Disciplined Inconsistency with Consistency Types

Brandon Holt James Bornholt Irene Zhang Dan Ports Mark Oskin Luis Ceze
University of Washington
{bholt,bornholt,iyzhang,drkp,oskin,luisceze}@cs.uw.edu

Abstract

Distributed applications and web services, such as online stores or social networks, are expected to be scalable, available, responsive, and fault-tolerant. To meet these steep requirements in the face of high round-trip latencies, network partitions, server failures, and load spikes, applications use eventually consistent datastores that allow them to weaken the consistency of some data. However, making this transition is highly error-prone because relaxed consistency models are notoriously difficult to understand and test.

In this work, we propose a new programming model for distributed data that makes consistency properties explicit and uses a type system to enforce *consistency safety*. With the *Inconsistent, Performance-bound, Approximate* (IPA) storage system, programmers specify performance targets and correctness requirements as constraints on persistent data structures and handle uncertainty about the result of datastore reads using new *consistency types*. We implement a prototype of this model in Scala on top of an existing datastore, Cassandra, and use it to make performance/correctness tradeoffs in two applications: a ticket sales service and a Twitter clone. Our evaluation shows that IPA prevents consistency-based programming errors and adapts consistency automatically in response to changing network conditions, performing comparably to weak consistency and 2-10× faster than strong consistency.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed databases

Keywords consistency, type system, programming model

1. Introduction

To provide good user experiences, modern datacenter applications and web services must balance the competing requirements of application correctness and responsiveness. For example, a web store double-charging for purchases or keeping users waiting too long (each additional millisecond of latency [26, 36]) can translate to a loss in traffic and revenue. Worse, programmers must maintain this balance in an unpredictable environment where a black and blue dress [42] or Justin Bieber [38] can change application performance in the blink of an eye.

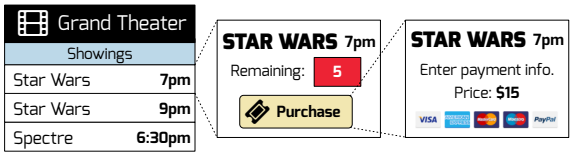
Recognizing the trade-off between consistency and performance, many existing storage systems support configurable consistency levels that allow programmers to set the consistency of individual operations [4, 11, 34, 58]. These allow programmers to weaken consistency guarantees only for data that is not critical to application correctness, retaining strong consistency for vital data. Some systems further allow adaptable consistency levels at runtime, where guarantees are only weakened when necessary to meet availability or performance requirements (e.g., during a spike in traffic or datacenter failure) [59, 61]. Unfortunately, using these systems correctly is challenging. Programmers can inadvertently update strongly consistent data in the storage system using values read from weakly consistent operations, propagating inconsistency and corrupting stored data. Over time, this *undisciplined* use of data from weakly consistent operations lowers the consistency of the storage system to its weakest level.

In this paper, we propose a more disciplined approach to inconsistency in the *Inconsistent, Performance-bound, Approximate (IPA)* storage system. IPA introduces the following concepts:

- *Consistency Safety*, a new property that ensures that values from weakly consistent operations cannot flow into stronger consistency operations without explicit endorsement from the programmer. IPA is the first storage system to provide consistency safety.
- *Consistency Types*, a new type system in which *type safety implies consistency safety*. Consistency types define the consistency and correctness of the returned value from every storage operation, allowing programmers to reason about their use of different consistency levels. IPA's type checker enforces the disciplined use of IPA consistency types statically at compile time.

SoCC '16, October 05 - 07, 2016, Santa Clara, CA, USA.

ACM 978-1-4503-4525-5/16/10…$15.00.
http://dx.doi.org/10.1145/2987550.2987559




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

// called from purchaseTicket & displayEvent
def getTicketCount(event: UUID): Int =
  // use weak consistency for performance
  readWeak(event+"ticket_count")

def purchaseTicket(event: UUID) = {
  val ticket = reserveTicket(event)
  val remaining = getTicketCount(event)
  // compute price based on inconsistent read
  val price = computePrice(remaining)
  display("Enter payment info. Price: ", price)
}
```

Figure 1. Ticket sales service. To meet a performance target in `displayEvent`, developer switches to a weak read for `getTicketCount`, not realizing that this inconsistent read will be used elsewhere to compute the ticket price.

- *Error-bounded Consistency*. IPA is a data structure store, like Redis [54] or Riak [11], allowing it to provide a new type of weak consistency that places *numeric error bounds* on the returned values. Within these bounds, IPA automatically adapts to return the strongest IPA consistency type possible under the current system load.

We implement an IPA prototype based on Scala and Cassandra and show that IPA allows the programmer 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 three applications: a simple counter, a Twitter clone based on Retwis [55] and a Ticket sales service modeled after FusionTicket [1].

2. The Case for Consistency Safety

Unpredictable Internet traffic and unexpected failures force modern datacenter applications to trade off consistency for performance. In this section, we demonstrate the pitfalls of doing so in an undisciplined way. As an example, we describe a movie ticketing service, similar to AMC or Fandango. Because ticketing services process financial transactions, they must ensure correctness, which they can do by storing data in a strongly consistent storage system. Unfortunately, providing strong consistency for every storage operation can cause the storage system and application to collapse under high load, as several ticketing services did in October 2015, when tickets became available for the new Star Wars movie [21].

To allow the application to scale more gracefully and handle traffic spikes, the programmer may choose to weaken the consistency of some operations. As shown in Figure 1, the ticket application displays each showing of the movie along with the number of tickets remaining. For better performance, the programmer may want to weaken the consistency of the read operation that fetches the remaining ticket count to give users an estimate, instead of the most up-to-date value. Under normal load, even with weak consistency, this count would often still be correct because propagation is typically fast compared to updates. However, eventual consistency makes no guarantees, so under heavier traffic spikes, the values could be significantly incorrect and the application has no way of knowing by how much.

While this solves the programmer's performance problem, it introduces a data consistency problem. Suppose that, like Uber's surge pricing, the ticket sales application wants to raise the price of the last 100 tickets for each showing to $15. If the application uses a strongly consistent read to fetch the remaining ticket count, then it can use that value to compute the price of the ticket on the last screen in Figure 1. However, if the programmer reuses `getTicketCount` which used a weak read to calculate the price, then this count could be arbitrarily wrong. The application could then over- or undercharge some users depending on the consistency of the returned value. Worse, the theater expects to make $1500 for those tickets with the new pricing model, which may not happen with the new weaker read operation. Thus, programmers need to be careful in their use of values returned from storage operations with weak consistency. Simply weakening the consistency of an operation may lead to unexpected consequences for the programmer (e.g., the theater not selling as many tickets at the higher surge price as expected).

In this work, we propose a programming model that can prevent using inconsistent values where they were not intended, as well as introduce mechanisms that allow the storage system to dynamically adapt consistency within predetermined performance and correctness bounds.

3. Programming Model

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 programming model, 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 programming model consists of three parts:

- Abstract data types (ADTs) implement common data structures (such as `Set[T]`) on distributed storage.
- Consistency policies on ADTs specify the desired consistency level for an object in application-specific terms (such as latency or accuracy bounds).

ADT / Method	Consistency(Strong)	Consistency(Weak)	LatencyBound(_)	ErrorTolerance(_)
Counter.read()	Consistent[Int]	Inconsistent[Int]	Rushed[Int]	Interval[Int]
Set.size()	Consistent[Int]	Inconsistent[Int]	Rushed[Int]	Interval[Int]
Set.contains(x)	Consistent[Bool]	Inconsistent[Bool]	Rushed[Bool]	N/A
List[T].range(x,y)	Consistent[List[T]]	Inconsistent[List[T]]	Rushed[List[T]]	N/A
UUIDPool.take()	Consistent[UUID]	Inconsistent[UUID]	Rushed[UUID]	N/A
UUIDPool.remain()	Consistent[Int]	Inconsistent[Int]	Rushed[Int]	Interval[Int]

Table 1. Example ADT operations; *consistency policies* determine the *consistency type* of the result.

- Consistency types track the consistency of operation results and enforce consistency safety by requiring developers to consider weak outcomes.

Programmmers annotate ADTs with consistency policies to choose their desired level of consistency. The *consistency policy* on the *ADT operation* determines the *consistency type* of the result. Table 1 shows some examples; the next few sections will introduce each of the policies and types in detail. 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 programming interface enables performance–correctness trade-offs, because consistency policies on ADTs allow the runtime to select a consistency level 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, 25, 31, 54] to the pseudo-relational model of Cassandra [32]. 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 [57].

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. Annotating ADTs with consistency policies 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. Consistency Policies

Previous systems [4, 11, 34, 58, 61] 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 programming model provides a set of consistency policies that can be placed on *ADT instances* to specify consistency properties for the lifetime of the object. Consistency policies come in two flavors: static and dynamic.

Static policies are fixed, 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 policies specify a consistency level in terms of application 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.

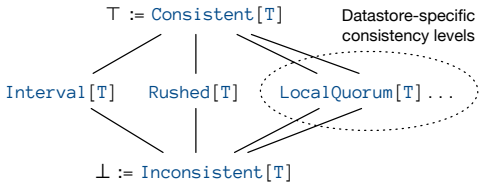


Figure 2. IPA Type Lattice parameterized by a type `T`.

Dynamic policies 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 policies 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 could 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 could have a relaxed accuracy policy, specified with `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. Consistency Types

The key to consistency safety in IPA is the consistency types—enforcing type safety directly enforces consistency safety. Read operations of ADTs annotated with consistency policies return instances of a *consistency type*. These consistency 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. This could be enforced dynamically, as in dynamic information flow control systems, but the static guarantees of a type system allow errors to be caught at compile time.

The consistency types encapsulate information about the consistency achieved when reading a value. Formally, the consistency 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 consistency types*), reflecting their possible inconsistency. The bottom element `Inconsistent[T]` specifies an object with the weakest possible (or unknown) consistency. The other consistency types follow a subtyping relation $\prec$ as illustrated in Figure 2.

The only possible operation on `Inconsistent[T]` is to *endorse* it. Endorsement is an upcast, invoked by `endorse(x)`, to the top element `Consistent[T]` from other types in the lattice:

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

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 where 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 consistency types which correspond to the dynamic 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 consistency type `Rushed[T]` is the result of read operations performed on an ADT with consistency policy `LatencyBound(x).Rushed[T]` is a *sum (or union) type*, with one variant per consistency level available to the implementation of `LatencyBound`. Each variant is itself a consistency 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 (that is, a quorum of replicas within a single datacenter). 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(x) => print(x+"???")
}
```

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 consistency

types, omitted cases can be matched by any type lower in the lattice, including the bottom element `Inconsistent(x)`; 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

Tagging values with a consistency level is useful because it helps programmers tell which operation reorderings are possible (e.g. strongly consistent operations will be observed to happen in program order). However, accuracy policies provide a different way of dealing with inconsistency by expressing it in terms of value uncertainty. They require knowing the *abstract behavior* of operations in order to determine the change in abstract state which results from each reordered operation (e.g., reordering increments on a Counter has a known effect on the value of reads).

The weak consistency 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` type is that the interval must include the result of some linearizable execution. 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 101 (provided no other updates are occurring). However, if `size` is annotated with `ErrorTolerance(5%)`, then it could return any interval that includes 101, such as $[95, 105]$ or $[100, 107]$, so the client cannot tell if the recent `add` was included in the size. 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 entire interval is above zero, then users can be assured that there are sufficient tickets remaining. In fact, because the interval could represent many possible linearizable executions, 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 could 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 consistency policies

The consistency policies introduced in the previous section allow programmers to describe application-level correctness properties. Static consistency policies (e.g. Strong) are enforced by the underlying storage system; the annotated ADT methods simply set the desired consistency level when issuing requests to the store. The dynamic policies each require a new runtime mechanism to enforce them: parallel operations with latency monitoring for latency bounds, and reusable reservations for error tolerance. But first, we briefly review consistency in Dynamo-style replicated systems.

To be sure a strong read sees a particular write, the two must be guaranteed to coordinate with overlapping sets of replicas (*quorum intersection*). 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].

Because overall consistency is dependent on both the strength of reads *and* writes, it really does not make sense to specify consistency policies on individual operations in isolation. Declaring consistency policies on an entire ADT, however, allows the implementer of the ADT to ensure that all combinations of reads and writes achieve the specified consistency.

4.1. Static bounds

Static consistency policies are typically enforced by the underlying datastore, but they require the designer of each ADT to carefully choose how to implement them. To support the `Consistency(Strong)` policy, the designer of each ADT must choose consistency levels for its operations which together enforce strong consistency. For example, if a developer knows that updates to a `Counter` are more common, they may choose to require the `read` operation to synchronize with all replicas (`ALL`), permitting `increment` and `decrement` to wait for only a single replica (`ONE`) without violating strong consistency.

4.2. 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.

Our implementation of latency-bound types takes a generic approach: it issues read requests at different consistency lev-

els in parallel. It composes the parallel operations and returns a result either when the strongest operation returns, or with the strongest available result at the specified time limit. If no responses are available at the time limit, it waits for the first to return.

This approach makes no assumptions about the implementation of read operations, making it easily adaptable to different storage systems. Some designs may permit more efficient implementations: for example, in a Dynamo-style storage system we could send read requests to all replicas, then compute the most consistent result from all responses received within the latency limit. However, this requires deeper access to the storage system implementation than is traditionally available.

4.2.1. Monitors

The main problem with our approach is that it wastes work by issuing parallel requests. 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.

4.3. Error bounds

We implement error bounds by building on the concepts of *escrow* and *reservations* [27, 44, 48, 50]. 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.

4.3.1. 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, 57].

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.

4.3.2. Enforcing error bounds

Reservations have been used previously to enforce hard global invariants in the form of upper or lower bounds on values [10], integrity constraints [9], or logical assertions [37]. 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 reser-

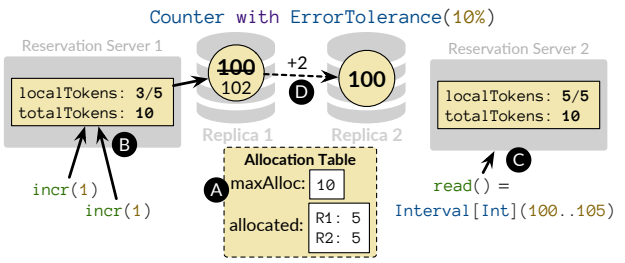


Figure 3. *Enforcing error bounds on a Counter:* (A) Each replica has some number of tokens allocated to it, must add up to less than the max (in this case, 10% of the current value). (B) Reservation Server 1 has sufficient tokens available, so both increments consume a token and proceed to Replica 1. (C) Reads return the range of possible values, determined by total number of allocated tokens; in this case, it reads the value 100, knows that there are 10 tokens total, but 5 of them (local to RS2) are unused, so it returns 100..105. (D) *Eventually,* when the increments have propagated, reservation server reclaims its tokens.

vations, so it knows that there cannot be more than 6 and can return the interval $[100, 106]$.

4.3.3. Narrowing bounds

Error tolerance policies set an *upper bound* on the amount of error; ideally, the interval returned will be more precise than the maximum error when conditions are favorable, such as when there are few update operations. Rather than assuming the total number of tokens is always the maximum allowable by the error bound, we instead keep an *allocation table* for each record that tracks the number of tokens allocated to each reservation server. If a reservation server receives an update operation and does not have enough tokens allocated, it updates the allocation table to allocate tokens for itself. The allocation table must preserve the invariant that the total does not exceed the maximum tokens allowed by the current value. For example, for a value of 100, 10 tokens were allowed, but after 1 decrement, only 9 tokens are allowed. Whenever this occurs, the server that changed the bound must give up the "lost" token out of its own allocations. As long as these updates are done atomically (in Cassandra, this is done using linearizable conditional updates), the global invariant holds. Because of this synchronization, reading and writing the allocation table is expensive and slow, so we use long leases (on the order of seconds) within each reservation server to cache their allocations. When a lease is about to expire, the server preemptively refreshes its lease in the background so that writes do not block unnecessarily.

For each type of update operation there may need to 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 [45] library for type-safe access to Cassandra data. Reservation server middleware is also built in Scala using Twitter's Finagle framework [63]. Communication is done between clients and Cassandra via prepared statements, and between clients and reservation 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 [35] and Spark [5], and at established companies such as Twitter and LinkedIn [2, 18, 29].

The IPA type system, responsible for consistency safety, is also simply part of our client library, 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.

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. Figure 4 shows how a Counter with error tolerance bounds is implemented using these pools. The library of reference ADTs includes:

```
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 with error bounds.

- `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 [28]) and simulated network conditions. We start with 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.

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 2. Network conditions for experiments: latency from client to each replicas, with standard deviation if high.

6.1. Simulating adverse conditions

To control for variability, 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. We use a local test cluster with nodes linked by standard ethernet and Linux's Network Emulation facility [62] (`tc netem`) to introduce packet delay and loss at the operating system level. We use Docker containers [24] to enable fine-grained control of the network conditions between processes on the same physical node.

Table 2 shows the set of conditions we use in our experiments. The *uniform 5ms* link simulates a well-provisioned datacenter; *slow replica* models contention or hardware problems that cause one replica to be slower than others, and *geo-distributed* replicates the latencies between virtual machines in the U.S., Europe, and Asia on Amazon EC2 [3]. These simulated conditions are validated by experiments on Google Compute Engine with virtual machines in four datacenters: the client in *us-east*, and the storage replicas in *us-central*, *europe-west*, and *asia-east*. We elide the results for *Local* (same rack in our testbed) except in Figure 11 because the differences between policies are negligible, so strong consistency should be the default there.

6.2. Microbenchmark: Counter

We start by measuring the performance of a 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 provide predictable performance while maximizing consistency; when latencies and load are low it is often possible to achieve strong consistency. Figure 5 compares the latency of 10ms and 50ms latency bounds with strong and weak consistency. As expected, there is a significant cost to strong consistency under all network conditions. IPA cannot achieve strong consistency under 10ms in any case, so the system must always default to weak consistency. With a 50ms bound, IPA can achieve strong consistency in conditions when network latency is low (i.e., the single dat-

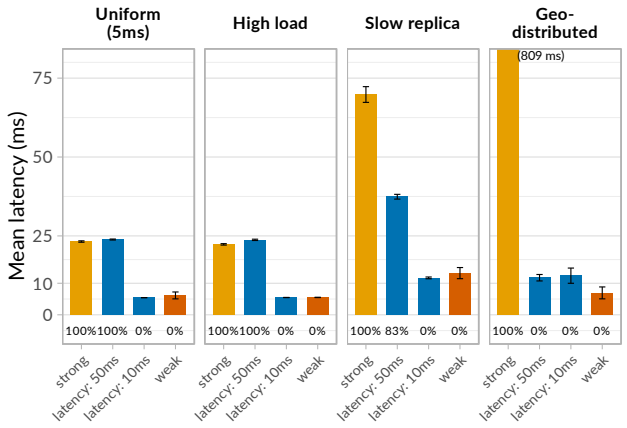


Figure 5. *Counter: latency bounds, mean latency*. Beneath each bar is the % of strong reads. Strong consistency is never possible for the 10ms bound, but 50ms bound achieves mostly strong, only resorting to weak when network latency is high.

acenter case). Cassandra load balances clients, so with one slow replica, IPA will attempt to achieve strong consistency for all clients but not succeed. In our experiments, IPA was able to get strong consistency 83% of the time. In the geo-distributed case, there are no 2 replicas within 50ms of our client, so strong consistency is never possible and IPA adapts to only attempt weak.

Figure 7 shows the 95th percentile latencies for the same workload. The tail latency of the 10ms bound is comparable to weak consistency, whereas the 50ms bound overloads the slow server with double the requests, causing it to exceed the latency 5% of the time. There is a gap between latency-bound and weak consistency in the geo-distributed case 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 measures the cost of enforcing error bounds using the reservation system described in §4.3, and its precision. Reservations move synchronization off the critical path: by distributing write permissions among replicas, reads can get strong guarantees from a single replica. Note that reservations impact write performance, so we must consider both.

Figure 6a shows latencies for error bounds of 1%, 5%, and 10%, plotting the average of read *and* increment operations. As expected, tighter error bounds increase latency because it forces more frequent synchronization between replicas. The 1% error bound provides most of the benefit, except in the slow replica and geo-distributed environments where it forces synchronization frequently enough that the added latency slows down the system. 5-10% error bounds provide latency comparable to weak consistency. In the geo-distributed case, the coordination required for reservations makes even the 10% error bound 4× slower than weak consistency, but this is still 28× faster than strong consistency.

While we have verified that error-bounded reads remain within our defined bounds, we also wish to know what error occurs in practice. We modified our benchmark to observe the actual error from weak consistency by incrementing counters a predetermined amount and reading the value; results are shown in Figure 6b. We plot the percent error of weak and strong against the actual observed *interval width* for a 1% error bound, going from a read-heavy (1% increments) to write-heavy (all increments, except to check the value).

First, we find that the mean interval is less than the 1% error bound because, for counters that are less popular, IPA is able to return a more precise interval. At low write rate, this interval becomes even smaller, down to .5% in the geo-distributed experiment. On the other hand, we find that the mean error for weak consistency is also less than 1%; however, the *maximum* error observed is up to 60%. This motivates error bounded consistency to ensure that applications never see drastically incorrect values from weakly consistent operations. Further, using the `Interval` type, IPA is able to give the application an estimate of the variance in the weak read, which is often more precise than the upper bound set by the error tolerance policy.

6.3. Applications

Next, we explore the implementation of two applications in IPA and compare their performance against Cassandra using purely strong or weak consistency on our simulated network testbed and Google Compute Engine.

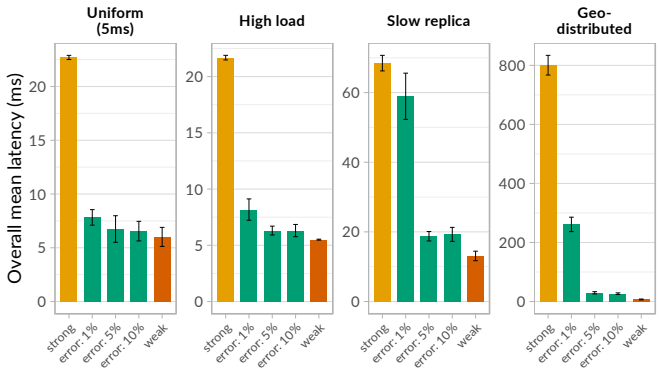
6.3.1. Ticket service

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 [65, 66]. We support the following actions:

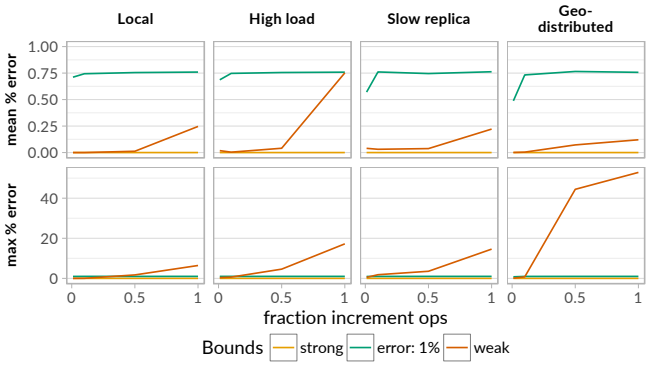
- `browse`: List events by venue
- `viewEvent`: View the full description of an event including number of remaining tickets
- `purchase`: Purchase a ticket (or multiple)
- `addEvent`: Add an event at a venue.

Figure 8 shows a snippet of code from the IPA implementation (compare with Figure 1). Tickets are modeled using the `UUIDPool` type, which generates unique identifiers to reserve tickets for purchase. The ADT ensures that, even with weak consistency, it never gives out more than the maximum number of tickets, so it is safe to `endorse` the result of the `take` operation (though there is a possibility of a false negative). We use an error tolerance annotation to bound the inaccuracy of ticket counts better than the weak read from Figure 1. Now `getTicketCount` returns an `Interval`, forcing us to decide how to handle the range of possible ticket counts. We decide to use the `max` value to compute the ticket price to be fair to users; the 5% error bound ensures we don't sacrifice too much profit.

We run a workload modelling a typical small-scale deployment: 50 venues and 200 events, with an average of 2000



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



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

Figure 6. *Counter benchmark: error tolerance*. In (a), we see that wider error bounds reduce mean latency because fewer synchronizations are required, matching *weak* around 5-10%. In (b), we see 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%.

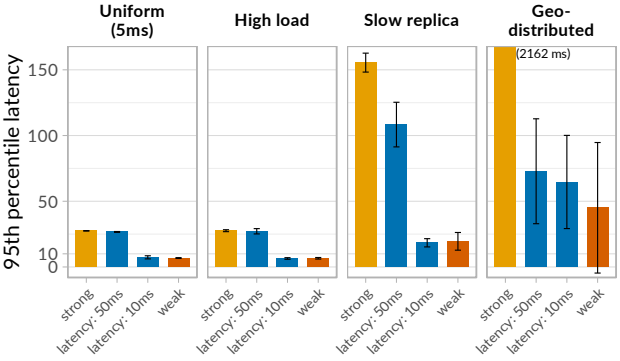


Figure 7. *Counter: 95th percentile latency*. Latency bounds keep tail latency down, backing off to weak when necessary.

tickets each (gaussian distribution centered at 2000, stddev 500); this ticket-to-event ratio ensures that some events run out tickets. Because real-world workloads exhibit power law distributions [20], we use a moderately skewed Zipf distribution with coefficient of 0.6 to select events.

Figure 9 shows the average latency of a workload consisting of 70% `viewEvent`, 19% `browse`, 10% `purchase`, and 1% `addEvent`. We use a log scale because strong consistency has over 5× higher latency. The `purchase` event, though only 10% of the workload, drives most of the latency increase because of the work required to prevent over-selling tickets. We explore two different IPA 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. Both perform with nearly the same latency as weak consistency. The latency bound version has 92% strong reads in low-latency conditions (*uniform* and *high load*), but falls back to weak for the more adverse conditions.

Figure 9 also shows results on Google Compute Engine *(GCE)*. We see that the results of real geo-replication validate the findings of our simulated geo-distribution results.

```
// 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))

// called from displayEvent (& purchaseTicket)
def getTicketCount(event: UUID): Interval[Int] =
  tickets(event).remaining()

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 remaining = getTicketCount(event)
      // use maximum count possible to be fair
      val price = computePrice(remaining.max)
      display("Ticket reserved. Price: $" + price)
      prompt_for_payment_info(price)
    case None =>
      display("Sorry, all sold out.")
  }
}
```

Figure 8. *Ticket service* code demonstrating consistency types.

On this workload, we observe that the 5% error bound performs well even under adverse conditions, which differs from our findings in the microbenchmark. This is because ticket pools begin *full*, with many tokens available, requiring less synchronization until they are close to running out. Contrast this with the microbenchmark, where counters started at small numbers (average of 500), where a 5% error tolerance means fewer tokens.

6.3.2. Twitter clone

Our second application is a Twitter-like service based on the Redis data modeling example, Retwis [55]. The data model is simple: each user has a `Set` of followers, and a `List` of tweets in their timeline. When a user tweets, the tweet ID is eagerly inserted into all of their followers' timelines. Retweets are tracked with a `Set` of users who have retweeted each tweet.

Figure 11 shows the data model with policy annotations: latency bounds on followers and timelines and an error bound

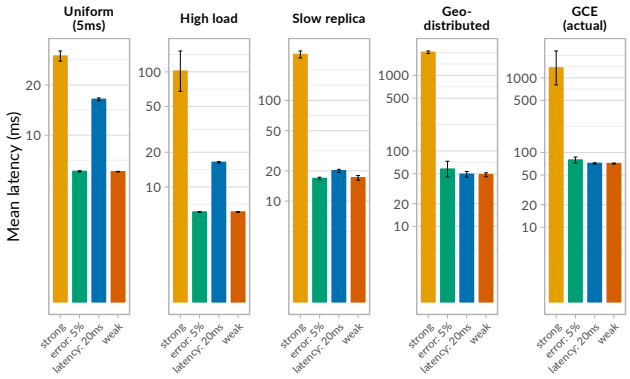


Figure 9. *Ticket service: mean latency,* ***log scale.*** Strong consistency is far too expensive (>10× slower) except when load and latencies are low, but 5% error tolerance allows latency to be comparable to weak consistency. The 20ms latency-bound variant is either slower or defaults to weak, providing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

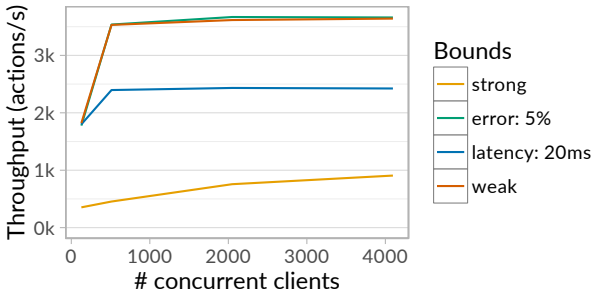


Figure 10. *Ticket service:* throughput on Google Compute Engine globally-distributed testbed. Note that this counts *actions* such as `tweet`, which can consist of multiple storage operations. Because error tolerance does mostly weak reads and writes, its performance tracks *weak*. Latency bounds reduce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

on the retweets. This ensures that when tweets are displayed, the retweet count is not grossly inaccurate. As shown in `displayTweet`, highly popular tweets with many retweets can tolerate approximate counts – they actually abbreviate the retweet count (e.g. "2.4M") – but average tweets, with less than 20 retweets, will get an exact count. This is important because average users may notice if a retweet of one of their tweets is not reflected in the count, and this does not cost much, whereas popular tweets, like Ellen Degeneres's record-breaking celebrity selfie [7] with 3 million retweets, have more slack due to the 5% error tolerance.

The code for `viewTimeline` in Figure 11 demonstrates how latency-bound `Rushed[T]` types can be destructured with a match statement. In this case, the timeline (list of tweet IDs) is retrieved with a latency bound. Tweet content is added to the store before tweet IDs are pushed onto timelines, so with strong consistency we know that the list of IDs will all be able to load valid tweets. However, if the latency-bound type returns with weak consistency (`Inconsistent` case), then this *referential integrity* property may not hold. In that case, we must guard the call to `displayTweet` and

```
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 data model with policy annotations, `Rushed[T]` helps catch referential integrity violations and `Interval[T]` represents approximate retweet counts.

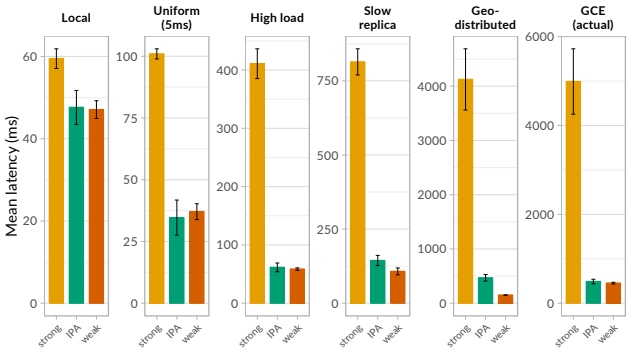


Figure 12. *Twitter clone: mean latency (all actions).* The IPA version performance comparably with weak consistency in all but one case, while strong consistency is 2-10× slower.

retry if any of the operations fails (e.g., if the retweet set wasn't created yet).

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 see in Figure 12 that for all but the local (same rack) case, strong consistency is over 3× slower. 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 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 used almost exclusively.

7. Related Work

Consistency models. IPA's consistency types could be extended to more of the thriving ecosystem of consistency models from *sequential consistency* [33] and *linearizability* [30], to *eventual consistency* [64]. A variety of intermediate models fit elsewhere in the spectrum, each making different tradeoffs balancing performance against ease of programming. Session guarantees, including *read-your-writes*, strengthen ordering for individual clients but reduce availability [60]. Many datastores allow fine-grained consistency control: Cassandra [4] per operation, Riak [11] on an object granularity, and others [34, 58]. The Conit consistency model [67] 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.

Higher-level consistency requirements. Some programming models allow users to express application correctness criteria directly. In Quelea [57], programmers write *contracts* to describe *visibility* and *ordering* constraints and the system selects the necessary consistency. In Indigo [9], programmers write *invariants* over abstract state and annotate postconditions on actions in terms of the abstract state and the system adds coordination logic, employing reservations for numeric bounds. 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 Pileus's [61] *consistency-based SLAs*. 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 [59], have given developers ways of expressing their desired performance and correctness requirements through *semantic cues*.

The principle that applications may be willing to tolerate slightly stale data in exchange for improve performance has a long history in databases [14, 46, 49, 51] and distributed caches [43, 47]. 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 techniques which could be used in IPA to trade off correctness in new ways. This work builds on those insights, introducing a new error tolerance mechanism, proposing ADT annotations rather than per-operation, but most importantly, providing *consistency safety* via consistency types, which ensure that all possible 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 developers use the information correctly.

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, 53] and Rely [19, 39] 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 [40]. One key difference for IPA is that inconsistent values can be strengthened if desired with additional synchronization.

Types for distributed and secure systems. *Convergent* (or *conflict-free*) *replicated data types* (CRDTs) [56] are data types designed for eventual consistency that guarantee convergence by forcing all updates to commute. CRDTs can be useful because they allow concurrent updates with meaningful 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. The Bounded Counter CRDT [10] informed the design of our reservations for error bounds, but enforces global value bounds and does not bound uncertainty. Information flow tracking systems [23, 41, 52], also use static type checking and dynamic analysis to enforce *non-interference* between sensitive data and untrusted channels, but, to the best of our knowledge, those techniques have not been applied to enforce consistency safety by separating weakly and strongly consistent data.

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 opportunities for optimization. Meanwhile, consistency 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 a sample of what is possible within the framework of *Inconsistent, Performance-bound, and Approximate* types.

Acknowledgments

We thank the anonymous reviewers for their feedback. This research was funded in part by NSF under grant #1518703, DARPA under contract FA8750–16–2–0032, C-FAR, one of

the six SRC STARnet Centers, sponsored by MARCO and DARPA, and gifts by Google and Microsoft.

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

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

[25] 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.

[26] 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.

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

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

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

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

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

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

[37] 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.

[38] 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/.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

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

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

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

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

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

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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

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

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

[65] 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.

[66] 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.

[67] 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.