most efficient coordination protocols. Our preliminary experience with example applications shows that we are able to determine a minimal coordination strategy.

1. Introduction

Replication is an essential technique to ensure the scalability of Internet services such as Google [3], Facebook [2], or Amazon [1]. Unfortunately, replication 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). This tension has been widely acknowledged by both industry solutions [10] and academic research [6, 22].

To relieve this tension, many proposals followed a hybrid approach, where different operations in the replicated system can have different semantics [13, 16, 17, 21]. In broad terms, some operations can be executed under strong consistency (like linearizability [11]) while other operations can be executed under weak consistency (like eventual consistency [9]). These approaches are effectively exploiting the fact that different consistency guarantees require different degrees of coordination among replicas for enforcing an order on the execution of operations. In fact, strong consistency requires operations to be totally ordered, which leads to significant coordination overheads, whereas weak consistency allows operations to be applied with few or even no restrictions, hence avoiding the performance penalty due to coordination.

To guide programmers to adapt their applications to these hybrid consistency models, our previous work of RedBlue Consistency [16] defines a set of sufficient classification conditions to assign different consistency levels to various operations: 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 works well for many web applications, but it can also lead to unnecessary coordination in some cases. We illustrate this with an auction service, where a `place_bid` operation creates a new bid for an item if the corresponding auction is still open, and a `close_auction` operation closes an auction for an item and declares its winner. In this example, the application-specific invariant is that the winner must be associated with the highest bid across all accepted bids.

In this example, the concurrent execution under weak consistency of a `place_bid` operation (taking as a parameter a bid that is higher than all accepted bids) and a `close_auction` operation can lead to the violation of the application invariant. This happens because `close_auction` will be unaware of the highest bid created by the concurrent `place_bid` operation. Unfortunately, the only way to address this issue in RedBlue Consistency is to label both operations as strongly consistent, i.e., all operations of either type will be totally ordered w.r.t. each other, which will incur in a high coordination overhead. Intuitively, however, there is no need to order pairs of `place_bid` operations, since a bid coming before or after another does not affect the winner selection. This highlights that our previous 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 address this issue, in this paper we attempt to offer a principled methodology that consists of three pillars. *First*,

* currently working at Google

PaPoC'15, April 21, 2015, Bordeaux, France.

http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2745947.2745955

we generalize the principles behind this binary classification by breaking down the coarse-grained constraint that totally orders all strongly consistent operations into a set of fine-grained restrictions, each of which only imposes an order between pairs of operations. Following this path, we propose a novel generic consistency definition, called *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 over operations. Weakening or strengthening the consistency semantics in the context of PoR Consistency is achieved by imposing fewer or more restrictions over relevant operations.

Second, given that adapting applications to PoR Consistency requires knowing which operations must be coordinated, we intend to find a set of conditions that operations must meet in order to require this coordination. The fundamental challenge with this task is that missing required restrictions will lead applications to violate invariants, while placing unnecessary restrictions will lead to a performance penalty due to the additional coordination. To overcome this, we aim at finding principles to identify a minimal set of restrictions. (By minimal we mean that removing a single restriction no longer ensures the desired properties.)

Third, we further observe that a key aspect to ensure good performance in a replicated service is to enforce these restrictions in an efficient way. In fact, 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 over time in which the system receives operations confined by a restriction, different coordination approaches lead to different performance tradeoffs. Therefore, to minimize the runtime coordination overhead, we also propose an efficient coordination service that helps replicated services use the most efficient protocol by taking into account the deployment characteristics measured at runtime.

The remainder of this paper is organized as follows. We introduce PoR Consistency in Section 2, and a set of principles to infer restrictions in Section 3; Section 4 presents a set of relevant case studies of PoR Consistency; and Section 5 proposes an efficient coordination service.

2. Partial Order-Restriction Consistency

The key intuition behind our proposal for a generic consistency model is that a consistency model can be perceived as a set of restrictions imposed over admissible partial orders across the operations of a system. Before introducing the consistency model, we start by defining a `restriction`, $r(u, v)$, as a binary relation between two operations u and v w.r.t. a partial order $P(U, \prec)$ over a set of operation instances U. This relation implies that for any pair of operations u and v in U, they must be ordered in $\prec$, such that $u \prec v \vee v \prec u$.

To provide an informal definition of PoR Consistency by combining the notions of partial order and restrictions, we say that: a replicated system $\mathscr{S}$ with a set of restrictions $R_{\mathscr{S}}$ is PoR Consistent if the following two conditions are met: 1) all restrictions are met in any partial order resulting from the execution of operations over $\mathscr{S}$; and 2) when an operation *op* is executed over $\mathscr{S}$, assume E be the set of operations that precede *op* in the corresponding partial order $\prec$, the state that is observed by *op* must be equivalent to the state reached through the application of all operations in E to an initial (known) state, according to a compatible linear extension of the partial order.

To demonstrate the power of PoR Consistency, we use it to express many different consistency requirements. For causal consistency [18] (excluding any restrictions to provide session guarantees), the restriction set is empty, since causality is already preserved in the definition of PoR Consistency by having $u \prec v \wedge v \prec w \implies u < w$. Regarding RedBlue Consistency, to capture the notion of strongly consistent operations, we define the following restriction set: for any pair of operations u, v, if u and v are strongly consistent, we have $r(u, v)$. Serializability [8] totally orders all operations, so its restriction set is as follows: for any pair of operations u, v, we have $r(u, v)$.

3. Restriction inference

When replicating a service under PoR Consistency, the first step is to infer restrictions to ensure the following two important system properties. First, to achieve state convergence, we take the same methodology adopted in prior research [16, 17, 20] to check operation commutativity in a pairwise fashion. However, unlike RedBlue Consistency, under which all non-commutative operations are totally ordered, PoR Consistency only requires that an operation must be ordered w.r.t. another one if they do not commute.

Second, to always preserve application-specific invariants, instead of conservatively totally ordering all operations that potentially break invariants if they execute without being aware of some other operations, we try to isolate the operations that contribute to an invariant violation from the remaining ones. To do so, we compose an execution containing a set of concurrent operations that leads to an invariant violation, and iteratively prune a single operation from the set and check if the violation persists. Once we reach a point where we cannot remove any operation and still create an execution that violates the invariant, then we say that this concurrent execution is minimal. By minimal, we mean that removing any concurrent operation from the execution will no longer lead to any violation. If such a composition is found, adding a restriction to force an order between any pair of remaining concurrent operations could be sufficient to avoid the problematic execution and hence, eliminate the

corresponding violation. Most importantly, this method may enable us to remove any unnecessary coordination by checking one by one which operations are relevant for the invariant violation.

4. Case studies

In this section, we present some relevant case studies of replicated services that can benefit from PoR Consistency.

Auction Service. We have previously introduced an auction service as a motivating example for the limitations of hybrid consistency models such as RedBlue Consistency. We are able to compose an invariant-violating execution with three concurrent operations over a shared auction, namely, a `close_auction` operation and two `place_bid` operations whose associated bids are higher than all accepted bids. According to our definition, this execution is not minimal since the composition of a `place_bid` and `close_auction` operation is already sufficient to trigger the violation. Thus, we refine the execution by removing one of the `place_bid` operations from it. As a way to preserve the invariant of this service under PoR Consistency, a restriction must be created between any pair of `place_bid` and `close_auction` operation, i.e., $r(\texttt{place\_bid}, \texttt{close\_auction})$. The advantage of this approach is that `place_bid` operations have no restrictions among themselves, which implies that no coordination mechanism is required when executing them concurrently. Given the fact that these operations are the most common in such a service, performance will be dramatically improved.

Banking Service. Another interesting example is an online banking service, that supports operations such as `deposit` and `withdraw` which, respectively, allow a user to add or remove a given amount to or from the current bank account balance, while a third operation, `accrueinterest`, updates the balance value of an account by taking into consideration its current balance and a given interest rate. We further assume an application-specific invariant which states that account balance values must be non-negative.

In this particular example, to ensure state convergence, `accrueinterest` operations must be executed with coordination in relation to `deposit` or `withdraw` operations, i.e., creating two restrictions $r(\texttt{accrueinterest}, \texttt{deposit})$ and $r(\texttt{accrueinterest}, \texttt{withdraw})$, as multiplication does not commute with addition or subtraction. Furthermore, to ensure the application-specific invariant, special care has to be given to `withdraw` operations, as two concurrent `withdraw` operations may drive the balance value to negative. As a result, these operations have to be coordinated among themselves. To capture this fact one has to add the restriction $r(\texttt{withdraw}, \texttt{withdraw})$.

5. Coordination protocols

There exist many possible coordination protocols to enforce ordering across pairs of operations, which can be used to impose concrete restrictions under PoR Consistency. One possibility is to take advantage of an existing coordination system, such as ZooKeeper [12], or rely on an implementation of Paxos [14]. An alternative is to use locks or leases to allow a single process to decide on the ordering of operations that require coordination. Although all these protocols provide the required ordering guarantees, their runtime overhead can be significantly different when considering the frequency of different operations. Therefore, it is not trivial to pick the right coordination strategy for a particular set of restrictions.

To circumvent this challenge, we propose to build a specialized coordination service that uses runtime information about the relative frequency of different types of operations to select a coordination mechanism for a given restriction that has the lowest cost. Next we discuss some coordination techniques and concrete scenarios where these mechanisms are more adequate. For that, we consider a single restriction between two operations u and v.

Barrier. A concrete materialization of a barrier would operate 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 this. This requires all replicas in the system to stop processing v operations. After all replicas acknowledge the entrance of r in the barrier for u, r can execute the operation, and then notify all replicas that it left the barrier (while at the same time propagating the effects of the operation u it executed). Such a coordination strategy might be interesting when one of the two operations in the restriction is rarely submitted to the system. For instance, in the auction example, `close_auction` is a candidate for being used as barrier, since `place_bid` dominates the operation space.

Paxos. Paxos can be leveraged as an external coordination technique to establish a total order across all u and v operations. This can be performed, for instance, by having these operations request from a Paxos instance a sequence number, and requiring all replicas to execute operations accordingly to the order defined by the Paxos participants. The restriction $r(\texttt{withdraw}, \texttt{withdraw})$ in the banking example would be a good candidate for using this coordination strategy.

Centralized sequencer. Another alternative for a coordination technique is to rely on a centralized component called a sequencer, which maintains a counter of the type $< seq_u, seq_v >$, where seq_u and seq_v represents the sequence number for u and v operations, respectively. Replicas maintain a local copy of the counter, and initially all local copies as well as the sequencer counter have all values set to zero. Whenever a u or v operation is received by a replica, that replica would contact the sequencer to increase the corresponding counter and get a fresh copy of the counter

maintained by the sequencer. Upon receiving the reply from the sequencer that replica can then verify the value of that counter for the complementary operation and compare with its local counter. 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 of the other type have been locally replicated. After replicating operations, the local copy of the counter will be brought up-to-date.

6. Related work

In the past decades, many proposals have been focusing on the reduction in coordination among concurrent operations to improve scalability in replicated systems [4, 5, 13, 15–17, 22]. Unlike previous proposals, which only allow programmers to choose from either strong or weak consistency, PoR Consistency offers a fine-grained classification using the visibility restrictions over operations to express consistency semantics. Visibility restrictions are analogous to conflict relations in Generic Broadcasting [19], but our approach significantly differs from it in that we intend to provide programmers with the ability to infer a minimal set of (fine-grained) restrictions to achieve state convergence and invariant preservation.

Most proposals [4, 5, 13, 15, 22] only take into account operation commutativity to determine the need for coordination, instead of invariant preservation, which is analyzed in our solution. Bailis et al. [6] proposed I-confluence to avoid coordination by proving if a set of transactions are I-confluent, i.e., they can be safely executed without coordination. We envision a less conservative approach by leveraging the fact that not all transactions in a non-I-confluent set must be coordinated. Indigo [7] 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. PoR consistency takes an alternative approach by modeling consistency as restrictions over operations, and enforcing these restrictions efficiently and in an adaptive fashion.

7. Conclusion

This paper proposed a research direction for building replicated system that employs a minimal amount of coordination in order to achieve both invariant preservation and state convergence. We aim to define a new generic consistency model, which maps consistency requirements to a minimal set of fine-grained restrictions over pairs of operations, while at the same time paving the way for an adaptive use of different coordination techniques for enforcing those restrictions.

Acknowledgments

The research of R. Rodrigues is funded by the European Research Council under ERC Starting Grant No. 307732. This work is partially funded by FCT under project PEst-OE/EEI/UI0527/2014.

References

[1] Amazon Web Services Webpage. `http://aws.amazon.com/`. [accessed 20-August-2014].

[2] Facebook Webpage. `https://www.facebook.com/`. [accessed 20-August-2014].

[3] Google Webpage. `www.google.com`. [accessed 20-August-2014].

[4] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR'11)*, 2011.

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

[6] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination-avoiding database systems. *Proc. VLDB Endow.*, 2015.

[7] V. Balegas, S. Duarte, C. Ferreira, N. Preguiça, R. Rodrigues, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, 2015.

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

[9] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical report, Microsoft Research, 2013.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, 2007.

[11] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 1990.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*, 2010.

[13] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 1992.

[14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 1998.

[15] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[16] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, 2012.

[17] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, 2014.

[18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual : Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011.

[19] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, 1999.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical report, INRIA, 2011.

[21] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually Consistent Byzantine-Fault Tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009.

[22] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011.