Data-centric Consistency Policies

A Programming Model for Distributed Applications with Tunable Consistency

Nosheen Zaza
Università della Svizzera italiana (USI)
nosheen.zaza@usi.ch

Nathaniel Nystrom
Università della Svizzera italiana (USI)
nate.nystrom@usi.ch

Abstract

The consistency level of operations over replicated data is an important parameter in distributed applications. It impacts correctness, performance, and availability. It is now common to find single applications using many different consistency levels at the same time; however, current commercial frameworks do not provide high-level abstractions for specifying or reasoning about different consistency properties of an application. Research frameworks that do tend to require a substantial effort from developers to specify operation dependencies, orderings and invariants to be preserved. We propose an approach for specifying consistency properties based on the observation that correctness criteria and invariants are a property of data, not operations. Hence, it is reasonable to define the consistency properties required to enforce various data invariants on the data itself rather than on the operations. The result is a system that is simpler to describe and reason about. In this paper, we outline an abstract model of programming language constructs and a static checker for data-centric consistency control, and demonstrate this model through a simple prototype programming language implementation.

Keywords distributed systems, eventual consistency, static analysis, domain-specific programming languages

1. Introduction

In modern distributed systems, data is often replicated to enhance availability and resilience to failure, and to improve performance by placing replicas closer to clients. However, managing the consistency of replicated data is a nontrivial problem. Ideally, all clients should observe the most recent data at all times. Achieving this consistency level through linearizability [18] or serializability [24] entails well-documented costs in terms of latency, availability, and partition tolerance [4, 21]. For this reason, many weaker consistency levels have been proposed, such as eventual consistency [28], causal consistency [23, 25], and snapshot isolation [19]. The choice of one of the above consistency levels depends on application correctness and performance requirements. For instance, in a flight booking application, there is no way eventual consistency alone can guarantee that a single airplane seat will not be assigned to two different people [4, 5]. However, eventual consistency works well in many non-critical cases, for example, creating and showing product recommendations, and is widely used in practice [12, 13, 17].

In current systems, to enforce a certain consistency property in an application, a developer selects a storage system that supports this property, then specifies consistency settings per operation or per sequence of operations. However, mapping high-level application consistency requirements to a combination of low-level consistency settings on operations is often tedious and error-prone. For instance, Cassandra [22] and Riak [9] allow specifying one of many consistency settings for operations: `one`, `two`, `quorum`, `all`, etc., describing the set of nodes that need to acknowledge the success of an operation. It is not immediately clear how consistency levels such as causal consistency or linearizability can be achieved, if at all, using these systems. The Cassandra documentation states that, to achieve "strong consistency"[1] clients must issue quorum reads and writes, however, nothing prevents a misbehaving client from issuing concurrent `one` writes and `quorum` reads on the same data, thus violating strong consistency for other clients. Another problem is when a client issues `all` reads and writes, which is completely unnecessary for enforcing any high-level consistency policies. Similar issues occur in relational database systems, where it cannot be guaranteed that a sequence of transactions running at different isolation levels preserves the intended invariants [16].

To tackle such problems, we propose a data-centric, declarative programming model inspired by [14], where high-level consistency requirements needed to enforce certain data integrity invariants are declared directly over data regions, and low-level consistency settings for operations interacting with regions are assigned or restricted accordingly. We describe the abstract model in Section 2, and in Section 3 we motivate its usefulness in addressing the above issues by showing an example in a prototype language implementing it.[2] We describe the mapping between our language's data-centric declarations and underlying operational consistency settings of the target storage system in Section 4. In section 5, we overview related work, and finally, in Section 6, we conclude.

2. Abstract System Model

Our system model allows developers to split the domain of stored objects into a collection of named, disjoint *regions*. Each region contains objects that are related by some application-specific integrity invariant. The developer annotates each region with a consistency policy strong enough to enforce the implied invariants. Supported consistency policies must form a partial order based on strength, as defined in [26]. To interact with regions, the developer declares *actions*, sequences of atomic read–write operations that

[1] Strong consistency in Cassandra implies neither linearizability or serializability [27].

[2] The language is a work in progress. The project that can be found at `https://github.com/amanjpro/languages-a-la-carte/`.

perform a single logical task, and may interact with multiple regions. An action succeeds if and only if all contained operations succeed. The visibility of each write, and the recency of data observed by each read within an action, for both failed and successful actions, is determined by the consistency policy of the region with which reads and writes interact. In this sense, actions are more flexible than relational database transactions, which enforce a single isolation and visibility policy for all contained operations.

This model allows controlled customization of the consistency policy of read operations; that is, it is allowed to request reads from a region at a weaker consistency level than the declared level of the region. It is not allowed to read at a consistency level stronger than that of the region because it cannot be enforced without also changing the consistency level of write operations. The model does not allow upward writes of data derived from low consistency reads to regions with a higher consistency level. However, an implementation may choose to allow explicit promotion of low-consistency data to be written into higher consistency regions, to increase the flexibility of the model.

Unlike reads, it is not possible to customize the consistency of write operations. This is justified by the assumption that each data region is annotated with a consistency policy strong enough to enforce any application-specific invariants. Selecting a stronger consistency policy is not only pointless, but also can worsen the availability or performance properties of the application. On the other hand, selecting a weaker consistency policy makes it impossible to ensure that application invariants are maintained by concurrent operations. If a weaker policy is in fact strong enough to enforce any application-specific invariants, it should be used as the annotation on the data region itself. These data-centric write restrictions act as data region guards, because they force all writes to go though the needed consistency protocol.

3. Data-centric Cloud Types

In this section, we present a simplified subset of our implementation of the proposed model, which is based on *cloud types* [10]. In our language, *DCCT* (Data-Centric Cloud Types), the data-centric model rules are enforced through static type checking. Objects are integers or strings, represented as replicated registers with last-writer-wins semantics [11]. At any point in time, different versions of each entity may co-exist, either on different replicas or in local client buffers. Each version is assigned a time stamp as well as other metadata that might be needed to enforce a consistency property.

We define a partial order of three consistency policies, eventual consistency (`EC`) $\prec$ monotonic atomic view (`MAV`) [4] $\prec$ serializability (`SR`), which apply on the action level. The programming language Quelea [26] provides similar consistency policies, as well as a proof of order validity. In short, consistency levels are represented as first-order logical predicates; ordering is given by entailment: stronger consistency policies entail weaker. All variables in our implementation are at least eventually consistent `EC`, which means it is possible to freely observe any version of the object (without monotonicity guarantees), and eventually all versions will converge. A region annotated `MAV` guarantees that all new versions written by an action are observed together, and that version updates are observed monotonically. The `SR` annotation on a region guarantees that the latest version of all contained objects is observed and manipulated at all times.

Figure 1 shows a simple schema with annotated regions and actions. Regions are annotated with their consistency policies. For simplicity, regions are simply fields, which may be records containing multiple simple fields. In this example, the developer requires `username`s to be unique, hence the field `Customer.username` is declared as serializable. It is still possible to read `username`s at lower consistency levels if needed (e.g., if we only need to display

```
class Customer {
  @SR username: String
  @EC name: String
  @EC gender: String
  @SR address:
    (street: String, city: String, country: String)
  @MAV paymentInfo:
    (paymentMethod: String, paymentInfo: String)
}
object mysteryProduct {
  @EC price: Int
  @SR amount: Int
}
class orderLog {
  @EC logRegion (
   username: String
   country: String
   date: String
   amount: Int
   orderSucceeded: Int
   )
}
action orderMysteryProduct(
 customer: Customer, amount: Int) {
  val prevAmount = mysteryProduct.amount
  if (prevAmount - amount >= 0) {
    mysteryProduct.amount = prevAmount - amount
    submitOrder(
     amount,
     customer.address,
     mysteryProduct.price * amount)
    orderLog.insert(customer.username,
      @EC customer.address.country,
      Date.now(), amount, 1)
  } else {
    displayLowStockError()
    orderLog.insert(customer.username,
      @EC customer.address.country,
      Date.now(), amount, 0)
  }
}
```

Figure 1. Example program in DCCT language with data-centric consistency policies

the name). `name` and `gender` are not very important, even if older data is shown or intermediate data is written, the customer can still correct it and eventually all replicas of this information will converge. `address` is strongly consistency to avoid shipping to an old address. Payment information is declared `MAV`, to avoid showing a mixed state of payment info. Using `MAV` consistency allows an older payment method to be used when charging the customer, however, the developer is willing to accept this anomaly for the sake of availability. `mysteryProduct.amount` is `SR` to ensure that all operations observe the latest version and avoid selling non-existent items. While it is possible to request a read at a lower level, our model ensures that we cannot write back an amount based on such reads. We show only one action `orderMysteryProduct`. This action does not have a single isolation level, as is the case in regular transactions; rather, the action manipulates data with many different consistency policies. However, we can see the flexibility and safety this approach gives the developer. It is possible to read and write at multiple consistency levels seamlessly while preserving the consistency requirements of each region.

Actions in our system can be compared to generic functions interacting with different data stores that enforce varying consistency guarantees. `orderMysteryProduct` can be seen as a function that reads and updates `mysteryProduct.amount`, stored in a serializ-

able relational database, and also stores operation logs in an eventually consistent, key-value store. In addition, actions can signal success or failure, and developers have to handle them appropriately. For brevity, we omit failure handling details here.

4. Supporting Data-Centric Consistency

The underlying storage system needed to support annotated regions and actions must support at least the strongest consistency level, and ideally all consistency levels specifiable in the language, with equivalent or stronger guarantees. For the implementation described here, we use a customized version of Kaiju [3], a prototype key-value store implemented to evaluate [6]. To implement `SR` regions, we associate a pair of Kaiju's long read and write locks with each region. The relevant locks are acquired when an action interacting with an `SR` region starts and released when action execution ends. To implement `MAV` regions, Kaiju provides `get_all` and `put_all` functions, which execute a RAMP [6] algorithm to fetch related values in a way consistent with our `MAV` consistency definition. As described in [6], at the beginning of each action, a call to `get_all` is issued for each `MAV` annotated region. The data fetched is placed in a local cache. Subsequent writes in the action are performed on the local cache. When the action completes successfully, a `put_all` call is issued on the local cache per region to write them back to the server. `EC` regions are read and written directly using the baseline protocol, which provides no concurrency control; the LWW semantics of our registers guarantee eventual consistency of represented objects.

The DCCT compiler translates actions to Java methods containing Kaiju calls. The generated code can be made more efficient. For instance, specialized program analysis can deduce that `SR` region guarantees can be satisfied with short read or write locks. Our aim here was to show the feasibility of implementing actions enforcing multiple consistency properties.

5. Related Work

Our approach is mainly inspired by Vaziri et al. [14]. In their work, the Java language is extended with constructs that allow programmers to declare *atomic sets* that contain places in memory for which operations must be serialized. The compiler inserts synchronization primitives for all operations interacting with such sets to ensure linearizability over the regions accessed by the operations. The main advantage they enlist is gathering the all synchronization logic at one code location rather than all control flow paths leading to a memory operation on shared data that must be dominated by a synchronization operation. Our model generalizes this idea, allowing consistency levels other than linearizability. Kraska et al. [20] have also proposed a data-centric approach supporting multiple consistency levels. Besides serializability and eventual consistency, they also allow some data to have adaptable consistency and accept invariant violations with some cost. While their aim is to reduce such costs, ours is to maintain the consistency level required to enforce certain invariants at all times, therefore simplifying reasoning about application correctness. Furthermore, their transactions are not restricted from updating data at lower consistency levels, which gives rise to anomalies.

Another line of work explores the relationship between application correctness invariants and consistency levels required to enforce them. Bailis el al. [5] propose a criteria they call *I-confluence*, to determine whether enforcing certain invariants require distributed coordination. Invariants that are I-confluent can be satisfied without strong consistency. An extension of I-confluence is proposed by Balegas et al. [8], who introduce a framework that allows declaring correctness invariants over data. However, the weakest supported consistency model is causal consistency, and not all types of invariants can be expressed. Gotsman et al. [15] propose a proof rule, automated in an SMT-based tool to show that a particular choice of consistency guarantees for operations ensures the preservation of programmer-defined application invariants. As in Balegas et al. [8], causal consistency is the lowest supported model.

The Quelea programming language [26] allows programmers to annotate operations with their consistency requirements in the form of first-order logical predicates. An SMT solver maps these requirements to the actual consistency levels provided by the underlying system. Quelea's contracts are lower-level than Gostman's, and do not ensure that the underlying integrity invariants are maintained. Quelea's model reasons about higher-level operations (e.g. deposit, withdraw), as opposed to our approach where we reason only about reads and writes. Hence, it is possible in Quelea for operations with multiple consistency levels to interact with the same data, while we require operations to follow the consistency policy declared on data. Their approach is thus more flexible in this sense, however, it requires keeping track of the entire history of operations, and describing all ordering and visibility properties among operations accessing the same data.

To reason about the correctness of systems with transactions running at different isolation levels, Adya [1] proposed a definition of *mixing-correct* and proved that in mixing-correct histories each transaction is provided the guarantees that pertain to its level. The implication is that transactions running at lower isolation levels must "know what they are doing", in terms of application invariants they must preserve, and must update the database consistently even if they observe an inconsistent state. Our data-centric approach, in contrast, requires all operations interacting with data for which invariants must be preserved to follow the declared consistency policy.

6. Conclusions, Current, and Future Work

The main advantage of our approach, compared to other proposals, is its simplicity, in terms of usage and description, and its ability to accommodate any non-probabilistic consistency properties that form a partial order. On the other hand, it does not attempt to prove that underlying application invariants will be maintained, putting it at the hands of the developer to decide on the suitable consistency policy needed for each region. We believe that it is possible to combine our work with [15] and [8], to show that a developer's selection of consistency policies maintains underlying application invariants. The data-centric consistency model presented here supports only disjoint regions. We plan to extend the model with support for nesting and merging of regions in the spirit of [14]. Temporarily merging regions is useful to avoid higher-level data races [2]. Furthermore, we also plan to survey commonly used simple invariants in data-backed applications to support declaring them directly, such as the uniqueness invariant [7], which is easier and more readable. To prove the soundness of the proposed model and programming language, we will formally specify the regions/actions model and their interactions, as well as the typing rules and operational semantics of the proposed language. Finally, a thorough evaluation, through implementing real world applications and benchmarks is necessary to to evaluate the ease of use and expressiveness of the language, as well as the performance of its generated code.

Acknowledgments

We thank Fernando Pedone and members of his research group for very helpful discussions and useful pointers. We would also like to thank our reviewers for their invaluable feedback.

References

[1] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 67–78. IEEE, 2000.

[2] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[3] P. Bailis. Code for scalable atomic visibility with RAMP transactions. `https://github.com/pbailis/ramp-sigmod2014-code`.

[4] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013.

[5] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, 2014.

[6] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 27–38. ACM, 2014.

[7] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1327–1342. ACM, 2015.

[8] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, page 6. ACM, 2015.

[9] Basho. Riak KV. `http://basho.com/products/riak-kv/`.

[10] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *ECOOP 2012–Object-Oriented Programming*, pages 283–307. Springer, 2012.

[11] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *ACM SIGPLAN Notices*, volume 49, pages 271–284. ACM, 2014.

[12] DataStax. Case study: Netflix: Netflix personalizes viewing for over 50 million customers with DataStax. `http://www.datastax.com/resources/casestudies/netflix`, 2011.

[13] DataStax. Case study: eBay: eBay engages customers with personalized recommendations. `http://www.datastax.com/resources/casestudies/ebay`, 2012.

[14] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):4, 2012.

[15] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In *Symposium on Principles of Programming Languages*, pages 371–384, 2016.

[16] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[17] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *In Proc. SOSP*. Citeseer, 2007.

[18] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[19] B. Kemme. *Database replication for clusters of workstations*. PhD thesis, Citeseer, 2000.

[20] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009.

[21] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[22] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[24] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[25] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *Distributed Algorithms*, pages 219–232. Springer, 1989.

[26] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 413–424. ACM, 2015.

[27] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Representation without taxation: A uniform, low-overhead, and high-level interface to eventually consistent key-value stores. *Data Engineering*, page 52, 2016.

[28] W. Vogels. Eventually consistent. *Communications of the ACM*, 52 (1):40–44, 2009.