Disciplined Inconsistency

Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, Luis Ceze

University of Washington

Submission Type: Research

Abstract

Distributed applications and web services, such as online stores or social networks, have tight performance requirements. They are expected to scale linearly, never lose data, always be available, and respond quickly to users around the world. To meet these needs in the face of high round-trip latencies, network partitions, server failures, and load spikes, applications must give up on strong consistency and use datastores with weaker guarantees, such as eventual consistency. Making this switch is highly error-prone because relaxed consistency models are notoriously difficult to understand and test. Introducing weak consistency to handle worst-case scenarios also 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 performance targets and correctness requirements as constraints on abstract data types (ADTs), and handle uncertainty about values with new *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 programmer-specified performance and correctness bounds with latencies comparable to weak consistency and 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. Programmers need to preserve application correctness (e.g., never double-charging for a purchase, or showing up-to-date results and accurate counts), while minimizing response times to meet contractual service level agreements (SLAs) or to keep user engagement high (e.g., Microsoft, Amazon and Google all note that every millisecond of latency translates to a loss in traffic and revenue [21, 30]). Worse, programmers must maintain this balance in an unpredictable environment where a black and blue dress [36] or Justin Bieber [32] can change application performance in the blink of an eye.

Recognizing this trade-off, many existing storage systems support configureable consistency levels; some allow programmers to set the consistency of the entire store or each operation, while other provide adaptable consistency SLAs. Ideally, programmers would weaken consistency guarantees only when it is necessary to meet performance requirements (e.g., during a spike in traffic or datacenter failure), or when it does not impact the application's correctness guarantees (e.g., if returning a slightly stale or estimated result is acceptable). Unfortunately, if programmers are *undisciplined* in their use of weakly consistent data (i.e., updating the storage system based on inconsistent reads), they can easily corrupt application data, lowering the consistency of the storage system to that of the weakest read or write operation.

In this paper, we propose a more disciplined approach to consistency: the *inconsistent, performance-bound, approximate (IPA)* programming model for distributed storage systems. 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. More specifically, the IPA type system provides the following features:

- *Abstract Data Types* (ADTs), supported by the storage system, that programmers can annotate with performance targets, which allow the storage system to automatically adapt to meet when conditions change, and consistency requirements, which allow programmers to express where preciseness is not required (e.g., the application plans to display an approximate count).
- *IPA types*, returned from the storage system, that express the consistency and correctness of potentially inconsistent values, and subtyping to allow applications to make decisions based on the actual consistency of data.

- *Type checking* that enforces the disciplined use of IPA types by requiring programmers to explicitly endorse the propagation of inconsistent values to strongly consistent ADTs.

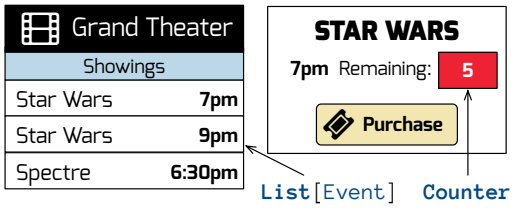
We explore the IPA model by implementing two important distributed runtime mechanisms: latency-bound operations, and a novel *error tolerance* reservation system. We describe how these mechanisms can be implemented in Scala for a distributed environment based on Cassandra, and explain how the IPA programming 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 correctness and performance to changing conditions with a counter microbenchmark and two applications: a simple Twitter clone based on Retwis [49] and a Ticket sales service modeled 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 how services like BuzzFeed [36] and Instagram [32] struggle with unpredictable Internet traffic. Before launching into the IPA programming model, we will explore an example application – a ticket sales service – to demonstrate the difficulties in trading off consistency to meet performance goals without breaking correctness.

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 [15]. 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 a minority of instances.

Let's look at how we would model this application using existing datastores. In order to make our service scalable, highly available, and fault-tolerant, we can use an off-the-shelf datastore like Cassandra. Cassandra is a Dynamo-style [17] datastore, meaning it keeps multiple complete replicas of its data, often replicating within a single datacenter and across geographically distributed datacenters. By default, clients are allowed to read or write to any available replica. Mechanisms within the datastore, such as anti-entropy, read repair, and gossip, share updates among replicas. Because propagation takes time, clients can observe many odd, inconsistent, states; updates can even appear to come and go. *Eventual consistency*, the weakest consistency typically available, only guarantees that all replicas will eventually reflect the same state some time after the last write [57]. However, these datastores can provide stronger consistency by synchronizing with more replicas. Strongly consistent reads are achieved by ensuring that a *quorum* of replicas agree on a value before returning it to the client.




```
// latency target: < 20ms
def load_page(event: UUID) = {
  val remaining = read_weak("{event}:tickets")
  // how accurate is this count?
  display("Remaining: ", remaining)
  // only allow purchase if tickets > 0
  if (remaining > 0)
    enable_purchase_button()
}
```

Figure 1. Ticket sales service. To meet latency target, `load_page` switches to a weak read, introducing a potential logical error when using this possibly-inconsistent value to determine if tickets can be purchased.

Figure 1 shows the application's structure. Each event (in this case movie, location, and time) has some number of available tickets. Visitors to the site may browse the events or movie showings, and view individual events to see how many tickets are remaining and purchase some.

The most important invariant is that tickets are not oversold: each available ticket must only go to one user, and the count of remaining tickets should never be below zero. However, there are also soft constraints:

- *Latency target:* Users browsing many events will become frustrated if pages take too long to load.
- *Accuracy bounds:* Users wish 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 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.

Using existing systems, we have few options for meeting all these requirements. We can ensure that the ticket count does not go negative by forcing operations to be linearizable [25, 51]. Linearizable operations (e.g., as achieved with Cassandra's "lightweight transactions") require coordination on every operation, and thus make reads of the count prohibitively slow. We could augment

this with a second, weakly consistent counter that approximates the remaining ticket count; this must be synchronized with the true value in the other counter, and can drift arbitrarily between synchronizations. As a result, their combination introduces significant implementation complexity to keep the two counters synchronized and manage the drift of the weakly consistent counter. Alternatively, a convergent replicated data type (CRDT) [50] called a `BoundedCounter` [7] 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.

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. If we support weak consistency, we must now handle new cases resulting from inconsistencies, such as late actions on deleted events. During low-traffic times, allowing 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 data structures (such as `Set[T]`) on distributed storage.
- 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 values from flowing into consistent values. Second, the IPA type system provides *performance*, because consistency annotations at the ADT level allow the runtime to select a consistency for each individual operation that maximizes performance in a constantly changing environment. Together, these systems allow applications to adapt to changing conditions with the assurance that the programmer has expressed how it should handle varying consistency.

3.2. Abstract Data Types

The base of the IPA type system is a set of abstract data types (ADTs) for distributed data structures. ADTs present a clear abstract model through a set of operations that query and update state, allowing users and systems alike to reason about their logical, algebraic properties rather than the low-level operations used to implement them. 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, the interface to these datatypes differs: from explicit sets of operations for each type in Redis, Riak, and Hyperdex [8, 20, 26, 48] to the pseudo-relational model of Cassandra [27]. IPA's extensible library of ADTs allows it to decouple the semantics of the type system from any particular datastore, though our reference implementation is on top of Cassandra, similar to [51].

Besides abstracting over storage systems, ADTs are an ideal place from which to reason about consistency and system-level optimizations. The consistency of a read depends on the write that produced the value. Expressing consistency properties on ADTs ensures the necessary guarantees for all operations are enforced, which we will expand on in the next section.

Custom ADTs can express application-level correctness constraints. IPA's `Counter` datatype allows reading the current value as well as increment and decrement operations. In our ticket sales example, we must ensure that the ticket count does not go below zero. Rather than forcing all operations on the datatype to be linearizable, this application-level invariant can be expressed with a more specialized ADT, such as a `BoundedCounter`, giving the implementation more latitude to choose how to enforce it. IPA's library is *extensible*, allowing applications to leverage common features to implement their own custom ADTs; see §4.5.

3.3. Policy Annotations

The IPA type system provides a set of annotations that can be placed on ADT instances to specify consistency policies. Annotations express 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.

Previous systems [3, 8, 29, 52, 54] 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 a fixed consistency policy for an ADT, such as `Consistency(Strong)` which states that an operation must have strongly consistent behavior. Static annotations provide the same direct control as existing approaches but simplify reasoning about correctness by applying them globally on the ADT. *Dynamic* annotations specify a consistency policy in terms of application-level requirements, allowing the system to decide at runtime how to meet the requirement for each executed operation. IPA offers two dynamic consistency policies:

- A latency policy `LatencyBound(x)` specifies a target latency for operations on the ADT (e.g., 20 ms). The runtime can choose the consistency level for each issued operation, optimizing for the strongest level that is likely to satisfy the latency bound.
- An accuracy policy `ErrorTolerance(x%)` specifies the desired accuracy for read operations 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.

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 will enforce safety by requiring the developer to consider the effects of weak operations.

Static and dynamic annotations can be declared globally for all operations or for specific methods of the ADT. For example, `Set[T] with Consistency(Strong)` or `List[T] with LatencyBound(50 ms)`, both declare that all operations on the object are performed with the specified bound. On the other hand, an instance of a `Set` may have relaxed consistency for its `size` but strong consistency for its `contains` predicate. The runtime is 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(5%)`. 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 used for events has a latency policy that also expresses a domain requirement—that pages on the website load in reasonable time.

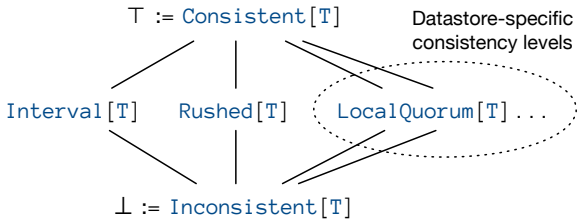


Figure 2. IPA Type Lattice parameterized by a type `T`.

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 non-interference property: results from weakly consistent operations cannot flow into computations with stronger consistency without explicit endorsement.

The IPA types encapsulate information about the consistency achieved when reading a value. Formally, the IPA types form a lattice parameterized by a primitive type `T`, shown in Figure 2. Strong read operations return values of type `Consistent[T]` (the top element), and so (by implicit cast) behave 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. The bottom element `Inconsistent[T]` specifies an object with the weakest possible (or unknown) consistency. The other IPA types follow a subtyping relation $\prec$, defined by:

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

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:

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

The core type system statically enforces safety by preventing weaker values from flowing into stronger computations. Forcing developers to explicitly endorse

`Inconsistent` values prevents them from accidentally using inconsistent data there they did not determine it was acceptable, essentially inverting the behavior of current systems where inconsistent data is always treated as if it was safe to use anywhere. However, endorsing values blindly in this way is not the intended use case; the key productivity benefit of the IPA type system comes from the other IPA types which correspond to the consistency policies in §3.3 which allow developers to handle dynamic variations in consistency, which we describe next.

3.4.1. Rushed types

The weak IPA type `Rushed[T]` is the result of read operations performed on an ADT with consistency policy `LatencyBound(x)`. `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 latency bound of 50 ms with a local quorum read. In this case, `Rushed[T]` 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.contains() 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).

3.4.2. 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 [10].

The key invariant of the `Interval[T]` type is that uses of the interval are *indistinguishable* from a linearizable order. Consider a `Set` with 100 elements. With linearizability, if we `add` a new element and then read the `size` (or if this ordering is otherwise implied), we *must* get back 101 (provided no other updates are occurring). However, if `size` is annotated with `ErrorTolerance(5%)`, then `size` could return intervals such as $[95, 105]$ or $[100, 107]$, so the client cannot tell if the `add` was incorporated. This frees the system to optimize to improve performance, such as by delaying synchronization. While any partially-ordered domain could be represented as an interval (e.g., a Set with partial knowledge of its members), in this work we consider only 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 the interval is well above zero, then users can be assured that there are sufficient tickets remaining. In fact, because the interval is indistinguishable from a linearizable order, in the absence of other user actions, a subsequent purchase must succeed. On the other hand, if the interval overlaps with zero, then there is a chance that tickets could already be sold out, so users should be warned. Note that ensuring that tickets are not over-sold is a separate concern requiring a different form of enforcement, which we describe in §4.5. The relaxed consistency of the interval type allows 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.

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. Type system

The IPA type system is implemented as a Scala library, simply leveraging Scala's sophisticated type system. Due to its type safety features, abstraction capability, and compatibility with Java, Scala is becoming increasingly popular for web service development, including widely-used frameworks such as Akka and Spark, and at established companies such as Twitter [cite some of that]. The IPA type lattice is implemented as a subclass hierarchy of parametric classes, leveraging Scala's support for higher-kinded types to allow them to be destructured in match statements, and implicit conversions for allowing `Consistent[T]` to be treated as type `T`. We use Scala traits ("mixins") to implement ADT annotations; when a `LatencyBound` trait is mixed into one of our ADT types, it wraps each of the ADT's methods, redefining it to have the new semantics and return the correct IPA type. Figure 4 shows an example trait.

4.2. Backing datastore

The core of the implementation is a a scalable, distributed storage system with fine-grained consistency control, as many popular Dynamo-style commercial datastores offer (Cassandra [3], Riak [8], etc.). Our implementation of the IPA model builds 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.

We introduced Dynamo-style replication and eventual consistency in §2. To be sure of seeing a particular write, strong reads must coordinate with a majority (*quorum*) of replicas and compare their responses. For a write and read pair to be *strongly consistent* (in the CAP sense [12]), 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 two 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`) [16]. Cassandra also supports limited linearizable conditional updates and degrees of weaker consistency, particularly to handle different locality domains (e.g. `LOCAL_QUORUM`).

4.3. Latency bounds

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.

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.

Unfortunately, Cassandra's client interface does not allow us to implement latency bounds exactly as described above: operations must specify a consistency level in advance. Instead, we issue read requests at different levels in parallel. We compose the parallel operations and respond either when the strong operation returns or with the strongest available result at the specified time limit. If no responses are available at the time limit, we wait for the first to return.

4.3.1. 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 these cases, we should back off and only attempt weaker consistency. To do this, the system monitors current traffic and predicts 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, grouped by operation and consistency level. 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 only the weakest if that is the prediction.

4.3.2. 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. Our implementation has ADTs choose their desired write level statically so that we know the strength of a read without checking. A more complex system might allow dynamic changes to an ADT's annotations (in the backing store) with clients checking for changes periodically. 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.

4.4. Error bounds

We implement error bounds by building on the concepts of *escrow* and *reservations* [22, 38, 42, 44]. These tech-

niques 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 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*: tokens 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.

4.4.1. Reservation Server

To implement reservations, we must 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 replica-local state, we must implement a custom middleware layer to handle reservation requests, similar to other systems which have built stronger guarantees on top of existing platforms [5, 7, 51].

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. 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.

4.4.2. Enforcing error bounds

Reservations have been used previously to enforce hard global invariants in the form of upper or lower bounds on values [7], integrity constraints [6], or logical assertions [31]. 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 3. 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.

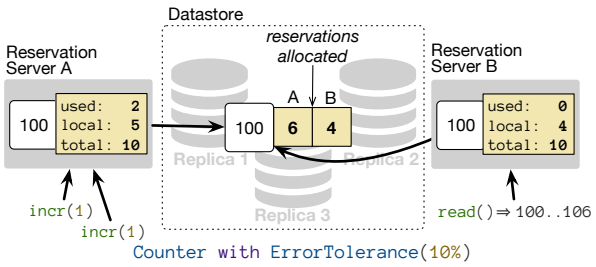


Figure 3. Reservations enforcing error bounds.

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 3, there are 10 reservations total, but Server B knows that it has not used its local reservations, so it knows that there cannot be more than 6 and can return the interval $[100, 106]$.

4.4.3. 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 can reflect this by returning more precise results. 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 could be sound. However, in the general case of eventual consistency, this is not 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.

4.5. Provided by IPA

The IPA system provides a number of primitives for building ADTs as well as some reference implementations of simple datatypes; some are shown in Figure 4. To support latency bounds, there is a generic `LatencyBound` trait that provides facilities for executing a specified read operation at multiple consistency levels within a time limit. For implementing error bounds, IPA provides a generic reservation pool which ADTs can use.

The IPA system implements several ADTs:

- `Counter` based on Cassandra's counter type, 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 [7] 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` generates unique identifiers, with a hard

```
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 4. Example facilities provided by IPA.

limit on the number of IDs that can be taken from it; builds on top of `BoundedCounter` and supports the same bounds.

- `List`: thin abstraction around a Cassandra table with a time-based clustering order, supports latency bounds. Used to implement Twitter timelines and Ticket listings.

Figure 4 shows 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 [39]. The Reservation Server is similarly built around futures using Twitter's Finagle framework [56]. 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 across varying network conditions using both a real-world testbed (Google Compute Engine [24]) and simulated changing network conditions. We begin using microbenchmarks on our simulated network to understand the performance of individual data types and explore each bound in isolation. Then, we move on to application benchmarks, modified to use IPA data types, on both our real-world and simulated test beds.

5.1. Simulating adverse conditions

To evaluate these bounding techniques, it is crucial to subject them to a variety of conditions. There are primarily three 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.

Network Condition	**Latencies (ms)**		
Simulated	*Replica 1*	*Replica 2*	*Replica 3*
Uniform / High load	5	5	5
Slow replica	10	10	100
Geo-distributed (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 replicas, with standard deviation if high.

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 [55] (`tc netem`) to introduce packet delay and loss at the operation system level. We use Docker containers [19] 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 a condition mimicking globally geo-replicated setups with latency distributions based on measurements of latencies between virtual machines in the U.S., Europe, and Asia on Amazon EC2 [2], to complement our real experiments on Google Compute Engine.

We elide the results for *Local* (same rack in our own testbed) except in Figure 10 because the differences in latencies are negligible for all the bounds. In such situations, strong consistency ought to be the default.

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

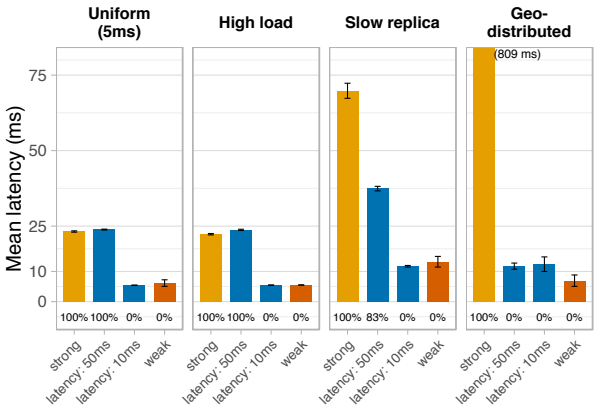


Figure 5. *Counter: latency bounds, mean latency*. Beneath each bar is the % of strong reads. Strong consistency is never possible for the 10ms bound, but 50ms bound gets mostly strong during good conditions and at high latency resorts to weak.

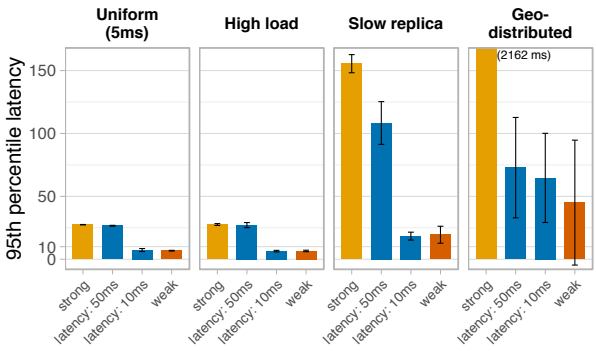


Figure 6. *Counter: 95th percentile latency*. Latency bounds help keep tail latency down by backing off to weak consistency when necessary.

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 5 shows the average latency of a counter with strong, weak, 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 and falls back to weak consistency. With a 50ms bound, the counter is able to get strongly consistent results if network latency is low. However, with one slow replica (out of 3), there is a chance that the `QUORUM` read needed for strong consistency will hit the slow replica, so IPA attempts both; in this case, 82% got strong consistency, and 18% timed out and went with weak. Finally, with our simulated geo-distributed environment, there are no 2 replicas within 50ms of our client, so strong consistency is never possible within our bounds; as a result, IPA adapts and only attempt weak reads in both cases.

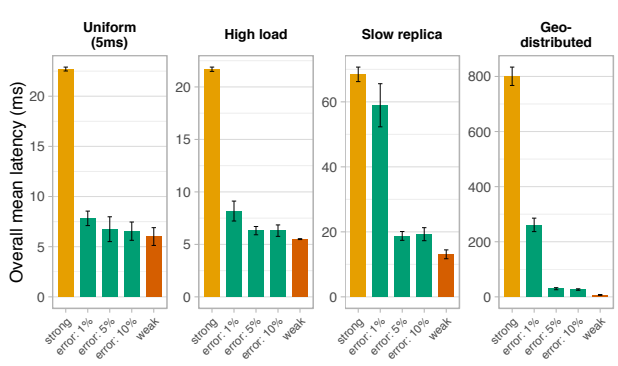
Figure 6 shows the 95th percentile latencies of the same workload. We see that the tail latency of the 10ms bound is comparable to weak, though the 50ms bound guesses incorrectly occasionally for the case of the slow replica. We see a latency gap between the latency-bound and weak in the geo-distributed 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

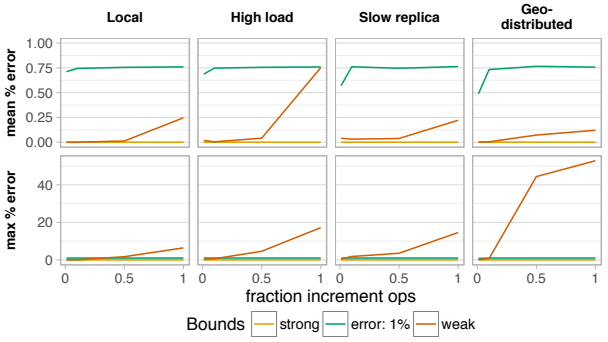
The goal of this experiment is to explore the empirical cost of enforcing error bounds using the reservation system described in §4.4, and to determine how tight error bounds can be while providing similar performance to weak consistency.

The intuition behind reservations is to move synchronization off the critical path: by distributing write permissions among replicas, clients can get strong guarantees while communicating only 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 considered only read latency because we didn't change the writes. However, reservations actually slow down writes, so we must consider that effect.

Figure 7a shows latencies for error bounds of 1%, 5%, and 10%. The *read* latency for error bounds is always equivalent to *weak*, so we plot the average of reads *and* increment operations to understand the overall performance. We can see that tighter bounds increase latency because it



(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 7. *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. For *weak*, the average error is less than 1%, but the *maximum* error can be extremely high (up to 60%), whereas error bounds never exceed their limit.

forces more synchronization operations, which must use consistency of `ALL`. 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.

While we have verified that error-bounded reads remain within our defined bounds, we may also wish to how much error occurs in practice without the 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 7b. 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, we find that the *mean* error of weak is significantly less than 1%. This shows that inconsistency is quite 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 outliers with significant error when writing is heavy: up to 60% error in the geo-replicated case. 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 application benchmarks in our simulated network testbed and on the geo-distributed, Google Compute Engine [24]. We ran virtual machines on Google Compute Engine in 4 different datacenters: the client in `us-east`, and the datastore replicas in `us-central`, `europe-west`, and `asia-east`.

5.3.1. Ticket service

Our Ticket sales web service, introduced in §2, is modeled after FusionTicket [1], which has been used as a benchmark in recent distributed systems research [58, 59]. 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 modeled using a `List` ADT. Tickets are modeled using the `UUIDPool` type, which 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, 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

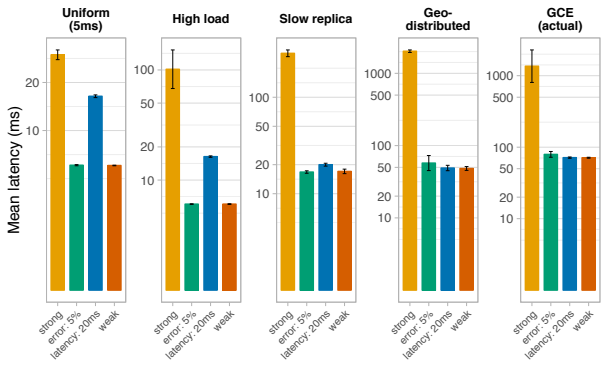


Figure 8. *Ticket service: mean latency,* ***log scale.*** Strong consistency is far too expensive (>10× slower) except when load and latencies are low, but 5% error tolerance allows latency to be comparable to weak consistency. The 20ms latency-bound variant is either slower or defaults to weak, providing little benefit. Note: the ticket `Pool` is safe even when weakly consistent.

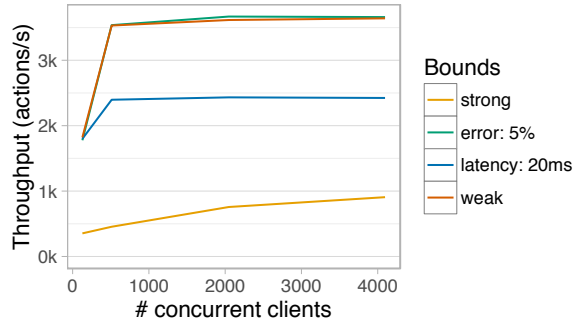


Figure 9. *Ticket service:* throughput on Google Compute Engine globally-distributed testbed. Note that this counts *actions* such as `tweet`, which can consist of multiple storage operations. Because error tolerance does mostly weak reads and writes, its performance tracks *weak*. Latency bounds reduce throughput due to issuing the same operation in parallel.

(this is true for all cases because the `BoundedCounter` has its own form of 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 [14], 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 8 shows the average latency of a workload consisting of 70% `viewEvent`, 19% `browse`, 10% `purchase`, and 1% `addEvent`. We plot with a log scale because strong consistency is consistently over 5× higher latency. The `purchase` event, though only 10% of the workload, drives most of the latency increase because of the additional work required by the `BoundedCounter` to prevent over-selling tickets. We explore two different implementations: one

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

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

Figure 10. Twitter application's (simplified) data model, with latency bounds for followers and timelines, and error tolerance for number of retweets (`size` of the retweets set).

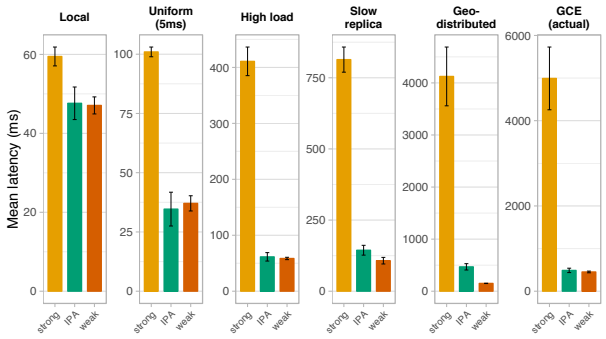


Figure 11. *Twitter clone: mean latency (all actions).* The implementation with IPA bounds is on-par with weak consistency in all but one of the conditions, while strong consistency is 2-10× slower.

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).

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 `Pools` 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).

5.3.2. Twitter clone

Our second application benchmark is a Twitter-like service based on the Redis data modeling example, Retwis [49]. The data model is simple: each user has a `Set` of followers, a `Set` of 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 Oscars [4]. 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 11 that for all but the local (same rack) case, strong consistency is over 3× 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* [28] and Herlihy's *linearizability* [25] on the strong side, to *eventual consistency* [57] 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 [53]. Many datastores support configuring consistency at a fine granularity: Cassandra [3] per operation, Riak [8] on an object or namespace granularity, as well as others [29, 52].

The Conit consistency model [60] 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 [51] 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 [6], 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

IPA's latency-bound policies were inspired by the *consistency-based SLAs* of Pileus [54]. 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, introducing greater annotation burden and preventing optimizations based on guarantees provided by other operations, which only ADT-level annotations provide.

A long history of systems have been built around the principle that applications may be willing to tolerate slightly stale data in exchange for improved performance, including databases [9, 40, 43, 45] and distributed caches [37, 41]. These systems generally require developers to explicitly specify staleness bounds on each transaction in terms of absolute time (although Bernstein et al.'s model can generate these from error bounds when a value's maximum rate of change is known). As a result, they are largely orthogonal to our work: these techniques can be used to build a datastore for IPA types, whereas our work introduces a type system to help developers manage their consistency requirements.

6.4. Types for distributed systems

Convergent (or *conflict-free*) *replicated data types* (CRDTs) [50] 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. 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. Particularly relevant to IPA, the Bounded Counter CRDT [7] enforces hard limits on the global value of a counter in a way similar to reservations but less general; this design informed our own implementation of reservations for error bounds.

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 [47] and Rely [13] track the flow of approximate values to prevent them from interfering with precise computation. Chisel [33] and DECAF [11] 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 [10] and to a long line of work on interval analysis [34]. 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.

Approximate type systems, including IPA, are inspired by a long line of work on information flow tracking (e.g., [18, 35, 46]). These systems use a combination of static type checking and dynamic analysis to enforce isolation between sensitive data and untrusted channels, guaranteeing a *non-interference* policy [23]: public information will never be influenced by confidential data.

7. Conclusion

The IPA programming model provides programmers with disciplined ways to trade consistency for performance in distributed applications. By specifying application-specific performance and accuracy targets in the form of latency and error tolerance bounds, they tell the system how to adapt when conditions change and provide it with release valves for optimization opportunities. Meanwhile, IPA types ensure consistency safety, ensuring that all potential weak outcomes are handled, and allowing applications to make choices based on the accuracy of the values the system returns. The policies, types and enforcement systems implemented in this work are only a sampling of the full range of possibilities within the framework of *inconsistent, performance-bound, and approximate* types.

References

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

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

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

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

[5] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*,

SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. doi:10.1145/2463676.2465279.

[6] 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.

[7] 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.

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

[9] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed currency serializability for middle-tier caching and replication. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, Chicago, IL, USA, June 2006. ACM.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20 (7): 504–513, July 1977.

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

[20] 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.

[21] 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.

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

[23] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy (Oakland '82)*, Oakland, CA, USA, April 1982. IEEE.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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/.

[33] 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.

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

[35] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programing Languages (POPL '99)*, San Antonio, TX, USA, January 1999. ACM.

[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] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA, USA, May 1999. ACM.

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

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

[40] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of the International Middleware Conference*, Toronto, Ontario, Canada, October 2004.

[41] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, October 2010. USENIX.

[42] 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.

[43] Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, CO, USA, May 1991. ACM.

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

[45] Uwe Röhm, Klemens Böhm, Hans-Jörg Schek, and Heiko Schuldt. FAS — a freshness-sensitive coordination middleware for a cluster of OLAP components. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, Hong Kong, China, August 2002.

[46] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21 (1): 1–15, January 2003.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] Twitter, Inc. Finagle. https://twitter.github.io/finagle/, March 2016.

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

[58] 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.

[59] 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.

[60] 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.