Writes: the dirty secret of causal consistency

Lorenzo Alvisi[‡] Natacha Crooks[†‡] Syed Akbar Mehdi[†]
[†]The University of Texas at Austin [‡]Cornell University

Abstract

Causal consistency offers geo-distributed systems what ought to be a sweet option between the poor performance of strong consistency and the weak guarantees of eventual consistency. Yet, despite its appealing properties, causal consistency has seen limited adoption in industry, where systems have instead been clustering around the two extremes of eventual and strong consistency. We argue that this reluctance stems primarily from how causal consistency handles writes—both in how they are propagated, and in how conflicting writes are applied. We present our experience in designing and building two recent systems, Occult [25] and TARDiS [11], that try to address or mitigate these problems, and highlight some of the open challenges that remain in this space.

1 Introduction

Causal consistency appears to be ideally positioned to respond to the needs of geographically replicated datastores that support today's large-scale web applications. It strikes a sweet point between the high latency of stronger consistency guarantees [10], and the programming complexity of eventual consistency [4, 7, 27]. Indeed, unlike eventual consistency, causal consistency preserves operation ordering and gives Alice assurance that Bob, whom she has defriended before posting her Spring-break photos, will not be able to access her pictures, even though Alice and Bob access the photo-sharing application using different replicas [5, 9, 20]. Crucially, causal consistency provides these guarantees while continuing to provide applications the ALPS properties [20] of availability, low latency, partition tolerance and high scalability.

These appealing properties have generated much interest in the research community. In the last few years, we have learned that no guarantee stronger than real-time causal consistency can be provided in a replicated datastore that combines high-availability with convergence [23], and that, conversely, it is possible to build causally-consistent datastores that can efficiently handle a large number of shards [3, 6, 15, 16, 20, 21]. Likewise, we have established increasingly sophisticated techniques [11, 14, 20, 23, 24, 32, 34] to ensure the convergence of objects to which conflicting updates have been applied updates.

Perhaps surprisingly, the interest has been rather more tepid in industry: to this day, causal consistency is rarely used in production, with companies preferring either strictly weaker guarantees like those provided in Riak [7], Redis Cluster [1], and Cassandra [4], or, at the other extreme, stronger guarantees like those offered by Google Spanner [10], FaunaDB [17], or CockroachDB [19]. As of the time of writing, the only commercially

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

available systems that report supporting causal consistency are AntidoteDB [30][1], Neo4j [28][2], MongoDB [27][3] and CosmosDB [26]. This, despite Facebook suggesting that causal consistency is a feature that they would like to support [2, 22]! We submit that industry's reluctance to deploy causal consistency is in part explained by the inability of its current implementations to handle *write operations* efficiently, while aligning with the semantics of the applications that run on top of these systems. Unlike reads, which can be served from any "up-to-date" replica [20,21,33] and return a state that includes the effects of all operations that could have influenced that state, writes come with few promises: existing causally consistent systems neither guarantee when a given write will eventually become visible, nor how merging conflicting writes on individual objects will affect the semantics of the global system state.

Write visibility In existing causal systems, such as COPS [20] or Eiger [21], a datacenter performs a write operation only after applying all writes that causally precede it. This approach guarantees that reads never block, as all replicas are always in a causally consistent state, but, in the presence of slow or failed shards, may cause writes to be buffered for arbitrarily long periods of time. These failures, common in large-scale clusters, can lead to the *slowdown cascade* phenomenon, where a single slow or failed shard negatively impacts the entire system, delaying the visibility of updates across many shards and leading to growing queues of delayed updates [2, 25]. Slowdown cascades thus violate a basic commandment for scalability: do not let your performance be determined by the slowest component in your system.

Merging of write-write conflicts Geographically distinct replicas can issue conflicting operations, and the write-write conflicts that result from these operations may cause replicas' states to diverge. To insulate applications from this complexity, systems like COPS [20] or Dynamo [14] attempt to preserve the familiar abstraction that an application evolves sequentially through a linear sequence of updates: they aggressively enforce per-object convergence, either through simple deterministic resolution policies, or by asking the application to resolve the state of objects with conflicting updates as soon as conflicts arise. These techniques, however, provide no support for meaningfully resolving conflicts between concurrent sequences of updates that involve *multiple* objects: in fact, they often destroy information that could have helped the application in resolving those conflicts. For example, as we describe further in §2.2, deterministic writer-wins [35], a common technique to achieve convergence, hides write-skew from applications, preventing applications from correcting for the anomaly. Similarly, exposing multivalued objects without context as in Dynamo [14] obscures cross-object semantic dependencies.

In the rest of this paper, we discuss in greater detail the effect of slowdown cascades on existing causal systems (§2.1), and the semantic challenges that are left unaddressed by current techniques for merging concurrent conflicting write operations (§2.2). We then sketch the outline of Occult (§3.1) and TARDiS (§3.2), two systems designed to mitigate the aforementioned issues, and conclude (§4) by highlighting the challenges that arise when attempting to combine insights from these two systems.

2 The challenges

2.1 Propagating writes

Causal consistency derives its performance advantage over stronger consistency guarantees from its ability to asynchronously propagate and apply writes to replicas. The flip side of this flexibility is the unpredictability of the relative timing at which causally dependent updates to different shards are ultimately applied at a remote replica. Not only causally dependent updates may reach distinct remote shards out of order, but updates may be arbitrarily delayed when shards experience performance anomalies (because of abnormally-high read or write

[1]Currently in alpha release.
[2]As of release 3.1.
[3]As of the release 3.6.

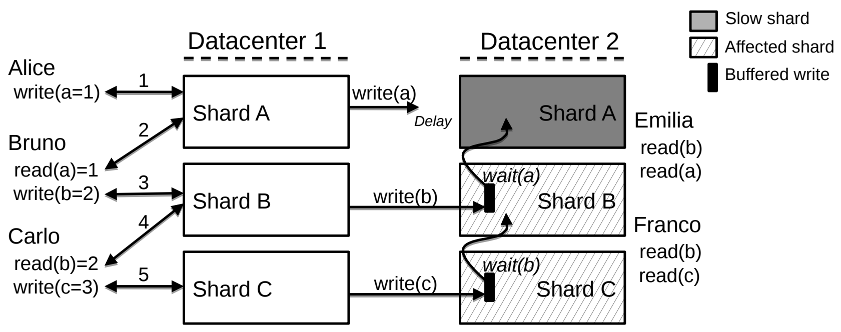

Figure 1: Example of a slowdown cascade in traditional causal consistency. Delayed replicated write(a) delays causally dependent replicated write(b) and write(c)

traffic, partially malfunctioning hardware, a congested top-of-rack switch, etc.) that are not only inevitable but indeed a routine occurrence in any system of sufficient size [12, 13].

Ideally, any drawback caused by such delays would be limited to the affected shard, and not spill over to the rest of the system. Unfortunately, to the best of our knowledge, all existing causally consistent systems [6, 16, 20, 21, 38] are *inherently* susceptible to spill overs, which ultimately can snowball into a system-wide *slowdown cascade*.

This vulnerability is fundamental to the design of these systems: they delay applying a write until after all writes that causally precede it have been applied. For example, Eiger [21] asks each replicated write w to carry metadata that explicitly identifies all writes that happened before w and have a single-hop distance from w in the causal dependency graph. When a replica receives w, it delays applying it until after all of w's dependencies have been applied; these dependencies in turn are delayed until *their* single-hop dependencies are applied; and so on). The visibility of a write within a shard can then become dependent on the timeliness of other shards in applying their own writes. Figure 1 illustrates this with an example. Shard A of DC_2 lags behind in applying the write propagating from DC_1, all shards in DC_2 must also wait before they make their writes visible. Shard A's limping inevitably affects Emilia's query, but also unnecessarily affects Franco's, which accesses exclusively shards B and C.

Industry has long identified the spectre of slowdown cascades as one of the leading reasons behind its reluctance to build strongly consistent systems [2, 8], pointing out how the slowdown of a single shard, compounded by *query amplification* (e.g., a single user request in Facebook can generate thousands of, possibly dependent, internal queries to many services), can quickly cascade to affect the entire system.

Figure 2 shows how a single slow shard affects the size of the queues kept by Eiger [21] to buffer replicated writes. Our setup is geo-replicated across two datacenters in Wisconsin and Utah, each running Eiger sharded across 10 physical machines. We run a workload consisting of 95% reads and 5% writes from 10 clients in Wisconsin and a read-only workload from 10 clients in Utah. We measure the average length of the queues buffering replicated writes in Utah. Larger queues mean that newer replicated writes take longer to be applied. If all shards proceed at approximately the same speed, the average queue length remains stable. However, if *any* shard cannot keep up with the arrival rate of replicated writes, then the average queue length across *all* shards grows indefinitely.

2.2 Merging conflicting writes

Propagating writes is not the only challenge that causal consistency faces: in multi-master systems (like COPS [20], Dynamo [14], Cassandra [4], Riak [7], or Voldemort [36]), conflicting writes can execute concurrently at different sites. The challenge is then not simply how to propagate these writes efficiently, but how to resolve

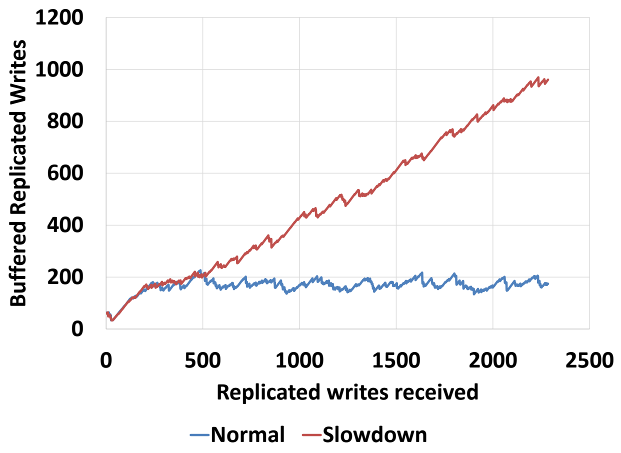

Figure 2: Average queue length of buffered replicated writes in Eiger under normal conditions and when a single shard is delayed by 100 ms.

the write-write conflict that arises from these operations. Though causal consistency elegantly guarantees that Emilia will observe a consistent state, causal consistency does not specify how to handle conflicting write operations. Write-write conflicts define a class of conflicts that causal consistency cannot prevent, detect, or repair.

To illustrate, consider the process of updating a Wikipedia page consisting of multiple HTML objects (Figure 3(a)). The page in our example, about a controversial politician, Mr. Banditoni, is frequently modified, and is thus replicated on two sites, A and B. Assume, for simplicity, that the page consists of just three objects—the content, references, and an image. Alice and Bruno, who respectively strongly support and strongly oppose Mr. Banditoni, concurrently modify the content section of the webpage on sites A and B to match their political views (Figure 3(b)). Carlo reads the content section on site A, which now favors Mr. Banditoni, and updates the reference section accordingly by adding links to articles that praise the politician. Similarly, Davide reads the update made by Bruno on site B and chooses to strengthen the case made by the content section by updating the image to a derogatory picture of Mr. Banditoni (Figure 3(c)). Eventually, the operations reach the other site and, although nothing in the preceding sequence of events violates causal consistency, produce the inconsistent state shown in Figure 3(d): a content section that exhibits a write-write conflict; a reference section in favor of Mr. Banditoni; and an image that is against him. Worse, there is no straightforward way for the application to detect the full extent of the inconsistency: unlike the explicit conflict in the content sections, the discrepancy between image and references is purely semantic, and would not trigger an automatic resolution procedure. To the best of our knowledge, this scenario presents an open challenge to current causally consistent systems. We conjecture that these systems struggle to handle such write-write conflicts for two reasons: they choose to syntactically resolve conflicts, and do not consider cross-object semantics.

Syntactic conflict resolution. We observe that the majority of weakly consistent systems use fixed, syntactic conflict resolution policies to reconcile write-write conflicts [4, 20] to maintain the abstraction of sequential storage. Returning to our previous example of the popular merging policy of deterministic writer-wins (DWW): DWW resolves write-write conflicts identically at all sites and ensures that applications never see conflicting writes, which guarantees eventual convergence. In our example, however, guaranteeing eventual convergence would not be sufficient to restore consistency: this policy would choose Bruno's update, and ignore the relationship between the content, references, and images of the webpage. Such greedy attempt at syntactic conflict resolution is not only inadequate to bridge this semantic gap, but in fact can also lose valuable information for reconciliation (here, Alice's update).

Lack of cross-object semantics. Some systems, like Dynamo [14] or Bayou [34], allow for more expressive

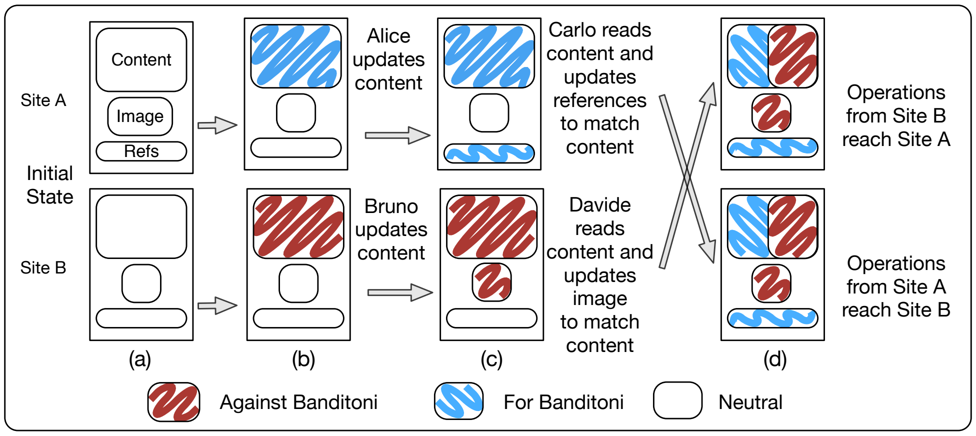
Site A
Initial State
Site B
Content
Image
Refs
Alice updates content
Carlo reads content and updates references to match content
Bruno updates content
Davide reads content and updates image to match content
Operations from Site B reach Site A
Operations from Site A reach Site B
(a)
(b)
(c)
(d)
Against Banditoni
For Banditoni
Neutral

Figure 3: Weakly-consistent Wikipedia scenario - A webpage replicated on two sites with asynchronous replications. The end state is a write-write conflict on the content and an inconsistent web-page

conflict resolution policies by pushing to conflict resolution to the application [14, 34], but on a per-object basis only. This approach allows for more flexible policies than a purely syntactic solution, but reduces conflict resolution to the merging of explicitly conflicting writes. As a result, it is often still overly narrow. For example, it would not detect or resolve inconsistencies that are not write-write conflicts, but instead result indirectly from conflicts between two writes, such as the one between the references and the image. Per-object reconciliation policies fail to consider that the effects of a write-write conflict on an object do not end with that object: Carlo and Davide update references and images as they do because they have *read* the conflicting updates to the original content section. Indeed, any operation that depends on one of two conflicting updates is potentially incompatible with all the operations that depend on the other: the shockwaves from even a single write-write conflict may spread to affect the state of the entire database.

To the best of our knowledge, there is currently no straightforward way for applications to resolve consistently the kind of multi-object, indirect conflicts that our example illustrates. Transactions [20, 21], an obvious candidate, are powerless when the objects that directly or indirectly reflect a write-write conflict are updated, as in our example, by different users. After Bruno's update, the application has no way to know that Davide's update is forthcoming: it must therefore commit Bruno's transaction, forcing Bruno's and Davide's updates into separate transactions. Nor would it help to change the granularity of the object that defines the write-write conflict—in our example, by making that object be the entire page. It would be easy to correspondingly scale up the example, using distinct pages that link each other. Short of treating the entire database as the "object", it is futile to try to define away these inconsistencies by redrawing the objects' semantic boundaries.

In essence, conflicting operations fork the entire state of the system, creating distinct *branches*, each tracking the linear evolution of the datastore according to a separate thread of execution or site. The Wikipedia for example, consists of two branches, one in support of Banditoni at Site A, and one against the politician (Site B).

3 Towards a solution?

3.1 Preventing slowdown cascades through client-centric causal consistency

The example (Figure 1 in §2.1) illustrates why causal consistency can be subject to slowdown cascades despite replicating writes asynchronously: writes share the fate of all other writes on which they causally depend. If one write is slow to replicate, all subsequent writes will incur that delay. Though this delay may sometimes be necessary - Emilia must wait for the delayed write to observe a causally consistent snapshot of the system -

inheriting the delays of causally preceding writes can also introduce gratuitous blocking. Franco, for instance, never reads Alice's write (a): delaying writes (b) and (c) until (a) is replicated is thus unnecessary. Otherwise said, observing writes (b) and (c) while write (a) is in flight does not lead to a consistency violation.

This observation, though seemingly fairly innocent, is in fact key to alleviating the problem of slowdown cascades in causal consistency, as it precisely captures what causal consistency *requires*. Causal consistency defines a contract between the datastore and its users that specifies, for a given set of updates, which values the datastore is allowed to return in response to user queries. In particular, it guarantees that each client observes a monotonically non-decreasing set of updates (including its own), in an order that respects potential causality between operations. Causal consistency thus mandates that Franco, upon observing write (c), also observes write (b), but remains silent on the fate of write (a). Existing causally consistent systems, however, enforce internally a stronger invariant than what causal consistency requires: to ensure that clients observe a monotonically non-decreasing set of updates, they evolve their data store *only* through monotonically non-decreasing updates. It is this strengthening that leaves current implementations of causal consistency vulnerable to slowdown cascades.

To resolve this issue, we propose to revisit what implementing causal consistency actually requires: a system is causally consistent from the point of view of a client if all executed queries return results that are consistent with a causal snapshot of the system. The underlying system itself need never store a causally consistent state, as long as it appears *indistinguishable* from a system that does!

To this effect, we designed a system, Occult (**O**bservable **C**ausal **C**onsistency Using **L**ossy **T**imestamps) that shifts the responsibility for the enforcement of causal consistency from the datastore to those who actually perceive consistency anomalies—the clients. Moving the output commit to clients allows Occult to make its updates available as soon as it receives them, without having to first apply all causally preceding writes. Causal consistency is then enforced by clients on reads, but only for those updates that they are actually interested in observing. In our example, Occult empowers Franco to independently determine that the result of its query is causally consistent: in general, Occult clients can access states of replicas that may not yet reflect some of the (unrelated) writes that were already reflected in a replica they had previously accessed.

Taking this read-centric approach to causal consistency may seem like a small step, but pays big dividends: as Occult is no longer compelled to delay writes to enforce consistency, Occult is impervious to slowdown cascades. This approach also conveys other benefits: Occult, for instance, no longer needs its clients to be sticky, a flexibility that is useful in real-world systems like Facebook, where clients sometimes must bounce between datacenters because of failures, load balancing, and/or load testing [2]).

At first blush, moving the enforcement of causal consistency to clients may appear fairly straightforward. Each client c in Occult maintains metadata to encode the most recent state of the datastore that it has observed. On reading an object o, c determines whether the version of o that the datastore currently holds is safe to read (i.e., if it reflects all the updates encoded in c's metadata). The datastore keeps, together with o, metadata of its own to encode the most recent state known to the client that created that version of o. If the version is deemed safe to read, then c needs to update its metadata to reflect any new dependency; if it is not, then c needs to decide how to proceed (among its options: try again; contact a master replica guaranteed to have the latest version of o; or trade safety for availability by accepting a stale version of o).

In reality, however, implementing client-centric causal consistency is non-trivial: the size of this metadata can quickly grow to have prohibitive cost. Vector clocks, the traditional way of capturing causal dependencies, require in principle an entry per each tracked object, a clearly unacceptable proposition in the large-scale systems that Occult targets. It is to sidestep this issue that current causally consistent datastores prefer the pessimistic approach to causal consistency: if a write can never be applied to a replica until all previous causally related writes have been replicated, it is sufficient to track the *nearest-writes* on which an operation depends [5, 20]. The core logic of Occult is thus geared towards minimizing metadata size while retaining the flexibility and resiliency to slowdown cascades that the read-centric approach conveys.

Occult makes this possible through a core technique: *compressed causal timestamps*. Causal timestamps are, essentially, vector clocks that, rather than tracking dependencies at the granularity of individual objects, do

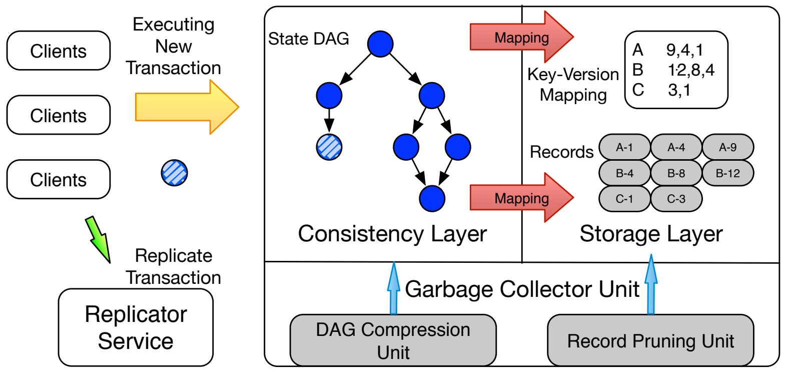

Figure 4: TARDiS architecture

so at the granularity of datacenter shards: each timestamp entry identifies the number of known writes from the corresponding shard. Occult uses causal timestamps to (i) encode the most recent state of the datastore observed by a client and (ii) capture the set of causal dependencies for write operations. Though conflating dependency tracking for all objects stored in a shard to to a single entry reduces overhead, they are still far from practical in large scale systems, which can have tens of thousands of shards. The challenge that Occult takes on is to devise a way to *compress* causal timestamps without significantly reducing their ability to track causal dependencies accurately.

To this end, Occult synthesizes a number of techniques that leverage structural and temporal properties to strike a sweet spot between metadata overhead and accuracy: for instance, using real-time rather than counters as entries in the causal timestamps allows Occult to eliminate timestamp inaccuracies that result from different shards seeing different write throughput. Likewise, Occult observes that more recently updated entries in the causal timestamp are more likely to generate spurious dependencies than older ones. Rather than using compression techniques that apply uniformly to all entries of a causal timestamp, Occult focuses the majority of its metadata budget to accurately resolve dependencies on the shards with the most recently updated vector entry. Specifically, clients assign an individual entry in their causal timestamp to the $n - 1$ shards with the most recently updated vector entry they have observed; all other shards are mapped to the vectors "catch-all" last entry. Though very coarse, conflating to a single timestamp entry the tracking of all the shards that have not been recently updated is likely to cause few consistency check failures: with high likelihood, that timestamp entry naturally reflects updates that have already had enough time to be accepted by every replica. The conference paper describing Occult [25] offers more details about these techniques and discusses the system's transactional capabilities.

Our experience suggests that Occult performs well: we find that our prototype of Occult, when compared with the eventually-consistent system (Redis Cluster) it is derived from, increases the median latency by only $50\mu s$, the 99th percentile latency by only $400\mu s$ for a read-heavy workload ($4ms$ for a write-heavy workload), and reduces throughput by less than 10%. More importantly, we find that a four-entry causal timestamp suffices to achieve an accuracy of 99.6% for a cluster with over 16,000 logical shards.

3.2 Improving merging through branches

As suggested in §2.2, conflicting operations fork the entire state of the system, creating distinct *branches*. These branches have traditionally been hidden or greedily merged by systems' syntactic and per-object resolution strategies, that try, at all cost, to maintain the abstraction of a sequential view of the world. The lack of support for enforcing the cross-object consistency demands expressed in many application invariants thus makes conflict resolution more difficult and error-prone, as our Wikipedia example highlights.

Attempting to isolate applications from the reality of conflicting write operations is therefore, we believe, a well-intentioned fallacy. Conflicting writes "contaminate" data in a way that the storage system cannot understand: application logic is often indispensable to resolve those conflicts. Replicas, however, only see a sequence of read/write operations and are unaware of the application-logic and invariants that relate these operations. We consequently argue that the storage system should avoid deterministic quick fixes, and instead give applications the information they need to decide what is best. The question becomes: how can one provide applications with the best possible system support when merging conflicting states.

To answer this question, we designed TARDiS [11], an asynchronously replicated, multi-master key-value store designed for applications built above weakly-consistent systems. TARDiS renounces the one-size-fits-all abstraction of sequential storage and instead exposes applications, when appropriate, to concurrency and distribution. TARDiS' design is predicated on a simple notion: *to help developers resolve the anomalies that arise in such applications, each replica should faithfully store the full context necessary to understand how the anomalies arose in the first place, but only expose that context to applications when needed.* By default in TARDiS, applications execute on a branch, and hence perceive storage as sequential. But when anomalies arise, TARDiS reveals to the users the intricate details of distribution. TARDiS hence gives applications the flexibility of deciding if, when, and how divergent branches should be merged.

TARDiS provides two novel features that simplify reconciliation. First, it exposes applications to the resulting independent branches, and to the states at which the branches are created (fork points) and merged (merge points). Second, it supports atomic merging of conflicting branches and lets applications choose when and how to reconcile them. Branches, together with their fork and merge points, naturally encapsulate the information necessary for semantically meaningful merging: they make it easy to identify all the objects to be considered during merging and pinpoint when and how the conflict developed. This context can reduce the complexity and improve the efficiency of automated merging procedures, as well as help system administrators when user involvement is required. Returning to our Wikipedia example, a Wikipedia moderator presented with the two conflicting branches would be able to reconstruct the events that led to them and handle the conflicting sources according to Wikipedias guidelines [37]. Note that merging need not simply involve deleting one branch. Indeed, branching and merging states enables merging strategies with richer semantics than aborts or rollbacks [32]. It is TARDiS's richer interface that gives applications access to content that is essential to reasoning about concurrent updates, reducing the complexity of programming weakly consistent applications. In many ways, TARDiS is similar to Git [18]: users operate on their own branch and explicitly request (when convenient) to see concurrent modifications, using the history recorded by the underlying branching storage to help them resolve conflicts.

Unlike Git, however, branching in TARDiS does not rely on specific user commands but occurs implicitly, to preserve availability in the presence of conflicts. Two core principles underpin TARDiS: branch-on-conflict and inter-branch isolation. Branch-on-conflict lets TARDiS logically fork its state whenever it detects conflicting operations and store the conflicting branches explicitly. Inter-branch isolation guarantees that the storage will appear as sequential to any thread of execution that extends a branch, keeping application logic simple. The challenge is then to develop a datastore that can keep track of independent execution branches, record fork and merge points, facilitate reasoning about branches and, as appropriate, atomically merge them – while keeping performance and resource overheads comparable to those of weakly consistent systems.

TARDiS uses multi-master asynchronous replication: transactions first execute locally at a specific site, and are then asynchronously propagated to all other replicas (§4). Each TARDiS site is divided into two components: a *consistency layer* and a *storage layer*:

- *Consistency layer* The consistency layer records all branches generated during an execution with the help of a directed acyclic graph, the *state DAG*. Each vertex in the graph corresponds to a logical state of the datastore; each transaction that updates a record generates a new state.

- *Storage Layer* The storage layer stores records in a disk-based B-tree. TARDiS is a multiversioned system: every update operation creates a new record version.

Much of TARDiS logic is geared towards efficiently mapping the consistency layer to the storage layer, and maintaining low metadata overhead. Two techniques are central to the system's design: *DAG compression* and *conflict tracking*:

- *DAG compression* TARDiS tracks the minimal information needed to support branch merges under finite storage. It relies on a core observation: most merging policies only require the fork points and the leaf states of a given execution. All intermediate states can be safely removed from the state DAG, along with the corresponding records in the storage layer.

- *Conflict tracking* To efficiently construct and maintain branches, TARDiS introduces the notion of conflict tracking. TARDiS summarizes branches as a set of fork points and merge points only (a *fork path*) and relies on a new technique, *fork point checking* to determine whether two states belong to the same branch. Fork paths capture *conflicts* rather than dependencies. As such, they remain of small size (conflicts represent a small percentage of the total number of operations), and allow TARDiS to track concurrent branches efficiently while limiting memory overhead. This is in contrast to the traditional dependency checking approach [16,20,24], which quickly becomes a bottleneck in causally consistent systems [5,16].

We do not attempt to further discuss the details of these techniques, and defer the reader to the corresponding SIGMOD paper [11]. Generally though, the system is promising: we find for example, that using TARDiS rather than BerkeleyDB [29] to implement CRDTs [31] – a library of scalable, weakly-consistent datatypes – cuts code size by half, and improves performance by four to eight times.

4 Future Work and Conclusion

This article highlighted one of the thornier aspects of causal consistency: the handling of writes. It focused on two problems, the merging of conflicting writes, and the implications of asynchronous write-propagation in the presence of failures. TARDiS and Occult are two systems that attempt to mitigate these issues. However, several open challenges remain: though Occult prevents slowdown cascades, it currently adopts a single-master architecture that prevents applications from executing writes at every datacenter, increasing latency and limiting availability. A multi-master Occult would be faced with the same challenge of merging conflicting writes that we previously outlined. Adding TARDiS's branches to Occult without reintroducing the danger of slowdown cascades is not straightforward: branching requires knowledge of a per-datacenter state DAG that is currently centralized. Naive approaches like sharding this datastructure may reintroduce the dangers of slowdown cascades. We believe that combining the approach of these systems is a promising avenue of future work.

Acknowledgements Occult and TARDiS are the result of a collaboration with many other co-authors: Nathan Bronson, Allen Clement, Nancy Estrada, Trinabh Gupta, Cody Littley, Wyatt Lloyd and Youer Pu. These projects were generously supported by grants from the National Science Foundation under grant number CNS-1409555, a Google Cloud Platform credit awards and a Google Faculty Research Award. Natacha Crooks was partially supported by a Google Doctoral Fellowship in Distributed Computing.

References

[1] Redis Cluster Specification. `http://redis.io/topics/cluster-spec`.

[2] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to Adopting Stronger Consistency at Scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland, 2015), HOTOS'15, USENIX Association.

[3] ALMEIDA, S., LEITÃO, J. A., AND RODRIGUES, L. Chainreaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic, 2013), EuroSys '13, ACM, pp. 85–98.

[4] APACHE. Cassandra. `http://cassandra.apache.org/`.

[5] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SOCC '12, pp. 22:1–22:7.

[6] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-On Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, 2013), SIGMOD '13, ACM, pp. 761–772.

[7] BASHO. Riak. `http://basho.com/products/`.

[8] BIRMAN, K., CHOCKLER, G., AND VAN RENESSE, R. Toward a Cloud Computing Research Agenda. *SIGACT News 40*, 2 (June 2009), 68–80.

[9] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[10] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pp. 251–264.

[11] CROOKS, N., PU, Y., ESTRADA, N., GUPTA, T., ALVISI, L., AND CLEMENT, A. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, ACM, pp. 1615–1628.

[12] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communications of the ACM 56*, 2 (Feb. 2013), 74–80.

[13] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, 2004), OSDI'04, USENIX Association, pp. 137–149.

[14] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pp. 205–220.

[15] DU, J., ELNIKETY, S., ROY, A., AND ZWAENEPOEL, W. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th ACM Symposium on Cloud Computing* (Santa Clara, California, 2013), SOCC '13, ACM, pp. 11:1–11:14.

[16] DU, J., IORGULESCU, C., ROY, A., AND ZWAENEPOEL, W. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pp. 4:1–4:13.

[17] FAUNADB. Faunadb. `https://fauna.com/`.

[18] GIT. Git: the fast version control system. `http://git-scm.com`.

[19] LABS, C. Cockroachdb. `https://www.cockroachlabs.com/`.

[20] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 401–416.

[21] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL, 2013), NSDI '13, USENIX Association, pp. 313–328.

[22] LU, H., VEERARAGHAVAN, K., AJOUX, P., HUNT, J., SONG, Y. J., TOBAGUS, W., KUMAR, S., AND LLOYD, W. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California, 2015), SOSP '15, ACM, pp. 295–310.

[23] MAHAJAN, P., ALVISI, L., AND DAHLIN, M. Consistency, availability, convergence. Tech. Rep. TR-11-22, Computer Science Department, University of Texas at Austin, May 2011.

[24] MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems 29*, 4 (2011), 12.

[25] MEHDI, S. A., LITTLEY, C., CROOKS, N., ALVISI, L., BRONSON, N., AND LLOYD, W. 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)* (Boston, MA, 2017), USENIX Association, pp. 453–468.

[26] MICROSOFT. Cosmosdb. `https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels`.

[27] MONGODB. Agility, Performance, Scalibility. Pick three. `https://www.mongodb.org/`.

[28] NEO4J. Neo4j. `https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/introduction/`.

[29] OLSON, M. A., BOSTIC, K., AND SELTZER, M. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99.

[30] REGAL, I. Antidote.

[31] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011.

[32] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 385–400.

[33] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 309–324.

[34] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pp. 172–182.

[35] THOMAS, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June 1979), 180–209.

[36] VOLDEMORT, P. Voldemort, a distributed database. `http://www.project-voldemort.com`.

[37] WIKIPEDIA. Wikipedia: Conflicting Sources. `http://en.wikipedia.org/wiki/Wikipedia:Conflicting_sources`.

[38] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada, 2015), Middleware '15, ACM, pp. 75–87.