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 [27, 37]). Worse, programmers must maintain this balance in an unpredictable environment where a black and blue dress [43] or Justin Bieber [39] 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 [4, 11, 35, 59]. Ideally, programs would only weaken consistency guarantees when necessary to meet availability requirements (e.g., during a spike in traffic or datacenter failure), or when it does not impact the application's correctness guarantees (e.g., returning a slightly stale or estimated result is acceptable); some have supported adaptable consistency in this way [60, 62]. Unfortunately, if programmers are *undisciplined* in their use of weakly consistent data – writing to other storage based on inconsistent reads, or using inconsistent data where they intended it to be strong, such as an irreversible purchase – they could 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), backed 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 precision is not required (e.g., the application plans to display an approximate count).
- *IPA types*, returned from storage system reads, that express the consistency and correctness of potentially inconsistent values, allowing 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 handle all potential consistency cases, or explicitly endorse the propagation of inconsistent values where appropriate.

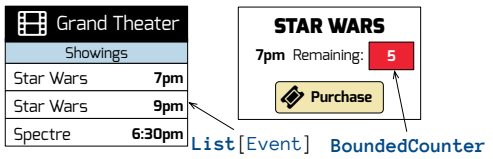
We explore the IPA model by implementing two important distributed runtime mechanisms: latency-bound operations, and a novel *error tolerance* reservation system. We describe how these mechanisms can be implemented in Scala for a distributed environment based on Cassandra, and explain how the IPA programming 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 [56] 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. The Case for Consistency Safety

There are many popular news stories about how services like BuzzFeed [43] and Instagram [39] 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 [21]. 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 [23] 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




```
// adjust price based on number of tickets left
def computePrice(ticketsRemaining: Int): Float

def purchaseTicket(event: UUID) = {
  val ticket = reserveTicket(event)
  // weak read of ticket_count for performance
  val remaining = read_weak(event+"ticket_count")
  // compute price based on inconsistent read
  val price = computePrice(remaining)
  display("Enter purchase info. Price: ", price)
}
```

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.

replicas will eventually reflect the same state some time after the last write [65]. 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.

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 [31, 58]. 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) [57] called a `BoundedCounter` [10] 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 tradeoff. 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 the policies. §4 will explain how the type system's guarantees are enforced.

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 have built-in support for more complex data structures such as sets, lists, and maps. However, the interface to these datatypes differs: from explicit sets of operations for each type in Redis, Riak, and Hyperdex [11, 26, 32, 55] to the pseudo-relational model of Cassandra [33]. 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 [58].

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` ADT 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 for enforcing it. IPA's library is *extensible*, allowing custom ADTs to build on common features; see §5.

3.3. Policy Annotations

Previous systems [4, 11, 35, 59, 62] 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. The IPA type system provides a set of annotations that can be placed on *ADT instances* to specify consistency policies for the lifetime of the object. IPA type annotations come in two flavors: static and dynamic.

Static annotations declare a fixed consistency policy, such as `Consistency(Strong)` which states that operations must have strongly consistent behavior. Static annotations provide the same direct control as previous 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 more performance from an application by relaxing the consistency of individual 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 apply to an entire ADT instance or on individual methods. For example, one could declare `List[Int] with LatencyBound(50 ms)`, in which case all read operations on the list are subject to the bound. Alternatively, one may wish to declare a`Set` with relaxed consistency for its `size` but strong consistency for its `contains` predicate. The runtime is responsible for managing the interaction between these 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` for each event's tickets has a relaxed accuracy policy: `ErrorTolerance(5%)`, allowing 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 violating correctness. 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.

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 $\prec$ as illustrated in Figure 2.

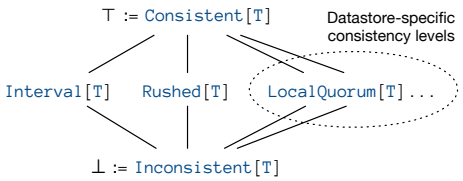


Figure 2. IPA Type Lattice parameterized by a type `T`.

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 [15].

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 §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. Enforcing dynamic policies

The dynamic policies introduced in the previous section allow programmers to describe application-level correctness properties but they require new runtime mechanisms to enforce. But first, we briefly review consistency in Dynamo-style replicated systems.

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 [17]), the replicas acknowledging the write (W) plus the replicas contacted for the read (R) must be greater than the total number of replicas ($W + R > N$). This can be achieved, for example, by writing to a quorum ($(N + 1)/2$) and reading from a quorum (`QUORUM` in Cassandra), or writing to N (`ALL`) and reading from 1 (`ONE`) [22]. Cassandra also supports limited linearizable conditional updates and varying degrees of weaker consistency, particularly to handle different locality domains (e.g. `LOCAL_QUORUM`).

4.1. Latency bounds

The time it takes to achieve a particular level of consistency depends on 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 while at peak traffic times 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 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.

4.1.1. Monitors

The main problem with this approach is that it wastes a lot of work, even if we didn't need to duplicate some mes-

sages due to the client 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 can share one). The monitor records the observed latencies of reads, grouped by operation and consistency level. 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, then issues one request at that consistency level and a backup at the weakest level, or only weak if none can meet the bound. In §6.2.1 we show empirically that even simple monitors allow clients to adapt to changing conditions.

4.2. Error bounds

We implement error bounds by building on the concepts of *escrow* and *reservations* [28, 45, 49, 51]. 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, whereas 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.

4.2.1. Reservation Server

Reservations require mediating requests to the datastore to prevent updates from exceeding the available tokens. Furthermore, each server must locally know how many tokens it has without synchronizing. We are not aware of a commercial datastore that supports custom mediation of requests and replica-local state, so we need a custom middleware layer to handle reservation requests, similar to other systems which have built stronger guarantees on top of existing datastores [8, 10, 58].

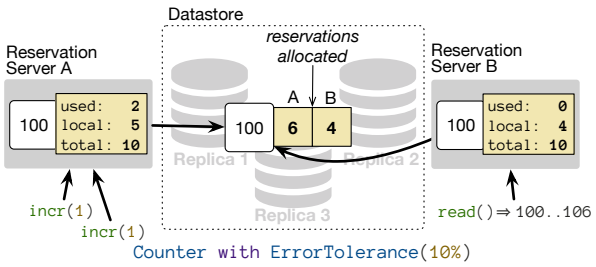


Figure 3. Reservations enforcing error bounds.

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 the backing store; 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; our implementation simply colocates a reservation server with each datastore server and uses the datastore's node discovery mechanisms to route requests to reservation servers on the same host.

4.2.2. Enforcing error bounds

Reservations have been used previously to enforce hard global invariants in the form of upper or lower bounds on values [10], integrity constraints [9], or logical assertions [38]. 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. If only 10 outstanding increments are allowed, reads are guaranteed to 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 temporarily agree on the value (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 $[100, 106]$.

4.2.3. Narrowing bounds

Error-tolerance policies give an *upper bound* on the amount of error; ideally, the interval returned will be more precise than the maximum error when conditions are favorable. The error bound determines the *maximum* number of reservations that can be allocated per instance. To allow a variable number of tokens, each ADT instance keeps a count of tokens allocated by each server, and when servers receive write requests, they increment their count to allocate tokens to use. Allocating must be done with strong consistency to ensure all servers agree, which can be expensive, so we use long leases (on the order of seconds) to allow servers to cache their allocations. When a 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 different error bounds on different read operations. It is up to the designer of the ADT to ensure that all error bounds are enforced with appropriate reservations. Consider a `Set` with an error tolerance on its `size` operation. This requires separate pools for `add` and `remove` to prevent the overall size from deviating by more than the bound in either direction, so the interval is $[v - \texttt{remove.delta}, v + \texttt{add.delta}]$ where v is the size of the set and `delta` computes the number of outstanding operations from the pool. In some situations, operations may produce and consume tokens in the same pool – e.g., `increment` producing tokens for `decrement` – but this is only allowable if updates propagate in a consistent order among replicas, which may not be the case in some eventually consistent systems.

5. Implementation

IPA is implemented mostly as a client-side library to an off-the-shelf distributed storage system, though reservations are handled by a custom middleware layer which mediates accesses to any data with error tolerance policies. Our implementation is built on top of Cassandra, but IPA could work with any replicated storage system that supports fine-grained consistency control, which many commercial and research datastores do, including Riak [11].

IPA's client-side programming interface is written in Scala, using the asynchronous futures-based Phantom [46] library for type-safe access to Cassandra data. Reservation server middleware is also built in Scala using Twitter's Finagle framework [64]. Communication is done between clients and Cassandra via prepared statements, and between clients and reservations servers via Thrift remote-procedure-calls [6]. Due to its type safety features, abstraction capability, and compatibility with Java, Scala has become popular for web service development, including widely-used frameworks such as Akka [36] and Spark [5], and at established companies such as Twitter and LinkedIn [2, 18, 30].

```
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, one per ADT instance.
   `max` recomputed as needed (e.g. for % error) */
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 ErrorTolerance {
  def error: Float // % tolerance (defined by instance)
  def maxDelta() = (cassandra.read(key) * error).toInt
  val pool = ReservationPool(maxDelta)

  def read(): Interval[Int] = {
    val v = cassandra.read(key)
    Interval(v - pool.delta, v + pool.delta)
  }
  def incr(n: Int): Unit =
    waitFor(pool.take(n)) { cassandra.incr(key, n) }
}
```

Figure 4. Some of the reusable components provided by IPA and an example implemention of a Counter.

The IPA type system, responsible for consistency safety, is also simply part of our client library, simply leveraging Scala's sophisticated type system. The IPA type lattice is implemented as a subclass hierarchy of parametric classes, using Scala's support for higher-kinded types to allow them to be destructured in match statements, and implicit conversions to allow `Consistent[T]` to be treated as type `T`. We use traits to implement ADT annotations; e.g. when the `LatencyBound` trait is mixed into an ADT, it wraps each of the methods, redefining them to have the new semantics and return the correct IPA type. Figure 4 shows an example.

IPA comes with a library of reference ADT implementations used in our experiments, but it is intended to be extended with custom ADTs to fit more specific use cases. Our implementation provides a number of primitives for building ADTs, some of which 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 library of reference ADTs includes:

- `Counter` based on Cassandra's counter, supporting increment and decrement, with latency and error bounds
- `BoundedCounter` CRDT from [10] 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.
- `Set` with `add`, `remove`, `contains` and `size`, supporting latency bounds, and error bounds on `size`.
- `UUIDPool` generates unique identifiers, with a hard limit on the number of IDs that can be taken from it; built 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.

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.

6. Evaluation

The goal of the IPA programming model and runtime system is to build applications that adapt to changing conditions, performing nearly as well as weak consistency but with stronger consistency and safety guarantees. To that end, we evaluate our prototype implementation under a variety of network conditions using both a real-world testbed (Google Compute Engine [29]) and simulated network conditions. We start with simple microbenchmarks to understand the performance of each of the runtime mechanisms independently. We then study two applications in more depth, exploring qualitatively how the programming model helps avoid potential programming mistakes in each and then evaluating their performance against strong and weakly consistent implementations.

6.1. Simulating adverse conditions

Evaluating replicated datastores under adverse conditions is challenging: tests conducted in a well-controlled environment where network latencies are low and variability is negligible will yield little of interest, whereas tests in production environments involve so many free variables that deciphering the results and reproducing them is difficult. An alternative approach is to *simulate* a variety of environments chosen to stress the system or mimic real challenging situations. We perform our experiments with a number of simulated conditions, and then validate our findings against experiments run on globally distributed machines in Google Compute Engine.

On our own test cluster with nodes linked by standard ethernet, we use Linux's Network Emulation facility [63] (`tc netem`) to introduce packet delay and loss at the operation system level. We use Docker containers [25] to enable fine-grained control of the network conditions between processes on the same physical node.

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. *Slow replica* simulates when one replica is significantly slower than others due to imbalanced load or hardware problems. Finally, we have a condition mimicking a geo-replicated setup with latency distributions based on measurements of latencies between virtual machines in the U.S., Europe, and Asia on Amazon EC2 [3] and Google Compute Engine. We elide the results for *Local* (same rack in our own testbed) except in Figure 11 because the differences between policies are negligible. In such situations, strong consistency ought to be the default.

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.

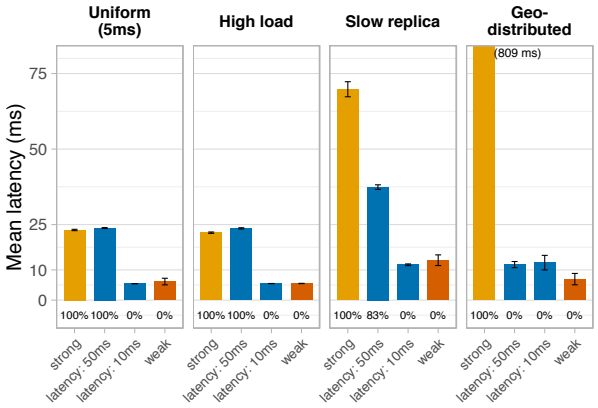


Figure 5. *Counter: latency bounds, mean latency*. Beneath each bar is the % of strong reads. Strong consistency is never possible for the 10ms bound, but 50ms bound achieves mostly strong, only resorting to weak when latency is high.

6.2. 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 (allowing up to 4000 concurrent operations).

6.2.1. 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 5 shows the average latency of a counter with strong, weak, and two 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's monitor predicts it will not get strong consistency and falls back to weak consistency. For

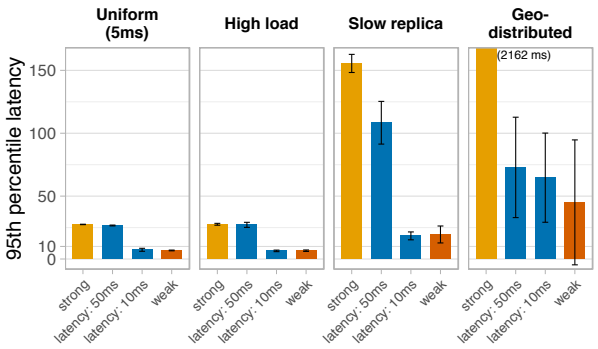


Figure 6. *Counter: 95th percentile latency.* Latency bounds keep tail latency down, backing off to weak when necessary.

a 50ms bound, the counter is able to get strong consistency 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 to only attempt weak 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 gap between 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.

6.2.2. Error bounds

This experiment determines the cost of enforcing error bounds using the reservation system described in §4.2, and to determine how tight error bounds can be while providing performance comparable to weak consistency. Reservations move synchronization off the critical path: by distributing write permissions among replicas, reads can get strong guarantees from a single replica. When evaluating the latency bounds, we considered only read latency because we didn't change the writes. Because reservations actually slow down writes, now we must consider both.

Figure 7a shows latencies for error bounds of 1%, 5%, and 10%, plotting the average of reads *and* increment operations because reads along would always be equivalent to weak. We see that tighter bounds increase latency because it forces more synchronization, which must use consistency of `ALL`. In most conditions, 5-10% error bounds have latency comparable to weak, except geo-distributed, where it seems that our implementation of reservations is not a good solution.

While we have verified that error-bounded reads remain within our defined bounds, we also wish to know what error occurs in practice without the bounds. We modified our benchmark to be able to observe the error from weak consistency by incrementing counters a predetermined amount and observing the value; results 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 write-heavy (all increments, except to check the value).

First, we find that the *mean* error is less than 1% – inconsistency is quite rare; it may be 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 for read-heavy workloads, the interval width (green line) is less than 1% because it dynamically adjusted as reservations were not needed.

6.3. Applications

Next, we explore how the IPA system performs on two application benchmarks in our simulated network testbed and on Google Compute Engine. We ran virtual machines on Google Compute Engine across four datacenters: the client in *us-east*, and the datastore replicas in *us-central*, *europe-west*, and *asia-east*.

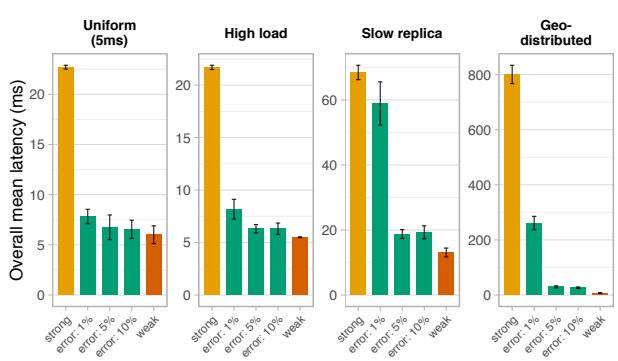
6.3.1. Ticket service

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 [66, 67]. We support the following actions:

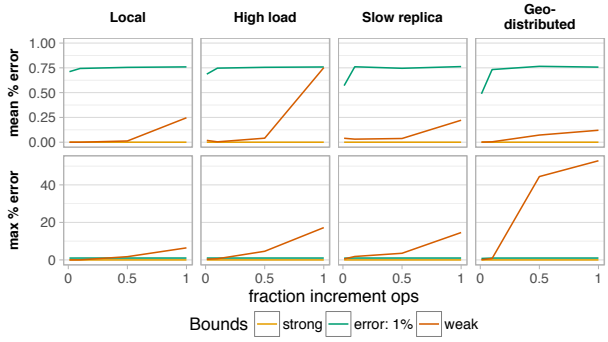
- `browse`: List events by venue
- `viewEvent`: View the full description of an event including number of remaining tickets
- `purchase`: Purchase a ticket (or multiple)
- `addEvent`: Add an event at a venue.

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 and ensures 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 (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



(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 actual error of *weak* compared with the actual interval for a 1% error bound with varying fraction of writes; average error is less than 1% but *maximum* error can be extremely high: up to 60%.

```
// creates a table of pools, so each event gets its own
// 5% error tolerance on `remaining` method, weak otherwise
val tickets = UUIDPool() with Consistency(Weak)
                with Remaining(ErrorTolerance(0.05))

def purchaseTicket(event: UUID) = {
  // UUIDPool is safe even with weak consistency (CRDT)
  endorse(tickets(event).take()) match {
    case Some(ticket) =>
      // imprecise count returned due to error tolerance
      val count: Interval[Int] = tickets(event).remaining()
      // use maximum count possible to be fair
      val price = computePrice(count.max)
      display("Ticket reserved. Price: $" + price)
      prompt_for_payment_info(price)
    case None =>
      display("Sorry, all sold out.")
  }
}
```

Figure 8. *Ticket service* code demonstrating use of IPA types.

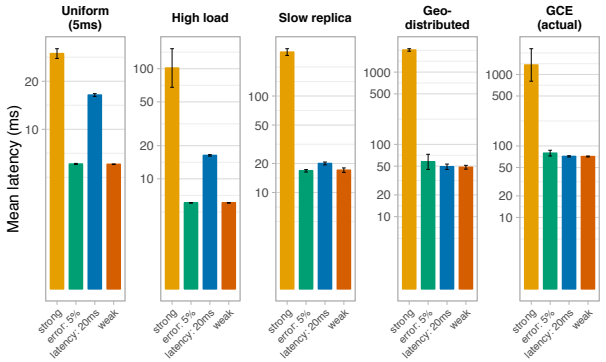


Figure 9. *Ticket service: mean latency*, ***log scale.*** Strong consistency is far too expensive (>10× slower) except when load and latencies are low, but 5% error tolerance allows latency to be comparable to weak consistency. The 20ms latency-bound variant is either slower or defaults to weak, providing little benefit. Note: the ticket `Pool` is safe even when weakly consistent.

has less margin for error at the end). Real-world workloads exhibit power law distributions [20], 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 9 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 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).

6.3.2. Twitter clone

Our second application is a Twitter-like service based on the Redis data modeling example, Retwis [56]. The data model is simple: each user has a `Set` of followers, a `Set` of

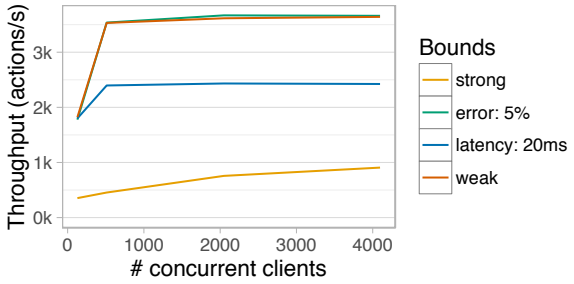


Figure 10. *Ticket service:* throughput on Google Compute Engine globally-distributed testbed. Note that this counts *actions* such as `tweet`, which can consist of multiple storage operations. Because error tolerance does mostly weak reads and writes, its performance tracks *weak*. Latency bounds reduce throughput due to issuing the same operation in parallel.

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 [7]. 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 12 that for all but the local (same

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

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

def viewTimeline(user: User) = {
  // `range` returns `Rushed[List[TweetID]]`
  user.timeline.range(0,10) match { // use match to unpack
    case Consistent(tweets) =>
      for (tweetID <- tweets)
        displayTweet(tweetID)
    case Inconsistent(tweets) =>
      // tweets may not have fully propagated yet
      for (tweetID <- tweets)
        // guard load and retry if there's an error
        Try { displayTweet(tweetID) } retryOnError
  }
}

def displayTweet(id: TweetID, user: User) = {
  val rct: Interval[Int] = tweets(id).retweets.size()
  if (rct > 1000) // abbreviate large counts (e.g. "2k")
    display("${rct.min/1000}k retweets")
  else if (rct.min == rct.max) // count is precise!
    display("Exactly ${rct.min} retweets")
  //...
  // here, `contains` returns `Consistent[Boolean]`
  // so it is automatically coerced to a Boolean
  if (tweets(id).retweets.contains(user))
    disable_retweet_button()
}
```

Figure 11. 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).

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.

7. Related Work

Consistency models. There is a thriving ecosystem of consistency models: from *sequential consistency* [34] and *linearizability* [31] on the strong side, to *eventual consistency* [65] 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 [61]. Many datastores support configuring consistency at a fine granularity: Cassandra [4] per operation, Riak [11] on an object or namespace granularity, as well as others [35, 59]. The Conit consistency model [68] breaks down the consistency spectrum into numerical error, order error, and staleness, but requires annotating each operation and explicit dependency tracking, rather than annotating ADTs.

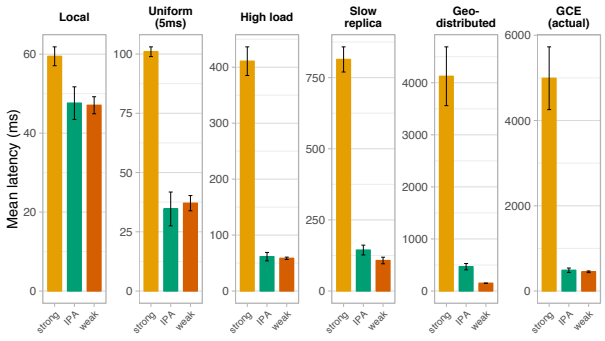


Figure 12. *Twitter clone: mean latency (all actions).* The IPA version is on-par with weak consistency in all but one of the conditions, while strong consistency is 2-10× slower.

Higher-level consistency requirements. Some programming models have gone beyond plain consistency models and allowed programmers to express correctness criteria directly. Quelea [58] has programmers write *contracts* to describe *visibility* and *ordering* constraints, independent of any particular consistency hierarchy, then the system automatically selects the consistency level necessary for each operation. In Indigo [9], programmers write *invariants* over abstract state and annotate post-conditions on actions in terms of the abstract state. The system analyzes annotated programs and adds coordination logic to prevent invariant violations, using a reservation system to enforce numer constraints. Neither Indigo nor Quelea, however, allow programmers to specify approximations or error tolerances, nor do they enforce any kind of performance bounds.

IPA's latency-bound policies were inspired by the *consistency-based SLAs* of Pileus [62]. 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. Other systems, including PRACTI [12], PADS [13], and WheelFS [60], have given developers ways of expressing their desired performance and correctness requirements through *semantic cues* and policies.

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 [14, 47, 50, 52] and distributed caches [44, 48]. 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).

The above techniques are relevant but largely orthogonal to our work: they provide many techniques which could be used in an IPA datastore to trade off correctness in new ways. This work builds on those insights, introducing a new error tolerance mechanism, proposing ADT-level annotations rather than per-operation, but most importantly, providing *type safety* via IPA types, which ensure that all possible edge cases are handled whenever the system adjusts consistency to meet performance targets. Previous systems gave some feedback to programs about achieved consistency, but did not provide facilities to ensure and help developers use the information correctly.

Types for distributed systems. *Convergent* (or *conflict-free*) *replicated data types* (CRDTs) [57] 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 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 [10] enforces hard limits on the global value of a counter in a way similar to reservations but less general; this design informed our own reservations system for error bounds.

Types for approximation. IPA's type system is inspired by work on *approximate computing*, in which computations can be selectively made inaccurate to improve energy efficiency and performance. EnerJ [16, 54] and Rely [19, 40] track the flow of approximate values to prevent them from interfering with precise computation. IPA's interval types are similar to Uncertain<T>'s probability distributions [15] and to interval analysis [41]. One key difference for IPA is that inconsistent values can be strengthened by forcing additional synchronization if necessary. IPA also builds on information flow tracking systems [24, 42, 53], which use static type checking and dynamic analysis to enforce *non-interference* between sensitive data and untrusted channels.

8. 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 *in-*

consistent, performance-bound, and approximate types.

References

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

[2] Scala in the enterprise. http://www.scala-lang.org/old/node/1658, March 2009.

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

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

[5] Apache Software Foundation. Apache spark - lightning-fast cluster computing. http://spark.apache.org/, 2016a.

[6] Apache Software Foundation. Apache thrift. https://thrift.apache.org/, 2016b.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267680.1267685.

[13] Nalini Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Mike Dahlin, and Robert Grimm. Pads: A policy architecture for distributed storage systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 59–73, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558982.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Travis Brown. Scala at scale at Twitter (talk). http://conferences.oreilly.com/oscon/open-source-2015/public/schedule/detail/42332, July 2015.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

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

[26] 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.

[27] 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.

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

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

[30] Susan Hall. Employers can't find enough scala talent. http://insights.dice.com/2014/04/04/employers-cant-find-enough-scala-talent/, March 2014.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] Lightbend Inc. Akka. http://akka.io/, 2016.

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

[38] 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.

[39] 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/.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

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

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

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

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

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

[54] 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.

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

[56] 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.

[57] 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.

[58] 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.

[59] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Op-*

erating Systems Principles - SOSP'11, SOSP. Association for Computing Machinery (ACM), 2011. doi:10.1145/2043556.2043592.

[60] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, NSDI'09, pages 43–58, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558981.

[61] 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.

[62] 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.

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

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

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

[66] 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.

[67] 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.

[68] 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.