Università della Svizzera italiana

Faculty of Informatics

Research Prospectus

September 24, 2014

Data-Centric Consistency Management

Nosheen Zaza

Abstract

The utilization of relaxed data consistency models has increased in recent years, especially in distributed applications, where maintaining strong consistency comes at the cost of availability and performance. Developing applications with correct consistency properties is difficult, because current development tools do not provide abstractions that enable programmers to intuitively write their code in a replicated, asynchronous environment with multiple consistency requirements. In this research prospectus, we survey various techniques and tools proposed to control data consistency, and describe our vision of a data-centric framework that would facilitate developing distributed applications with replicated data.

Research Advisor
Prof. Nate Nystrom

Academic Advisor
Prof. Nate Nystrom

Review Committee
Prof. Committee Member1, Prof. Committee Member2

Research Advisor's approval (Prof. Nate Nystrom): Date: .....

PhD Director's approval (Prof. Stefan Wolf): Date: .....

1 Introduction and Problem Domain

In an ideal concurrent environment, it is reasonable for programmers to expect that the latest update issued to a shared data abstraction will be the one immediately visible to the next read issued by any client. However, a system will almost always include multiple copies of the same data abstraction through caching, buffering and replication, and since updates do not propagate among the copies instantaneously, it is possible to observe multiple versions of data at the same time. Furthermore, updates are often reordered, either to allow optimizations, which is the case in many CPU architectures such as ARM [10], or because of communication delays. Consequently, defining what 'latest update' means is no longer straightforward. Programming in such environments requires defining a consistency model [1] which serves as a contract between the data abstraction and its clients. Strong consistency models such as sequential consistency [26] and linearizability [22] come close to fulfilling the above mentioned expectation, but they are expensive for certain applications, because they disallow many execution interleavings through some form of synchronization, ultimately forcing clients to wait. They also forbid some reordering-based optimizations. In practice, most frameworks provide more relaxed consistency models, such as causal consistency [37] and eventual consistency [43].

Even when built on top of such frameworks, the vast majority of multicore applications attempt to exhibit a strongly-consistent behavior. The exceptions such as the concurrent skip-list map [34], PageRank [29] and branch-and-bound [12] tend to be probabilistic in nature, or tolerate nondeterminism, which are not common properties of multicore applications. On the contrary, the Internet has brought many more use cases where relaxed consistency is employed on application-level with little compromise. As a motivating example, consider an online shopping application that replicates product data. When a client requests a list of all products that have 5-star ratings, it is not necessary to query all replicas in order to get the most up-to-date rating data. In fact, requiring acknowledgments from all replicas would cause unnecessary system load for a task that is inherently subjective. Some real-world systems that employ relaxed consistency include metrics collection and product recommendations at eBay [36], and inbox search at Facebook [25]. Both are backed by Cassandra [25], which is an eventually-consistent storage system [46].

Relaxed consistency is rarely sufficient on its own for the requirements of a complete application. Typically, components requiring different consistency policies co-exist and interoperate. Great care must be taken to ensure correct program semantics in such cases. Continuing with the previous example, consider when the client adds a product from the list of items with 5-star ratings to his shopping cart. Because the list was not constructed by querying all replicas, it is possible that it includes some items that are no longer in stock. If the programmer forgets to query all replicas to ensure that the selected item is in stock prior to checkout, the client will end up buying a non-existing item.

Employing weak and strong consistency side-by-side, on the application level in a distributed environment is a relatively recent shift from the traditional approach of developing strongly consistent applications on top of weakly consistent frameworks. We believe that many programming pitfalls, similar to those mentioned above can be avoided by providing programmers with the right development environment and tools. Ongoing research has produced many methodologies and tools that facilitate consistency management, and we are investigating how to combine and extend existing ideas in order to construct a framework that allows programmers to safely employ both strong and relaxed consistency in their distributed applications.

The core idea we are studying is based on a declarative, data-centric approach to consistency management, originally proposed by Vaziri el al. [44]. This approach allows programmers to declaratively express atomicity as a property of data, and we are investigating how to allow expressing and inferring other consistency properties and constraints as well, so that it is known at compile time which parts of the program are strongly consistent and which parts are not. This way, programs become clearer, and it becomes possible to delegate many parts of consistency management logic to the compiler and runtime. In this research prospectus, I start by describing the core ideas of this approach, then demonstrate how it can be adapted to support multiple consistency levels of replicated data. After that I will list related work on data abstractions, consistency constraints and program properties that we believe are useful for programmers to express declaratively. Finally, that I will overview the research directions and challenges, and how we plan to conduct our work.

2 Data-Centric Atomicity Constraints

When programming in a shared-memory environment, programmers must ensure that concurrent accesses to the same memory location do not lead to data races and inconsistent results. This is traditionally achieved by ensuring that instructions accessing shared data acquire locks, or are within a transaction. Such approaches have an operational, control-centric flavor. The problem is that data races may occur if the programmer forgets to synchronize

even a single control-flow path. Vaziri et al. [44] outlined a solution in 2006, that allows associating synchronization constraints with data, and based on these constraints, the compiler generates synchronization code for all operations that access this data. For example, to protect a field or more of an object from data races, an atomic set is declared, which is a memory region to which all accesses must be synchronized, then object fields that need to be protected are added to this set. The compiler then synchronizes all public methods that interact with the atomic set. The system also allows atomic sets that span multiple objects, and provides other features as well. The only consistency constraint available in this system in atomicity, which provides strong consistency guarantees. However, there are cases where it is desirable to have both strong and weak consistency in an application. In the following section, we describe our idea of extending data-centric atomic sets to accommodate other consistency properties as well.

2.1 Adding Replicated Data Consistency Constraints

In many replicated key-value storage systems, such as Cassandra [25], Dynamo [20], and Riak [38], clients can tune consistency by specifying the number of replicas on which an operation must succeed before returning an acknowledgment to the client application. In Cassandra, consistency is configured on the level of cluster, data center, or individual I/O operations. Going back to our online shopping example, instead of relying solely on the programmer to choose the correct consistency level every time he writes code that interacts with the contents of the shopping cart, we can attach a policy to shopping cart data region, stating that all operations must have the consistency level ALL, meaning all replicas must acknowledge the successful completion a query. We gain two benefits following this data-centric approach: it is now possible to infer the correct consistency level for all operations dealing with the shopping cart data, and if the programmer specifies a lower consistency level, such as ONE, which only requires an acknowledgment from one replica, it is now possible to show an error/warning stating that this is probably not what is meant to be done. Furthermore, since the list of 5-star rated items was read with consistency level ONE, it represents a memory region that is not strongly-consistent, so it is not possible to assign an item from this region to the strongly-consistent shopping cart region.

3 Related Work

Based on the idea of atomic sets, we aim to introduce a programming framework that enables developing distributed applications with better data abstractions and consistency control. In the section above, we have shown one way to extend atomic sets. However, coming up with a generic and practical framework requires further research. There are two main aspects we need to investigate: defining data region abstractions, and identifying data and operation properties and constraints that would be useful to infer/express. In the following sections, I describe both research problems, and show related work.

3.1 Data Region Abstractions

We need to determine which data region abstraction(s) to employ, because choosing the right abstractions eases defining constraints on data. Many data abstractions in various systems come with baked-in consistency constraints. Having good default definitions of such constraints often relieves programmers from having to define them themselves. We also believe that having replication defined as an explicit property of data makes reasoning about it easier for the programmer. The following list contains various data abstractions and built-in consistency guarantees.

1. The Java memory model [35] ensures atomic semantics for built-in operations on 32-bit primitive types primitive types, such as `int`. `long` and `double` are not required to be atomic by the language specification.
2. In many database systems, row updates are atomic. For instance, In MySQL [32], the MYISAM storage engine locks tables when an update is issued, while InnoDB engine locks the row being updated. In Cassandra, rows are atomic units.
3. Objects in object-oriented languages span over multiple memory locations, and group together data and operations [39]. Objects can be arbitrarily nested and passed around to form complex graphs, and accesses to object data are typically unordered and synchronous. Objects present a difficult case for constraint attachment, because many threads can have references to different parts of an object and update them concurrently, possibly making temporary invariant violation visible among threads. Even if each object in the graph is properly synchronized, the collection of objects might still have incorrect semantics [5]. Furthermore, it is hard to reason about interaction and nesting of objects with conflicting constraints. There is a large body of ongoing research on ownership types and alias management that aim to restrict arbitrary referencing and nesting of objects [11, 18, 31].

4. In the current implementation of atomic sets [14], Vaziri el al. employ a simple ownership type system that ensures objects are only accessed via their owners. Without this constraint, it would be hard to perform static analysis. Atomic sets presented in this work are static data containers that may span the data of multiple objects. data within an atomic set shares the same lock. However atomic sets only express "shallow-locking", if nested objects must be locked, they need to be explicitly added to the containing object's atomic set. It is also not possible to dynamically grow or shrink atomic sets.
5. Rust, [30] a programming language developed by Mozilla, employs ownership, alias management, and regions to ensure memory safety and data-race freedom, which contribute to building programs that are strongly consistent.
6. DPJ [7] regions are hierarchal heap partitions used to disambiguate accesses to distinct objects, as well as distinct parts of the same object. A type-checker then ensures that there are no conflicting accesses to overlapping memory regions between concurrent tasks.
7. X10 [9] supports places [45], which are a distribution-friendly data abstraction that encapsulate the binding of activities and globally addressable memory. An activity may synchronously access data items only at the place in which it is running. It may atomically update one or more data items, but only in the current place. If it becomes necessary to read or modify an object at some other place, a place-shifting operation can be used. This approach ensures that expensive operations (e.g., those which require communication) are more readily visible in the code. The implicit constraint in this system is the locality of data.
8. The actor model [2] differs in two fundamental ways from the object model: all accesses to actor state are serial and asynchronous. This model is a better fit for a distributed environment, and is gaining popularity, with languages such as Erlang [4] and Scala [33] employing it successfully. Typically, actors must not expose their internal data by any means other than message passing. In practice, systems such as Akka [8] do not enforce that, and rely on the programmer to maintain actor encapsulation. Because of their properties, actors are easily distributable, however it is hard to introduce concurrency within an actor. Habanero Java and Habanero Scala [23] enable controlled concurrency within an actor.

3.2 Consistency-Related Properties and Constraints

We now list some other properties and constraints that enable better control of consistency by making some program properties explicit to the programmer or the development environment.

1. Monotonicity: this property means that the final order or content of input will never cause any earlier output to be "revoked" once it has been generated. This means that any order of operation generates the same consistent result. Bloom [3] is a distributed language that exploits this property by providing programmers with order-independent abstractions, which encourages a programming style that minimizes coordination requirements and guarantees consistency.
2. Latency and failure: X10, Scala and other languages support futures [19], which materialize latency and failure as datatypes and operation effects. Their advantage is that they make these code properties explicit, which is important in a distributed environment. While multicore programmers need not be concerned with a CPU core going down or instructions getting lost among system buses, network developers always face node failure and message loss. Futures make failure handling a first-class action in a distributed language.
3. Locality: Loci [47] is a pluggable type system that enables programmers to make thread locality of data explicit and statically checkable. This has many advantages: programmers need not to worry about the consistency of data proved to be local, and compilers can perform more optimizations with such data.
4. Commutativity and Inverses: Commutativity permits re-ordering of method calls while ensuring deterministic semantics, and inverses allow creating more compact rollback logs for transactional memory. Systems that employ these properties include DPJ, the transactional boosting method [21], IceCube [24] CRDTs [40] and Sagas [16].
5. Isolation Levels: In databases, isolation levels can be attached to transactions [6]. The consistency of dataset resulting from a transaction would depend on the specified isolation level.
6. Effects: A problem with object-oriented systems is that operations on data are grouped as functions. A function binding in many systems is determined at runtime. Functions can perform arbitrary reads and writes, and because they can be dynamically bound, it is hard to tell at compile-time which pieces of data are being read or written to by a function. This complicates reasoning about data consistency. Effect systems allow the programmer to declaratively attach effects to methods, and some of them can infer the effects partially [28].

Read/write effect systems can solve the previous problem [27]. Another interesting effect system, proposed by Cormac and Shaz [15] allows declaring atomicity as a property of methods.

7. Transactional Memory: Software transactional memory [41] borrows the ideas of database transactions and apply them to generic programs. Such systems are known for they high overhead because of the delay of creating operation logs and performing roll-backs. Not all software systems can tolerate such overheads. However transactions are easier to reason about than locking, and they are composable.
8. Other replicated-data consistency guarantees: There are many other consistency level between ONE and ALL mentioned earlier. Cassandra also support QUORUM, which employs a quorum-based technique [17]. Session guarantees [42] provide an application with a view of the replicated database that is consistent with its own reads and writes performed in a session even though these operations may be directed at different servers. Guarantees are provided within the context of a session. a single application may create several sessions that it uses to exercise fine-grained control over the guarantees it desires.

4 Current State of Work and Research Directions

After we noted the potential of extending atomic sets [14] to accommodate other data properties and constraints, we implemented the system in the Scala programming language, and attempted to design an extension to permit declaring sets that do not enforce strict atomicity. For the system to function properly, it would require the programmer to declare read/write effects and ownership constraints in addition to the declared set policies. As a sample use-case of the extended system, I analyzed the implementation of concurrent skip list map [34] from the Java collections library, because it contains both strongly and weakly consistent data regions. The underlying algorithm is fairly complex, and concurrency control is extremely fine-grained and sophisticated. At this point, I believe that multicore applications which employ weak consistency have similar properties, hence the proposed system would be coarse-grained, complex and conservative for such applications, and would provide few code simplicity or performance benefits, if not the contrary. The other application domain I studied is weakly-consistent distributed applications. I examined some eventually consistent, distributed key-value stores, focusing on Cassandra, and also studied the actor model and reactive programming to learn distributed programming patterns. As explained in Section 1, weak consistency has more obvious application-level uses in distributed applications, and the need for better data abstractions and management tools as we propose is clearer. As mentioned in Section 2.2, enabling programmers to codify consistency policies, as well as other data properties and constraints not only makes code clearer and safer, but also opens new perspectives of runtime-optimizations based on this knowledge. As a first step towards a generic consistency-management framework, I am currently designing and building a domain-specific language to allow attaching Cassandra consistency policies to data regions in Scala, similar to what I described in Section 2.2. I am also building a sample application to demonstrate the uses and benefits of this language. Given our focus on distributed applications, we need to ensure that our approach integrates well with a distributed programming model, which includes, but is not limited to the following: programmers should be able to intuitively program assuming asynchronous-by-default operations, similar to that proposed by the actor model, which is now popular because it truly reflects how the network operates. Furthermore, current development frameworks also make other properties of a distributed application first-class citizens in code. These include failure, as in Scala's reactive streams and try types, latency represented by futures, as implemented in Java, Scala and X10, and data residing at different places as implemented in Resilient X10 [13].

As we discussed in Section 3.1, objects in object-oriented programming languages impose many challenges for the design of our framework, yet they are very commonly used and familiar to developers, thus we need to consider them. To tame the object model, we need to implement ownership and alias management type systems, as well as various effect systems, which ultimately complicate our framework. We may also need to introduce other simpler data abstractions, similar to DPJ regions or X10 places. The main challenge in this work is to keep our framework simple enough for use yet expressive enough to be useful.

5 Achievements

So far I have achieved the following:

- Attended a workshop on software correctness and reliability at ETH Zurich.
- Attended the following courses: Bugs 2013 (4 ECTS) and Introduction to Ph.D. studies (2 ECTS)
- Attended ECOOP 2014 with co-located workshops and summer schools.

- Served as a TA for the following courses: Programming Fundamentals 3 (4 ECTS) and Automata and Formal Languages (2 ECTS). I am currently the TA of Massively Parallel Programming (3 ECTS)

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] G. A. Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[3] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a calm and collected approach. In *CIDR*, pages 249–260. Citeseer, 2011.

[4] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent programming in Erlang. 1993.

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

[6] P. A. Bernstein. SQL isolation levels. In *Encyclopedia of Database Systems*, pages 2761–2762. Springer, 2009.

[7] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, pages 4–4. USENIX Association, 2009.

[8] J. Boner. Akka. `http://akka.io/`, 2014.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.

[10] N. Chong and S. Ishtiaq. Reasoning about the ARM weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '08, pages 16–19, New York, NY, USA, 2008. ACM.

[11] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. *Minimal ownership for active objects*. Springer, 2008.

[12] J. Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[13] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. Saraswat, M. Takeuchi, and O. Tardieu. Resilient x10: efficient failure-aware programming. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 67–80. ACM, 2014.

[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] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN Notices*, volume 38, pages 338–349. ACM, 2003.

[16] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, Dec. 1987.

[17] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM Symposium on Operating Systems Principles*, pages 150–162. ACM, 1979.

[18] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP 2010–Object-Oriented Programming*, pages 354–378. Springer, 2010.

[19] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Futures and promises. *URL http://docs. scala-lang. org/overviews/core/futures. html*.

[20] 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.

[21] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 2008.

[22] 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.

[23] S. M. Imam and V. Sarkar. Integrating task parallelism with actors. In *ACM SIGPLAN Notices*, volume 47, pages 753–772. ACM, 2012.

[24] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 210–218, New York, NY, USA, 2001. ACM.

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

[26] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[27] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[28] D. Marino and T. Millstein. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation*, pages 39–50. ACM, 2009.

[29] F. McSherry. A uniform approach to accelerated PageRank computation. In *Proceedings of the 14th international conference on World Wide Web*, pages 575–582. ACM, 2005.

[30] Mozilla. The Rust programming language homepage. `http://www.rust-lang.org/`.

[31] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming*, volume 263. Fernuniversität Hagen, 1999.

[32] MySQL. Mysql. `http://dev.mysql.com/`, 2014.

[33] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, 2004.

[34] Oracle. java.util.concurrent.ConcurrentSkipListMap. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html`.

[35] Oracle. Java language specification, chapter 17. `http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4`, 2014.

[36] F. Qu and A. Jambhekar. Cassandra at eBay scale. `http://www.slideshare.net/planetcassandra/5-feng-qu`, 2013.

[37] M. Raynal and A. Schiper. From causal consistency to sequential consistency in shared memory systems. In *Foundations of Software Technology and Theoretical Computer Science*, pages 180–194. Springer, 1995.

[38] E. Redmond and J. R. Wilson. *Seven databases in seven weeks: a guide to modern databases and the NoSQL movement*. Pragmatic Bookshelf, 2012.

[39] T. Rentsch. Object-oriented programming. *ACM SIGPLAN Notices*, 17(9):51–57, 1982.

[40] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[41] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[42] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 140–149. IEEE, 1994.

[43] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[44] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. *SIGPLAN Not.*, 41(1):334–345, Jan. 2006.

[45] Vijay Saraswat et al. X10 language specification. `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`, 2014.

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

[47] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for Java. In *ECOOP 2009–Object-Oriented Programming*, pages 445–469. Springer, 2009.