Incremental Consistency Guarantees for Replicated Objects

Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi*

School of Computer and Communication Sciences,
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{*rachid.guerraoui, matej.pavlovic, dragos-adrian.seredinschi*}*@epfl.ch*

Abstract

Programming with replicated objects is difficult. Developers must face the fundamental trade-off between consistency and performance head on, while struggling with the complexity of distributed storage stacks. We introduce *Correctables*, a novel abstraction that hides most of this complexity, allowing developers to focus on the task of balancing consistency and performance. To aid developers with this task, Correctables provide *incremental consistency guarantees*, which capture successive refinements on the result of an ongoing operation on a replicated object. In short, applications receive both a preliminary—fast, possibly inconsistent—result, as well as a final—consistent—result that arrives later.

We show how to leverage incremental consistency guarantees by speculating on preliminary values, trading throughput and bandwidth for improved latency. We experiment with two popular storage systems (Cassandra and ZooKeeper) and three applications: a Twissandra-based microblogging service, an ad serving system, and a ticket selling system. Our evaluation on the Amazon EC2 platform with YCSB workloads A, B, and C shows that we can reduce the latency of strongly consistent operations by up to 40% (from 100*ms* to 60*ms*) at little cost (10% bandwidth increase, 6% throughput drop) in the ad system. Even if the preliminary result is frequently inconsistent (25% of accesses), incremental consistency incurs a bandwidth overhead of only 27%.

1. Introduction

Replication is a crucial technique for achieving performance—i.e., high availability and low latency—in large-scale applications. Traditionally, strong consistency protocols hide replication and ensure correctness by exposing a single-copy abstraction over replicated objects [26, 46]. There is a trade-off, however, between consistency and performance [14, 21, 33]. Weak consistency [28] boosts performance, but introduces the possibility of incorrect (anomalous) behavior.

A common argument in favor of weak consistency is that such anomalous behavior is rare in practice. Indeed, studies reveal that on expectation, weakly consistent values are often correct even with respect to strong consistency [19, 55]. Applications which primarily demand performance thus forsake stronger models and resort to weak consistency [16, 28].

There are cases, however, where applications often diverge from correct behavior due to weak consistency. As an extreme example, an execution of YCSB workload A [25] in Cassandra [45] on a small $1K$ objects dataset can reveal stale values for 25% of weakly consistent read operations (Figure 7 in §6). This happens when using the *Latest* distribution, where read activity is skewed towards popular items [25]. In other cases, even very rare anomalies are unacceptable (e.g., when handling sensitive data such as user passwords), making strongly consistent access a necessity. For this class of applications, correctness supersedes performance, and strong consistency thus takes precedence [26].

There is also a large class of applications which do not have a single, clear-cut goal (either performance or correctness). Instead, such applications aim to satisfy *both* of these conflicting demands. These applications fall in a *gray zone*, somewhere in-between the two previous classes, as we highlight in Figure 1. Typically, these applications aim to strike an optimal balance of consistency and performance by employing different consistency models, often at the granularity of individual operations [18, 24, 43, 51, 66]. Choosing the appropriate consistency model, even at this granularity, is hard, and

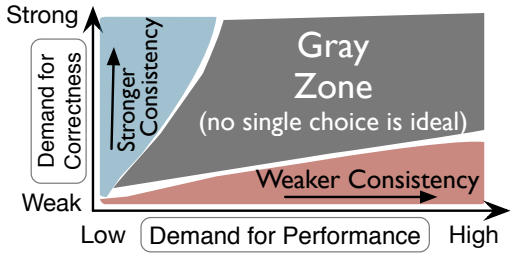

Figure 1: Many applications fall into a *gray zone*, torn between the need for both performance and correctness.

*Author names appear in alphabetical order.

the result is often sub-optimal, as developers still end up with fixing a certain side of the consistency/performance trade-off (and sacrificing the other side).

Moreover, programming in the gray area is difficult, as developers have to juggle different consistency models in their applications [24, 43]. If programming with a single consistency model (such as weak consistency [26]) is non-trivial, then mixing multiple models is even harder [50]. In their struggle to optimize performance with consistency, developers must go up against the full complexity of the underlying storage stack. This includes choosing locations (cache or backup or primary replica), dealing with coherence and cache-bypassing, or selecting quorums. These execution details reflect as a burden on developers, complicate application code, and lead to bugs [31, 55].

Our goal is to help with the programming of applications located in the gray area. We accept as a fact that no single consistency model is ideal, providing both high performance and strong consistency (correctness) at the same time [14, 33]. Our insight is to approach this ideal in complementary steps, by *combining consistency models in a single operation*. Briefly, developers can invoke an operation on a replicated object and obtain multiple, incremental *views* on the result, at successive points in time. Each view reflects the operation result under a particular consistency model. Initial (preliminary) views deliver with low latency—but weak consistency—while stronger guarantees arrive later. We call this approach *incremental consistency guarantees* (ICG).

We introduce Correctables, an abstraction which grants developers a clean, consistency-based interface for accessing replicated objects, clearly separating semantics from execution details. This abstraction reduces programmer effort by hiding storage-specific protocols, e.g., selecting quorums, locations, or managing coherence. Correctables are based on *Promises* [53], which are placeholders for a single value that becomes available in the future. Correctables generalize Promises by representing not a single, but multiple future values, corresponding to incremental views on a replicated object.

To the best of our knowledge, our abstraction is the first which enables applications to build on ICG. As few as *two* views suffice for ICG to be useful. The advantage of ICG is that applications can speculate on the preliminary view, hiding the latency of strong consistency, and thereby improving performance [71]. Speculating on preliminary responses is expedient considering that, in many systems, weak consistency provides correct results on expectation [19, 55].

Speculation with ICG is applicable to a wide range of scenarios. Consider, for instance, that a single application-level operation can aggregate multiple—up to hundreds of—storage-level objects [16, 27, 52, 65]. Since these objects are often inter-dependent, they can not always be fetched in parallel. With ICG, the application can use the fast preliminary view to speculatively prefetch any dependent objects. By the time the final (strongly consistent) view arrives, the prefetching would also finish. If the preliminary result was correct (matching the final one), then the speculation is deemed successful, reducing the overall latency of this operation.

Alternatively, ICG can open the door to exploiting application-specific semantics for optimizing performance. Imagine an application requiring a monotonically increasing counter to reach some pre-defined threshold (e.g., number of purchased items in a shop required for a fidelity discount). If a weakly consistent view of the counter already exceeds this threshold, the application can proceed without paying the latency price of a strongly consistent view.

The high-level abstraction centered on consistency models, coupled with the performance benefits of enabling speculation via ICG, are the central contributions of Correctables. We evaluate these performance benefits by modifying two well-known storage systems (Cassandra [45] and ZooKeeper [39]). We plug Correctables on top of these, build three applications (a Twissandra-based microblogging service [10], an ad serving system, and a ticket selling system), and experiment on Amazon EC2.

Our evaluation first demonstrates that there is a sizable time window between preliminary and final views, which applications can use for speculation. Second, using YCSB workloads A, B, and C, we show that we can reduce the latency of strongly consistent operations by up to 40% (from 100*ms* to 60*ms*) at little cost (10% bandwidth increase, 6% throughput drop) in the ad system. The other two applications exhibit similar improvements. Even if the preliminary result is often inconsistent (25% of accesses), incremental consistency incurs a bandwidth overhead of only 27%.

In the rest of this paper, we overview our solution in the context of related work (§2) and present the Correctables interface (§3). We show how applications use Correctables (§4), and describe the bindings to various storage stacks (§5). We then give a comprehensive evaluation (§6) and conclude (§7).

2. Overview & Related Work

This paper addresses the issue of programming and speculating with replicated objects through a novel abstraction called Correctables. In this section, we overview the main concepts behind Correctables, and we contrast our approach with related work.

2.1 Consistency Choices

There is an abundance of work on consistency models. These range from strong consistency protocols [40, 46,

68], some optimized for WAN or a specific environment [26, 29, 44, 48, 72, 74], through intermediary models such as causal consistency [30, 54], to weak consistency [28, 67]. As a recent development, storage systems offer multiple—i.e., *differentiated*—consistency guarantees [24, 43, 62]. This allows applications in the abovementioned gray zone to balance consistency and performance on a per-operation basis: the choice of guarantees depends on how sensitive the corresponding operation is.

Differentiated guarantees can take the form of SLAs [66], policies attached to data [43], dynamic quorum selection for quorum-based storage systems such as Dynamo [28] or others [8, 45], or even ad-hoc operation invariants [18]. In practice, two consistency levels often suffice: weak and strong [1, 5]. Sensitive operations (e.g., account creation or password checking) use the strong level, while less critical operations (e.g., remove from basket) use weak guarantees [43, 66, 73] to achieve good performance.

For instance, in Gemini [51], operations are either Blue (fast, weakly consistent) or Red (slower, strongly consistent). For sensitive data such as passwords, Facebook uses a separate linearizable sub-system [55]. Likewise, Twitter employs strong consistency for "certain sets of operations" [64], and Google's Megastore exposes strong guarantees alongside read operations with "inconsistent" semantics [20]. Another frequent form of differentiated guarantees appears when applications bypass caches to ensure correctness for some operations [16, 60].

Given this great variety of differentiated guarantees, we surmise that applications can benefit from mixing consistency models. The notable downside of this approach is that application complexity increases [50]. Developers must orchestrate different storage APIs and consider the interactions between these protocols [16, 18, 69]. Our work subsumes results in this area. We propose to hide different schemes for managing consistency under a common interface, Correctables, which can abstract over a varying combination of storage tiers and reduce application complexity. In addition, we introduce the notion of *incremental consistency guarantees* (ICG), i.e., progressive refinement of the result of a *single* operation.

2.2 ICG: Incremental Consistency Guarantees

Applications which use strong consistency—either exclusively or for a few operations—do so to avoid anomalous behavior which is latent in weaker models. Interestingly, recent work reveals that this anomalous behavior is rare in practice [19, 55]. There are applications, however, which cannot afford to expose even those rare anomalies.

For instance, consider a system storing user passwords, and say it has 1% chance of exposing an inconsistent password. If such a system demands correctness—as it should—then it is forced to pay the price for strong consistency on *every* access, even though this is not necessary in 99% of cases. We propose ICG to help applications avert this dilemma, and pay for correctness only when inconsistencies actually occur.

With ICG, an application can obtain both weakly consistent (called *preliminary*) and strongly consistent (called *final*) results of an operation, one by one, as these become available. While waiting for the final result, the application can speculatively perform further processing based on the preliminary—which is correct on expectation. Following our earlier example, this would help hide the latency of strong consistency for 99% of accesses.

The full latency of strong consistency is only exposed in case of misspeculation, when the preliminary and final values diverge because the preliminary returned inconsistent data [71]. These are the 1% cases where strong consistency is needed anyway. Speculation through ICG can lessen the most prominent argument against strong consistency, namely its performance penalty. With ICG we pay the latency cost of strong consistency only when necessary, regardless of how often this is the case.

Speculation is a well-known technique for improving performance. Traditionally, the effects of speculation in a system remain hidden from higher-level applications until the speculation confirms, since the effects can lead to irrevocable actions in the applications [41, 57, 59, 71]. Alternatively, it has been shown that leaking speculative effects to higher layers can be beneficial, especially in user-facing applications, where the effects can be undone or the application can compensate in case of misspeculation [36, 47, 49, 61]. We propose to use eventual consistency as a basis for doing speculative work, as a novel approach for improving performance in replicated systems. Also, more generally, we allow the application itself (which knows best), to decide on the speculation boundary [70]—whether to externalize effects of speculation, and later to undo or compensate these effects, or whether to isolate users from speculative state.

Besides speculation, ICG is useful in other cases as well. For instance, applications can choose dynamically whether to settle with a preliminary value and forsake the final value altogether. This is a way to obtain application-specific optimizations, e.g., to enforce tight latency SLAs. Alternatively, we can *expose* the preliminary response to users and revise it later when the final response arrives. This strategy is akin to compensating in case of misspeculation, as mentioned earlier.

Clearly, not all applications are amenable to exploiting ICG. In Table 1 we give a high-level account on three categories of applications: (1) those which have no additional benefit from strong consistency or ICG; (2) those which require correct results but are not amenable to speculation; and at last (3) applications that can obtain

Category	Synopsis	Applications and use cases
Weak Consistency	Use the **weakest, but fastest** consistency model, e.g., by using partial quorums, or going to the closest replica or cache. No benefit from ICG.	Computation on static (BLOBs) content, e.g., thumbnail generator for images and videos, accessing cold data, fraud analysis, disconnected operations in mobile applications, etc.
Strong Consistency	Use the **strongest** available model, e.g., by going to the primary replica. Applications require correct results.	Infrastructure services (e.g., load-balancing, session stores, configuration and membership management services), stock tickers, trading applications, etc.
Incremental Consistency Guarantees (ICG)	Use multiple, **incremental** models. Applications benefit from weakly consistent values (e.g., by speculating or exposing them), but prefer correct results.	E-mail, calendar, social network timeline, grocery list, flight search aggregation, online shopping, news reading, browsing, backup, collaborative editing, authentication and authorization, advertising, online wallets, etc.

Table 1: Different patterns and their corresponding use cases. Many applications can benefit from ICG.

performance without sacrificing correctness by leveraging ICG.

2.3 Client-side Handling of ICG

To program with ICG, applications need to wait asynchronously for multiple replies to an operation (where each reply encapsulates a different guarantee on the result) while doing useful work, i.e., speculate. To the best of our knowledge, no abstraction fulfills these criteria. To minimize the effort of programming with ICG, we draw inspiration from *Promises*, seminal work on handling asynchronous remote procedure calls in distributed systems [53].

A Promise is a placeholder for a value that will become available asynchronously in the future. Given the urgency to handle intricate parallelism and augmenting complexity in applications, it is not surprising that Promises are becoming standard in many languages [6, 2, 12, 31]. We extend the binary interface of Promises (a value either present or absent) to obtain a multi-level abstraction, which incrementally builds up to a final, correct result.

The Observable interface from reactive programming can be seen as a similar generalization of Promises. Observables abstract over asynchronous data streams of arbitrary type and size [56]. Our goal with Correctables, in contrast, is to grant developers access to consistency guarantees on replicated objects in a simple manner. The ProgressivePromise interface in Netty [7] also generalizes Promises. While it can indicate progress of an operation, a ProgressivePromise does not expose preliminary results of this operation.

3. Correctables

This section presents the Correctables interface for programming and speculating with replicated data. Applications use this interface as a library, as Figure 2 depicts. At the top of this library sits the application-facing API. The library is connected to the storage stack using a storage binding, which is a module that encapsulates all storage

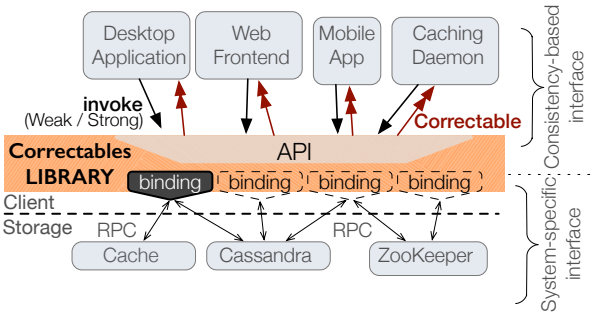

Figure 2: High-level view of Correctables, as an interface to the underlying storage.

system specific interfaces and protocols. Correctables fulfill two critical functions: (i) translate API calls into storage-specific requests via a binding, and (ii) orchestrate responses from the binding and deliver them—in an incremental way—to the application, using *Correctable* objects. Each call to an API method returns a Correctable which represents the progressively improving result (i.e., a result with ICG).

3.1 From Promises to Correctables

As mentioned earlier, Correctables descend from Promises. To model an asynchronous task, a Promise starts in the *blocked* state and transitions to *ready* when the task completes, triggering any callback associated with this state [53]. Promises help with asynchrony, but not incrementality. To convey incrementality, a Correctable starts in the *updating* state, where it remains until the final result becomes available or an error occurs (see Figure 3). When this happens, the Correctable *closes* with that result (or error), transitioning to the *final* (or *error*) state. Upon each state transition, the corresponding callback triggers. Preliminary results trigger

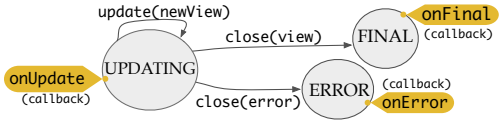

Figure 3: The three states, transitions, and callbacks associated with a Correctable.

a same-state transition (from *updating* to *updating*). A Correctable can have callbacks associated with each of its three states. To attach these callbacks, we provide the `setCallbacks` method; together with `speculate`, these two form the two central methods of a Correctable, which we examine more closely in §4.

3.2 Decoupling Semantics from Implementation

The Correctables abstraction decouples applications from storage specifics by adopting a thin, consistency-based interface, centered around *consistency levels*. This enables developers—who naturally reason in terms of consistency rather than protocol specifics—to obtain simple and portable implementations. With Correctables, applications can transparently switch storage stacks, as long as these stacks support compatible consistency models.

Our API consists of three methods:

1. `invokeWeak`(*operation*),
2. `invokeStrong`(*operation*), and
3. `invoke`(*operation*[, *levels*]).

The first two allow developers to select either weak or strong consistency for a given *operation*. The returned Correctable never transitions from *updating* to *updating* state and only closes with a final value (or error). These two methods follow the traditional practice of providing a single result which lies at one extreme of the consistency/performance trade-off.

The third method provides ICG, allowing developers to operate on this trade-off at run-time, which makes it especially relevant for applications in the above-mentioned gray area. Instead of a single result (as is the case with the two former methods), `invoke` provides incremental updates on the operation result. Optionally, `invoke` accepts as argument the set of consistency levels which the result should—one after the other—satisfy. If this argument is absent, `invoke` provides all available levels. This argument allows some optimizations, e.g., if an application only requires a subset of the available consistency levels, this parameter informs a binding to avoid using the extraneous levels; we omit further discussion of this argument due to space constraints. The available consistency levels depend on the underlying storage system and binding, which we discuss in more detail in §5.

In the next section, we show how to program with Correctables through several representative use-cases. In code snippets we adopt a Python-inspired pseudocode for readability sake. For brevity we leave aside error handling, timeouts, or other features inherited from modern Promises, such as aggregation or monadic-style chaining [12, 31, 53].

```
1 from pylons import app_globals as g # cache access
2 from r2.lib.db import queries       # backend access

4 def user_messages(user, update = False):
5   key = messages_key(user._id)
6   trees = g.permacache.get(key)
7   if not trees or update:
8       trees = user_messages_nocache(user)
9       g.permacache.set(key, trees) # cache coherence
10  return trees
11 def user_messages_nocache(user):
12  # Just like user_messages, but avoiding the cache...
```

Listing 1: Different consistency guarantees in Reddit [13], as an example of tight coupling between applications and storage. Developers must manually handle the cache and the backend.

```
1 def user_messages(user, strong = False):
2   key = messages_key(user._id)
3   # coherence handled by invoke* functions in bindings
4   if strong: return invokeStrong(get(key))
5        else: return invokeWeak(get(key))
```

Listing 2: Reddit code rewritten using Correctables.

4. Correctables in Action

This section presents examples of how Correctables can be useful on two main fronts. (1) Decoupling applications from their storage stacks by providing an abstraction based on consistency levels. (2) Improving application performance by means of ICG, e.g., via speculation or exploiting application-specific semantics.

4.1 Decoupling Applications from Storage

We first discuss a simple case of decoupling, where we illustrate the use the first two functions in our API, namely `invokeWeak` and `invokeStrong`. As discussed in §2, many applications differentiate between weak and strong consistency to balance correctness with performance. In practice, applications often resort to ad-hoc techniques such as cache-bypassing to achieve this, which complicates code and leads to errors [16, 31]. Listing 1 shows code from Reddit [13], a popular bulletin-board system and a prime example of such code. Developers have to explicitly handle cache access (lines L6 and L9), make choices based on presence of items in the cache (L7), manually bypass the cache (L8) under specific conditions, and write duplicate code (L12).

Instead of explicit cache-bypassing, we can employ `invokeWeak` and `invokeStrong` to substantially simplify the code by replacing ad-hoc abstractions like `user_messages` and `user_messages_nocache`, as Listing 2 shows. Furthermore, we can replace other near-identical functions for differentiated guarantees, elimi-

```
invoke(read(...))
  .speculate(speculationFunc[, abortFunc])
  .setCallbacks(onFinal = (res) => deliver(res))
```

Listing 3: Generic speculation with Correctables. The square brackets indicate that `abortFunc` is optional.

nating duplicate logic.[1] Cache-coherence and bypassing is completely handled by the storage-specific binding. This reduces both programmer effort and application-level complexity.

The third method in our library is `invoke`. Correctables are crucial for this method, since it captures ICG. `invoke` allows applications to speculate on preliminary values (hiding the latency of strong consistency), or exploit application-specific semantics, as we show next.

4.2 Speculating with Correctables

Many applications are amenable to speculating on preliminary values to reap performance benefits. To understand how to achieve this, we consider any non-trivial operation in a distributed application which involves reading data from storage. Using `invoke` to access the storage, applications can perform speculation on the preliminary value. If this preliminary value is confirmed by the final value, then speculation was correct, reducing overall latency [71]. Examples where speculation applies include password checking or thumbnail generation (as mentioned in [66]), as well as operations for airline seat reservation [73], or web shopping [43].

Listing 3 depicts how this is performed in practice with Correctables. Even though such speculation can be orchestrated directly by using the `onUpdate` and `onFinal` callbacks of a Correctable object, we provide a convenience method called `speculate` that captures the speculation pattern (L2). It takes a speculation function as an argument, applying it to every new view delivered by the underlying Correctable if this view differs from the previous one. The `speculate` method returns a new Correctable object which closes with the return value of the user-provided speculation function. If the final view matches a preliminary one (which is the common case), the new Correctable can close immediately when the final view becomes available, confirming the speculation. Otherwise, it closes only after the speculation function is (automatically) re-executed with correct input. In the latter case, an optional abort function is executed, undoing potential side-effects of the preceding speculation. Next, we discuss an ad serving system as an example application that can benefit from such speculation.

[1] Similar pairs of ad-hoc functions exist in Reddit for accessing other objects. Perhaps accidentally, these other functions contain comments referring to `user_messages` instead of their specific objects. We interpret this as a strong indication of "copy-pasting" code, which Correctables would help prevent.

```
def fetchAdsByUserId(uid):
  invoke(getPersonalizedAdsRefs(uid))
    .speculate(getAds) # fetch & post−process ads
    .setCallbacks(onFinal = (ads) => deliver(ads))
```

Listing 4: Example of applying speculation in an advertising system to hide latency of strong consistency.

Advertising System. Typically, ads are personalized to user interests. These interests fluctuate frequently, and so ads change accordingly [42]. Given their revenue-based nature, advertising systems have conflicting requirements, as they aim to reconcile consistency (freshness of ads) with performance (latency) [24, 26]. We thus find that they correspond to our notion of gray area, and are a suitable speculation use-case.

Listing 4 shows how we can use ICG while fetching ads. First, we obtain a list of *references* to personalized ads using the `invoke` method (L2). This method returns both a preliminary view (with weak guarantees) and a final (fresh) view. Using the references in the preliminary view, we fetch the actual ads content and media, and do any post-processing, such as localization or personalization (L3). If the final view corresponds to the preliminary, then speculation was correct, and we can deliver (L4) the ads fast; otherwise, `getAds` re-executes on the final view, and we deliver the result later. We use this application as our first experimental case-study (§6.3.2).

The pattern of fetching objects based on their references—which themselves need to be fetched first—is widespread. It appears in many applications, such as reading the latest news, the most recent transactions, the latest updates in a social network, an inventory, the most pressing items in a to-do list or calendar, and so on. In all these cases, the application needs to chase a pointer (reference) to the latest data, while weak consistency can reveal stale values, which is undesirable. We avoid stale data by reading the references with `invoke`, and we mask the latency of the final value by speculatively fetching objects based on the preliminary reference.

4.3 Exploiting Application Semantics

Applications can exploit their specific semantics to leverage the preliminary and the final values of `invoke`. For instance, consider the web auction system mentioned by Kraska et al. [43], where strong consistency is critical in the last moments of a bid, but is not particularly helpful in the days before the bid ends, when contention is very low and anomalous behavior is unlikely. Another example is selling items from a predefined stock of such items. If a preliminary response suggests that the stock is still big, it is safe to proceed with a purchase. Otherwise, if the stock is almost empty, it would be better to wait for the arrival of the final response. This is the case, for instance, for a system selling tickets to an event, which we describe next.

```
def purchaseTicket(eventID):
  done = false
  invoke(dequeue(eventID)).setCallbacks(
    onUpdate = (weakResult) =>
      if weakResult.ticketNr > THRESHOLD:
        done = true # many tickets left, so we can buy
        confirmPurchase()
    onFinal = (strongResult) =>
      if not done and strongResult is not null:
        confirmPurchase() # we managed to get a ticket
      else: display("Sold out. Sorry!"))
```

Listing 5: Dynamic selection of consistency guarantees in a ticket selling system. If there are many tickets in the stock, we can safely use weak consistency.

Selling Tickets for Events. For this application system, we depart from the popular key-value data type. First, as we want to avoid overselling, we need a stronger abstraction to serialize access to the ticket stock. Simple read/write objects (without transactional support) are fundamentally insufficient [37]. Second, we want to demonstrate the applicability of ICG to other data types. We thus model the ticket stock using a queue, which is a simple object, yet powerful enough to avoid overselling.

Event organizers enqueue tickets and retailers dequeue them. This data type allows us to serialize access to the shared ticket stock [15, 43]. We assume, however, that tickets bear no specific ordering (i.e., there is no seating). Clients are interested in purchasing *some* ticket, and it is irrelevant which exact element of the queue is dequeued. We can thus resort to weak consistency most of the time, and use strong consistency sparingly. We consider a weakly consistent result of an operation to be the outcome of simulating that operation on the local state of a single replica (see §5.2).

Listing 5 shows how we can selectively use strong consistency in this case, based on the estimated stock size. For each purchase, retailers use `invoke` with the dequeue operation. This yields a quick preliminary response, by peeking at the queue tail on the closest replica of the queue. If the preliminary value indicates that there are many tickets left (e.g., via a ticket sequence number, denoting the ticket's position in the queue), which is the common case, the purchase can succeed without synchronous coordination on dequeue, which completes in the background. This reduces the latency of most purchase operations. As the queue drains, e.g. below a predefined threshold of 20 tickets, retailers start waiting for the final results, which gives atomic semantics on dequeuing, but incurs higher latency. This system represents our second experimental case study (§6.3.2).

4.4 Exposing Data Incrementally

In some cases, it is beneficial to expose even incorrect (stale) data to the user if this data arrives fast, and amend the output as more fresh data becomes available. Indeed, a quick approximate result is sometimes better than an overdue reply [28, 66]. Many applications update their output as better results become available. A notable example is flight search aggregators [9], or generally, applications which exhibit high responsiveness by leaking to the user intermediary views on an ongoing operation [47, 49], e.g., previews to a video or shipment tracking. We can assist the development of this type of applications, as we describe next.

```
invoke(getLatestNews()).setCallbacks(
    onUpdate = (items) => refreshDisplay(items))
```

Listing 6: Progressive display of news items using Correctables. The `refreshDisplay` function triggers with every update on the news items.

Smartphone News Reader. Consider a smartphone news reader application for a news service replicated with a primary-backup scheme [66]. Additionally, recently seen news items are stored in a local phone cache. With ICG provided by Correctables, the application can be oblivious to storage details. It can use a single logical storage access to fetch the latest news items, as Listing 6 shows. The binding would translate this logical access to three actual requests: one to the local cache, resolving almost immediately, one to the closest backup replica, providing a fresher view, and one to a more distant primary replica, taking the longest to return but providing the most up-to-date news stories.

4.5 Discussion: Applicability of ICG

In a majority of use-cases, we observe that two views suffice. Correctables, however, support arbitrarily many views. Note that this does not add any complexity to the interface and can be useful, as the news reader application shows.

There are other examples of applications which can benefit from multiple views. A notable use-case are blockchain-based applications (e.g., Bitcoin [58]), where Correctables can track transaction confirmations as they accumulate and eventually the transaction becomes an irrevocable part of the blockchain, i.e., strongly-consistent with high probability. This is a use-case we also implemented, but omit for space constraints. In larger quorum systems (e.g., BFT), Correctables can represent the majority vote as it settles. Search or recommenders, likewise, can benefit from exposing multiple intermediary results in subsequent updates.[2]

Intuitively, multiple preliminary views are helpful for applications requiring live updates. On the one hand, several preliminary values would make the application more interactive and offer users a finer sense of progress. This is especially important when the final result has high latency (Bitcoin transactions take tens of minutes). On the other hand, as the replicated system delivers more

[2] We are grateful to our anonymous OSDI reviewers for this particularly constructive idea.

preliminary views for an operation, less operations can be sustained and overall throughput drops. Thus, applications which build on ICG with multiple incremental views observe a trade-off between interactivity and throughput. This trade-off can be observed even when the system delivers only two views (§6.2.1).

In order to be practical, the cost of generating and exploiting the preliminary values of ICG must not outweigh their benefits. The cost of generating ICG is captured in the trade-off we highlighted above; the cost of exploiting ICG is highly application-dependent. If used for speculation, the utility of 2+ views depends on how expensive it is to re-do the speculative work upon misspeculation. This can range from negligible (simply display preliminary views) to potentially very expensive (prefetch bulky data). Additionally, the utility also depends on how often misspeculation actually occurs. This depends on the workload characteristics: workloads with higher write ratios elicit higher rates of inconsistencies, and thus more misspeculations (§6.2.1–Divergence).

There are also cases when using ICG is not an option. This is either due to the underlying storage providing a unique consistency model and lacking caches, or due to application semantics, which can render ICG unnecessary—we give examples of this in the first two rows of Table 1. Correctables, however, are beneficial beyond ICG. This abstraction can hide the complexity of dealing with storage-specific protocols, e.g., quorum-size selection. The application code thus becomes portable across different storage systems.

5. Bindings

Our library handles all the instrumentation around Correctable objects. This includes creation, state transitions, callbacks, and the API inherited from Promises [12, 31]. Bindings are storage-specific modules which the library uses to communicate with the storage. These modules encapsulate everything that is storage system specific, and thus draw the separating line between consistency models—which Correctables expose—and implementations of these models. In this section, we describe the binding API, and show how bindings can facilitate efficient implementation of ICG with server-side support.

5.1 Binding API

An instance of our library always uses one specific binding. A binding establishes: (1) the concrete configuration of the underlying storage stack (e.g., Memcache on top of Cassandra) together with (2) the *consistency levels* offered by this stack, and (3) the implementation of any storage specific protocol (e.g., for coherence, choosing quorums). This allows the library to act as a client to the storage stack.

When an application calls an API method (§3.2), the library immediately returns a Correctable. In the background, we use the *binding API* to access the underlying storage. The binding forwards responses from the storage through an upcall to the library. The library then updates (or closes) the associated Correctable, executing the corresponding callback function.

The binding API exposes two methods to the library. First, `consistencyLevels()` advertises to the library the supported consistency levels. It simply returns a list of supported consistency levels, ordered from weakest to strongest. In most implementations, this will probably be a one-liner returning a statically defined list. The second function is `submitOperation(op, consLevels, callback)`. The library uses this function to execute operation `op` on the underlying storage, with `consLevels` specifying the requested consistency levels. The `callback` activates whenever a new view of the result is available. The binding has to implement the protocol for executing `op` and invoke `callback` once for each requested consistency level.

Listing 7 shows the implementation of a simple binding for a primary-backup storage, supporting two consistency levels. A more sophisticated binding could access the backup and primary in parallel, or could provide more than two consistency levels. We designed the binding API to be as simple as possible; contributors or developers wishing to support a particular store must implement this API when adding new bindings. We currently provide bindings to Cassandra and ZooKeeper.

5.2 Efficiency and Server-side Support

On a first glance, ICG might seem to evoke large bandwidth and computation overheads. Indeed, if the `invoke` method comprises multiple independent single-consistency requests, then storage servers will partly redo their own work. Also, as the weakly and strongly consistent values often coincide, multiple responses are frequently redundant. Such overheads would reduce the practicality of ICG.

```
1  def consistencyLevels():
2    return [WEAK, STRONG]

4  def submitOperation(operation, consLevels, callback):
5    if WEAK in consLevels:
6      backupResult = queryClosestBackup(operation)
7      callback(backupResult, WEAK)
8    if STRONG in consLevels:
9      primaryResult = queryPrimary(operation)
10     callback(primaryResult, STRONG)
```

Listing 7: Simple binding to a storage system with primary-backup replication.

With server-side support, however, we can minimize these overheads. For instance, we can send a *single* request to obtain all the incremental views on a replicated object. An effective way to do this is to hook into the coordination mechanism of consistency protocols. This mechanism is the core of such protocols, and the provided consistency model and latency depend on the type of coordination. For example, asynchronous (off the critical path) coordination ensures eventually consistent results with low-latency [28]. Coordination through an agreement protocol, as in Paxos [46], yields linearizability [38], but with a higher latency.

Our basic insight is that we can get a good guess of the result already before coordinating, based on a replica's local state. In fact, this same state is being exposed when asynchronous coordination is employed, and as we alreay mentioned, this state is consistent on expectation. The replica can leak a preliminary response—with weak guarantees—to the client prior to coordination (Figure 4). Moreover, we can reduce bandwidth overhead by skipping the final response if it is the same as the preliminary: a small *confirmation* message suffices, to indicate that the preliminary response was correct. Indeed, with such an optimization, ICG has minor bandwidth overhead (§6.2.1).

An additional benefit from this approach compared to sending two independent requests is that it prevents certain types of unexpected outcomes. For instance, strong consistency might be more stale than weak consistency if responses to two independent requests were reordered by the WAN [66]. Using this approach, we modify two popular systems—Cassandra and ZooKeeper—to provide efficient support for ICG. Other techniques (e.g., master leases [23]) or replication schemes (e.g., primary-backup) can provide final views fast, skipping the preliminary altogether.

Cassandra. Cassandra uses a quorum-gathering protocol for coordination [32]. In our modified version of Cassandra—called Correctable Cassandra (CC)—the coordinating node sends a preliminary view after obtaining the *first* result from any replica. This view has low latency, obtained either locally (if the coordinator is itself a replica) or from the closest replica. Our binding

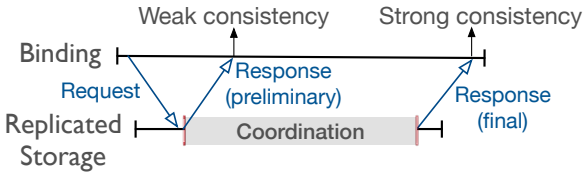

Figure 4: Simple server support for efficient ICG. The storage system sends a preliminary response before coordinating. Note that for a single request, the storage provides two responses.

to CC supports two consistency levels, *weak* (involving one replica) and *strong* (involving two or more). To minimize bandwidth overhead of `invoke`, CC uses the confirmation messages optimization we mentioned earlier.

ZooKeeper. To demonstrate the versatility of Correctables, we consider a different data type, namely replicated queues, which ZooKeeper can easily model [11]. Our binding supports operations `enqueue` and `dequeue`, with weak and strong consistency semantics, accessible via `invokeWeak` and `invokeStrong`, respectively; `invoke` supplies both consistency models incrementally.

The vanilla ZooKeeper implementation (ZK) has strong consistency [39]. For efficient ICG, we implement Correctable ZooKeeper (CZK) by adding a fast path to ZK: a replica first simulates the operation on its local state, returning the preliminary (weak) result. After coordination (via the Zab protocol [40]), this replica applies the operation and returns the strong response.

Causal Consistency and Caching. We also implement a binding to abstract over a causally consistent store complemented by a client-side cache. The `invoke` function reveals two views: one from cache (very fast, possibly stale), and another from the causally consistent store. This binding ensures write-through cache coherence, allows cache-bypassing (`invokeStrong`) or direct cache access (`invokeWeak`), e.g., in case of disconnected operations for mobile applications [62]. Given the space constraints we focus on the two other bindings.

6. Evaluation

Our evaluation focuses on quantifying the benefits of ICG. Before diving into it, it is important to note that any potential benefit of ICG is capped by performance gaps among consistency models. Briefly, if strong consistency has the same performance as weaker models (or the difference is negligible) then applications can directly use the stronger model. This is, however, rarely the case. In practice, there can be sizable differences—up to orders of magnitude—across models [17, 66].

We first describe our evaluation methodology, and then show that such optimization potential indeed exists. We do so by looking at the performance gaps between weak and strong consistency in quorum-based (Cassandra) and consensus-based (ZooKeeper) systems. We then quantify the performance gain of using ICG in three case studies: a Twissandra-based microblogging service [10], an ad serving system, and a ticket selling application.

6.1 Methodology

We run all experiments on Amazon's EC2 with m4.large instances and a replication factor of 3, with replicas distributed in Frankfurt (FRK), Ireland (IRL), and N. Virginia (VRG). Unless stated otherwise, to obtain WAN conditions, the client is in IRL and uses the replica in

FRK; note that colocating the client with its contact server (i.e., both in IRL) would play to our advantage, as it would reduce the latency of preliminary responses and allow a bigger performance gap. We also experiment with various other client locations in some experiments.

For Cassandra experiments, we compare the baseline Cassandra v2.1.10 (labeled C), with our modified Correctable Cassandra (CC). We use superscript notation to indicate the specific quorum size for an execution, e.g., C^1 denotes a client reading from Cassandra with a read quorum $R = 1$ (i.e., involving 1 out of 3 replicas). For the ZooKeeper queue, we compare our modified Correctable ZooKeeper (CZK) against vanilla ZooKeeper (ZK), v3.4.8. The cumulative implementation effort associated with CC and CZK, including three case studies, is modest, at roughly $3k$ lines of Java code.

6.2 Potential for Exploiting ICG

To determine the potential of ICG, we examine their behavior in practice. Studies show that large load on a system and high inter-replica latencies give rise to large performance gaps among consistency models [17, 66]. To the best of our knowledge, however, there are no studies which consider a combination of incremental consistency models in a single operation. We first investigate this behavior in Cassandra and then in ZooKeeper.

6.2.1 Potential for Exploiting ICG in Cassandra

Cassandra can offer us insights into the basic behavior of ICG in a quorum system. As explained in §5, CC offers two consistency models: weak, which yields the *preliminary* view ($R = 1$), and strong, giving the *final* view ($R = 2$ or $R = 3$, depending on the requested quorum size). For write operations, we set $W = 1$. We use microbenchmarks and YCSB [25] to measure single-request latency and performance under load, respectively. For each CC experiment, we run three 60-second trials and elide from the results the first and last 15 seconds. We report on the average and 99th percentile latency, omitting error bars if negligible.

Single-request Latency. We use a microbenchmark consisting of read-only operations on objects of $100B$. We are interested in the performance gap between preliminary and final views as provided by ICG, and we contrast these with their vanilla counterparts. We thus compare CC^2 ($R \in \{1,2\}$) and CC^3 ($R \in \{1,3\}$) with C^1 ($R = 1$), C^2 ($R = 2$), and C^3 ($R = 3$). For CC, R has two values: the read quorum size for the preliminary (weak) and for the final (strong) replies, respectively.

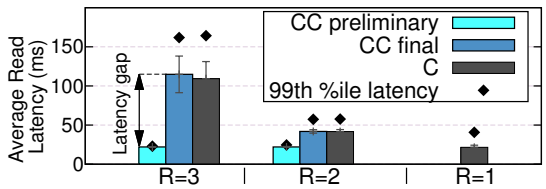

Figure 5: Single-request latencies in Cassandra for different quorum configurations. A bigger latency gap means a larger time window available for speculation.

Figure 5 shows the results for all these configurations, grouped by their read quorum size. The average latency of preliminary views—whether it is for CC^2 or CC^3—follows closely the latency of C^1, which coincides with the $20ms$ RTT between the client and the coordinator. Preliminary views reflect the local state on the replica in FRK, having the same consistency as C^1. Final views of CC^2 and CC^3 follow the trend of the requested quorum size and reflect the behavior of C^2 and C^3 respectively.

The performance gap between the preliminary and final view for CC^2 is $20ms$. The coordinator (FRK) is gathering a quorum of two: itself and the closest replica (IRL). The gap indeed corresponds to the RTT between these two regions. For CC^3, the gap is much larger: up to $140ms$ for the 99th percentile, due to the larger distance to reach the third replica (VRG). By speculating on the preliminary views, applications can hide up to $20ms$ (or $140ms$) of the latency for stronger consistency. In practice, such differences already impact revenue, as users are highly-sensitive to latency fluctuations [28, 35].

Performance Under Load. We also study the performance gap using YCSB workloads A (50:50 read/write ratio), B (95:5 read/write ratio), and C (read-only) [25]. To stress the systems and obtain WAN conditions, we deploy 3 clients, one per region, with each client connecting to a remote replica. For brevity, we only report on the results for the client in IRL and $R = \{1,2\}$. Figure 6 presents the average latency as a function of throughput. We plot the evolution of both the preliminary and final views individually.

We observe that CC trades in some throughput due to the load generated on the coordinator, which handles ICG. We observe this behavior in all three workloads. This is to be expected, considering the modifications necessary to implement preliminary replies (§5.2). Briefly, we add another step to every read operation that uses quorums larger than one. This step, called *preliminary flushing*, occurs at any coordinator replica serving read operations as soon as that replica finishes reading the requested data from its local storage—and prior to gathering a quorum from other replicas. This step generates additional load on the coordinator replica, explaining the throughput drop of CC^2 compared to baselines. Related work on replicated state machines (RSM) suggests an optimization [71] which resembles our flushing technique. Perhaps unsurprisingly, the optimized RSM exhibits a similar throughput drop [71, §6.2] as we notice in these experiments.

The latency gap between preliminary and final views

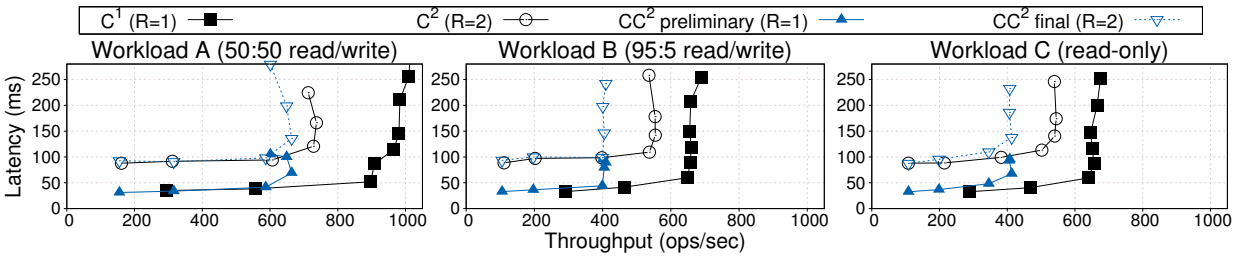

Figure 6: Performance of Correctable Cassandra (CC) compared to baseline Cassandra (C). Note that the measurements for CC^2 have two results, one for the preliminary view and another for final. These two have the same throughput but different latencies.

is the same as the one we observe in the microbenchmarks. To conclude, our results confirm that the performance gaps while using ICG are noticeable, and hence there is room for hiding latency.

Divergence. To obtain more insight about the behavior of ICG, we use CC and the YCSB benchmark to measure how often preliminary values diverge from final results. We achieve this by using `invoke` and comparing the preliminary view to the final one. We run this experiment with a small dataset of $1K$ objects. We aim at obtaining the conditions of a highly-loaded system where clients are mostly interested in a small (popular) part of the dataset.

Figure 7 shows our result for a mix of representative YCSB workloads (A and B) and access patterns (Zipfian and Latest) with default settings. Notably, workload A (50:50 read/write) under Latest distribution (read activity skewed towards recently updated items) exhibits high divergence, up to 25%. Under such conditions, using $R = 1$ would yield many stale results. Indeed, some applications with high write ratios, e.g., notification or session stores [25, 34], tend to use $R = 2$, even though this forces *all* read operations to pay the latency price [19].

In fact, even if less than 1% of accessed objects are inconsistent, these are typically the most popular ("linchpin" [16, 60]) objects, being both read- and write-intensive. Such anomalies have a disproportionate effect at application-level, since they reflect in many more than 1% application-level operations. Applications with high update ratios as modeled by workload A, e.g., social networks [24], can thus benefit from exploiting ICG to avoid anomalies.

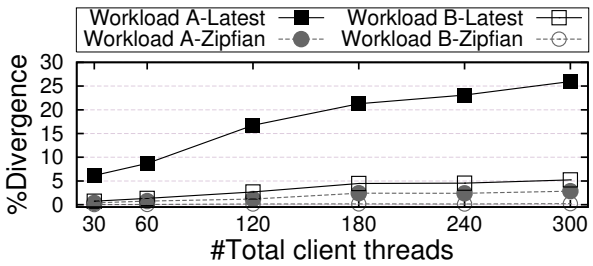

Figure 7: Divergence of preliminary from final (correct) views in Correctable Cassandra with various YCSB configurations.

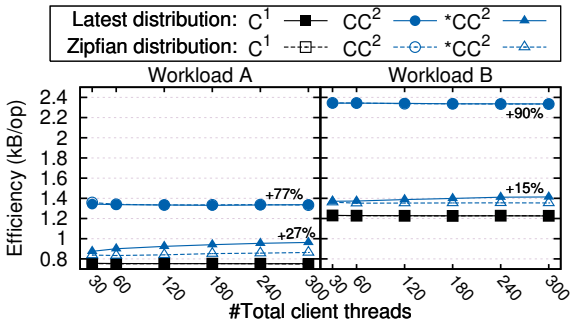

Figure 8: Efficiency (bandwidth overhead) of the ICG implementation in Correctable Cassandra (CC).

Bandwidth Overhead. In addition to the throughput drop mentioned above, client-replica bandwidth is the next relevant metric which ICG can impact. Yet, optimizations can cut the cost of this feature (§5.2). We implement such an optimization in CC, whereby a final view contains only a small confirmation—instead of the full response—if it coincides with the preliminary view. We note that in all experiments thus far we did not rely on this optimization, which makes our comparisons with Cassandra conservative.

To obtain a worst-case characterization of the costs of ICG, we consider the scenario where divergence can be maximal, as this will lessen the amount of bandwidth we can save with our optimization. Hence, we consider the exact conditions we use in the divergence benchmark, where we discovered that divergence can rise up to 25%. In this experiment, we measure the average data transferred (KB) per operation. We contrast three scenarios. First, as baseline, we use C^1, where clients request a single consistency version using weak reads. The other two systems are CC^2 (without optimization) and $^*CC^2$ (optimized to reduce bandwidth overhead).

Figure 8 shows our results. As expected, if divergence is very high—notably in workload A—then many preliminary results are incorrect. This means that final views cannot be replaced by confirmations, increasing the data cost by up to 27%. Without any optimization, this would drive the cost up by 77%. Workload B has a smaller write ratio (5%), so a lower divergence and more optimization potential: we can reduce the overhead from 90% down

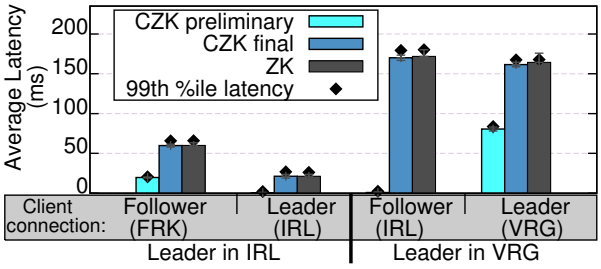

Figure 9: Latency gaps between preliminary and final views on the result of dequeue operations in Correctable ZooKeeper (CZK) compared to ZooKeeper (ZK). Client is in IRL.

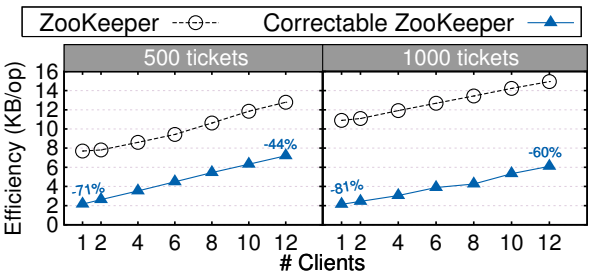

Figure 10: Efficiency (bandwidth overhead) for dequeuing operation in Correctable ZooKeeper (CZK) and ZooKeeper (ZK). Overhead in CZK is independent of queue size.

to 15% (since most final views are confirmations).

Our experiments prove that ICG have a modest cost in terms of data usage. This cost can be further reduced through additional techniques (§5.2). We remark that our choice of baseline, C^1, is conservative, because CC^2 offers better guarantees than C^1. A different baseline would be a system where clients *send two requests*—one for $R = 1$ and one for $R = 2$—and *receive two replies*. While such a baseline offers the same properties as CC^2, it would involve bigger data consumption, putting our system at an advantage.

6.2.2 Potential for Exploiting ICG in ZooKeeper

Latency Gaps. We also measure performance gaps in ZooKeeper queues for various locations of the leader and the replica which the client (in IRL) connects to. We show the results for four representative configurations for adding elements to a queue (we discuss dequeuing in the context of a ticket selling system in §6.3.2). The elements are small, containing an identifier of up to 20*B* (e.g., ticket number). Figure 9 shows the latency gaps when we use ICG in Correctable ZooKeeper (CZK) compared to baseline ZooKeeper (ZK).

In all cases, the latency of the preliminary view (containing the name of the assigned znode) corresponds to the RTT between the client and the contacted replica. This latency ranges from 2*ms* (when client and replica are both in IRL), through 20*ms* (the RTT from IRL to FRK), up to 83*ms* (the RTT between IRL and VRG). The most appealing part of this result is perhaps the substantial gap which appears when the client and the closest follower are in IRL and the leader is distant (in VRG), in the third group of results in Figure 9.

Bandwidth Overhead. Storing big chunks of data is not ZooKeeper's main goal. The client-server bandwidth is usually not dominated by the payload, reducing the benefits of the confirmation optimization. For enqueuing, the bandwidth cost thus increases by roughly 50%, from 270 to 400 bytes/operation. As expected, this corresponds to one additional (preliminary) response message in addition to the original request and (final) response.

While queues are a common ZooKeeper use-case, a problem appears in standard dequeue implementations due to message size inflation [3]. Specifically, clients first read the *whole queue* and then try to remove the tail element. To evade this problem in CZK, clients only read the constant-sized tail relevant for dequeuing. Figure 10 compares the bandwidth cost per dequeue operation in CZK and ZK for different queue sizes as we increase the number of contending threads. While the cost still increases with contention in both cases, in CZK we make it independent of queue size, which is not the case for ZK. As future work, we plan to make the dequeue cost also independent of contention using tombstones [63].

6.3 Case Studies for Exploiting ICG

Given the optimization potential explored so far, we now investigate how to exploit it in the context of three applications: the Twissandra microblogging service [10], an ad serving system, and a ticket selling system. The first two build on CC and use speculation. The last application uses CZK queues.

6.3.1 Speculation Case Studies

For Twissandra, we are interested in `get_timeline` operation, since this is a central operation and is amenable to optimization through speculation. This operation proceeds in two-steps: (1) fetch the timeline (tweet IDs), and then (2) fetch each tweet by its ID. We re-implement this function to use `invoke` on step (1) and leverage the preliminary timeline view to speculatively execute step (2) by prefetching the tweets. If the final timeline corresponds to the preliminary, then the prefetch was successful and we can reduce the total latency of the operation. In case the final timeline view is different, we fetch the tweets again based on their IDs from this final view.

Our second speculation case study is the ad serving system we describe in §4.2. The goal is to reduce the total latency of `fetchAdsByUserId` operation without sacrificing consistency, so we exploit ICG by speculating on preliminary values (Listing 4).

For both systems, we adapt their respective operations to use `invoke` ($R = \{1,2\}$) and plug them in the YCSB framework. We compare these operations using a baseline that uses only the strongly consistent result ($R = 2$),

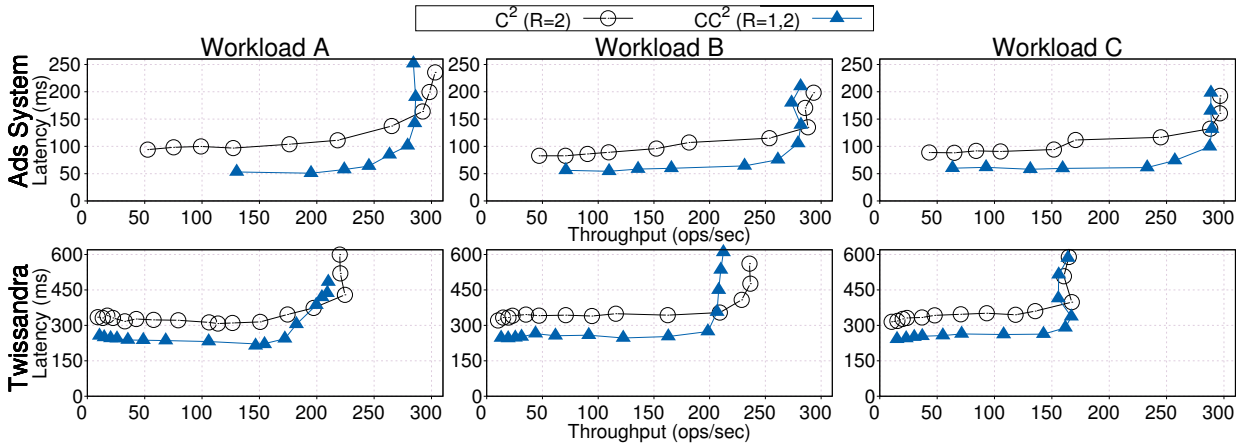

Figure 11: Using speculation via ICG to improve latency in the advertising system and in Twissandra (`get_timeline` operation). Correctable Cassandra (CC) improves latency by up to 40% in exchange for a throughput drop of 6%.

and does not leverage speculation. For Twissandra we use a corpus of 65k tweets [4] spread over 22k user timelines; the ad serving system uses a dataset of 100k user-profiles and 230k ads, where each profile references between 1 and 40 random ads.

The results are in Figure 11. In contrast to our other experiments, we deploy Twissandra replicas in Virginia, N. California, and Oregon EC2 regions. The goal is to see how performance gains vary based on deployment scenario. The ads system uses the same configuration as before. The client is in IRL for both experiments.

We first explain the results for the ads system. As can be seen, these are consistent with our earlier findings from Cassandra experiments (Figure 6). We trade throughput for better latency. Prior to saturation, we can serve ads with an average latency of 60ms. In the same conditions, the baseline achieves 100ms average latency (improvement by 40%). In turn, the throughput drop is most noticeable in workload A, by 18ops/sec (reduced by 6%). The smaller throughput drop compared to the raw results of Figure 6 is explained by the fact that each `fetchAdsByUserId` entails two storage accesses. Only the first access, however, uses ICG (to speculate). The second storage access is hidden inside `getAds` (Listing 4, L3); this is a read with $R = 2$, incurring no extra cost.

For Twissandra, we observe a lower throughput and higher latency, as the client is farther from the coordinator and replicas are also more distant from each other. But otherwise we draw similar conclusions. Notably, across both of these case-studies, divergence was consistently under 1%, so the applications encountered very few misspeculations.

6.3.2 Selling Tickets to Events

A second notable use-case of ICG is exploiting application semantics, as we discuss in the ticket selling system from §4.3 (see Listing 5). Here we exploit the fact that

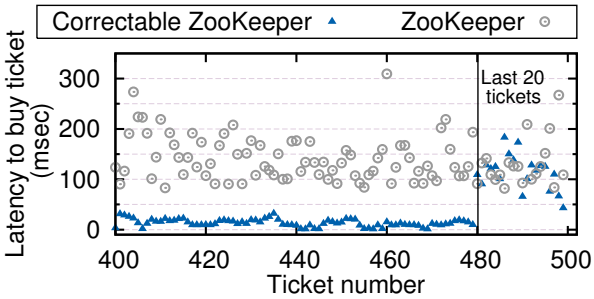

Figure 12: Selling tickets with ZK and CZK. The last 20 tickets incur high latency due to strong consistency.

the position of a ticket in the queue is irrelevant. Thus, in the common case, we can rely on the preliminary value. Strong consistency (atomicity), however, becomes critical when ticket retailers are contending over the last few remaining tickets. Using ICG, we can switch dynamically between using the preliminary or the final results when the stock becomes low, to avoid overselling.

We consider 4 retailers concurrently serving (dequeuing) tickets from a fixed-size stock of 500 tickets. Retailers are colocated with a CZK follower in FRK, the leader being in IRL. We wait for the final (atomic, equivalent to ZK) response for the last 20 tickets, otherwise we use the preliminary one. This is a conservative bound; in our experiments, only the last two tickets were "revoked" by the final view on average, with a maximum of six.

Figure 12 shows individual ticket purchase latencies, averaged over five runs, compared to latencies with vanilla CZK. As long as there are more than 20 tickets left, we reduce the purchase latency substantially. The high variability of final view latencies is caused by contention between the retailers, which does not affect preliminary views. We experiment also with larger ticket stocks (1000), but the queue length has no practical ef-

fect on latencies. To support more contention (more retailers) in practice, such a ticketing service can scale-out. For instance, we can shard the ticket stock and instantiate multiple replicated CZK services, each of them serving a partition of the overall stock, ensuring scalability [22].

7. Conclusions

We have presented Correctables, an abstraction for programming with replicated objects. The contribution of Correctables is twofold. First, they *decouple* an application from its underlying storage stack by drawing a clear boundary between consistency guarantees and the various methods of achieving them. This reduces developer effort and allows for simpler and more portable code.

Second, Correctables provide *incremental consistency guarantees* (ICG), which allow to compose multiple consistency levels within a single operation. With this type of guarantees we aim to fill a gap in the consistency/performance trade-off. Namely, applications can make last-minute decisions about what consistency level to use in an operation while this operation is executing. This opens the door to new optimizations based on speculation or on concrete, application-specific semantics.

We evaluated the performance and overhead of ICG, as well as the impact of this novel type of guarantees on three practical systems: (1) a microblogging service and (2) an ad serving system, both backed by Cassandra, and (3) a ticket selling system based on ZooKeeper queues. We modified both Cassandra and ZooKeeper to support ICG with little overhead. We showed how ICG provided by Correctables bring substantial latency decrease for the price of small bandwidth overhead and throughput drop.

We believe that Correctables provide a new way to structure the interaction between applications and their storage by exploiting incrementality, and hence a new way to build distributed applications.

Acknowledgements

We thank our shepherd, Timothy Roscoe, and the anonymous OSDI reviewers for their thoughtful comments which greatly improved the quality of our paper. We are also grateful to our colleagues from the Distributed Programming Laboratory (LPD) for putting up with our recurring requests for feedback, and for the insightful discussions we had along the way with Martin Odersky (who also suggested us the name *Correctables*), Edouard Bugnion, Willy Zwaenepoel, John Wilkes, Aleksandar Dragojević, Julia Proskurnia, Vlad Ureche, and Jad Hamza. A special thanks goes to Kenji Relut for his help with ZooKeeper. This work has been supported in part by the European ERC Grant 339539 - AOC and the Swiss FNS grant 20021_147067.

References

[1] Amazon SimpleDB. https://aws.amazon.com/simpledb/.

[2] C++ Futures at Instagram. http://instagram-engineering.tumblr.com/post/121930298932/c-futures-at-instagram.

[3] Distributed queue. netflix/curator. https://github.com/Netflix/curator/wiki/Distributed-Queue.

[4] Followthehashtag / 170,000 Apple tweets. http://followthehashtag.com/datasets/170000-apple-tweets-free-twitter-dataset/.

[5] Google appengine. https://appengine.google.com/.

[6] google/guava wiki: ListenableFutureExplained. https://github.com/google/guava/wiki/ListenableFutureExplained.

[7] ProgressivePromise (Netty 4.0 API). http://netty.io/4.0/api/io/netty/util/concurrent/ProgressivePromise.html.

[8] Riak KV, distributed NoSQL database. http://basho.com/products/riak-kv/.

[9] Skyscanner. http://www.skyscanner.ch.

[10] Twissandra. https://github.com/twissandra/twissandra/.

[11] ZooKeeper Recipes and Solutions. http://tiny.cc/zkqueues.

[12] Futures for C++11 at Facebook, 2015. https://code.facebook.com/posts/1661982097368498.

[13] reddit/r2/r2/lib/comment_tree.py:308, Accessed March, 2016. Source: https://github.com/reddit/reddit.

[14] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, (2), 2012.

[15] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[16] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *HotOS XV*, 2015.

[17] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *VLDB*, 7(3), 2013.

[18] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *SIGMOD*, 2015.

[19] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *VLDB*, 5(8), 2012.

[20] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[21] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2), 2012.

[22] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[23] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC*, 2007.

[24] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *VLDB*, 1(2), 2008.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3), 2013.

[27] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2), 2013.

[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo. In *SOSP*, 2007.

[29] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *SOSP*, 2015.

[30] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *SoCC*, 2014.

[31] M. Eriksen. Your server as a function. In *PLOS*, 2013.

[32] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.

[33] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[34] C. Hale and R. Kennedy. Using Riak at Yammer, March 2011. http://dl.dropbox.com/u/2744222/2011-03-22_Riak-At-Yammer.pdf.

[35] J. Hamilton. The cost of latency. http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/, 2009.

[36] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[37] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 1991.

[38] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[39] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[40] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[41] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *OSDI*, 2012.

[42] Y. Koren. Collaborative filtering with temporal dynamics. *CACM*, 53(4), 2010.

[43] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *VLDB*, 2(1), 2009.

[44] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data Center Consistency. In *EuroSys*, 2013.

[45] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.

[46] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2), 1998.

[47] J. R. Lange, P. A. Dinda, and S. Rossoff. Experiences with Client-based Speculative Remote Display. In *USENIX ATC*, 2008.

[48] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *SOSP*, 2015.

[49] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *MobiSys*, 2015.

[50] C. Li, J. Leitao, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX ATC*, 2014.

[51] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[52] D. Li, J. Mickens, S. Nath, and L. Ravindranath. Domino: Understanding Wide-Area, Asynchronous Event Causality in Web Applications. In *SoCC*, 2015.

[53] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, 1988.

[54] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, 2013.

[55] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *SOSP*, 2015.

[56] E. Meijer. Your mouse is a database. *CACM*, 55(5), 2012.

[57] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *NSDI*, 2010.

[58] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system.

[59] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *SOSP*, 2005.

[60] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *NSDI*, 2013.

[61] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. PLANET: Making Progress with Commit Processing in Unpredictable Environments. In *SIGMOD*, 2014.

[62] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu. Simba: Tunable End-to-end Data Consistency for Mobile Apps. In *EuroSys*, 2015.

[63] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1), 2005.

[64] P. Schulle. Manhattan, distributed database for Twitter scale. http://tiny.cc/twitmanhattan, 2014.

[65] M. Schwarzkopf. *Operating system support for warehouse-scale computing*. PhD thesis, University of Cambridge Computer Laboratory, 2015.

[66] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[67] D. B. Terry, M. M. Theimer, K. Petersen, a. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5), 1995.

[68] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[69] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In *CIDR*, 2011.

[70] B. Wester, P. M. Chen, and J. Flinn. Operating System Support for Application-Specific Speculation. In *EuroSys*, 2011.

[71] B. Wester, J. A. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, 2009.

[72] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining ACID and BASE in a distributed database. In *OSDI*, 2014.

[73] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, 2000.

[74] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *SOSP*, 2015.