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

Applications often wish to guarantee a certain response time to keep users engaged or meet a contractual SLA. However at the same time, they wish to present the most consistent view possible to users. The time it takes to achieve a particular level of consistency is highly variable and depends on many parameters: individual nodes can be slower, they could temporarily be exceptionally slower to respond due to a garbage collection or operating system pause, or be overloaded with exceptionally high traffic. Furthermore, the same application may typically operate mostly within a single datacenter, but occasionally have clients from other parts of the world interact, causing geo-distributed replicas with high-latency or unreliable networks to coordinate.

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 `Consistent[T]`, an object with the strongest possible consistency, which has an implicit cast to type `T` available. The other IPA types follow a subtyping relation $\prec$, defined by:

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

The IPA type system is very similar to the probability type system of DECAF [@TODO], which uses types to track the quality of results computed on approximate hardware. Their non-interference property is also similar: a result of low quality cannot flow into a result of higher quality without explicit endorsement. We elide a thorough formal development of the IPA type system due to its close similarity with DECAF.

2.4.1. Weak IPA types

The IPA types encapsulate information about the consistency policy used to perform a read operation. Strong read operations return values of type `Consistent[T]`, and so (by implicit casting) appear to developers as any other instance of type `T`. Intuitively, this equivalence is because the results of strong reads are known to be consistent.

Weaker read operations return values of some type lower in the lattice (*weak IPA types*), reflecting their possible inconsistency. At the bottom of the lattice, the weak IPA type `Inconsistent[T]` encapsulate a value with unknown consistency. The only possible operation on `Inconsistent[T]` is to *endorse* it. Endorsement is an upcast, invoked by `Consistent(x)`, to the top element `Consistent[T]` from other types in the lattice:

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

While `Inconsistent[T]` has value as a reminder to developers about consistency, the key productivity benefit of the IPA type system is in the other weak IPA types. Each of the consistency policies in Section §2.3 has a corresponding weak IPA type for operations performed under that policy.

Rushed types The weak IPA type `Rushed[T]` is the result of read operations performed on an ADT with consistency policy `LatencyBound(x ms).Rushed[T]` is a *sum type*, with one variant per consistency level available to the implementation of `LatencyBound`. Each variant is itself an IPA type (though the variants obviously cannot be `Rushed[T]` itself). The effect is that values returned by a latency-bound object carry with them their actual consistency level. A result of type `Rushed[T]` therefore requires the developer to consider the possible consistency levels of the value.

For example, a system with geo-distributed replicas may only be able to satisfy a read latency bound of 50 ms using a local quorum. In this system, the `Rushed[T]` type would be the sum of three types `Consistent[T]`, `LocalQuorum[T]`, and `Inconsistent[T]`. A match statement destructures the result of a latency-bound read operation:

```
set.size() match {
  case Consistent(x) => print(x)
  case LocalQuorum(x) => print(x + ", locally")
  case Inconsistent(_) => print("unknown")
}
```

The application may want to react differently to a local quorum as opposed to a strongly or weakly consistent value. Note that because of the subtyping relation on IPA types, any omitted case will be matched by `Inconsistent(_)`, so other cases need only be added if the application should respond differently to them.

Interval types The weak IPA type `Interval[T]` is the result of operations performed on an ADT with consistency policy `ErrorTolerance(x%).Interval[T]` represents an interval of numbers within with the true (strongly consistent) result lies. The interval reflects uncertainty in the true value created by relaxed consistency, in the same style as work on approximate computing [8].

The key invariant of the `Interval[T]` type is that operations within the interval are linearizable. In other words, [Brandon has a nice example of this]

Lower bounds Weak IPA types enforce consistency safety by ensuring developers address the worst case results of weak consistency. However, the weak IPA types are *lower bounds* on weakness: one valid implementation of a system using IPA types is to always return strongly consistency values. Moreover, the runtime guarantees that if every value returned has strong consistency, then the execution is linearizable, as if the system were strongly consistent from the outset.

- 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[Bool`
  - `Counter with ErrorTolerance(5%) -> read(): Interval[Long]`
- IPA type lattice
  - Inconsistent (⊥)
  - `Rushed | Interval | Leased`
  - Consistent (⊤)
- Rushed
  - Consistency level achieved
  - Consistency levels are themselves ordered (lattice something something), so one could imagine writing an application with fewer type bounds than are supported by the underlying system, and it would simply fall back to the strongest lower bound or whatever.

- Example: write an app only handling "Strong" and "Weak": if the system supports intermediate levels that's fine, but the program will see all of them as "weak"
- Not 100% sure how to describe this

- 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

The IPA type system provides users with controls to specify performance and correctness criteria and abstractions for handling uncertainty. It is the job of the IPA implementation to enforce those bounds.

3.1. Backing datastore

At the core, we need a scalable, distributed storage system with the ability to adjust consistency at a fine granularity. In Dynamo-style [14] eventually consistent datastores, multiple basic consistency levels can be achieved simply by adjusting how many replicas the client waits to synchronize with. Many popular commercial datastores such as Cassandra [3] and Riak [7] support configuring consistency levels in this way. Our implementation of the IPA model in this work is built on top of Cassandra, so we will use Cassandra's terminology here, but most of the techniques employed in our implementation would port easily to Riak or others.

Eventual consistency, or the property that all replicas will eventually reflect the same state if updates have stopped [30], only requires clients to wait until a single replica has acknowledged receipt. Weak eventually consistent reads can similarly be satisfied by a single replica that has the requested data. A number of mechanisms within the datastore, such as anti-entropy, read repair, and gossip share updates among replicas, and operations are designed to ensure convergence (falling back to some form of last-writer-wins in case of conflicts). However, because clients can read or write to any replica, and writes take time to propagate, reads may not reflect the latest state, leading to potential confusion for users. [this eventual consistency primer should probably be earlier]

In order to be sure of seeing a particular write, clients must coordinate with a majority (*quorum*) of replicas and compare their responses. In order for a write and a read operation to be *strongly consistent* (in the CAP sense [9]), the replicas acknowledging the write plus the replicas contacted for the read must be greater than the total number of replicas ($W + R > N$). This can be achieved in a couple ways: write to a quorum ($(N + 1)/2$), and read from a quorum (`QUORUM` in Cassandra), or write to N (`ALL`), and read from 1 (`ONE`) [13]. Cassandra additionally supports limited linearizable ([17, 18]) conditional updates, and varying degrees of weaker consistency, particularly to handle different locality domains (same datacenter or across geo-distributed datacenters). In this work, we keep our discussion in terms of this simple model of consistency.

3.2. Latency bounds

As discussed earlier, applications often wish to guarantee a certain response time to keep users engaged or meet an SLA. However at the same time, they wish to present the most consistent view possible to users. The time it takes to achieve a particular level of consistency depends on the current conditions and can vary over large time scales (minutes or hours) but can also vary significantly for individual operations. During normal operation, strong consistency may have acceptable performance, but during those peak times under adverse conditions, the application would fall over.

Latency bounds specified by the application allow the system to *dynamically* adjust to maintain comparable performance under varying conditions. Stronger reads in Dynamo-style datastores are achieved by contacting more replicas and waiting to merge their responses. Therefore, it is conceptually quite simple to implement a dynamically tunable consistency level: send read requests to as many replicas as necessary for strong consistency (depending on the strength of corresponding writes it could be to a quorum or all), but then when the latency time limit is up, take however many responses have been received and compute the most consistent response possible from them.

Cassandra's client interface unfortunately does not allow us to implement latency bounds exactly as described above: operations must specify a consistency level beforehand. We implement a less optimal approach by issuing read requests at different levels in parallel. The Scala client driver we use is based on *futures*, allowing us to compose the parallel operations and respond either when the strong operation returns, with the strongest available at the specified time limit, or exceeding the time limit waiting for the first response. Pseudocode for this is shown in [#fig-latency-bound].

3.2.1. Monitors

The main problem with this approach is that it wastes a lot of work, even if we didn't need to duplicate some messages due to Cassandra's interface. Furthermore, if the system is responding slower due to a sudden surge in traffic, then it is essential that our efforts not cause additional burden on the system. In cases where it is clear that strong consistency is unlikely to succeed, it should back off and attempt weaker consistency. To do this, the system must monitor current traffic and predict the latency of different consistency levels.

Each client in the system has its own Monitor (though multi-threaded clients share one). The monitor records the observed latencies of read operations, grouping them by operation and consistency level. All of the IPA ADTs are implemented in terms of Cassandra *prepared statements*, so we can easily categorize operations by their prepared identifier. The monitor uses an exponentially decaying reservoir to compute running percentiles weighted toward recent measurements, ensuring that its predictions continually adjust to current conditions.

Whenever a latency-bound operation is issued, it queries the monitor to determine the strongest consistency likely to be achieved within the time bound. It then issues 1 request at that consistency level and a backup at the weakest level (or possibly just the one weakest if that is the prediction).

3.2.2. Adjusting write level

Remember that the achieved consistency level is determined by the combination of the write level and read level. By default, we assume a balanced mix of operations on an ADT, so writes are done at `QUORUM` level and strong reads can be achieved with the matching `QUORUM` level. However, sometimes this is not the case: if a datatype is heavily biased toward writes, then it is better to do the weakest writes, and adjust reads to compensate. This would also be helpful in cases where even the weakest reads fail to meet latency requirements because quorum writes are overloading the servers.

Changing the write level must be done with care because it changes the semantics of downstream reads. We have ADT implementations choose their desired write level statically so that we know the strength of a read without checking. One could imagine a more complex system allowing dynamic changes to an ADT's metadata (in the backing store), with clients checking for changes periodically, but we do not implement this. Applications wishing to get more dynamic behavior in our implementation could create alternate versions of ADTs with different static write levels and mediate the transition themselves.

3.3. Error bounds

We implement error bounds by building on the concepts of *escrow* and *reservations* [15, 22–24]. These techniques have been used in storage systems to enforce hard limits, such as an account balance never going negative, while permitting concurrency. The idea is to set aside a pool of permissions to perform certain update operations (we'll call them *reservations* or *tokens*), essentially treating operations as a manageable resource. If we have a counter that should never go below zero, there could be a number of *decrement* tokens equal to the current value of the counter. When a client

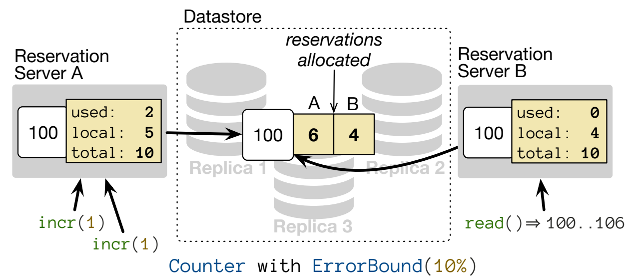

Figure 1. Reservations enforcing error bounds.

wishes to decrement, it must first acquire sufficient tokens before performing the update operation. Correspondingly, in this scheme, increments produce new tokens. The insight is that the coordination needed to ensure that there are never too many tokens can be done *off the critical path*: they can be produced lazily if there are enough around already, and most importantly for this work, they can be *distributed* among replicas. This means that replicas can perform some update operations *safely* without coordinating with any other replicas.

3.3.1. Reservation Server

To implement reservations, we must be able to mediate requests to the datastore to prevent updates from exceeding the available reservations. Furthermore, we must be able to track how many reservations each server has locally without synchronization. Because Cassandra does not allow custom mediation of requests, nor does it support replica-local state, we must implement a custom middleware layer to handle reservation requests.

Any client requests requiring reservations are routed to one of a number of *reservation servers*. These servers then forward operations when permitted along to the underlying datastore. All persistent data is kept in Cassandra; these reservation servers keep only transient state tracking available reservations. Our design is similar to the middleware for implementing bounded counters on top of Riak [6]. The number of reservation servers can theoretically be decoupled from the number of datastore replicas; however, our design simply co-locates a reservation server with each Cassandra server and uses Cassandra's discovery mechanisms to route requests to reservation servers on the same host.

3.3.2. Enforcing error bounds

Reservations have been used previously to enforce hard global invariants in the form of upper or lower bounds on values or integrity constraints [5, 6], or logical assertions [20]. However, enforcing error tolerance bounds presents a new design challenge because the bounds are constantly shifting.

Consider a `Counter` with a 10% error bound, shown in Figure 1. If the current value is 100, then 10 increments can be done before anyone must be told about it. However, we have 3 reservation servers, so these 10 reservations are distributed among them, allowing each to do some increments without synchronizing. Because only 10 outstanding increments are allowed, reads will maintain the 10% error bound.

In order to perform more increments after a server has exhausted its reservations, it must synchronize with the others, sharing its latest increments and receiving any changes of theirs. This is accomplished by doing a strong write (`ALL`) to the datastore followed by a read. Once that synchronization has completed, those 3 tokens become available again because the reservation servers all agree that the value is now, in this case, at least 102.

Read operations for these types go through reservation servers as well: the server does a weak read from any replica, then determines the interval based on how many reservations there are. For the read in Figure 1, there are 10 reservations total, but Server B knows that it has not used its local reservations, so it knows that there are as many as 6 outstanding increments, so it returns the interval $[100, 106]$.

3.3.3. Narrowing bounds

The maximum number of reservations that can be allocated for an ADT instance is determined by the statically defined error bound on the ADT. However, as with latency bounds, when conditions are good, or few writes are occurring, reads should reflect this. In the previous example, Server B only knew how many of its own reservations were used; it had to be conservative about the other servers. To allow error bounds to dynamically shrink, we have each server *allocate* reservations when needed and keep track of the allocated reservations in the shared datastore. Allocating must be done with strong consistency to ensure all servers agree, which can be expensive. However, we can use long leases (on the order of seconds) to allow servers to cache their allocations. When a server receives some writes, it allocates some reservations for itself. If it consistently needs more, it can request more, and if it is still using those reservations when the lease is about to expire, it preemptively refreshes its lease in the background so that writes do not block.

For each type of update operation there may be a different pool of reservations. Similarly, there could be multiple error bounds on read operations. It is up to the designer of the ADT to ensure that all error bounds are met with the right set of reservations. For instance, the full implementation of a Counter includes decrement operations. These require a different pool of reservations to ensure that there are never more decrements than the error bound permits.

In some cases, multiple operations may consume or produce reservations in the same pool. Consider a `Set` with an error bound on the size. This requires separate reservation pools for `add` and `remove` to prevent the overall size from deviating by more than the desired error bound. In this case, we calculate the interval for `size` to be:

```
Interval(min = v - removePool.delta()
         max = v + addPool.delta())
```

Where `v` is the size of the set read from the datastore, and `delta` is the number of possible outstanding operations from the pool, or:

```
delta(): pool.total - (pool.local - pool.used)
```

It is tempting to try to combine reservations for inverse operations into the same pool. For instance, it would seem that decrements would cancel out increments, allowing a single reservation server receiving matching numbers of each to continue indefinitely. In some situations, such as if sticky sessions can guarantee ordering from one reservation server to one replica, this is sound. However, in the general case of eventual consistency, this would not be valid, as the increments and decrements could go to different replicas, or propagate at different rates. Therefore it is crucial that ADT designers think carefully about the guarantees of their underlying datastore. Luckily, the abstraction of ADTs hides this complexity from the user — as long as the ADT is implemented correctly, they need only worry about the stated error bounds of the type they are using.

3.4. Provided by IPA

The IPA System implementation provides a number of primitives for building ADTs as well as some reference implementations of simple datatypes. We show some in Figure 2. To support latency bounds, there is a generic `Rushable` trait that provides facilities for executing a specified read operation at multiple consistency levels within the specified time limit. For implementing error bounds, IPA provides a generic reservation pool which ADT implementations can use.

The IPA system currently has a small number of datatypes implemented:

- `Counter` based on Cassandra's counter datatype, supporting increment and decrement, with latency and error bounds
- `Set` with `add`, `remove`, `contains` and `size`, supporting latency bounds, and error bounds on `size`.
- `BoundedCounter` CRDT from [6] that enforces a hard lower bound even with weak consistency. Our implementation adds the ability to bound error on the value of the counter, and set latency bounds.
- `UUIDPool` that generates unique identifiers, but has a hard limit on the number of ids that can be taken from it, built on top of `BoundedCounter` and supporting the same bounds.
- `List`: thin abstraction around a Cassandra table with a time-based clustering order, with latency bounds. Used to implement Twitter timelines and Ticket listings.

[#fig-intefaces] shows conceptual-level Scala code using reservation pools to implement a Counter with error bounds. The actual implementation splits this functionality between

```
trait Rushable {
  def rush[T](bound: Duration,
              readOp: ConsistencyLevel => T): Rushed[T]
}

/* Generic reservaton pool, conceptually one per ADT instance
 * `max` recomputed as needed (e.g. for percent error) */
abstract class ReservationPool(max: () => Int) {
  def take(n: Int): Boolean // try to take some tokens
  def sync(): Unit          // sync to regain used tokens
  def delta(): Int          // possible ops outstanding
}

/* Counter with ErrorBound (simplified) */
class Counter(key: UUID) with ErrorBound {
  def error: Float // error bound

  def calculateMax(): Int = (cass.read(key) * error).toInt

  val incrPool = ReservationPool(computeMax)
  val decrPool = ReservationPool(computeMax)

  def value(): Interval[Int] = {
    val v = cass.read(key)
    Interval(v - decrPool.delta,
             v + incrPool.delta)
  }

  def incr(n: Int): Unit = {
    waitFor(incrPool.take(n)) {
      cass.incr(key, n)
    }
  }
}
```

Figure 2. Example facilities provided by IPA.

the client and the reservation server. It is also all implemented using an asynchronous futures-based interface to allow for sufficient concurrency, based on the Phantom Scala client for Cassandra [@Phantom]. The Reservation Server is similarly built around futures using Twitter's Finagle framework. Communication is done between clients and Cassandra via prepared statements to avoid excessive parsing, and Thrift remote-procedure-calls between clients and the Reservation Servers.

4. Evaluation

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

4.1. Counter microbenchmark

- Latency bound

Condition label	Latencies (ms)		
Local	<1	<1	<1
Uniform/High load	5	5	5
Slow replica	10	10	100
Google	1 ± 0.3	110 ± 5	160 ± 5
Amazon	1 ± 0.3	80 ± 10	200 ± 50

Table 1. Simulated network conditions

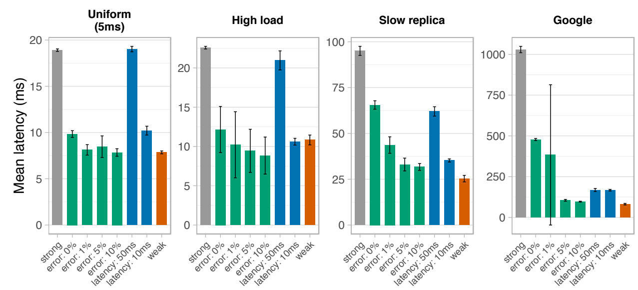

Figure 3. Counter benchmark: mean latency for a random mix of counter ops (20% increment, 80% read), with various IPA bounds, under various conditions.

  - 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 3 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 favorable conditions, when latencies and load are low, it is often possible to achieve strong consistency. Figure Figure 3 shows the average latency of

4.1.2. Error bounds

We use the reservation system described in [#reservations] to enforce error bounds. Though error bounds represent a relaxation from a strict strongly consistent read, they still have Our

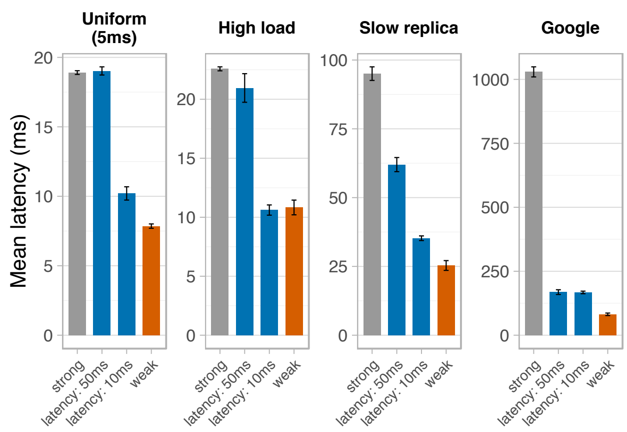

Figure 4. Consistency of latency-bound operations. Strong consistency is rarely possible within 10ms. With one slow replica, most reads can still achieve strong consistency, but with high network latencies or heavy load, it degrades to use weak consistency.

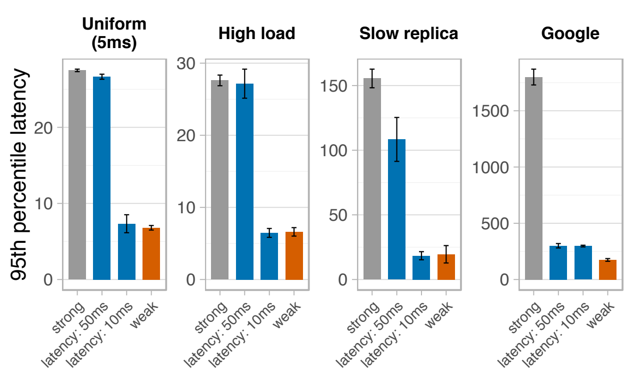

Figure 5. Latency bounds reduce unpredictable tail latency.

goal is to explore how expensive it is in practice to enforce these error bounds, and in particular to determine if reasonable error bounds that programs could rely on are achievable with performance comparable to weak consistency.

The general intuition behind reservations is to move synchronization off the critical path: by distributing write permissions among replicas, clients can get strong guarantees while only communicating with a single replica. This shifts the majority of the synchronization burden off of reads, which are typically more common. However, this balance must be carefully considered when evaluating the performance of reservations, more so than the other techniques.

Figure 3 contains results for error bounds ranging from 0% to 10%, showing the average latency of both reads and increments for a mix with 20% increments. We can see that how narrow the bounds are does affect performance, but in

Error Bound	80% read	99% read
0%	0	0
1%	5.4	0.03
5%	45.7	1.36
10%	98.2	3.5

Table 2. Reservations adapt to varying write loads, resulting in narrower intervals (tighter error bounds) than clients requested. [add more mixes, gen line plot with %reads on x axis]

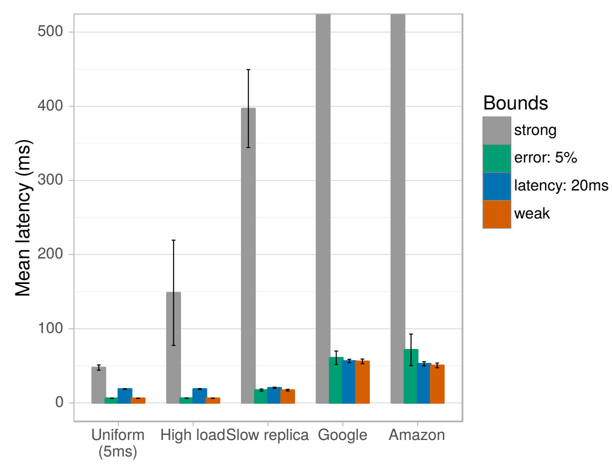

Figure 6. Ticket-sales app: mean latency of `purchase` action under various conditions. The `BoundedCounter` underlying ticket sales is safe even when weakly consistent, but latency bounds allow users to see strong consistency [??]% of the time, while reservations bound error to less than 5% with similar performance.

most cases, the latency of 5-10% error bounds have roughly the same performance as weak consistency.

[plot of actual error with weak consistency compared to bounded error]

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 [31, 32])
- 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* [18] and Herlihy's *linearizability* [17] on the strong side, to *eventual consistency* [30] 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* [28], 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 [7] on an object or namespace granularity, as well as others [19, 27].

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 [29], 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 [16], leases have since been used in a multitude of ways: in Facebook's Memcache system [21] for invalidations, Google's Chubby [10] and Spanner [12] to adjust the frequency of heartbeat messages, and on mobile clients with exo-leases [26]. Warranties [20] 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) [25] 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, 11] 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> [8]
- Conit-based Continuous Consistency Model [33]

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] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys, pages 6:1–6:16, New York, NY, USA, 2015a. ACM. ISBN 978-1-4503-3238-5. doi:10.1145/2741948.2741972.

[6] Valter Balegas, Diogo Serra, Sergio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. *34th International Symposium on Reliable Distributed Systems (SRDS 2015)*, September 2015b.

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

[8] 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.

[9] Eric A. Brewer. Towards robust distributed systems. In *Keynote at PODC (ACM Symposium on Principles of Distributed Computing)*. Association for Computing Machinery (ACM), 2000. doi:10.1145/343477.343502.

[10] 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.

[11] 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.

[12] 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.

[13] Datastax, Inc. How are consistent read and write operations handled? http://docs.datastax.com/en/cassandra/3.x/cassandra/dml/dmlAboutDataConsistency.html, 2016.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi:10.1145/1294261.1294281.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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 ser-*

vices - MobiSys 03, MobiSys. Association for Computing Machinery (ACM), 2003. doi:10.1145/1066116.1189038.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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`.

[32] 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.

[33] 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.