Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications

João Ferreira Loff
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Daniel Porto
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

João Garcia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Jonathan Mace
Max Planck Institute for Software Systems and Microsoft Research

Rodrigo Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Abstract

Modern internet-scale applications suffer from *cross-service inconsistencies*, arising because applications combine multiple independent and mutually-oblivious datastores. The end-to-end execution flow of each user request spans many different services and datastores along the way, implicitly establishing ordering dependencies among operations at different datastores. Readers should observe this ordering and, in today's systems, they do not.

In this work, we present Antipode, a bolt-on technique for preventing cross-service consistency violations in distributed applications. It enforces cross-service consistency by propagating *lineages* of datastore operations both alongside end-to-end requests and within datastores. Antipode enables a novel *cross-service causal consistency* model, which extends existing causality models, and whose enforcement requires us to bring in a series of technical contributions to address fundamental semantic, scalability, and deployment challenges. We implemented Antipode as an application-level library, which can easily be integrated into existing applications with minimal effort, is incrementally deployable, and does not require global knowledge of all datastore operations. We apply Antipode to eight open-source and public cloud datastores and two microservice benchmark applications. Our evaluation demonstrates that Antipode is able to prevent cross-service inconsistencies with limited programming effort and less than 2% impact on end-user latency and throughput.

ACM Reference Format:
João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. 2023. Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3600006.3613176

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613176

1 Introduction

Modern internet-scale applications, such as social networks, online forums and e-commerce sites, are global-scale, decentralized distributed systems that comprise many different services and back-end datastores. In these systems, the end-to-end request flow of end-user interactions is complex, spanning multiple different services and machines and involving interactions with multiple different datastores [2, 23, 24, 48, 58, 65]. Popular design patterns like microservice architectures further reinforce this complexity: services are loosely coupled, each implements a small slice of application logic, and each makes independent choices of datastores and consistency model [34, 42].

Cross-service inconsistencies are a new challenge that arises in this setting. Although consistency is well-studied in the context of individual distributed datastores, new issues appear when an application uses *multiple independent distributed datastores*. In particular, a single end-to-end request can make multiple writes to multiple different datastores over the course of its execution; these writes are issued by the different services (and machines) traversed by the request. As a whole, the request establishes an implicit visibility ordering for its writes, which readers must respect if they are to be consistent.

However, in today's systems, datastores are independent and mutually oblivious. Each datastore implements its own consistency model; there is no coordination between datastores when replicating updates; and no single service has global knowledge of all datastore interactions of an end-to-end application request. Consequently, existing systems can neither detect cross-service consistency violations, nor enforce a visibility ordering for readers. This challenge has emerged recently, from both user bug reports [29–32] and reports from practitioners [2, 4, 42, 59, 66].

In this work, our goal is to provide developers with principles and tools to prevent cross-service inconsistencies in distributed applications. This goal is particularly challenging

due to the size and complexity of the collection of services and their interaction patterns, as we further motivate in §2, using experiments with multiple services and a large-scale trace from Alibaba [48]. Furthermore, the fact that different systems are developed independently calls for a solution that can be incrementally adopted and deployed by each system that wants to prevent this class of inconsistencies.

To address these challenges, we present Antipode, a system that enforces *cross-service causal consistency* for applications with requests that span multiple processes and interact with multiple datastores. The design of Antipode brings together four main concepts, all of which are essential to the effectiveness and practicality of our solution.

First, we extend Lamport's causal consistency model to a new cross-service causal consistency definition. The new definition introduces the concept of *lineages*, embodying the dependent actions of a request across multiple processes. Furthermore, developers can select the relevant subset of dependencies that are amassed as requests percolate, leading to a sensible balance between semantics and scalability.

Second, we designed Antipode in a way that does not need global knowledge of all services and datastores. Instead, services only need to communicate *lineage metadata* with the end-to-end execution flow of requests and within datastore operations, piggybacking on existing request context propagation mechanisms [50, 54].

Third, to avoid stalling every read to check for possible incoming dependent updates, Antipode allows developers to selectively enforce causal relationships through a simple and generic API, with a service-specific implementation. This is key to avoiding user-visible delays, while also decoupling the generic specification aspects from the implementation that is specific to each service.

Finally, to facilitate an incremental deployment on top of existing service implementations, Antipode takes a pragmatic *bolt-on* approach, inspired by prior work in causal consistency [14], where a service developer integrates Antipode as an application-level library and datastore shim.

We demonstrate the practical benefits of Antipode through experiments using eight popular cloud and open-source datastores (MySQL, Dynamo, Redis, S3, SNS, AMQ, MongoDB, and RabbitMQ), an end-to-end evaluation on the DeathStarBench [36] and TrainTicket [66] microservices benchmarks, and a public-cloud microbenchmark. Our experimental evaluation pinpoints the existence of cross-service inconsistencies in these systems, and shows that Antipode is able to effectively prevent them with minimal performance impact.

In summary, the contributions of this paper are as follows:

- We exemplify empirically how cross-service consistency violations arise both in open-source benchmark applications and using public cloud datastores.
- We propose *cross-service causal consistency* (XCY), an extension of Lamport's causal consistency to systems where requests span multiple processes and interact with multiple datastores.
- We present Antipode, a system designed to enforce cross-service causal consistency. The design of Antipode brings together the four main concepts mentioned above, to produce a solution that is (1) scalable, (2) performant, and (3) easy to deploy through an incremental integration with existing microservice systems and datastores (in merely tens of LoC).
- We demonstrate experimentally that Antipode can effectively enforce cross-service causal consistency with minimal impact on end-to-end performance.

The remainder of this paper is structured as follows: in §2, we motivate how the increasingly complex architecture of modern applications renders them vulnerable to cross-service inconsistencies. §3 delves into the challenges associated with enforcing cross-service consistency and presents the insights offered by Antipode to tackle them. In §4, we define XCY and introduce the notion of lineages, and in §5 we detail how Antipode effectively tracks and enforces these concepts. The design and implementation details of Antipode are discussed in §6, and in §7 we present an experimental evaluation of Antipode using a combination of public cloud datastores and microservice benchmarks. In §8 we compare Antipode to existing approaches, and in §9 we conclude by summarizing our findings.

2 Motivation

2.1 On the complexity of modern applications

At an intuitive level, the more complex an application's architecture and the patterns of interaction between its components, the higher the chances of cross-service inconsistencies occurring – in particular when a single request accesses multiple datastores and triggers numerous state externalizations.

This is particularly relevant for modern applications [2, 23, 24, 58, 65], as they prevailingly resort to architectural patterns that prescribe a loose coupling of services, using various different consistency models for datastores, and allowing for independence between different services [34, 42]. Furthermore, the complexity of end-to-end request flows and resulting graphs of interacting services can be daunting: a single user request may span hundreds of sub-queries that traverse multiple microservices [2, 36, 59]. We therefore argue that these architectural features are fertile ground for cross-service inconsistencies.

Substantiating this claim through real-world examples is, however, a challenging task. While there are many anecdotal reports of inconsistencies in large-scale services, they rarely have enough detail to diagnose whether their cause is due to cross-service issues or not. (An exception is a report from Facebook [2], which we elaborate on in the next section.)

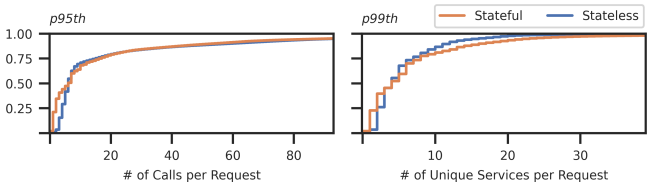

Figure 1. (Left) CDF of the number of calls to services per request, and (right) CDF of unique services called per request, on Alibaba's trace dataset. For ease of presentation, we cut off the long tail to only show values within the 95th and 99th percentile, respectively.

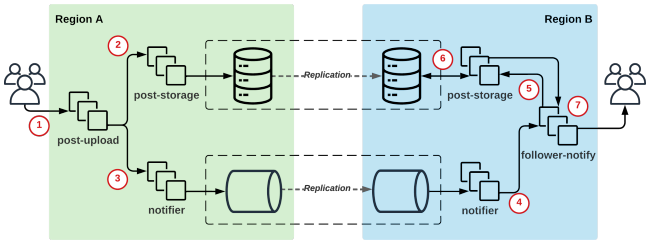

Figure 2. Request flow for publishing a post in the Post-Notification example application.

Fortunately, Alibaba recently released a comprehensive dataset [48], from which we can glean deeper insights regarding both deployment size, and, more interestingly, the respective request patterns. In terms of the scale of the deployment, we found that out of Alibaba's more than 17k microservices, more than 80% are stateful services, namely databases, caches or message queues. The prevalence of stateful services in the requests' large call graphs is also very high. In fact, Fig. 1 shows that more than 20% of requests perform 20 or more calls to stateful services. Moreover, more than half of the requests call 5 or more unique stateful services, with 10% calling more than 20. Furthermore, Alibaba's situation is by no means an exception: Uber has also disclosed comparable findings regarding the intricacies of their call graphs [65]. In particular, they report that a single request can call up to 1400 unique endpoints, has an average of 112 RPC calls per request – and a maximum of 275k – and has an average request depth of 8.5 and a maximum of 35.

Overall, all these findings confirm not only the sheer scale of modern large-scale applications, but also how complex the call graphs of a typical request have become. Furthermore, the analysis shows that it is common for a single request to externalize state through multiple stateful services, which we argue can lead to cross-service inconsistencies.

Next, to motivate our work more concretely and precisely characterize cross-service inconsistencies, we abstract away from the complexity of Alibaba's trace, and focus on a distilled, but realistic example that we use throughout the paper.

2.2 Example: Post-Notification application

The following motivating example, which closely follows a description of a real problem in Facebook's infrastructure [2], illustrates cross-service consistency violations, and highlights how and why these occur in distributed applications. We consider a simplified version of a *post-notification* application, depicted in Fig. 2. In this application, users can upload posts and followers receive notifications, similarly to applications such as social networks, online forums, and e-commerce sites.

Internally, the application comprises four different services, each responsible for a different task in the end-to-end request flow, namely: a post-upload service that works as a proxy for the clients; a post-storage service responsible for storing and processing the contents of posts; a notifier service in charge of disseminating notification events; and a follower-notify service that notifies followers of new posts.

Cross-service consistency violations can occur in this application: followers in Region B can be notified of posts that do not yet exist in that region. Concretely, we step through the end-to-end request flow to illustrate how this can occur:

① A user in Region A invokes the top-level post-upload API, which internally makes an RPC to post-storage passing the post data as argument.
② post-storage writes the post data to its internal replicated datastore, returning a post ID as the RPC response.
③ post-upload makes an RPC to notifier passing the post ID and user ID. notifier writes a notification event to its internal replicated queue.
④ Eventually, the notification arrives to Region B, where it is dequeued by notifier and delivered to follower-notify.
⑤ To deliver notifications, follower-notify first retrieves the post data by calling post-storage and passing the post ID.
⑥ post-storage reads the post data from its internal replicated datastore and replies to follower-notify.
⑦ follower-notify delivers the post and notification to followers in that region.

The cross-service inconsistency is visible in step ⑥ of this request flow. The internal replicated datastore of post-storage might not have replicated the post data to Region B yet, and thus reading the post could yield an empty or stale result. This inconsistency occurs because our example uses *two independent datastores* in the end-to-end flow of application logic: the posts datastore at ② and the notifications queue at ③. Individually, each of these datastores correctly implements some consistency model. Yet together, we fail to achieve our expected consistent behavior: post data should be visible by the time the notification event is delivered.

2.3 Exploring cross-service inconsistencies

To further highlight the risk of cross-service inconsistencies and their sensitivity to deployment aspects, we performed an experiment to measure these inconsistencies in several cloud datastores from Amazon Web Services (AWS). We implemented the post-notification example application described in §2.2 and deployed the services across two global regions. We measure inconsistencies as previously described:

		post-storage			
		MySQL	DynamoDB	Redis	S3
notifier	SNS	95%	95%	88%	100%
	AMQ	8%	7%	13%	100%
	DynamoDB	0%	0%	0%	13%

Table 1. Percentage of observed inconsistencies for several combinations of off-the-shelf AWS services. Both post-storage and notifier are geo-replicated between US and EU using the built-in mechanisms for the respective service (experimental setup details in §7).

how often a notification is received before the corresponding post can be read. We used multiple different cloud datastores, namely MySQL (a RDBMS), DynamoDB (a NoSQL database), S3 (an object store), and Redis (an in-memory cache) for posting data; and SNS (a publish-subscribe system), AMQ (a message broker), and DynamoDB for notification events.

Table 1 details the percentage of observed inconsistencies. The results show that some combination of systems experience more inconsistencies than others. For example, using S3 as post-storage, we observed numerous inconsistencies across all notifier datastores, suggesting a slower internal replication. In contrast, with DynamoDB as notifier, we observed low rates of inconsistencies across all post-storage datastores, suggesting a less optimized replication for the notification's specific type of payload – which enables the post to replicate sooner than the notification. We study this scenario in more depth in §7.

3 Challenges & Insights

To enforce cross-service consistency in the challenging setting of large-scale deployments with several independent microservices, the design of Antipode addresses the following main challenges through the corresponding key insights.

3.1 Extending causal consistency

First, we need to understand at a conceptual level what is the disconnect between the current view on causal consistency and the architecture of modern distributed systems, and how to address that mismatch. Recall that the original definition of causality stems from the *happened-before* partial order defined by Lamport [43], which states that two events are (causally) related by this partial order if they are either consecutive events from the same process, or the sending and corresponding receiving events for a given inter-process message (plus the transitive closure of the previous two classes). This definition assumed a simple model where a system was a collection of processes and events were either executing a single machine instruction (or subprogram) or sending and receiving messages between processes.

This definition was later extended in the context of the ISIS system to causal broadcast [16] and shortly after by Ahamad et al. to causal memory [1]. In particular, the latter definition makes the observation that, in a shared memory system, communication occurs via reading and writing from shared memory positions, and therefore the causality partial order needs to be extended with a *writes-into* order, capturing the reading of a value that was written by another process.

Insight #1. In this paper, we observe that the brave new world of microservices and large scale distributed systems no longer has a simple and logically centralized view of a shared memory, through which processes read and write all the side effects that need to be shared among them. Instead, we have complex patterns of interactions between services that stem from a single user request, as exemplified in our simple post-notification example (but translated to a much larger scale, as evidenced by the Alibaba trace [48]). To address this, we strengthen Lamport's notion of causality to have a broader view of the meaning of "writing to a shared memory", so that it encompasses all the side effects that percolate throughout different services – a novel concept named *lineage*.

3.2 Capturing cross-service dependencies

Addressing cross-service inconsistencies through a causal approach also poses inherent scalability challenges, namely due to the fact that the number of dependencies that are amassed in our target scenarios can be prohibitively large.

The most common approach for tracking these dependencies is to use vector clocks, where each entry contains the most recent version observed for each process. More recent solutions optimize this by coalescing vector clocks into a single scalar [18, 27] (at the cost of requiring frequent state dissemination). Regardless of the technique used to track and enforce causal dependencies, the large dependencies trees amassed (and the corresponding metadata size) have been shown to lead to scalability and performance bottlenecks [13, 17, 20, 21, 51]. We argue the effects will be largely magnified in a cross-service setting. For example, in an ecosystem as large as Alibaba's [48], this would require enforcing dependencies from possibly hundreds of services (each of which will depend on the same order of other services, and so on). At the protocol level, this implies a proportional number of entries in a vector clock, or frequent expensive cross-service synchronization calls.

On top of that, we argue that some of these dependencies may not be worth enforcing. Going back to our running example, if the post also triggered a write to a continuously updated data analytics stream, it would not make sense to wait for the entire set of analytics to be recomputed before reading any data that was produced from it.

Insight #2. Our goal is to devise a system that empowers the developer with tools for capturing and enforcing these dependencies, in a way that is as automated as possible, while also giving the developer some control over the subset of these dependencies that need to be enforced. To automatically detect the relevant dependencies, we track dependent operations across services by communicating metadata, conveying this set of dependencies both within datastores and alongside end-to-end requests. This metadata can easily be

extracted and conveyed through existing systems by leveraging causal tracing frameworks [40, 50, 53, 56, 60, 63] already commonly deployed. Furthermore, we provide developers with an API to explicitly add and remove dependencies that will be enforced at a later stage. We argue that this combination leads to a smaller dependency graph, resulting in improved performance at scale, at the cost of requiring some intervention from developers, which our evaluation shows to be simple in practice.

3.3 Enforcing cross-service consistency

Enforcing cross-service consistency is particularly challenging within these increasingly complex environments, especially since each service individually lacks both (a) the end-to-end knowledge about previous datastore operations made by other services, and also (b) the knowledge regarding the protocol implementation and semantics of other datastores. To illustrate this through our running example, notifier lacks both (a) the knowledge of the initial write to post-storage, and (b) the knowledge of the asynchronous replication that was triggered in response to the write post operation.

As straw man solutions, we could ameliorate both classes of problems by strengthening the guarantees of post-storage to make its replication synchronous, but this introduces undesirable delays that are discouraged in practice [34, 47, 48]. We could also try to incorporate more global knowledge about end-to-end requests. For example, the notifier service could manually check the post-storage service before delivering the notification. Alternatively, all datastores could synchronize their replication progress. Generically, this requires developers to enforce consistency at an application-wide scale – which, although it is the status-quo in microservice-based applications [34, 42], is precisely the burden we aim to minimize. Overall, these approaches break the design philosophy of microservice applications, which intentionally imposes strict boundaries and loose coupling between services, to enable rapid and independent development [34, 42]. Recently, Google introduced *Service Weaver* [38], a framework that enables developers to build microservice-based application using a programming model similar to writing a monolith application. While this framework helps developers tame the complexity of managing a microservice deployment, it is not meant to address either data placement or possible cross-service inconsistencies.

A related challenge is that existing approaches typically enforce the visibility of causal dependencies at either read or write operations [3, 5, 18, 26, 27, 44–46, 55], which, in a scenario with cross-service dependencies, can inadvertently add user-facing delays that degrade the user experience. For instance, in the post-notification application, the application delivers a notification to the end-user, which triggers the fetching of the corresponding post from post-storage. However, if this read is performed with a set of cross-service dependencies, it may result in the end-user having to wait for the replication process to complete before obtaining a consistent view of both the notification and post objects.

Insight #3. In order to avoid the performance penalty of checking and enforcing cross-service dependencies at every single datastore operation, Antipode decouples the enforcement into a separate `barrier` call. This offers a more flexible approach by allowing the developer to selectively enforce the visibility of cross-service dependencies through a specific API call with a generic interface and semantics, but a service-specific implementation that is opaque to other services. Additionally, it allows the developer to select the best `barrier` placement in order to hide the delay from the end-users, independently of datastore operations.

3.4 Incremental deployment

Many existing solutions for preventing cross-service inconsistencies require architectural and internal changes to existing applications (as detailed in §8). However, we believe that the instrumentation for such prevention should not require a forklift upgrade of the entire set of applications: we should aim for a minimal and self-contained set of changes that allows each individual service to benefit from the new consistency semantics.

Insight #4. Inspired by prior work [12], Antipode takes a pragmatic bolt-on design, where its logic runs as a shim layer around existing services. This approach does not require deep changes to the internals of services or datastores (unlike FlightTracker [59]), making it a more flexible and adaptable solution for gradually correcting cross-service inconsistencies. In addition, Antipode is agnostic to the interface or semantics of the services that comprise the system, and provides an API that does not require end-to-end application knowledge.

4 Cross-Service Causal Consistency (XCY)

In this section, we define XCY, which, unlike prior causality definitions, captures data inconsistencies such as those exemplified in §2 and allows for an efficient and scalable design (§5) that can be readily applied to existing applications (§6).

4.1 Lineages

The concept of a *lineage* captures a tree of events (or actions) across different services, corresponding to the various branches that are spawned as a consequence of a given client request. For example, in the case of the post-notification example (§2.2), there are two distinct lineages. The first has two concurrent branches corresponding to the write post and notification operations, including their respective replications. A separate lineage starts when the follower-notify reads the notification, followed by the post read at post-storage.

The concept of lineage has actually been used extensively within the distributed tracing community [19, 33, 50, 54, 60], but was never formalized nor incorporated into Lamport's

causal consistency. (We provide a formal definition of the system model and the concept of lineage in appendix.)

Although lineages are a simple concept, their instantiation can be very complex. For instance, at Alibaba, user requests typically form a tree, where more than 10% of stateless microservices fan out to at least five other services, and where the average call depth is greater than four [48]. Additionally, this tree contains, on average, more than five stateful services (Fig. 1). This attests to how scattered application state is in microservice-based applications, which makes it more challenging to track and aggregate the dependencies between states into lineages.

4.2 XCY definition

To define XCY, we begin by outlining the abstract model on which it operates. We restrict our model to a system that encompasses two operations: `write(k,v)` and `read(k)`. Generalizing to complex queries and updates would be straightforward. We denote lineages as $\mathcal{L}$ and we define $\mathcal{L}(a)$ to be the lineage $\mathcal{L}$ such that operation $a \in \mathcal{L}$. XCY also makes use of the Lamport *happened-before* relationship [43], denoted $\rightarrow$, which relates operations that either succeed each other in the same execution thread or the sending and receiving the same message across processes.

We denote the cross-service causal order between operations as $\rightsquigarrow$, which extends the canonical *happened-before* to our setting. Given two operations a and b, if $a \rightsquigarrow b$, we use the terms that b depends on a or a is a dependency of b. Three rules define this relationship:

1. **Happened-before**. If $a \rightarrow b$ then $a \rightsquigarrow b$
2. **Reads-from-lineage**. If a' is a write operation and b is a read operation that returns the value written by a', then $a \rightsquigarrow b, \forall a \in \mathcal{L}(a')$
3. **Transitivity**. For any operations a, b, and c, if $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$.

Note that Lamport's causality definition only uses rules 1 and 3, and therefore the additional dependencies stemming from rule 2 create a stronger definition, as we elaborate next.

We can now define XCY consistency in a similar way to causal memory [1], i.e., by taking the $\rightsquigarrow$ operator, and imposing that each process sees the execution of operations in an order that respects $\rightsquigarrow$, i.e.:

Definition. An execution x is XCY consistent if, for each process p_i, there is a serialization of the all `write` and p_i's `read` events of x that respects $\rightsquigarrow$.

Intuitively, what we capture is that, after an operation b from one lineage reads the value written by an operation a from another lineage, any *further* operations that depend on b must observe the effects of the entire $\mathcal{L}(a)$ (and not just a itself, as in classical definitions). In the context of our post-notification example, given that writing the post and the notification belong to the same lineage, then reading the notification will not only establish a causal dependency to that operation, but also to the post itself. This causal dependency will then be transitively carried to subsequent operations, namely reading the post (which thus precludes a `post not found` scenario).

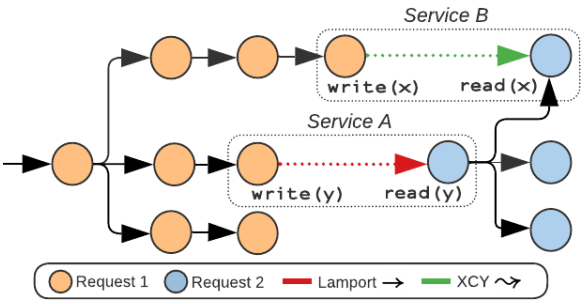

Figure 3. This example illustrates the difference between Lamport's $\rightarrow$ and XCY's $\rightsquigarrow$. While the red dependency is present in both definitions, the green dependency stems from the concept of lineage and is only present in $\rightsquigarrow$.

How XCY relates to Lamport's causal consistency. Given the commonalities between XCY and traditional causal consistency definitions [1, 43], it is natural to ask whether one is stronger than the other. The answer is that XCY is stronger (i.e., more restrictive) than Lamport's causality. This is because a read operation adds a causal dependency to the entire lineage and not just the operation that wrote the value that was read. In other words, read operations that observe the result of a particular write operation require causally subsequent operations to observe the entire offshoot of execution branches resulting from the external client request that originally generated the observed write.

Fig. 3 illustrates this distinction between the happens-before relation as defined by Lamport [43] and extended by Ahamad et al. [1], and XCY's definition of $\rightsquigarrow$. This example includes two requests: request 1 (R_1) highlighted in orange and request 2 (R_2) in blue. R_1 begins with an initial action that triggers multiple subsequent events across various services. Meanwhile, R_2 originates from reading a value previously written by R_1 at Service A. Note that these two events, `write(y)` and `read(y)`, are considered to be causally related according to traditional happens-before relation by Lamport [43] and extended by Ahamad et al. [1]. Then, after that relation was established, R_2 percolated through the application and one of its branches ended in Service B, where it performs `read(x)`. Furthermore, one of the branches R_1 performed a write operation, `write(x)`, also at Service B. At this point, this raises the question of whether `write(x)` and `read(x)` should be causally related. The answer in Lamport's happens-before relation is that these events are concurrent, since they come from different branches of execution, meaning that they can be ordered in any way. In contrast, XCY is more restrictive than Lamport's causality definition, because as soon as there is a read of a value that was written by another lineage – such as `read(y)` and `write(y)` – a new

relation is established between lineage R_1 and the new event from R_2, which means that all events from R_1 are ordered before `read(y)` and the causally subsequent events from R_2. Therefore, within XCY, `read(x)` should wait for the effects of `write(x)` to be visible.

Note, however, that for scalability and incremental deployment considerations, our implementation enables developers to relax XCY by selectively choosing a relevant subset of operations, as we will describe next.

5 Enforcing XCY

This section presents the main design choices of Antipode, a system that enforces XCY in a scalable way.

5.1 Tracking dependencies in Antipode

Keeping track of dependencies within lineages entails a design choice that was articulated in a recent system [12], namely between potential and explicit causality. *Potential causality* refers to the traditional definition [43], where all possible influences via (transitive) dependencies are implicitly established. Transparently tracking all dependencies leads to large dependency graphs, which can degrade the performance and scalability of the system [13, 17, 20, 51]. In turn, *explicit causality* [13, 41] requires applications to explicitly identify and declare dependencies between write operations. For example, applications can mark these dependencies based on user interactions (e.g., replying to a post establishes a dependency between the post and the reply). This results in smaller dependency graphs (and hence better performance at scale), at the cost of requiring developer intervention and knowledge.

In Antipode, we aim to strike a balance between the two approaches, namely by automating the dependency tracking process to the fullest extent, while giving developers control over relevant dependencies so that the dependency set is reduced, leading to improved performance at scale.

Implicit dependency tracking. By leveraging context propagation tools, such as the ones used for distributed tracing [40, 50, 53, 56, 60, 63], we are able to automatically track dependencies across the graph of traversed services that communicate via message passing. However, this is not sufficient to track and propagate dependencies through replicated datastores (due to writing and then reading from the same key, in possibly different replicas). To address this, we use a *bolt-on* approach [14], where the interaction with the underlying datastores is conducted indirectly via a shim layer, allowing datastores to remain unchanged. This enables us to interpose write operations and automatically capture dependencies across replicas of traversed datastores. Both techniques allow us to collect dependencies gradually and automatically over time, as requests percolate through the application.

Explicit dependency tracking. To allow developers to explicitly add or remove dependencies from the current lineage context, Antipode provides a generic `append` and `remove` API. This granular control over dependency management enables developers to, in exceptional cases, capture dependencies that were not automatically detected and remove irrelevant dependencies for an optimized user experience.

In addition, Antipode further reduces the dependency set by dropping the ongoing set when the execution ends (or when `stop` is called). While this is Antipode's default behavior, developers can selectively override it by explicitly calling the `transfer` procedure. This procedure transfers the dependency set from one lineage to a subsequent one, explicitly establishing the transitivity between them.

The rationale behind this design choice is simple: if a lineage already has a high number of dependencies, blindly transferring dependency sets between lineages might result in an explosion of the dependency graph, a challenge even traditional causal consistency approaches wrestle with [13]. This is especially relevant for objects that are constantly read and written (known as linchpin objects [2]). By giving developers the ability to control this behavior, we empower them to make informed decisions to manage the dependency graph effectively and optimize performance. While this API increases the programming overhead and changes the original semantics (namely the transitivity rule of the $\rightsquigarrow$ operator) we found this to be an acceptable trade-off that promotes scalability, since, by default, long dependency chains across lineages are truncated.

As a result of the previous design choice, developers are required to use `transfer` to ensure a correct XCY order of the application. To illustrate this scenario, we extend our running example (Fig. 2) so that, before writing a new post, user Alice blocks her follower Bob by writing to an access control list, held in a geo-replicated storage. This results in two lineages: $\mathcal{L}_{block}$, the request from Alice to block user Bob in the acl-storage, and $\mathcal{L}_{post}$, the request from Alice to create a post. In this case, after Alice blocks Bob, Bob should not receive a notification for the subsequent post. However, this would happen in case the acl-storage replication is slower than the post-storage replication, allowing Bob to see the notification and the post, resulting in an XCY consistency violation. To correct this scenario with Antipode, the dependency set of $\mathcal{L}_{block}$ (containing the write to the acl-storage) is copied to $\mathcal{L}_{post}$ by having the developer explicitly call `transfer(`$\mathcal{L}_{block}$`,`$\mathcal{L}_{post}$`)`.

Overall, the combination of the implicit and explicit approaches (1) facilitates the tracking of dependencies within a lineage, while also (2) preventing the system from amassing huge sets of causal dependencies, and (3) allowing developers to pinpoint which lineages are logically connected.

5.2 Enforcing dependencies in Antipode

While dependency tracking provides one dimension of XCY, there is also the need to enforce the visibility of captured dependencies before a new operation takes effect. Traditionally, dependency enforcement is done implicitly, i.e., the

underlying service is able to resolve a list of causal dependencies without developer intervention. In a cross-service setting, however, this approach has a significant drawback: it requires the services to enforce the set of dependencies at every read or write operation. This is undesirable in our context since it requires frequent cross-service communication, which introduces unacceptable delays.

For this reason, Antipode uses explicit dependency enforcement, allowing developers to select the places where XCY dependencies must be enforced. To this end, developers place `barrier` calls in selected locations of their applications.

This primitive takes the current lineage and enforces an order of operations that is consistent with the definition of XCY by unpacking the causally-dependent operations that are currently being carried by the lineage. It then enforces the visibility of these operations at the corresponding services by invoking the `barrier` operation, which has the semantics of blocking until those dependent operations are made visible (or superseded by more recent operations). This approach has two advantages. First, it provides developers with a fine-tuned balance between correctness (by enforcing important dependencies) and performance (by bypassing irrelevant ones). Second, it gives developers control over the best location for that enforcement to happen, which is crucial to avoid negatively affecting the user experience. We elaborate on `barrier` placement in §6.3 and showcase its tradeoffs in §7.4.

One argument that can be made against `barrier` is that it is as explicit as today's application-level solutions, since both of them require the developer to manually select its locations. What makes Antipode's approach better suited is not only `barrier`, but its combination with the implicit/explicit dependency tracking, which keeps services loosely coupled and does not require end-to-end knowledge of *what* to enforce. For instance, in the previous ACL example, `barrier` enforces dependencies that were automatically gathered from all datastores involved in the `post` lineage (acl-storage and post-storage) in a way that does not require knowledge about which systems were involved and how they are implemented.

As we mentioned, a downside of relying on developer input for enforcing the visibility of dependent operations is that not all consistency violations will be necessarily prevented, and some undesired behaviors may be observed when developers do not place the required `barrier` calls. We envision that Antipode can also be helpful in this context by additionally working as a testing tool: instead of exhaustively trying to prevent every possible variant of XCY violation, developers can (as part of their development cycle) use Antipode to incrementally correct them.

6 Antipode

Antipode is an application-level library for enforcing XCY. Table 2 outlines Antipode's API operations, and Fig. 5 illustrates their interactions. Integrating Antipode entails three

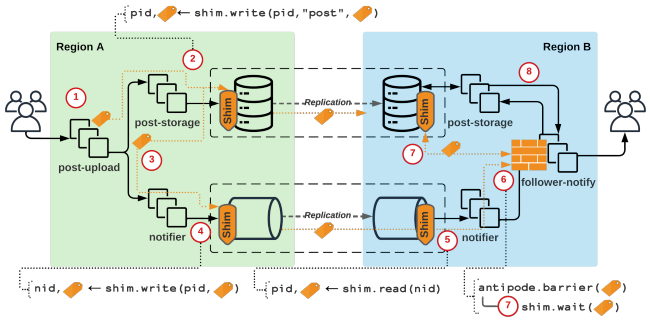

Figure 4. Antipode prevents the identified XCY violation in the Post-Notification example application (Fig. 2).

concerns for service developers. First, datastore invocations (`read` and `write`) are replaced with proxy calls to Antipode's Shim API, in the same manner as prior work [14]. Second, Antipode's Lineage API piggybacks on the system's context propagation [49, 54], which requires instrumentation if not already present in the system. In practice, context propagation is widespread in microservice systems, as it is a prerequisite for commonplace tools like distributed tracing [40, 50, 53, 56, 60, 63]. (In our evaluation, for example, context propagation already exists in the benchmark systems.) Third, to actually *enforce* causal dependencies, developers selectively add calls to Antipode's `barrier` API to block an execution until its causal dependencies are satisfied.

End-to-end flow. Internally, Antipode comprises several components that together enforce XCY. Fig. 4 depicts the end-to-end flow of the example from §2.2, annotated with Antipode interactions; we refer to the numbers in the figure in our description.

① The request begins at the post-upload service which starts a new lineage. The lineage is passed with the RPC to post-storage.
② The call to `write` on the post-storage database is proxied via Antipode's Shim API, and the lineage is included as an argument. The `write` call returns an updated lineage that reflects this new database operation.
③ In its RPC response, post-storage includes the updated lineage. Likewise, post-upload then passes the updated lineage with the RPC to notifier.
④ The call to `write` on the notifier queue is proxied via Antipode's Shim API, passing the lineage as an argument.
⑤ When the notification has replicated and is read in Region B, the `read` call also returns the lineage.
⑥ The notifier includes the lineage in calling follower-notify. Here, the follower-notify service calls `barrier`.
⑦ Internally, the call to `barrier` at follower-notify will inspect the dependencies contained in the lineage, then await replication to finish at the relevant datastores.
⑧ `barrier` will only return once replication has completed; follower-notify can now safely read the post and deliver the notification to followers.

Core API	
barrier($\mathcal{L}$)	Enforces lineage's dependencies
Shim API	
$\mathcal{L}' \leftarrow$ **write**(k,$\langle v, \mathcal{L}\rangle$)	Writes key along with lineage
$\langle v, \mathcal{L}\rangle \leftarrow$ **read**(k)	Reads key and returns the lineage
wait($\mathcal{L}$)	Waits for all the lineage dependencies
Lineage API	
$\mathcal{L} \leftarrow$ **root**()	Initialize lineage in the running process
stop($\mathcal{L}$)	Closes lineage in the running process
append($\mathcal{L}$,dep)	Appends dependency to a lineage
remove($\mathcal{L}$,dep)	Removes dependency from a lineage
transfer($\mathcal{L}_a$,$\mathcal{L}_b$)	Transfers $\mathcal{L}_b$ dependencies into $\mathcal{L}_a$
$s \leftarrow$ **serialize**($\mathcal{L}$)	Serializes the lineage
$\mathcal{L} \leftarrow$ **deserialize**(s)	Deserializes the lineage

Table 2. Antipode API reference. Shim layer method arguments might change according to the underlying datastore.

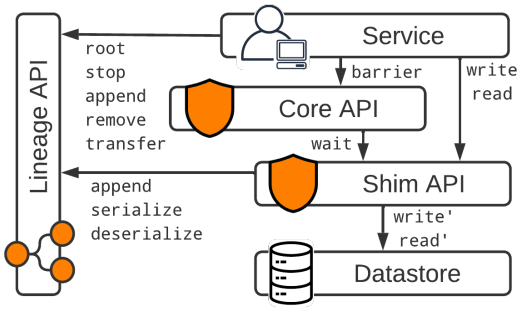

Figure 5. Overview of the interactions between a service and the Antipode APIs. `write'` and `read'` are the original operations on the underlying datastore.

6.1 Creating and updating lineages

We implement lineages as a set of *write identifiers*. A write identifier uniquely identifies a write to a datastore (e.g., a $\langle datastore, key, version\rangle$ [45]). Antipode relies on the underlying datastore to generate the unique write identifiers (e.g., at ② and ④), by assuming a versioned key-object model (we believe that this does not reduce generality, e.g., FlightTracker [59] makes a similar assumption). Furthermore, many of the existing datastores natively offer this model, such as the `rowversion` of Azure SQL Database [11] or `hlc` in CockroachDB [25], while others can be trivially extended to support it, such as AWS DynamoDB [6]. These identifiers are later used by calls to `barrier` (e.g., at ⑥), in order to check if writes have been replicated.

Every Shim API `write` call takes a lineage as an argument and returns an updated lineage. The returned lineage simply extends the lineage given as input argument to include the new write identifier.

6.2 Propagating lineages

Antipode propagates lineages in two dimensions: alongside end-to-end requests as they traverse services via RPC calls (e.g., ③, ⑥); and with data values as they are replicated within datastores (e.g., ⑤, ⑦).

Request propagation. To maintain and propagate lineages with end-to-end requests, developers use Antipode's Lineage API. At each point in time when a request is executing in a thread, it will have a corresponding lineage; typically, this is stored in a pre-existing (thread-local) request context [50]. The `root` API call initializes an empty lineage; it is used only at the beginning of a request's execution, before any reads or writes have occurred. Conversely, the `stop` API call discards a lineage from the request context. In practice, `stop` calls are rare because execution tends to just end, discarding contexts (and lineages) in the process. After a call to `write`, the returned lineage is written to the request context. Services must include their lineages with all RPC requests and responses. The `transfer` API call establishes continuity between two lineages by combining their dependency sets (as detailed in §5.1).

Datastore propagation. Antipode propagates lineages alongside data values within datastores. For each call to the Shim `write` API, Antipode serializes the lineage given as an argument, and `writes` it alongside the data value in the underlying datastore. For a subsequent call to the Shim `read` API elsewhere, Antipode calls `read'` (i.e., the read operation from the underlying datastore) and deserializes the data value and its corresponding lineage. The caller of `read` can then combine the returned lineage with their own current lineage using the Lineage `transfer` API. Datastore propagation requires datastore-specific Shim Layer implementations. However, as described below, this entailed no more than 50 LoC for each of the 8 datastores in our evaluation.

6.3 Enforcing consistency

Antipode's `barrier` API call enforces the visibility of a lineage. It takes a lineage as an argument and will block until all writes contained in the lineage are visible in the underlying datastores. Internally, a `barrier` will inspect the write identifiers in the lineage and contact the corresponding datastores. For each datastore, `barrier` will call the datastore-specific `wait` API, which will block until the write identifier is *visible* in that datastore. Note that `wait` is datastore-specific because visibility depends on the design choices and consistency model of the underlying datastore. Once `wait` has returned for all identifiers in the lineage, `barrier` will return.

For developers adopting Antipode, placing `barrier` calls is the main new implementation decision. Developers are free to decide where in the code `barrier` should be called. Naïvely we could place a `barrier` call immediately preceding any `read` call, and this would achieve XCY. While this fully automated solution is attractive, by placing `barrier` on the critical path of every `read` request we would add unacceptable delays and lead to user-visible slowdowns (as we show in §7.4). For a better user experience, it may be more sensible to call `barrier` earlier in a request's execution – Antipode gives developers the flexibility to make this choice.

To guide developers in choosing barrier locations, Antipode can work as a *passive consistency checker* by providing a *dry-run* mode which allows developers to simulate the enforcement of barrier locations without actually enforcing them. This procedure returns developers insights into lineages that were unable to be enforced during the first attempt, which might hint at the presence of cross-service inconsistencies. In our experience, we found that relationships between different datastores can be empirically detected by developers, namely from commonalities between their data models and schemas. E.g., notifications objects in notifier have a `post-id` referring to the corresponding post in post-storage. These foreign key-like relationships provide a practical way of identifying necessary `barrier` locations.

We also implemented other variants of `barrier` that accept a timeout, and an asynchronous `barrier` that triggers a callback to application code once dependencies are visible. Furthermore, we implemented a practical optimization strategy specifically tailored for geo-replicated datastores. This involves implementing the `wait` procedure to enforce dependencies only from replicas that are co-located with its caller, thereby avoiding (whenever the underlying datastore allows it) global enforcement.

6.4 Implementation

Antipode APIs. We have implemented Antipode's Lineage API and Core API in C++ (250 LoC), Java (175 LoC), and Python (40 LoC). Antipode piggybacks lineage metadata on OpenTelemetry baggage [53] and thus requires minimal additional implementation for lineage propagation.

Shim layers. Antipode requires a new shim layer implementation for any supported datastore. The shim layer's purpose is to implement datastore-specific lineage propagation and `wait` logic. This often requires a one-time change by the developer to the underlying data model schema, which is still preferable to changing the complex internals of the underlying datastores. Note that, in a bolt-on approach, concerns such as replication, fault-tolerance, and availability are delegated to the underlying datastore [14].

As we mentioned, Antipode is able to enforce XCY irrespective of the consistency level of the underlying systems, as long as it is possible to implement `wait` with at least *monotonic reads* [62] semantics. For example, even though DynamoDB is originally eventually consistent – which could lead to semantics that make it hard to ensure that an object is visible – we were able to implement `wait` by simply leveraging the available strongly consistent reads [8]. We implemented Antipode Shim layers for the following datastores: MySQL, DynamoDB, Redis, S3, SNS, AMQ, MongoDB and RabbitMQ. No shim layer implementation exceeded 50 LoC. All code is open source and publicly available [10].

7 Evaluation

We evaluate Antipode in terms of its effectiveness and performance by applying it to three benchmarks representative of real-world, microservice-based applications: a Post-Notification microbenchmark, the DeathStarBench benchmark [36], and the Train Ticket benchmark [66]. Our analysis is structured around two questions, which capture the main costs and benefits of using Antipode:

- How prevalent are XCY violations, and does Antipode effectively prevent them?
- What is the overhead of enforcing XCY dependencies?

7.1 Case studies

Post-Notification. As a microbenchmark, we implemented a serverless version of the post-notification example in §2.2, using various public cloud geo-replicated storage solutions. In our implementation, we have two cloud functions (Reader and Writer), which access off-the-shelf services for the post-storage and notifier logic. Each client request spawns a Writer call, which writes a new post to post-storage, and then creates a new notification in the notifier. A new Reader is spawned when a new notifier replication event is received. For this scenario, we consider that an XCY violation occurs when reading a post outputs `object not found`. We solve this violation by placing a `barrier` right after the Reader receives the notification replication event. For the off-the-shelf datastores, we used our previously developed shim layers. These changes resulted in modifying less than 20 LoC.

DeathStarBench. The DeathStarBench is a suite of microservice based common web applications. One such application is a social network application where users can perform standard actions like writing posts, following users and reading their timelines. In comparison to the first benchmark, DeathStarBench has a scale that is closer to a real-world application: it has more than 30 unique microservices, with a mix of datastores, cache services, and pub-sub queues, mainly implemented in C++. For our evaluation, we extended the original services with geo-replication logic.

We focus on the interaction where a user publishes a new post, which causes an asynchronous task to be placed on a pub-sub queue. When that message is processed, the corresponding post is fetched, and the timeline of each follower is updated with the contents of the post. An XCY violation occurs when the follower tries to read a post, and it outputs `object not found`[1]. Antipode solves this error with a `barrier` call right after it dequeues the notification object. We implemented lineage tracking by leveraging the already existing context propagation tool (Jaeger). We developed the shim layers for the respective datastores (RabbitMQ

[1]We found that DeathStarBench's *media service* had a similar violation, whereas *hotel reservation* has a very simple architecture with no cross-datastore references, resulting in no XCY violations being found.

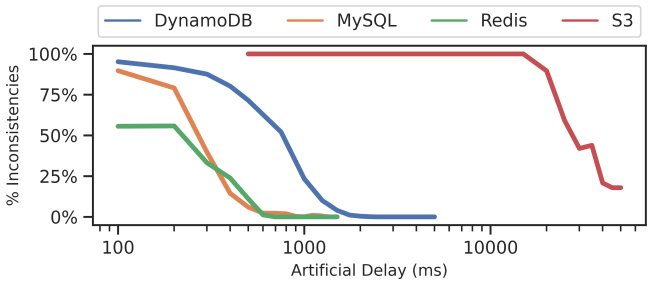

Figure 6. Percentage of inconsistencies found in Post-Notification after an artificial delay was added before publishing the notification. The notifier is always SNS.

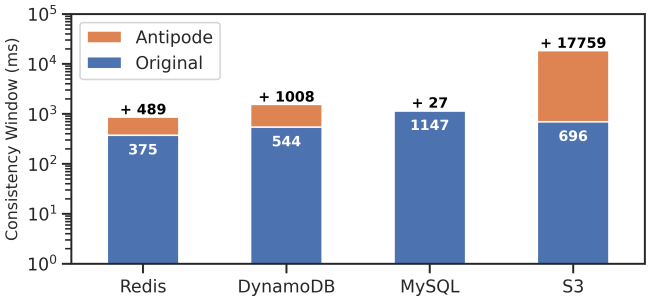

Figure 7. Consistency window in Post-Notification for the original post-storage, and with Antipode enabled. notifier is always SNS.

and MongoDB). All changes considered, these modifications comprised under 350 LoC.

TrainTicket. Developed as a testbed for replicating industrial faults, the TrainTicket benchmark [66] is a microservice-based application that provides typical ticket booking functionalities, such as ticket reservation and payment. It is implemented in Java and it consists of more than 40 services including web servers, datastores and queues.

We focus on the XCY violation that occurs when a user cancels a ticket. This operation is split into two tasks: (*a*) changing the status of the ticket to be cancelled, and (*b*) refunding the ticket price to the client. These events are performed by different services, interacting with different datastores. A violation happens when the refund (*b*) is delayed, resulting in the customer not seeing the refunded amount right away. This scenario was identified as a prevalent issue in the fault-analysis survey performed by the benchmark authors [66]: using asynchronous tasks within a request might result in events being processed in a different order, which might lead to incorrect application behavior. Unlike previous applications, no replication was needed to observe an XCY violation. Antipode was added by placing a `barrier` before returning the cancellation output to the user. We detail the consequences of this in §7.4. We implemented lineages leveraging the already existing context propagation tool and reused previously developed shim layers, thereby fixing the violation in just 10 LoC.

7.2 Experimental setup

Post-Notification. We deployed the Post-Notification application on the AWS Lambda platform. For the datastores, we used several off-the-shelf products with built-in global replication features. For MySQL, S3 and Redis, the post object size is roughly 1MB. For DynamoDB, post objects are 400KB (which is the maximum allowed object size). Notification objects, in turn, are a ⟨`notification-id,post-id`⟩ pair of about 120B. For each experiment, we submit 1000 post creations requests at the Frankfurt (EU) data center, and notifications were read from a data center in Central US (US).

DeathStarBench. We deployed DeathStarBench on Google Cloud platform, through a Docker-based deployment. Each service was deployed on machines with 4 vCPUs and 16 GB RAM, and data was replicated across East US (US) and either Frankfurt (EU) or Singapore (SG). Clients ran on machines with 2 vCPUs and 8 GB RAM, for 5 minutes in open-loop, using a load varying between 50 and 150 requests per second.

TrainTicket. We deployed TrainTicket on Google Cloud platform, through a Docker-based deployment. Each service was deployed on machines with 8 vCPUs and 32 GB RAM. Clients ran on machines with 2 vCPUs and 8 GB RAM, for 5 minutes in open-loop. For both benchmarks, we repeat each experiment 15 times and report the average.

7.3 Does Antipode prevent XCY violations?

As a first experiment, we determined the prevalence of XCY violations, and validated whether Antipode prevented them.

Post-Notification. Table 1 shows the percentage of inconsistencies found. The observed differences in their prevalence is caused by different levels of delay by different services in replicating data asynchronously.

To confirm this, we ran an experiment where we allow more time for post replication by adding an artificial delay before publishing the notification. Fig. 6 shows the results of this experiment, where each line corresponds to a different post-storage datastore type (the notifier datastore is always SNS). The results show that, as the notification delay increases, fewer inconsistencies are found. This is because, by adding a delay before publishing the notification, we allow the post to replicate sooner than the notification, and hence reduce the possibility of an XCY violation.

Finally, we report that, regardless of the combination of individual datastore consistency semantics, by applying Antipode, we saw that this inconsistency was always corrected.

DeathStarBench. On average, we observed 0.1% of violations for the US→EU replication pair, whereas for the US→SG replication pair we observed 34%. We note that we found a standard deviation of 42% for the US→SG scenario. A likely explanation for the discrepancy is that this results from a mix between the network conditions and MongoDB's replication protocol (which is reported to suffer in the presence of network latency [52]). When we applied Antipode, the inconsistency was always corrected.

post-storage					notifier	
DynamoDB	MySQL	Redis	S3	MongoDB	SNS	RabbitMQ
+42B (0.01%)	+14kB (1.5%)	+105B (2%)	+320B (0.03%)	+46B (9%)	+32B (4.8%)	+87B (20%)

Table 3. Average object size increase from the original applications to the Antipode enabled version.

TrainTicket. On average, 0.57% violations were found in normal behavior – this relatively low value is expected for an application that has no replication, and where all services are in the same datacenter. When we applied Antipode, the inconsistency was always corrected.

7.4 What is the overhead of enforcing XCY dependencies?

Preventing XCY violations imposes a coordination penalty on the application, which is materialized when Antipode enforces the visibility for a set of dependencies in a `barrier` call. In this set of experiments, we quantify the performance overhead of Antipode in terms of latency and throughput. In addition, we introduce a *consistency window* metric:

Consistency Window. This refers to the time window between one client issuing an initial write and another client *attempting* to read the written data. We measure the consistency window regardless of whether a consistent result is returned – in baseline experiments, many attempted reads result in XCY violations. However, when Antipode is used, the consistency window represents the *time-to-consistency*, since the `barrier` call prevents progress until a consistent read is possible.

Post-Notification. In Fig. 7 we show the results of an experiment that measures the consistency window for the Post-Notification application, comparing its original version with the one using Antipode. For this application, we consider that the consistency window spans from the moment the post is written at the Writer, until the Reader tries to read it. In the original application, reads are allowed to proceed immediately, despite returning an inconsistent result. With Antipode, the `barrier` call will block until a consistent result is available. Consequently, the consistency window of the applications increases proportionally to the datastore's replication delay. This delay is substantially different across datastores, and thus the consistency window varies based on how long `barrier` must block. For instance, AWS states that S3 can spend up to 15 minutes to fully propagate an object [9]. In contrast, MySQL uses a faster replication scheme, and propagation happens within 1 second of the initial write [7]. These observations are consistent with the longer consistency windows in Fig. 7, but also with the measured inconsistencies in Fig. 6: e.g., with 50 seconds of artificial delay, S3 presents a 20% chance of observing XCY violations. In this case, Antipode was able to fix violations by

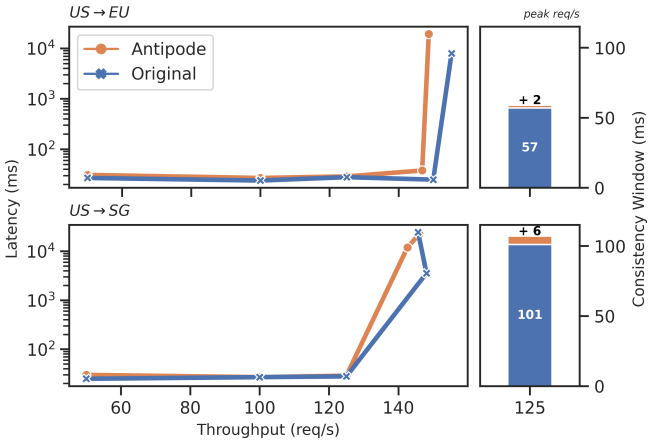

Figure 8. (Left) Average throughput vs. Latency and (Right) Consistency window (at peak 125 req/s) in the original DeathStarBench, and with Antipode enabled. Results split between US→EU and US→SG replication pairs.

waiting for a replication confirmation from S3, which took ≈18 seconds on average.

Overall, we conclude from this experiment that the overheads induced by the lineage propagation instrumentation and shim layer mechanisms of Antipode are negligible and that the increased duration of the consistency window stems almost exclusively from replication delays of the underlying systems and the consequent wait for consistency.

DeathStarBench. In Fig. 8 we compare DeathStarBench with and without Antipode, under two replication pairs: US→EU and US→SG. For this application, we consider that the consistency window ranges from the post being written to the datastore (MongoDB), until reading the notification from the message queue (RabbitMQ) and fetching the corresponding post to be added to the followers' timeline.

The left side of Fig. 8 shows the throughput-latency results of the experiment from the point of view of the post writers. Since we placed the `barrier` call right after the asynchronous read from RabbitMQ of the notification object, the impact of the `barrier` call is not felt by the writer. Therefore, in this case, we only observe the effect of creating and propagating lineages and using the shim layer. Regarding lineage metadata size, we found that the maximum size was below 200 bytes. This was also the maximum metadata size across all experiments. As a follow-up, we use the Alibaba dataset to assess how metadata size would fare in a realistic deployment. Assuming the worst-case scenario where all stateful operations are part of the dependency chain, we found that for 99% of requests the maximum metadata size is below 1 KB and, on average, just 200 bytes. In addition, we assessed the impact and overhead of modifying the datastore schema to accommodate Antipode's metadata, as summarized in Table 3. The results show that the increase in average object size is under 200 bytes, which is in line with the reported metadata size. The only notable exception is MySQL,

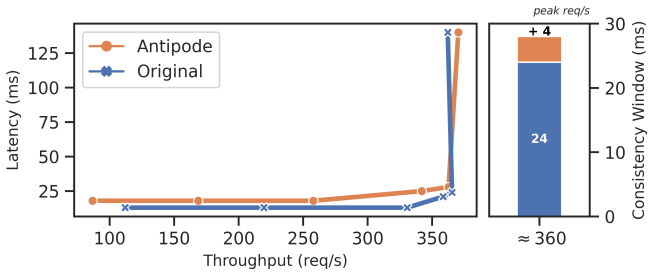

Figure 9. (Left) Average throughput vs. Latency and (Right) Consistency window (at peak 360 req/s) with and without Antipode enabled for TrainTicket. Latency is increased due to `barrier` placement in the request's critical path.

where the average object size increased by 14 KB, which we attribute to more complex data structures surrounding the new column and index created for lineage identifiers.

Overall, Antipode is able to fix all XCY violations without incurring in a significant performance impact, as we observe a maximum of 2% penalty on the DeathStarBench throughput.

The placement of the `barrier` creates a wait for consistency that postpones notification message delivery, increasing the consistency window. The right side of Fig. 8 shows that, at peak throughput, this increase accounted for a maximum of 2ms for US→EU, and 6ms for US→SG. The extension of the consistency window is significantly smaller than in the previous application due to the use of MongoDB, which has a faster replication strategy.

This scenario highlights the advantages of explicit enforcement, which allows developers to choose the best placement for `barrier`, in order to minimize the negative effects for the end user experience. In this instance, it resulted in just a small delay in the time to deliver the notification.

TrainTicket. In contrast to the previous two applications, where the XCY violation resulted from a race condition between the replication protocols of two different datastores, in the TrainTicket application, the identified XCY violation results from the *"lack of sequence control in the asynchronous invocations of multiple message delivery microservices"* [66]. More concretely, the two messages that lack coordination are the cancel order message, and the refund money message. In this scenario, the goal would be to have a consistent output where both the order is cancelled and the refund is issued.

In order to fix this scenario, the `barrier` call must be placed in the request's critical path, thereby forcing the user to actively wait for the conclusion of both actions. Fig. 9 showcases the impact of this enforcement on performance. Compared with the original application, the Antipode corrected version (without inconsistent outputs) exhibits just over 15% overhead on throughput and 17% on latency.

This TrainTicket scenario highlights the trade-offs between performance and correctness that `barrier` allows to developers. For instance, in the DeathStarBench scenario we were able to hide the consistency window penalty on the reader side – outside the writer request critical path – whereas in TrainTicket that is not possible. Consequently, due to the placement of the barrier in the critical path, the consistency window delays the barrier imposes, are then directly reflected in the throughput-latency analysis (around 4ms which corresponds to the same 17% increase). In this instance, the developer has to decide whether to expose to the user the increased latency caused by ensuring that no inconsistent intermediate states will be observed.

8 Related Work

Existing noteworthy systems that address cross-service consistency take one of the following approaches: they either wrap requests in other abstractions, resort to centralized coordination mechanisms, add transactions to the design, or propose an overall revision of the system architecture.

Wrapping approaches. Developed at Facebook, the FlightTracker [59] metadata server was designed to provide read-your-writes (RYW) guarantees across a variety of datastores. It identifies a user session through a *ticket* abstraction, to which all of a user's write operations are associated. Tickets are created and updated through a metadata server, and are passed between different services and datastores.

FlightTracker requires applications to correctly identify a user session, which – as its authors acknowledge – is not always easy. In contrast, request contexts in Antipode can be started on-demand without the need for user sessions. Furthermore, FlightTracker requires changes to the datastore public and internal APIs. We argue that this is a heavy implementation burden, whose cost is acceptable only to a few large industry players. Antipode requires no such changes, and is usable with off-the-shelf cloud-provider datastores. Additionally, FlightTracker is an all-or-nothing approach that does not allow for gradual corrections of violations, unlike Antipode.

Coordination-based approaches. Traditionally, for multiple systems to interact, the use of a logically centralized coordination mechanism like Zookeeper [39] was the natural design choice. However, strongly consistent coordination systems introduce a performance bottleneck and go against microservice principles. In particular, the proponents of this class of architectures encourage the community to embrace eventual consistency due to its better performance and higher scalability [34]. Antipode provides cross-service consistency in a way that is decentralized, can be gradually adopted, and allows for more behaviors than strongly consistent solutions.

Transaction-based approaches. Distributed transactions can also be applied in this context, e.g., in the form of 2PC protocols [15]. Just as coordination-based approaches, distributed transactions suffer from low performance [61]. Faced with this problem, the community migrated towards an approach known as Sagas [37]. A saga is a continuous sequence of local transactions. If any saga transaction fails, a costly

series of compensating transactions that undo the already applied effects must be run. Although sagas gained acceptance within the community [22, 28], they fall short when compensating mechanisms are not possible or hard to achieve. Reversal is especially challenging when transactions trigger third-party side effects [4]. Furthermore, Sagas often still rely on an orchestrator-like entity that sequences the steps of a saga.

Antipode's approach is fundamentally different for several reasons: (1) it is decentralized and therefore does not require any orchestrator, (2) it can be gradually adopted, since it does not require coordination between all entities in the same request; and (3) it does not require compensation mechanisms since violations are prevented in the first place.

Full-rewrite approaches. There are also proposals that opt for a complete redesign of the application, often ending up merging different services – and their respective datastores – into a single one. A good example of this approach is Diamond [64], where a reactive application with two different datastores (distributed storage and notification service), was merged into a single service that provided both functionalities. In fact, service re-design is a hot topic in the microservice community [22, 35]. Aegean [4] proposes a redesign of the replication layer of datastores that are based on state machine replication [57], with the intention of ensuring that dependent services always see a strongly consistent view of ongoing operations. While this approach provides strong consistency across services, it assumes that the individual replicated systems implement a serializable state machine, unlike Antipode which supports different underlying consistency models.

In our view, this represents an alternative approach to the problem, where Antipode has the advantage of not needing profound changes to existing code bases or protocols.

9 Conclusion

Microservices emerged as an architectural style that provides loosely coupled and independently deployable services, leading to good scalability, performance, and maintainability. And while they fulfilled this promise, data consistency was sacrificed and developers were left with accepting eventual consistency as the norm.

We presented Antipode, a library with a simple yet powerful API, allowing developers to enforce XCY consistency with: (1) no need to rewrite their entire applications, (2) no global management of large-scale applications, and (3) gradual and independent adoption according to the microservice ethos. Our evaluation with eight open-source and cloud-based datastores, and using two large microservice benchmarks, shows that Antipode prevents inconsistencies with a limited programming effort and low performance overhead.

Acknowledgements

This paper is dedicated to the memory of Daniel Porto. We sincerely thank our shepherd, Mahesh Balakrishnan, as well as the anonymous reviewers of not only SOSP but also OSDI and EuroSys, for their tireless efforts and insightful feedback. We would also like to thank Nuno Preguiça and Carlos Baquero for insights into earlier iterations of XCY and lineages. We also wish to thank the INESC-ID DPSS and MPI-SWS Systems groups for their support and feedback. This research was supported by the Fundação para a Ciência e a Tecnologia, under grants UIDB/50021/2020 and PTDC/CCI-INF/6762/2020.

References

[1] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995), 37–49. https://doi.org/10.1007/BF01784241 (§3.1, 4.2, 4.2, and A).

[2] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS'15)*. https://www.usenix.org/conference/hotos15/workshop-program/presentation/ajoux (§1, 2.1, 2.2, and 5.1).

[3] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguica, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *36th IEEE International Conference on Distributed Computing Systems (ICDCS '16)*. https://doi.org/10.1109/ICDCS.2016.98 (§3.3).

[4] Remzi Can Aksoy and Manos Kapritsos. 2019. Aegean: replication beyond the client-server model. In *27th ACM Symposium on Operating Systems Principles (SOSP '19)*. https://doi.org/10.1145/3341301.3359663 (§1 and 8).

[5] Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction. In *8th ACM European Conference on Computer Systems (EuroSys '13)*. https://doi.org/10.1145/2465351.2465361 (§3.3).

[6] Amazon Web Services. 2020. Implementing version control using Amazon DynamoDB. https://aws.amazon.com/blogs/database/implementing-version-control-using-amazon-dynamodb/ (§6.1).

[7] Amazon Web Services. 2022. Amazon Aurora Global Database. https://aws.amazon.com/rds/aurora/global-database/ (§7.4).

[8] Amazon Web Services. 2022. Amazon DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html (§6.4).

[9] Amazon Web Services. 2022. Amazon S3. https://aws.amazon.com/s3/features/replication/ (§7.4).

[10] Antipode. 2023. Artifacts. https://github.com/Antipode-SOSP23 (§6.4).

[11] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The new SQL server in the cloud. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'19)*. https://doi.org/10.1145/3299869.3314047 (§6.1).

[12] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *VLDB Endowment* 8, 3 (2014), 185–196. https://doi.org/10.14778/2735508.2735509 (§3.4 and 5.1).

[13] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2012. The potential dangers of causal consistency and an explicit solution. In *3rd ACM Symposium on Cloud Computing (SoCC '12)*. https://doi.org/10.1145/2391229.2391251 (§3.2 and 5.1).

[14] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. https://doi.org/10.1145/2463676.2465279 (§1, 5.1, 6, and 6.4).

[15] Philip A Bernstein, Nathan Goodman, and Vassos Hadzilacos. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. https://dl.acm.org/doi/book/10.5555/17299 (§8).

[16] Kenneth P. Birman and Robbert van Renesse. 1994. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press. https://doi.org/10.1109/M-PDT.1996.532142 (§3.1).

[17] Manuel Bravo, Nuno Diegues, Jingna Zeng, Paolo Romano, and Luís Rodrigues. 2015. On the use of Clocks to Enforce Consistency in the Cloud. *IEEE Computer Society Technical Committee on Data Engineering* 38, 1 (2015), 18–35. https://dblp.org/rec/journals/debu/BravoDZRR15.html (§3.2 and 5.1).

[18] Manuel Bravo, Luis Rodrigues, and Peter Van Roy. 2017. Saturn: a Distributed Metadata Service for Causal Consistency. In *11th European Conference on Computer Systems (EuroSys '17)*. https://doi.org/10.1145/3064176.3064210 (§3.2 and 3.3).

[19] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. 2007. Whodunit: Transactional Profiling for Multi-tier Applications. *ACM SIGOPS Operating Systems Review* 41, 3 (2007), 17–30. https://doi.org/10.1145/1272998.1273001 (§4.1).

[20] Bernadette Charron-Bost. 1991. Concerning the size of logical clocks in distributed systems. *Inform. Process. Lett.* 39, 1 (1991), 11–16. https://doi.org/10.1016/0020-0190(91)90055-M (§3.2 and 5.1).

[21] David R. Cheriton and Dale Skeen. 1993. Understanding the limitations of causally and totally ordered communication. *ACM SIGOPS Operating Systems Review* 27, 5 (1993), 44–57. https://doi.org/10.1145/173668.168623 (§3.2).

[22] Chris Richardson. 2021. Microservices.io. https://microservices.io/ (§8).

[23] Jeremy Cloud. 2013. Decomposing Twitter: Adventures in Service-Oriented Architecture. In *QConNY'13*. https://www.infoq.com/presentations/twitter-soa/ (§1 and 2.1).

[24] Adrian Cockcroft. 2014. Migrating to Cloud Native with Microservices. In *GOTO Conference '14*. http://gotocon.com/dl/goto-berlin-2014/slides/AdrianCockcroft_MigratingToCloudNativeWithMicroservices.pdf (§1 and 2.1).

[25] Cockroach Labs. 2023. CockroachRB: Transaction Layer. https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer (§6.1).

[26] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *4th ACM Symposium on Cloud Computing (SoCC '13)*. https://doi.org/10.1145/2523616.2523628 (§3.3).

[27] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks Jiaqing. In *5th ACM Symposium on Cloud Computing (SoCC '14)*. https://doi.org/10.1145/2670979.2670983 (§3.2 and 3.3).

[28] Eventuate. 2021. Eventuate. https://eventuate.io/ (§8).

[29] Facebook Help Community (Entry now inaccessible) Retrieved 2017-06-03. 2017. Anyone know why I can click on a post and I get the page not found? (§1).

[30] Facebook Help Community (Entry now inaccessible) Retrieved 2017-06-03. 2017. Notification links with picture only brings to page not found.

[31] Facebook Help Community (Entry now inaccessible) Retrieved 2017-06-03. 2017. Why am I not receiving all of my notifications on posts that I comment on?

[32] Facebook Help Community (Entry now inaccessible) Retrieved 2017-06-03. 2017. Why when I get notifications but then not showing up on my page? (§1).

[33] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: a pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework (§4.1).

[34] Martin Fowler. 2015. Microservice Trade-Offs. https://martinfowler.com/articles/microservice-trade-offs.html (§1, 2.1, 3.3, and 8).

[35] Jonas Fritzsch, Justus Bogner, Stefan Wagner, and Alfred Zimmermann. 2019. Microservices Migration in Industry: Intentions, Strategies, and Challenges. In *EEE International Conference on Software Maintenance and Evolution (ICSME '19)*. https://doi.org/10.1109/ICSME.2019.00081 (§8).

[36] Yu Gan, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Yanqi Zhang, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, Christina Delimitrou, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, and Brian Ritchken. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. https://doi.org/10.1145/3297858.3304013 (§1, 2.1, and 7).

[37] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. *ACM SIGMOD Record* 16, 3 (1987), 249–259. https://doi.org/10.1145/38714.38742 (§8).

[38] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. 2023. Towards Modern Development of Cloud Applications. In *19th Workshop on Hot Topics in Operating SystemsJune 2023 (HotOS '23)*. https://doi.org/10.1145/3593856.3595909 (§3.3).

[39] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (ATC '10)*. https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems (§8).

[40] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Vekataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *26th ACM Symposium on Operating Systems Principles (SOSP '17)*. https://doi.org/10.1145/3132747.3132749 (§3.2, 5.1, and 6).

[41] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems* 10, 4 (11 1992), 360–391. https://doi.org/10.1145/138873.138877 (§5.1).

[42] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data management in microservices. *VLDB Endowment* 14, 13 (2021), 3328–3361. https://doi.org/10.14778/3484224.3484232 (§1, 2.1, and 3.3).

[43] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563 (§3.1, 4.2, 4.2, 4.2, and 5.1).

[44] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li (§3.3).

[45] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *23rd ACM Symposium on Operating*

Systems Principles (SOSP '11). https://doi.org/10.1145/2043556.2043593 (§6.1).

[46] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd (§3.3).

[47] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*. https://doi.org/10.1145/2815400.2815426 (§3.3).

[48] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *2021 ACM Symposium on Cloud Computing (SoCC '21)*. https://doi.org/10.1145/3472883.3487003 (§1, 2.1, 3.1, 3.2, 3.3, and 4.1).

[49] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *13th European Conference on Computer Systems (EuroSys '18)*. https://doi.org/10.1145/3190508 (§6).

[50] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic causal monitoring for distributed systems. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*. https://doi.org/10.1145/2815400.2815415 (§1, 3.2, 4.1, 5.1, 6, and 6.2).

[51] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi (§3.2 and 5.1).

[52] MongoDB. 2021. Replication Lag Causes. https://docs.mongodb.com/manual/tutorial/troubleshoot-replica-sets (§7.3).

[53] OpenTelemetry. 2021. OpenTelemetry. https://opentelemetry.io/ (§3.2, 5.1, 6, and 6.4).

[54] Austin Parker, Daniel Spoonhower, Jonathan Mace, Rebecca Isaacs, and Ben Sigelman. 2020. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media. https://www.oreilly.com/library/view/distributed-tracing-in/9781492056621/ (§1, 4.1, and 6).

[55] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. 2014. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. In *33rd International Symposium on Reliable Distributed Systems Workshops (SRDSW '14)*. https://doi.org/10.1109/SRDSW.2014.33 (§3.3).

[56] Raja R. Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R. Ganger. 2014. *So, you want to trace your distributed system? Key design insights from years of practical experience*. Technical Report. Parallel Data Laboratory - Carnegie Mellon University. https://www.pdl.cmu.edu/PDL-FTP/SelfStar/CMU-PDL-14-102.pdf (§3.2, 5.1, and 6).

[57] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *Comput. Surveys* 22, 4 (1990), 299–319. https://doi.org/10.1145/98163.98167 (§8 and A).

[58] Malte Schwarzkopf. 2015. *Operating system support for warehouse-scale computing*. Ph. D. Dissertation. University of Cambridge. https://doi.org/10.17863/CAM.26443 (§1 and 2.1).

[59] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. 2020. FlightTracker: Consistency across read-optimized online stores at Facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. https://www.usenix.org/conference/osdi20/presentation/shi (§1, 2.1, 3.4, 6.1, and 8).

[60] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google. https://research.google/pubs/pub36356/ (§3.2, 4.1, 5.1, and 6).

[61] Michael Stonebraker and Uğur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *21st International Conference on Data Engineering (ICDE '05)*. https://doi.org/10.1109/ICDE.2005.1 (§8).

[62] Doug Terry. 2013. Replicated data consistency explained through baseball. *Commun. ACM* 56, 12 (2013), 82–89. https://doi.org/10.1145/2500500 (§6.4).

[63] Cory G. Watson. 2013. Observability at Twitter. https://blog.twitter.com/engineering/en_us/a/2013/observability-at-twitter.html (§3.2, 5.1, and 6).

[64] Irene Zhang, Niel Lebeck, Ariadna Norberg, Pedro Fonseca, Brandon Holt, Raymond Cheng, Arvind Krishnamurthy, and Henry M Levy. 2016. Diamond: Automating Data Management and Storage for Wide-area, Reactive Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-irene (§8).

[65] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou (§1 and 2.1).

[66] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Transactions on Software Engineering* 22, 4 (2021), 243–260. https://doi.org/10.1109/TSE.2018.2887384 (§1, 7, 7.1, and 7.4).

A System Model

We model the system as a collection of N_p processes that collectively implement N_s services.

Processes are modelled as state machines [57] that can perform five types of actions: (a) perform a LOCAL computation, (b) SEND a message to another process, (c) RECEIVE a message from another process, (d) INVOKE an operation on a service, or (e) read the corresponding REPLY from a service. We define an execution of a process to be a sequence of states and actions: $s_0, a_1, s_1, a_2, s_2, \cdots$ obeying the state machine specification for that process.

Services, in turn, are available through a request/response interface. Since we may not know how a given service is implemented internally, we model a service as a black box. However, an opaque service cannot be simply modeled as a state machine, since its implementation may cause behaviors that differ from a linear sequence of transitions between states. For example, in an eventually consistent storage service, the invocation of a write operation does not imply that subsequent operations will see that state change. As such, services support two operations with the following semantics: (a) receiving an INVOKE action (k_n), and (b) conveying the corresponding REPLY action (e_n) to the process, where the execution of a service is a sequence of (paired but not necessarily alternating) invocations and replies $k_0, k_1, e_0, k_2, e_1, e_2, \ldots$, with the specification that each reply reflects *some subset* of the preceding invocations of the sequence. In the definition of XCY, we further specialize the existence of a storage service with a `read/write` interface, which allows us to provide a definition that is better suited for the consistency of replicated data, in the same way as in the original causal memory definition [1].

B Lineages

We define a lineage as a partial order of dependent actions that stem from an initial ROOT action and end in one or more STOP actions. The ROOT action corresponds to the initial invocation of the application (e.g., a client request, or the launching of a periodic job). A STOP action marks the end of the handling of an external invocation at each process. We formally define a lineage as follows:

Definition. The *lineage* of a given ROOT action r_i, denoted as $\mathcal{L}_{r_i}$ is a directed acyclic graph (DAG) defined as follows[2]:

1. The vertex r_i is the single root of $\mathcal{L}_{r_i}$.
2. $a_1 \rightarrow a_2 \in \mathcal{L}_{r_i}$, if a_1 (a vertex of $\mathcal{L}_{r_i}$ and $\neq$ STOP) precedes a_2 in the execution of a process p.
3. $a_1 \rightarrow a_2 \in \mathcal{L}_{r_i}$, if a_1 (a vertex of $\mathcal{L}_{r_i}$) is a SEND from process p_1 and a_2 is the corresponding RECEIVE at process p_2, where both p_1 and p_2 belong to the same service S.

[2]To simplify our model, we assume that when a process receives a message, it handles the corresponding request without interleaving it with the processing of other messages until the STOP action, allowing for a one to one mapping between actions and lineages.

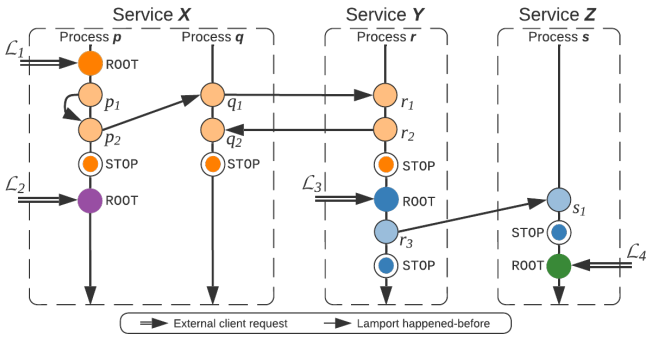

Figure 10. Depiction of our lineage definition.

4. $k \rightarrow a_k \in \mathcal{L}_{r_i}$ if k (a vertex of $\mathcal{L}_{r_i}$) is the INVOKE action from service S to service R, and a_k is the action corresponding to the service execution at R.
5. $e \rightarrow a_e \in \mathcal{L}_{r_i}$ if e (a vertex of $\mathcal{L}_{r_i}$) is the REPLY to a previous INVOKE on service R, and a_e is the corresponding action of the REPLY at the caller service S.

Fig. 10 depicts an example of the previous definition: (1) each lineage begins with a single root from an external client request; (2) any successive LOCAL actions within the same process ($p_1 \rightarrow p_2$), belong to the same lineage; (3) successive actions between different processes of the same service ($p_2 \rightarrow q_1$), belong to the same lineage; (4) INVOKE related actions ($q_1 \rightarrow r_1$), belong to the same lineage; and (5) REPLY related actions ($r_2 \rightarrow q_2$), belong to the same lineage. Lineage $\mathcal{L}_1$ is delimited by STOP actions at processes p, q and r.