Preprint

Disciplined Inconsistency

Double-blind submission

Abstract

To keep users happy and meet service level agreements, web services must respond quickly to requests and be highly available. In order to always meet these tight performance goals, despite network partitions or server failures, developers often must give up strong consistency and migrate to some form of eventual consistency. Making this switch can be error-prone because the guarantees of weaker consistency models are notoriously difficult to understand and test. Furthermore, introducing weak consistency to handle worst-case scenarios creates an ever-present risk of inconsistency even for the common case, when everything is running smoothly.

In this work, we propose a new programming model for distributed data that uses types to provide a *disciplined* way to trade off consistency for performance safely. Programmers specify their performance and correctness targets as constraints on abstract data types (ADTs). Meeting performance targets introduces uncertainty about values, which are represented by a new class of types called *inconsistent, performance-bound, approximate (IPA)* types. We demonstrate how this programming model can be implemented in Scala on top of an existing datastore, Cassandra, and show that it provides sufficient flexibility in terms of performance and correctness to handle a variety of adverse scenarios for applications including a shopping cart, Twitter clone, and ticket vendor.

1. Introduction

- Applications have performance requirements
    - Sometimes explicit in the form of SLAs, promising a certain latency or availability
    - Sometimes more implicit (i.e. every additional ms of latency reduces revenue)
- Constantly balancing performance vs correctness / programmability
    - If it isn't scaling well, or latencies are too high, then relax consistency in some places and hope...
- This is error prone: every time you change consistency, there are new reorderings and conditions to consider
    - new edge cases to handle, *implicit* in the consistency model
    - accidentally leak into places that weren't intended to be weakened
- Worse: conditions can change at any moment; node goes down, network unreliable, traffic surges
    - In test environment, inconsistency is typically unlikely
    - Adverse conditions in production can cause errors that never appeared in testing, or are very difficult to test for
    - No way to know if you've caught them all
- Furthermore, when conditions are good, there's no need to resort to weak consistency
- It would be great if we had a way to:
    - Express performance bounds
        - Have the system help achieve them
    - Make inconsistency explicit and restricted
        - handle different cases in a disciplined way
        - restrict possible values, and where they can be used
- So the question is: *where to introduce this abstraction?*
    - As part of the data type!
    - Couples the effects of mutating operations with reads
    - Concise and modular: re-use data types, no annotations on individual operations
    - Safe: inconsistency expressed as return types

2. Type System

We propose a programming model for distributed data that uses types to control the consistency–performance trade-off. The *inconsistent, performance-bound, approximate* (IPA) type system helps developers to trade consistency for performance in a disciplined manner. This section presents the IPA type system, including the available consistency policies and the semantics of operations performed under those policies. Section [#Implementation] presents the implementation of the IPA type system.

2.1. Overview

The IPA type system consists of three parts:

- Abstract data types (ADTs) implement common distributed data structures (such as `Set[T]`).
- Policy annotations on ADTs specify the desired consistency level for an object in application-specific terms (such as latency or accuracy bounds).
- IPA types track the consistency of operation results and enforce consistency safety by requiring developers to consider weak outcomes.

Together, these three components provide two key benefits for developers. First, the IPA type system enforces *consistency safety*, tracking the consistency level of each result and preventing inconsistent data from flowing into consistent data without explicit endorsement, in the style of EnerJ ([@TODO]). Second, the IPA type system provides *performance*, because consistency annotations at the ADT level allow the runtime to dynamically select the consistency for each individual operation that maximizes performance.

2.2. Abstract Data Types

The base of the IPA type system is a set of abstract data types for common distributed data structures. [copy some stuff from Claret here?]

2.3. Policy Annotations

The IPA type system provides a set of annotations that can be placed on ADT instances to specify consistency policies. Previous systems [@TODO] require annotating each read and write operation with a desired consistency level. This per-operation approach complicates reasoning about the safety of code using weak consistency, and hinders global optimizations that can be applied if the system knows the consistency level required for future operations.

IPA type annotations come in two flavors. *Static* annotations declare an explicit consistency policy for an ADT. For example, a `Set` ADT with elements of type `T` can be declared as `Set[T] with Consistency(Strong)`, which states that all operations on that object are performed with strong consistency. Static annotations provide the same direct control as existing approaches, but simplify reasoning about correctness. *Dynamic* annotations specify a consistency policy in terms of application-level requirements. For example, a `Set` ADT can be declared as `Set[T] with LatencyBound(50 ms)`, which states that operations on that object are performed with a target latency bound in mind. The runtime is free to dynamically choose, on a per-operation basis, whichever consistency level is necessary to meet this bound.

The IPA type system features two dynamic consistency policies:

- A latency policy `LatencyBound(x ms)` specifies a target latency for each operation performed on the ADT. The runtime can then determine the consistency level for each individual operation issued, optimizing for the cheapest level that will likely satisfy the latency bound.
- An accuracy policy `ErrorTolerance(x%)` specifies the desired accuracy for read operations performed on the ADT. For example, the `size` of a `Set` ADT may only need to be accurate within 5% tolerance. The runtime can optimize the consistency of write operations so that reads are guaranteed to meet this bound.

Policy annotations are central to the flexibility and usability of the IPA type system. Dynamic policy annotations allow the runtime to extract the maximum performance possible from an application by relaxing the consistency of its operations, safe in the knowledge that the IPA type system has enforced safety by requiring the developer to consider the effects of weak operations. Moreover, policy annotations are expressed in terms of application-level requirements, such as latency or accuracy. This higher-level semantics absolves developers of manipulating the consistency of individual operations to maximize performance while maintaining safety.

As an extension, ADTs can also have different consistency policies for each method. For example, a `Set` ADT might have a relaxed consistency policy for its `size`, but a strong consistency policy for its `contains?` predicate method. The runtime is then responsible for managing the interaction between these consistency policies.

2.4. IPA Types

The keys to the IPA type system are the IPA types themselves. Read operations performed on ADTs annotated with consistency policies return instances of an *IPA type*. These IPA types track the consistency of the results, and enforce a fundamental non-interference property: results from weakly consistent operations cannot flow into computations with stronger consistency without explicit endorsement.

Formally, the IPA types form a lattice parameterized by a primitive type `T`. The bottom element `Inconsistent[T]` specifies an object with the weakest possible consistency. The top element is `T` itself, an object with the strongest possible consistency. The other IPA types follow a subtyping relation $\prec$, defined by:

$$\frac{\tau \text{ is weaker than } \tau'}{\tau'[T] \prec \tau[T]}$$

There is also an explicit *endorsement* upcast to the top element `T`:

$$\frac{\gamma \vdash e_1 : \tau[T]}{\gamma \vdash \text{Consistent}(e_1) : T}$$

- High-level goals
  - Explicit performance bounds (latency)
  - Explicit approximation bounds (error tolerance)
  - Results in IPA types which express the resulting uncertainty
- ADTs
  - can't just express these on the *read* side, most require knowing how the *write* was done
  - e.g. `Consistency = Read.Consistency + Write.Consistency`, so `Write.ALL + Read.ONE = Strong`, or `Write.QUORUM + Read.QUORUM`
  - Other benefits of annotating ADTs:
    - portable / reusable
    - modular
  - Similar to Indigo's ([@Indigo]) invariants, but expressing performance and approximation bounds
- Types of annotation

  - "static" bounds like `Consistency(Strong)` that fix a policy upfront
  - "dynamic" bounds like `LatencyBound(50 ms)` that choose a policy at invocation time
  - per-method bounds for ADTs (e.g. `Set[ID]` has `size` and `contains?` methods that could have different bounds)
- Bounds
  - `Set[ID] with Consistency(Strong)`
  - `Set[ID] with LatencyBound(50 ms) -> contains(ID): Rushed[Boolean]`
  - `Counter with ErrorTolerance(5%) -> read(): Interval[Long]`
- IPA type lattice
  - Inconsistent (⊥)
  - `Rushed | Interval | Leased`
  - Consistent (⊤)
- Rushed
  - Consistency level achieved
- Interval
  - min, max, contains?, etc
  - linearizable within the error bound – as long as we stay within the bound, everything is strongly consistent
- Leased goes away
- Semantics of mixed consistency levels?
  - If every operation comes back strong, it's just like strong consistency was chosen in advance – so everything is linearizable
- Futures
  - (talk about how everything is implemented with futures, or just elide that?)
- All writes are statically at a certain consistency level
  - Why? So we don't have to reason about interactions with reads (would need flow analysis)

3. Implementation

We demonstrate one possible instantiation of the Disciplined Inconsistency model with an implementation of a Scala client, using Cassandra as the backing store. Most of the functionality required to implement the model is relatively datastore-agnostic; most Dynamo-style datastores support some form of tunable consistency, so porting our implementation to another backing datastore such as Riak should be possible.

[explain the basics of how Cassandra's consistency levels work] see: Cassandra Consistency

3.1. Latency bounds

As discussed earlier, a common desire is to be able to guarantee a certain response time, for example in order to meet an SLA. However, within that window of time, we would like to provide the strongest guarantees possible, so that users typically observe consistent, up-to-date data.

Conceptually, any Dynamo-style datastore implements configurable consistency levels by adjusting the number of replicas that a client request waits for a response from. [explain first how it would work conceptually by sending read requests to a quorum of replicas, and then proceeding with whatever we have when time is up; then explain how in Cassandra we have to cheat by issuing reads in parallel]

Condition label	Latencies (milliseconds)
Local	<1, <1, <1
Uniform/High load	5, 5, 5
Slow replica	10, 10, 100
Google	1±.3, 110±5, 160±5
Amazon	1±.3, 80±10, 200±50

Table 1. Simulated network conditions

3.2. Reservations

In order to implement the `Interval` bounds, we build on the concept of *escrow* and *reservations* [10, 17–19].

We implement reservations as a middleware layer: a reservation server runs alongside each Cassandra server. Any operations with error tolerance bounds are routed to a reservation server, using the Cassandra client's knowledge of which replicas are up.

3.3. Leases

[???]

4. Evaluation

[explain how we simulate network conditions using `tc netem` and `docker` ?]

4.1. Counter microbenchmark

- Latency bound
  - show how it can meet various latency bounds, compared with Strong and Weak
  - show that 95th percentile still meets latency bound!
  - show how many achieved stronger consistency, and how that correlates with actual consistency violations
- Reservations
  - show link between tighter bounds and lower performance
  - tie performance to the number of strong reads/reservation refreshes we had to do
  - show how interval width gets smaller with fewer writes

We start by measuring the performance of a very simple application that randomly increments and reads from a number of counters with different IPA bounds. Figure 1 shows the average latency of a 20% increment, 80% read workload over 200 counters randomly selected using a zipfian distribution.

4.1.1. Latency bounds

Latency bounds aim to provide predictable performance for clients while attempting to maximize consistency. Under fa-

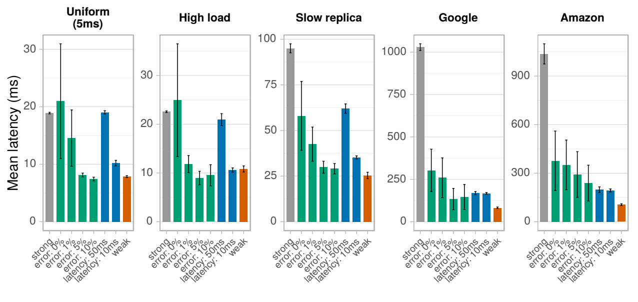

Figure 1. Counter benchmark: mean latency for a random mix of counter ops (20% increment, 80% read), with various IPA bounds, under various conditions.

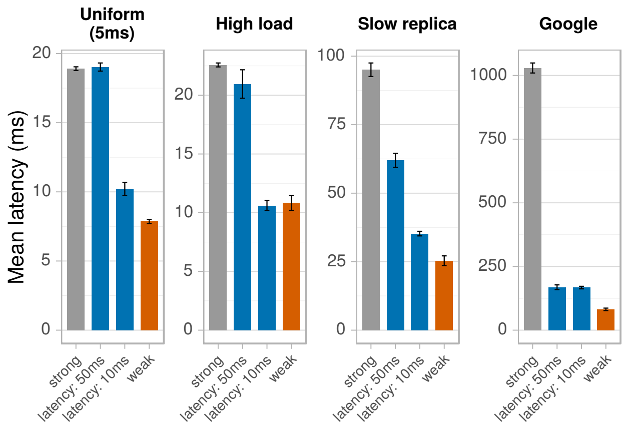

Figure 2. Consistency of latency-bound operations. Strong consistency is rarely possible within 10ms. With one slow replica, most reads can still achieve strong consistency, but with high network latencies or heavy load, it degrades to use weak consistency.

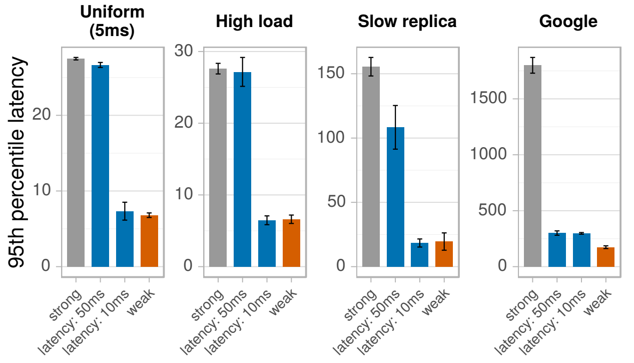

Figure 3. Latency bounds reduce unpredictable tail latency.

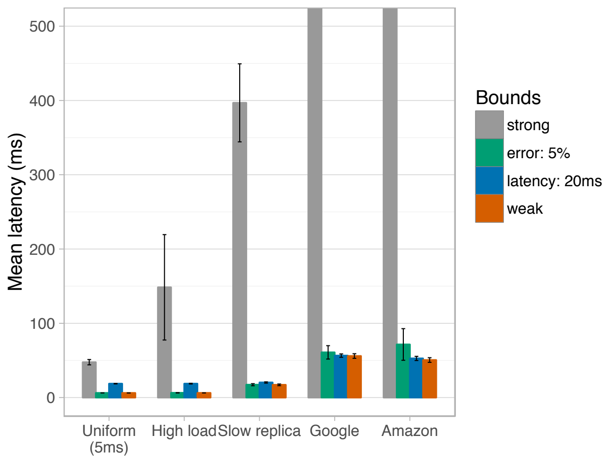

Figure 4. Ticket-sales app: mean latency of `purchase` action under various conditions. The `BoundedCounter` underlying ticket sales is safe even when weakly consistent, but latency bounds allow users to see strong consistency [??]% of the time, while reservations bound error to less than 5% with similar performance.

vorable conditions, when latencies and load are low, it is often possible to achieve strong consistency. Figure Figure 1 shows the average latency of

4.2. Applications

4.2.1. Shopping Cart

[demonstrate loading cart with a latency bound, but not allowing users to check out without doing a strong read]

4.2.2. TicketSleuth

- Modeled after FusionTicket (benchmark in [26, 27])
- Demonstrates

[ticket sales app demonstrating hard lower bounds on counters]

4.2.3. Twitter clone

[demonstrating error tolerance for Counter (number of retweets), and latency bound for loading the timeline]

5. Related Work

5.1. Consistency Models

A vast number of consistency models have been proposed over the years. From Lamport's *sequential consistency* [13] and Herlihy's *linearizability* [12] on the strong side, to *eventual consistency* [25] at the other extreme. A variety of intermediate models fit elsewhere in the spectrum, each making different trade-offs balancing high performance and availability against ease of programming. For example, a family of models including *read-your-writes* and *monotonic reads* use *sticky sessions* [23], which reduces availability in a small way, but provides users with a bit more certainty about what values they will observe.

A single global consistency model for an entire database or application is restrictive; some datastores support configuring consistency at a finer granularity: Cassandra [3] per operation, Riak [5] on an object or namespace granularity, as well as others [14, 22].

5.2. Explicit performance bounds

It is difficult for programmers to determine the correct consistency level for each operation. Ideally, everything would be as consistent as possible, but in some situations, performance needs (such as availability) force inconsistency.

[will probably have to introduce this earlier when explaining Rushed, but putting the text here for now] With *consistency-based SLAs* in Pileus [24], programmers can explicitly trade off consistency for latency. A consistency SLA specifies a target latency and a consistency level (e.g. 100 ms with read-my-writes). In this programming model, operations specify a set of desired SLAs, each associated with a *utility*. Using a prediction mechanism similar to PBS, Pileus attempts to determine which SLA to target to maximize utility, typically to achieve the best consistency possible within a certain latency.

In Pileus, SLAs are specified on each *read* operation, which returns both the value it got and the achieved consistency level. This allows programs to behave different depending on changing conditions. Our Rushed IPA types, which were inspired by Pileus, provide a more disciplined way to let programmers express how behavior should depend on consistency, protecting them from inadvertently misusing the returned value. In addition, Pileus's SLAs are assigned only to individual reads; writes are all assumed to be the same, and data type is not considered. Working with latency bounds at the ADT level allows reads and writes to be coupled, enabling more potential optimizations.

[are there other systems with explicit performance bounds enforced by the system?]

5.3. Controlling staleness

Most eventually consistent models provides no guarantees about how long it will take for updates to propagate. However, there are several techniques to help bound the staleness of reads.

Leases are an old technique that essentially gives reads an *expiration date*: the datastore promises not to modify the value that was just read until the lease term is over. First proposed to avoid explicit invalidations in distributed file system caches [11], leases have since been used in a multitude of ways: in Facebook's Memcache system [16] for invalidations, Google's Chubby [7] and Spanner [9] to adjust the frequency of heartbeat messages, and on mobile clients with exo-leases [21]. Warranties [15] are a generalization of leases, allowing arbitrary assertions over state or behavior. [explain how our leases relate (if they get implemented)]

[Probabilistically bounded staleness? (4)]

5.4. Types for distributed systems

Convergent (or *conflict-free*) *replicated data types* (CRDTs) [20] are data types designed for eventual consistency. Similar to how IPA types express weakened semantics which allow for implementation on weak consistency, CRDTs guarantee that they will converge on eventual consistency by forcing all update operations to commute. For example, Set `add` and `remove` typically do not commute, but a CRDT called an OR-Set re-defines them so that `add` wins over `remove`, making them commute again. CRDTs can be enormously useful because they allow concurrent updates with sane semantics, but they are still only eventually (or causally) consistent, so users must still deal with temporary divergence and out-of-date reads, and they do not incorporate performance bounds or variable accuracy.

Bloom [1, 2, 8] is a language and runtime system for defining whole applications that are guaranteed to converge. Based around a conceptual monotonically growing set of facts, the language encourages coordination-free computation, but automatically creates synchronization points where necessary.

[Session types?]

5.5. Approximate types / Trading off correctness

- Cite some approximate computing papers
- Something something Uncertain<T> [6]
- Conit-based Continuous Consistency Model [28]

References

[1] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R Marczak. Consistency analysis in bloom: a calm and collected approach. In *Conference on Innovative Data Systems Research (CIDR)*, CIDR, pages 249–260. Citeseer, 2011.

[2] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis for distributed programs. In *IEEE International Conference on Data Engineering*. Institute of Electrical & Electronics Engineers (IEEE), March 2014. doi:10.1109/icde.2014.6816639.

[3] Apache Software Foundation. Cassandra. http://cassandra.apache.org/, 2015.

[4] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5 (8): 776–787, April 2012. doi:10.14778/2212351.2212359.

[5] Basho Technologies, Inc. Riak. http://docs.basho.com/riak/latest/, 2015.

[6] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<T>: A First-Order Type for Uncertain Data. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS 14*, ASPLOS. Association for Computing Machinery (ACM), 2014. doi:10.1145/2541940.2541958.

[7] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association, 2006.

[8] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC 12*, SoCC. ACM Press, 2012. doi:10.1145/2391229.2391230.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 251–264, 2012. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387905`.

[10] Dieter Gawlick and David Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Engineering Bulletin*, 8 (2): 3–10, 1985.

[11] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP. Association for Computing Machinery (ACM), 1989. doi:10.1145/74850.74870.

[12] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12 (3): 463–492, July 1990. doi:10.1145/78969.78972.

[13] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28 (9): 690–691, September 1979. doi:10.1109/tc.1979.1675439.

[14] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li`.

[15] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 503–517, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/liu_jed`.

[16] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX. ISBN 978-1-931971-00-3. URL `https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala`.

[17] Patrick E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11 (4): 405–430, December 1986. doi:10.1145/7239.7265.

[18] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st international conference on Mobile systems, applications and services - MobiSys 03*, MobiSys. Association for Computing Machinery (ACM), 2003. doi:10.1145/1066116.1189038.

[19] Andreas Reuter. *Concurrency on high-traffic data elements*. ACM, New York, New York, USA, March 1982.

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS, pages 386–400, 2011. ISBN 978-3-642-24549-7.

[21] Liuba Shrira, Hong Tian, and Doug Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Middleware 2008*, Middleware, pages 42–61. Springer Science $$ Business Media, 2008. doi:10.1007/978-3-540-89856-6_3.

[22] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles - SOSP'11*, SOSP. Association for Computing Machinery (ACM), 2011. doi:10.1145/2043556.2043592.

[23] 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 *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, PDIS. Institute of Electrical & Electronics Engineers (IEEE), 1994. doi:10.1109/pdis.1994.331722.

[24] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP 13*. ACM Press, 2013. doi:10.1145/2517349.2522731.

[25] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52 (1): 40, January 2009. doi:10.1145/1435417.1435432.

[26] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/xie`.

[27] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-Performance ACID via Modular Concurrency Control. In *ACM Symposium on Operating Systems Principles (SOSP)*, SOSP, pages 276–291, 2015. ISBN 978-1-4503-2388-8. doi:10.1145/2517349.2522729.

[28] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20 (3): 239–282, 2002.