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 used to make performance/correctness tradeoffs in two applications: a ticket sales service and a Twitter clone. Our evaluation shows that IPA's runtime system can enforce the programmer-specified performance and correctness bounds, achieving latencies comparable with weak consistency, 2–10× better than strong consistency under a variety of adverse conditions.

1. Introduction

To provide good user experiences, modern datacenter applications and web services must balance many competing requirements. At a minimum, they must be scalable, highly available, and fault tolerant. On top of that, they often wish to guarantee certain response times to meet contractual service level agreements (SLAs) or to keep user engagement high; for example, Microsoft, Amazon and Google have all noted that every additional millisecond of latency translates directly to a loss in traffic and revenue [22, 31]. On the other hand, every application has correctness criteria that can be hard requirements, such as never double-charging a user for a purchase, or a soft requirement, such as always showing the most recent tweets.

Developers of these applications typically balance these performance and correctness requirements by trading off consistency. To that end, the replicated NoSQL datastores used to implement these services, such as Cassandra [5] and Riak [10], often support multiple levels of consistency: from *linearizability* all the way down to *eventual consistency*. Whenever part of an application is not scaling well, or response latencies are unacceptably high, developers can choose to perform some operations with weaker consistency. However, they must then understand what can go wrong now that new reorderings of operations are possible and handle them. Failure to do so can lead to lost updates, duplications, stale data, and more. Relaxed consistency models are notoriously difficult to understand, especially when an application intertwines several models for different records. It is dangerously easy for inconsistencies to leak into operations that were intentionally kept as strongly consistent.

Even once an application's consistency model is tuned and tested to meet performance targets for one execution environment, the conditions in which web services operate is in constant flux. Sharing resources with many other colocated services, each of which also undergoes occasional garbage collection or operating system pauses, leads to unpredictable performance. Moreover, traffic coming in, from the outside world or via other services, can be highly unpredictable: when a black and blue dress goes viral [36], or Justin Bieber posts a selfie [33], the web service is subject to highly irregular load that overloads some nodes. In unusual situations (such as the World Cup or a natural disaster), traffic usually restricted to a single datacenter may cross multiple geo-replicated datacenters, with significantly different performance and consistency characteristics.

Using today's datastore programming models, it is difficult and often impossible to handle this wide variety of conditions within a single application. Interfaces to datastores such as Cassandra offer consistency control at the level of individual operations, but provide no support for application-level reasoning about consistency. Failed by these abstractions, developers face a choice. One option is to opt for strong consistency, making the program easier to reason about but with overheads so high — 2–10× slower in our experiments — that the service falls over under even slightly adverse conditions. The other option is to choose a worst-case scenario to target and build the application to handle it, but with the risk

of inconsistency causing application failures even during normal, good conditions, and with significant productivity costs.

We need a better programming model to help developers build fast and correct distributed applications that adapt to changing conditions. Current abstractions for distributed datastores do not offer the level of safety and abstraction that developers expect from modern programming languages. Unfortunately, the CAP theorem [13, 24] tells us that developers will always need to balance correctness against performance. But the right programming language support can help developers to make these tradeoffs efficiently and, most importantly, safely. Moreover, programming language support for application-level reasoning about consistency offers semantic information that opens new runtime optimizations that are unavailable to applications where consistency levels are specified at the per-operation level.

We propose the *inconsistent, performance-bound, approximate (IPA)* programming model for distributed datastores. The IPA model provides a type system for which type safety implies *consistency safety*: values from weakly consistent operations cannot flow into stronger consistency operations without explicit endorsement. Applications built with the IPA model allow developers to annotate *abstract data types* with performance targets, so the system can automatically adapt to meet them when conditions change. IPA annotations also express where incorrectness is acceptable, to give the system a release valve where it can relax consistency to meet those performance goals. The keys to our model are *IPA types* that express the consistency and correctness of any potentially inconsistent values. Type checking ensures that programmers handle potential error cases, and subtyping allows applications to make decisions based on the actual consistency of data.

We present the IPA model instantiated with types and annotations that implement two important distributed runtime mechanisms: latency-bound operations, and a novel *error tolerance* reservation system. We describe how these mechanisms can be implemented in a distributed environment based on Cassandra, and explain how the IPA programming model allows the system to trade off performance and consistency, safe in the knowledge that the type system has checked the program for consistency safety. We demonstrate experimentally that these mechanisms allow applications to dynamically adapt the correctness and performance trade-off with data-structure microbenchmarks and two applications: a simple Twitter clone based on Retwis [43] and a Ticket sales service modelled after FusionTicket [1]. Our results show that IPA applications adapt to changing execution environments while respecting application-level correctness requirements.

2. Background

There are many popular news stories about different services like BuzzFeed [36] and Instagram [33] and their struggles with unpredictable Internet traffic. Before launching into the programming model and implementation, we will motivate the difficulties with implementing web services by delving more deeply into one example — a ticket sales service — which we will use throughout the rest of the paper.

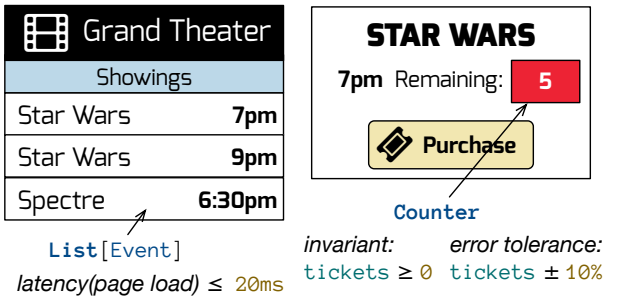


Figure 1. Ticket sales service with performance and correctness requirements.

In October 2015, movie ticket pre-sales became available for the much-anticipated new movie in the Star Wars franchise. The demand for these tickets was so high that many movie ticket sites, including big players such as AMC and Fandango, saw catastrophic performance drops; some smaller vendors even had crashes which resulted in loss of purchases and significant media backlash [17]. Missing the opening night of a movie may not be the most dire of circumstances, but it illustrates that web services must be prepared for all kinds of situations, even if they only happen in the smallest minority of instances.

Let's look at how we would model this application using existing datastore options. Figure 1 shows a high-level schematic of a ticket sales application. Each event (in this case movie, location, and time) has some number of available tickets. Very large events such as big conventions or conferences are likely to cause even more problems than movie showings if tickets are likely to sell out quickly. Visitors to the site may browse the events, and *view* individual events to see how many tickets are remaining. Finally, they can *purchase* some tickets.

The most important invariant is that tickets are not oversold: each available ticket must only go to one user. However, there are other, softer, constraints for this application:

1. Users browsing many events will become frustrated if pages take too long to load, so there is an implicit latency target.
2. Users want to know how many tickets are remaining to determine how quickly they need to purchase them.

If the latency target is not met, users may choose to take their business to other sites. Similarly, if the remaining ticket count is off, such as if the count is stale and there are in fact fewer remaining tickets, then users may not get the tickets they wanted. Some amount of inaccuracy in the count is

unavoidable; there is an absolute bound on the round trip time, for instance, but we may hope to keep this count as up-to-date as possible.

Using existing systems, we have few options for meeting these requirements. In order to ensure that the ticket count does not go negative, we need linearizability [26, 45]. In Cassandra, we cannot do this with the `Counter` datatype but could with their "lightweight transactions" on a normal record, but this makes reads of the count prohibitively slow. We could use a second, weakly consistent counter, to approximate the remaining ticket count; this could be arbitrarily off and could drift if we do not synchronize periodically with the true value. Overall, it introduces significant implementation complexity. A convergent replicated data type (CRDT) [44] called a `BoundedCounter` [9] can enforce the invariant we want even on eventual consistency, which gives good performance and safety, but does not give us any bound on how accurate the count is at any time. What we need is a *datatype* that encodes the hard lower-bound and the soft error bound, which we will in §3.2 and §3.4.

Ensuring low-latency browsing is also difficult. If we blindly use weak consistency, then users could occasionally miss new events (especially relevant to anyone refreshing waiting for Star Wars tickets), or see events that should have been deleted. Internally, we must now handle new potential inconsistencies, such as late actions on deleted events. During low-traffic times, these inconsistencies may be unnecessary. If we know the load is low, we should try to use stronger consistency to retrieve the results, and in times of heavy load, we could indicate to the user that the results may be inaccurate.

3. 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 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. §4 will explain how the type system's guarantees are enforced for a distributed datastore.

3.1. 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 [41]. 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 in a constantly changing environment.

3.2. Abstract Data Types

The base of the IPA type system is a set of abstract data types (ADTs) for common distributed data structures. Though the simplest key/value stores only support primitive types like strings for values, many popular datastores now have built-in support for more complex data structures such as sets, lists, maps, and other specialized types. However, each datastore has its own interface to these types: Redis's [42] types are implicit in the command used to operate on them, Cassandra [28] uses a more relational-style model with tables and columns and composite keys, while Riak [10] and Hyperdex [21, 27] support more direct data structure access. Our library of ADTs allows us to decouple our type system from any particular datastore, though our reference implementation is on top of Cassandra, similar to [45].

Besides abstracting over various storage systems, ADTs are an ideal place from which to begin reasoning about consistency and system-level optimizations. ADTs present a clear abstract model through a set of operations which query and update the state, allowing users and systems alike to reason about their logical, algebraic properties rather than the low-level operations used to implement them. Moreover, as we will see in the coming sections, annotating datatypes, rather than individual operations, is crucial because ADTs are where writes and reads meet and affect one another.

In the ticket sales application, each event (i.e., movie screening) has an instance of the `Counter` ADT to track the number of tickets available. The counter provides a read method to get its current value, and increment and decrement operations to update its value. Describing the ticket sales behavior as an ADT allows the system to reason more strongly about the values by, for example, ruling out arbitrary state updates.

3.3. Policy Annotations

The IPA type system provides a set of annotations that can be placed on ADT instances to specify consistency policies. Previous systems [5, 10, 30, 46, 48] 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. In the case of a conflict between two bounds, the system can be conservative and choose stronger policies than specified without affecting correctness.

In the ticket sales application, the counter ADT for each event's tickets has an accuracy policy `ErrorTolerance(10%)`. This relaxed accuracy policy allows the system to quickly read the count of tickets remaining. An accuracy policy is appropriate here because it expresses a domain requirement—users want to see accurate ticket counts. As long as the system meets this requirement, it is free to relax consistency and maximize performance without sacrificing application quality. The list ADT for events has a latency policy, which also expresses a domain requirement—that pages on the website load in reasonable time.

3.4. IPA Types

The keys to the IPA type system are the IPA types themselves. Read operations of ADTs annotated with consistency policies return instances of an *IPA type*. These IPA types track the consistency of the results, and enforce a fundamental noninterference 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 [12], which uses types to track the quality of results computed on approximate hardware. Their noninterference 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.

3.4.1. 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, which corresponds to the control flow in conventional (non-distributed) applications.

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 §3.3 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, omitted cases can be matched by any type lower in the lattice, including the bottom element `Inconsistent(_)`; other cases therefore need only be added if the application should respond differently to them. This subtyping behavior allows applications to be portable between systems supporting different forms of consistency (of which there are many).

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 values within which 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 [11].

The key invariant of the `Interval[T]` type is that uses of the interval are *indistinguishable* from a linearizable order. For example, consider a `Set` with 100 elements, with its `size` operation annotated with error tolerance of 5%. With linearizability, if we `add` a new element, then read the `size` (or if there is any way to know that the one precedes the other), we must get back 101 (provided no other updates are occurring). However, 5% error tolerance means that `size` could return $[95, 105]$, or $[100, 107]$, among many others. In which case the client is unable to tell if the `add` was incorporated or not. This frees the underlying system up to apply a number of optimizations, including delaying synchronization, that can significantly improve performance.

In theory, any type with a partial order over values could be represented as an interval (e.g. a Set with some items that may or may not be in the set). However, in this work we limit them to numeric types.

In the ticket sales example, the counter ADT's accuracy policy means that reads of the number of tickets return an `Interval[Int]`. If this interval does not contain zero, then it is safe (in the absence of intervening events) to sell a ticket. Because the interval is indistinguishable from a linearizable order, the write event to sell the ticket will succeed. The relaxed consistency of the interval type allowed the system to optimize performance in the common case where there are many tickets available, and dynamically adapt to contention when the ticket count diminishes.

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.

4. 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.

4.1. 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 [19] 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 [5] and Riak [10] 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 [50], 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 [13]), 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`) [18]. Cassandra additionally supports limited linearizable ([26, 29]) 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.

4.2. 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.

4.2.1. 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).

4.2.2. 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.

4.3. Error bounds

We implement error bounds by building on the concepts of *escrow* and *reservations* [23, 37, 39, 40]. 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 wishes to decrement, it must first acquire sufficient tokens before performing the update operation. Correspondingly, in

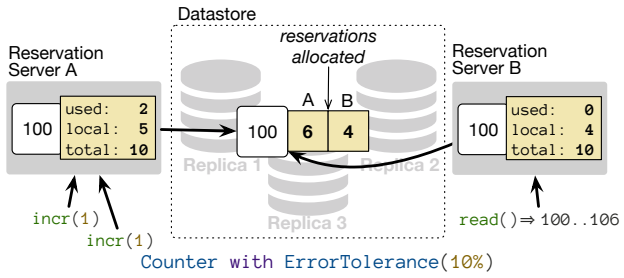


Figure 2. Reservations enforcing error bounds.

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.

4.3.1. 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 [9]. 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.

4.3.2. 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 [8, 9], or logical assertions [32]. 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 2. 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 2, 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 $[100, 106]$.

4.3.3. 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())
```

```
trait LatencyBound {
  // execute readOp with strongest consistency possible
  // within the latency bound
  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 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 computeMax(): 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 3. Example facilities provided by IPA.

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.

4.4. 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 3. 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 [9] 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.

Figure 3 shows conceptual-level Scala code using reservation pools to implement a Counter with error bounds. The actual implementation splits this functionality between 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 [38]. 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.

5. Evaluation

Distributed applications must be able to run in a constantly changing environment. To evaluate IPA's ability to help programmers cope with extremes in performance, we answer the following questions in this section:

1. Do IPA's techniques meet the stated latency and error bounds under varying network conditions?
2. Can IPA enforce these bounds with reasonable overhead for a range of network conditions?
3. How do different bounds affect performance under different network conditions?

We answer these questions first using microbenchmarks to understand the performance of individual data types and ex-

plore each bound in isolation. Then, we move on to some miniature applications built using these data types, employing bounds tailored to their use cases. However, before discussing the results, we begin by detailing how we simulate changing network conditions in a controlled manner.

5.1. Simulating adverse conditions

To evaluate these bounding techniques, it is crucial to subject them to a variety of conditions. One typically has two two extreme choices when evaluating this kind of system: perform experiments in a controlled environment where latencies are typically very low and performance variability is negligible, or to throw them into a complex, real-world system, subject to the whims of unpredictable network conditions and resource sharing and try to piece together how they performed. A third option is to *simulate* a variety of environments, chosen to stress the system or mimic reality, within a more controlled environment.

In our experiments, we employ all three to best understand how these techniques behave. On our own test cluster, with standard ethernet linking nodes within the same rack, we run controlled experiments, simulating adverse network conditions. We use Linux's Network Emulation facility [49] (`tc netem`) to introduce packet delay and loss at the operation system level. We use Docker containers [20] to enable fine-grained control of the network conditions between processes on the same physical node (`netem` is one of the properties isolated within the container).

Table 1 shows the set of conditions we use in our experiments to explore the behavior of the system. To simulate latencies within a well-provisioned datacenter, we have a *uniform 5ms* condition which is predictable and reliable but slower than our raw latency which is typically less than 1ms. Another condition demonstrates what happens when one replica is significantly slower to respond than the others either due to imbalanced load or hardware problems. Finally, we have two conditions mimicking globally geo-replicated setups with latency distributions based on measurements of latencies between virtual machines in the U.S., Europe, and Asia on Google Compute Engine [25] and Amazon EC2 [4].

5.2. Microbenchmark: Counter

We start by measuring the performance of a very simple application that randomly increments and reads from a number of counters with different IPA policies. Random operations (`incr(1)` and `read`) are uniformly distributed over 100 counters from a single multithreaded client (using Scala futures to allow up to 4000 concurrent operations).

5.2.1. 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 4 shows the average latency of a counter with strong, weak,

Network Condition	**Latencies (ms)**		
Simulated	*Replica 1*	*Replica 2*	*Replica 3*
Uniform / High load	5	5	5
Slow replica	10	10	100
Geo-replicated (EC2)	1 ± 0.3	80 ± 10	200 ± 50
Actual	*Replica 1*	*Replica 2*	*Replica 3*
Local (same rack)	<1	<1	<1
Google Compute Engine	30 ± <1	100 ± <1	160 ± <1

Table 1. Network conditions for experiments: latency from client to each of the replicas, with standard deviation if significant.

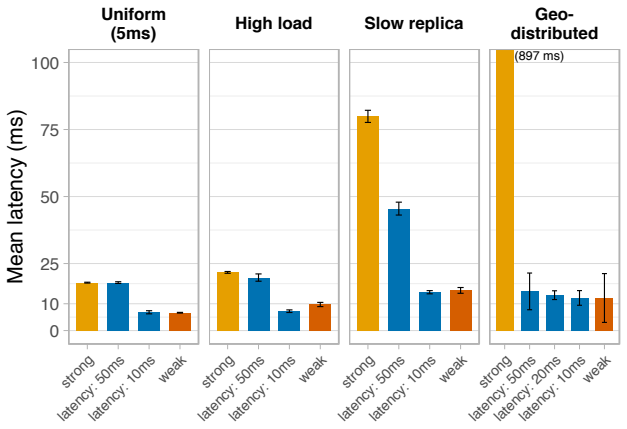


Figure 4. *Counter microbenchmark: latency bounds*. Strong consistency is rarely possible within 10ms bound, but for the 50ms bound, in low-latency conditions, it achieves strong consistency, and with high-latency conditions, weak. [find Local results]

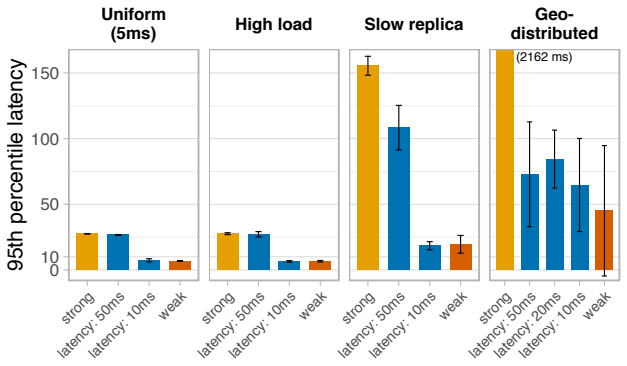


Figure 5. *Counter: 95th percentile latency*. Latency bounds help keep tail latency down by backing off to weak consistency when necessary.

and 2 latency-bounds under various conditions. We can see that there is a significant difference in latency between strong and weak. In these conditions, it is almost never possible to get strong consistency within 10ms, so the 10ms-bound counter predicts it will not get strong consistency uses weak consistency. In the case of the 50ms bound, in the cases with low latency it gets strongly consistent results. With one slow replica (out of 3), there is a chance that the `QUORUM` read needed for strong consistency will go to the slow replica, so it attempts both; in this case, 30% returned weak after the strong read timed out. Finally, with our simulated geo-replicated environment, there are no 2 replicas within 50ms of each other, so strong consistency is never possible in 50ms, so the monitor adapts only attempt weak reads in both cases.

Figure Figure 5 shows the 95th percentile latencies of the same workload. We see that the tail latency of the 10ms bound is comparable to weak, though the 50ms bound guesses incorrectly occasionally for the case of the slow replica. We see a latency gap between the latency-bound and weak in the geo-distributed case. This is because the `weak` condition uses weak reads *and* writes, while our rushed types, in order to have the option of getting strong reads without requiring a read of `ALL`, must do `QUORUM` writes.

5.2.2. Error bounds

We use the reservation system described in §5.2.2 to enforce error bounds. Error bounds represent an intermediate level between strict strongly consistent reads and inconsistent reads that could, in theory, return anything. Our goal is to explore how expensive it is in practice to enforce these error bounds, and in particular to determine what error bounds 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. When evaluating the latency bounds, we only considered the read latency because we didn't change the writes. However, reservations actually slow down writes, so we must consider that effect.

Figure 6a shows latencies for error bounds ranging from 0% to 10%, The *read* latency for error bounds is always equivalent to *weak*, so we plot the average of reads *and* increment operations combined to understand the overall performance. We can see that tighter bounds increase latency because it forces more synchronization operations, which must use consistency of `ALL`. Under high load we see higher variance in performance: if too many increments arrive together, some end up waiting for reservations. In most conditions, we see that 5-10% error bounds have comparable latency with weak, with the exception of the geo-distributed condition, where it seems that at least this implementation of reservations is not a good solution.

We also wish to know how much error actually happens in practice, and how this compares with our error bounds. We modified our benchmark to be able to observe the error resulting from weak consistency: using a single multi-threaded client, we keep track locally of how many increments to each counter we have done; then when we have completed a random, predetermined number of increments, we stop and read the count. The resulting error measurements are shown in Figure 6b. We plot the percent error of weak and strong against the actual observed interval width for a 1% error bound, going from a read-heavy (1% increments) to a write-heavy (all increments, except to check the value).

First of all, the *mean* error of weak is significantly less than 1%. This verifies that inconsistency really is pretty rare; it is probably higher in practice when operations come from more than one client, but then we would not have been able to observe the error in the way we did. However, even with this experiment, we see that there are outliers with significant error when writing is heavy: up to as high as 60% error in the geo-replicated case. These maximum errors averaged over several experiments. Finally, it is worth noting that the average interval width is less than the maximum of 1% and is lower for more read-heavy workloads that do not need as many reservations.

5.3. Applications

Next, we explore how the IPA system performs on two miniature application benchmarks. In addition to running with our simulated network conditions, for these applications we also ran some experiments on a real globally-distributed network. In Google Compute Engine [25], we ran virtual machines in 4 different datacenters: the client in `us-east`, and the datastore replicas in `us-central`, `europe-west`, and `asia-east`.

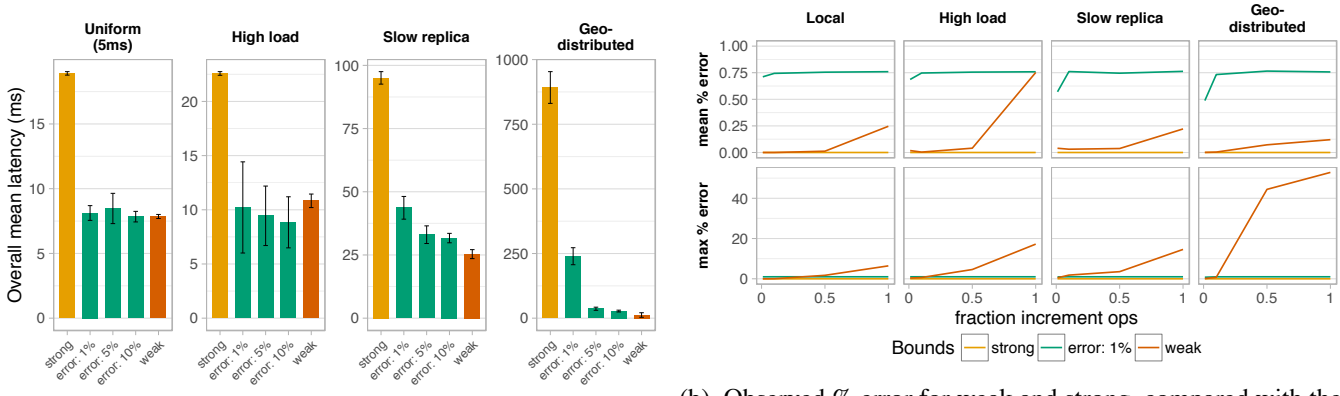
5.3.1. Tickets

Our Ticket sales web service, introduced in §2, is modelled after FusionTicket [1], which has been used as a benchmark in recent distributed systems research [51, 52]. We support the following actions:

- `browse`: List events by venue
- `viewEvent`: View the full description of an event including number of remaining tickets
- `purchase`: Purchase a ticket (or multiple)
- `addEvent`: Add an event at a venue.

Event listings by venue are modelled using a `List` ADT. Tickets are modelled using a `Pool` type that generates unique identifiers as proof of purchase, using a `BoundedCounter` at its base to ensure that, even with weak consistency, it never gives out more than the maximum number of tickets.

Our workload attempts to model typical use of a small-scale deployment: we start with 50 venues and 200 events,



(a) Mean latency (increment *and* read).

(b) Observed % error for weak and strong, compared with the actual interval widths returned for 1% error tolerance.

Figure 6. *Counter benchmark: error tolerance*. In (a), we see that wider error bounds reduce mean latency because fewer synchronizations are required, matching *weak* around 5-10%. In (b), we see that error increases with heavier write workloads. The average error for *weak* is less than 1%, but the *maximum* error can be extremely high (up to 60%).

with an average of 2000 tickets each (gaussian distribution centered at 2000, stddev 500). We chose the ticket to event ratio so that during the course of a run, we should have some events run out of tickets. This is important because the behavior of the `Pool` changes as it runs out of tokens (this is true for all bounds because the `BoundedCounter` has its own reservations for ensuring the lower bound, it is doubly true for the `Pool` with error bounds, which also has less margin for error at the end). Real-world workloads exhibit power law distributions [16], where a small number of keys are much more popular than the majority. We model event popularity using a Zipf (power law) distribution with a coefficient of 0.6, which is moderately skewed.

Figure 7 shows the average latency of a workload consisting of 70% `viewEvent`, 19% `browse`, 10% `purchase`, and 1% `addEvent`. We plot it with a log scale because strong consistency is consistently over 5x higher latency. The `purchase` event, though only 10% of the workload, drives most of the latency increase in this workload because of the additional work required in the `BoundedCounter` to prevent over-selling tickets. We explore two different implementations: one with a 20ms latency bound on all ADTs, aiming to ensure that both `viewEvent` and `browse` complete quickly, and one where the ticket pool size ("tickets remaining") has a 5% error bound. We see that both perform with nearly the same latency as weak consistency. With the low-latency condition (*uniform* and *high load*), 20ms bound does 92% strong reads, 4% for *slow replica*, and all weak on both *geo-distributed* conditions.

This plot also shows results on Google Compute Engine *(GCE)*. We see that the results of real geo-replication confirm the findings of our simulated geo-distribution results (which were based on measurements of Amazon EC2's US/Europe/Asia latencies).

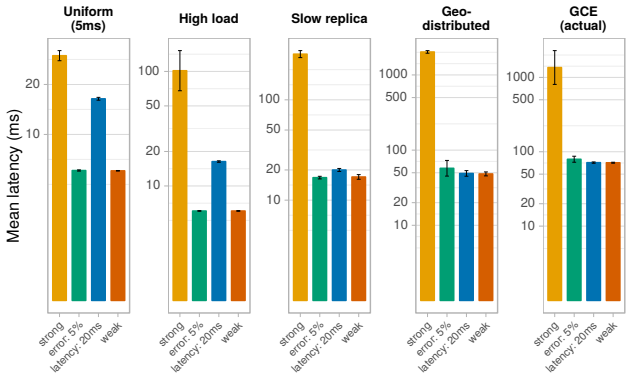


Figure 7. *Ticket-sales app: mean latency*, ***log scale.*** Strong consistency is far too expensive (>10× slower) except when load and latencies are low, but 5% error tolerance allows latency to be comparable to weak consistency. The 20ms latency-bound variant is either slower or defaults to weak, providing little benefit. Note: the ticket `Pool` is safe even when weakly consistent.

On this workload, we observe that the reservations used for the 5% error bound perform well even with high latency, which is different than our findings for the counter. This is because, in this case, the `Pool`s start out *full*, with sufficient reservations to be distributed among the replicas so that they can usually complete locally. Contrast this with the `Counter` experiments, where they start at typically smaller numbers (average initial value less than 500).

```
User(id: UserID,
  name: String,
  followers: Set[UserID] with LatencyBound(20ms),
  timeline: List[TweetID] with LatencyBound(20ms)
)

Tweet(id: TweetID,
  user: UserID,
  text: String,
  posted: DateTime,
  retweets: Set[UserID] with Size(ErrorTolerance(5%))
)
```

Figure 8. Twitter application's (simplified) data model, with latency bounds for followers and timelines, and error tolerance for the number of retweets (the `size` of the retweets set).

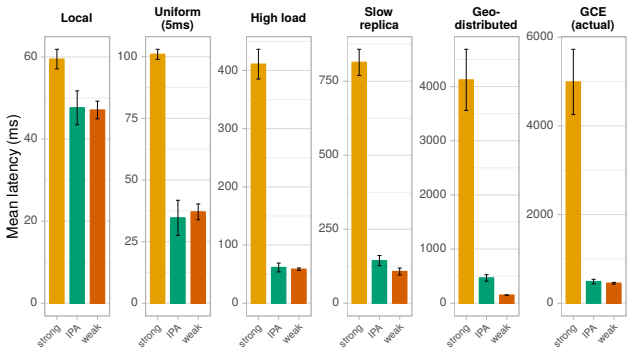


Figure 9. *Twitter clone: mean latency (all actions).* [fixed experiments in progress]

5.3.2. Twitter clone

Our second application benchmark is a Twitter-like service based on the Redis data modelling example, Retwis [43]. The data model is simple: each user has a `Set` of followers, a `Set` they users they follow, and a `List` of their tweets. Each user's timeline is kept materialized as a `List` of tweet IDs — when a user tweets, the new tweet ID is eagerly appended to all of their followers' timelines. Retweets are tracked with a `Set` of users who have retweeted each tweet. The retweet count, loaded for each tweet when a timeline is loaded, is represented in this ADT model by the size of the set.

Retweets are another good example of a place where error tolerance bounds faithfully represent an application-level correctness constraint. Most tweets by an average user on Twitter will have few retweets; in these situations, even a small error can be glaringly obvious: a user may get a notification, then look at the tweet and get frustrated when they cannot see the retweet. However, for a highly popular tweet that has been retweeted millions of times already, one more tweet does not need to be reflected immediately in the count. In fact, most views of Twitter will truncate large values to something like "3.4M", which is a perfect way to use an `Interval` result. Moreover, situations with massive numbers of retweets can be a performance bottleneck if not handled correctly, as Twitter learned when Ellen Degeneres's celebrity selfie brought it to a standstill at the 2014 Oscar's [6]. For the user's timeline, on the other hand, we do not need to know the size, but we do want it to load quickly to keep users engaged, so we use a latency bound. Finally, the follower list, which is changed infrequently, could be left as strongly consistent to ensure that new followers get all the latest tweets, but we want to keep performance as a priority, so use another latency bound, which gives us the option to warn the user if they may need to refresh to get more tweets.

Retwis doesn't specify a workload, so we simulate a realistic workload by generating a synthetic power-law graph, using a Zipf distribution to determine the number of followers per user. Our workload is a random mix with 50% `timeline` reads, 14% `tweet`, 30% `retweet`, 5% `follow`, and 1% `newUser`.

We can see in Figure 9 that for all but the local (same rack) case, strong consistency is over 4× slower, but our implementation combining latency and error-bounds performs comparably with weak consistency, but with stronger guarantees for the programmer. Our simulated geo-distributed condition turns out to be the worst-case scenario for IPA's Twitter, with latency over 2× slower than weak consistency. This is because weak consistency performed noticeably better on our simulated network, which had one very close (1ms latency) replica that it could use almost exclusively.

6. Related Work

6.1. Consistency Models

A vast number of consistency models have been proposed over the years. From Lamport's *sequential consistency* [29] and Herlihy's *linearizability* [26] on the strong side, to *eventual consistency* [50] 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. Session guarantees, including *read-your-writes*, strengthen ordering for individual clients but reduce availability [47]. Many datastores support configuring consistency at a fine granularity: Cassandra [5] per operation, Riak [10] on an object or namespace granularity, as well as others [30, 46].

The Conit consistency model [53] explores the spectrum between weak and strong consistency, breaking consistency down along three axes: numerical error, order error, and staleness, with algorithms to bound each of them. The programming model, however, requires annotating each operation and making dependencies explicit in order to track these metrics, rather than annotating ADTs as in IPA.

6.2. Explicit correctness requirements

Some programming models have gone beyond plain consistency models and allowed programmers to express correctness criteria directly. Quelea [45] has programmers write *contracts* to describe *visibility* and *ordering* constraints between operations, then the system automatically selects the consistency level for each operation necessary to satisfy all the contracts. The contracts are independent of any particular consistency hierarchy; applications are portable, provided the constraint solver can find a solution given the consistency levels supported by the target platform. Quelea encourages programmers to use contracts to describe the semantics of application-specific datatypes which can later be reused.

In Indigo [8], programmers write *invariants* over abstract state and state transitions and annotate post-conditions on actions to express their side-effects in terms of the abstract state. With these in place, the system can determine where in the program these invariants could be violated and adds coordination logic to prevent it. Indigo uses a similar reservation system to enforce numeric constraints. Neither Indigo nor Quelea, however, allow programmers to specify approximations or error tolerances, nor do they enforce any kind of performance bounds.

6.3. Explicit performance bounds

The IPA model's latency-bound policies were inspired by Pileus's *consistency-based SLAs* [48]. Consistency SLAs specify a target latency and consistency level (e.g. 100 ms with read-my-writes), associated with a *utility*. Each operation specifies a set of SLAs, and the system predicts which is most likely to be met, attempting to maximize utility, and returns both the value and the achieved consistency level.

Consistency SLAs are more expressive than IPA's latency bounds, allowing multiple acceptable targets to be indicated and weighted. On the other hand, the IPA type system provides more assistance in working with the results: `Rushed[T]` values can statically ensure that all possible cases are handled, and protect programmers from inadvertently using weak values where they should not. Additionally, Pileus's SLAs are specified on individual read operations, a greater annotation burden, and preventing the potential optimizations to writes that come from coupling the two at the ADT level.

Probabilistically bounded staleness (PBS) [7] are a form of performance and correctness bound. Using a predictive model, PBS is able to quantify the degree of inconsistency that is likely for a given operation, but it has not been used to directly improve programming models. This kind of knowledge could be applied within IPA's runtime system to make better latency predictions or to provide a new IPA probabilistic IPA type similar to `Interval[T]`.

6.4. Types for distributed systems

Convergent (or *conflict-free*) *replicated data types* (CRDTs) [44] 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 [2, 3, 15] 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.

6.5. Types for approximation

As discussed in §3, IPA's programming model bears similarities with the type systems developed for *approximate computing*, in which values are tagged with their accuracy. EnerJ [41] and Rely [14] track the flow of approximate values to prevent them from interfering with precise computation. Chisel [34] and DECAF [12] extend this information-flow tracking to automatically infer the necessary accuracy of a computation's inputs to achieve application-level quality guarantees. IPA's interval types are similar in motivation to Uncertain<T>'s probability distributions [11] and to a long line of work on interval analysis [35]. IPA targets a different class of applications to these existing approaches. The key difference is the source of approximation: rather than lossy hardware that can lose the true value forever, inconsistent values can be strengthened if necessary by forcing additional synchronization.

7. Conclusion

The IPA programming model provides programmers with disciplined ways to trade consistency for performance in distributed applications. By specifying performance targets in the form of latency bounds, they tell the system how to adapt when conditions change. IPA types help programmers ensure they are handling potential inconsistencies, while providing release valves where the system knows it is acceptable to optimize by allowing uncertainty. The IPA policies, types and enforcement systems are only a sampling of the full range of possibilities for using types to safely deal with weak consistency.

References

[1] Fusion ticket. http://fusionticket.org.

[2] 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.

[3] 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.

[4] Amazon Web Services, Inc. Elastic compute cloud (ec2) cloud server & hosting – aws. https://aws.amazon.com/ec2/, 2016 .

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

[6] Lisa Baertlein. Ellen's Oscar 'selfie' crashes Twitter, breaks record. http://www.reuters.com/article/2014/03/03/us-oscars-selfie-idUSBREA220C320140303, March 2014.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 470–487, 2015. doi:10.1145/2814270.2814301.

[13] 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.

[14] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*, pages 33–52, 2013. doi:10.1145/2509136.2509546.

[15] 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.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing - SoCC 10*. Association for Computing Machinery (ACM), 2010. doi:10.1145/1807128.1807152.

[17] Hayley C. Cuccinello. 'star wars' presales crash ticketing sites, set record for fandango. http://www.forbes.com/sites/hayleycuccinello/2015/10/20/star-wars-presales-crash-ticketing-sites-sets-record-for-fandango/, October 2015.

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

[19] 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.

[20] Docker, Inc. Docker. https://www.docker.com/, 2016.

[21] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex. In *Proceedings of the ACM SIGCOMM Conference*. Association for Computing Machinery (ACM), August 2012. doi:10.1145/2342356.2342360.

[22] Brady Forrest. Bing and google agree: Slow pages lose users. Radar, June 2009. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html.

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

[24] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33 (2): 51, June 2002. doi:10.1145/564585.564601.

[25] Google, Inc. Compute engine — google cloud platform. https://cloud.google.com/compute/, 2016.

[26] 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.

[27] Hyperdex. Hyperdex. http://hyperdex.org/, 2015.

[28] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper.*

Syst. Rev., 44 (2): 35–40, April 2010. ISSN 0163-5980. doi:10.1145/1773912.1773922.

[29] 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.

[30] 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.

[31] Greg Linden. Make data useful. Talk, November 2006. http://glinden.blogspot.com/2006/12/slides-from-my-talk-at-stanford.html.

[32] 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.

[33] Cade Metz. How Instagram Solved Its Justin Bieber Problem, November 2015. URL http://www.wired.com/2015/11/how-instagram-solved-its-justin-bieber-problem/.

[34] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014*, pages 309–328, 2014. doi:10.1145/2660193.2660231.

[35] Ramon E. Moore. *Interval analysis*. Prentice-Hall, 1966.

[36] Dao Nguyen. What it's like to work on buzzfeed's tech team during record traffic. http://www.buzzfeed.com/daozers/what-its-like-to-work-on-buzzfeeds-tech-team-during-record-t, February 2015.

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

[38] outworkers ltd. Phantom by outworkers. http://outworkers.github.io/phantom/, March 2016.

[39] 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.

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

[41] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 164–174, 2011. doi:10.1145/1993498.1993518.

[42] Salvatore Sanfilippo. Redis. http://redis.io/, 2015a.

[43] Salvatore Sanfilippo. Design and implementation of a simple Twitter clone using PHP and the Redis key-value store. http://redis.io/topics/twitter-clone, 2015b.

[44] 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.

[45] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, PLDI. Association for Computing Machinery (ACM), 2015. doi:10.1145/2737924.2737981.

[46] 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.

[47] 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.

[48] 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.

[49] The Linux Foundation. netem. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem, November 2009.

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

[51] 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.

[52] 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.

[53] 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.