Bilateral Anti-Entropy for Eventual Consistency

Rebecca Bilbro & Benjamin Bengfort
{rebecca,benjamin}@rotational.io
Rotational Labs, LLC
Queenstown, Maryland, USA

Peter Keleher
keleher@cs.umd.edu
University of Maryland Department of Computer Science
College Park, Maryland

Abstract

Eventually consistent systems are often more cost-effective to implement and maintain than their strongly consistent cousins. Gossip-based anti-entropy methods can be used to improve the consistency of such systems. However, observational data suggests that such improvements come at the cost of perennial data egress, significantly mitigating savings in a geo-replicated context. This paper presents observations collected from an eventually consistent system deployed in production clusters across the US, Germany, and Singapore. A bilateral anti-entropy process facilitates data synchronization for an application accessed globally and around-the-clock. Replication metrics such as latency, conflict, and the duration and productivity of anti-entropy sessions seem to indicate that it is possible to quantify tradeoffs between consistency and visibility latency such that staleness can be not only bounded, but fine-tuned.

Keywords: eventual consistency, geo-replication, anti-entropy replication, distributed systems

ACM Reference Format:
Rebecca Bilbro & Benjamin Bengfort and Peter Keleher. 2022. Bilateral Anti-Entropy for Eventual Consistency. In *Principles and Practice of Consistency for Distributed Data (PaPoC '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3517209.3524083

1 Introduction

While the literature on eventual consistency now extends back many decades, new and important use cases are emerging for globally connected systems (e.g. regulation, global health, machine learning/MLOps, cryptocurrency, etc) that are bringing these concepts to the foreground for applications developers. This paper presents lessons learned from implementing and running a geo-replicated deployed eventual consistency store for a production application serving users around the world. As architects and maintainers of the system, we have a unique opportunity to track and observe the patterns and behavior that emerge from the system's underlying bilateral anti-entropy routine which serves as its eventual consistency engine.

Anti-entropy is a mechanism for achieving eventual consistency by enabling remote peers to periodically synchronize, compare object versions, and make local repairs. Our implementation is inspired by the Bayou system [22] as well as subsequent research into gossip protocols [13] and their optimizations [16, 18]. More recently, we've worked on using adaptive methods to strengthen consistency via probabilistic guarantees [10].

Our interest in exploring replication conflicts in particular stems from the literature on conflict-free replicated data types [17, 21] as well as foundational concepts about coordination-free consistency measures [14]. In particular, we are interested in exploring observed conflict patterns during replication that can provide insights into the fine-tuning of consistency and staleness [7, 8] with respect to the cost-effectiveness of the overall system.

2 A Global Directory Service

This paper explores the behavior of the GDS ("Global Directory Service"), an eventually consistent system deployed in production clusters across the US, Germany, and Singapore

The GDS, shown in Figure 1, is a complete network; all nodes can replicate with any other node in the network. The GDS also produces full replication, meaning all nodes receive all objects. Writes are made by clients to replicas in all regions. GDS administrators, who generate a large proportion of writes, are located in the US, which results in a write imbalance.

The GDS replicated data store supports an application that facilitates secure, low-latency transfer of private beneficiary and sender compliance information for international cryptocurrency transactions regulated under the Travel Rule. Organizations that agree to adopt the Travel Rule Information Sharing Architecture (TRISA), a shared protocol for information transmission [15], may enroll themselves using a web application that enables a certificate authority via a series of APIs to generate, verify, and deliver their identity and signing certificates. Public certificates are stored together with organizational details to enable lookups and verification during cryptocurrency compliance transactions. Messages containing these secure envelopes are transmitted

PaPoC '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9256-3/22/04…$15.00
https://doi.org/10.1145/3517209.3524083

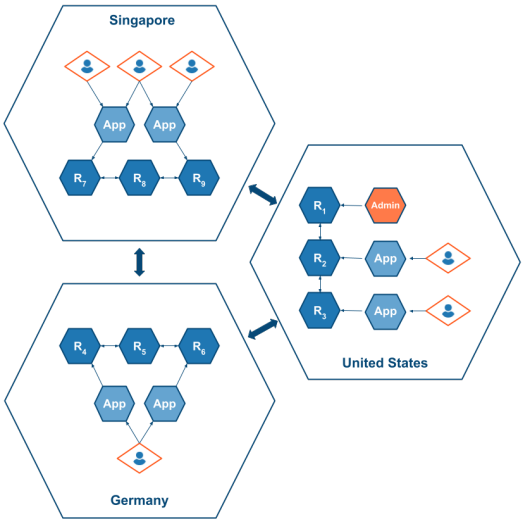

Figure 1. The GDS Architecture implements full replication and is a complete network.

via mutual authentication (mTLS) [5] connections through a suite of gRPC [3] services.

The Travel Rule only applies to international virtual asset transactions, so by definition the GDS is required to be a globally distributed network. Most geo-replicated systems [9, 11, 24, 25] prioritize scaling to large numbers of nodes for high throughput and wide-area durability. The GDS flips the script – as the facilitator of a peer-to-peer network, we prioritize lower latency via physical proximity (not to mention the compliance benefits of region-aware data storage). The size of the GDS replicated data store is therefore very small relative to its geographic footprint: our goal is to host primary and secondary nodes in separate availability zones per cloud region so that TRISA members querying the directory have the lowest possible latency for the best durability.

TRISA is an overlay on the blockchain; it was therefore a critical requirement that the GDS *not* participate directly in compliance information exchanges, merely facilitate them (lest it be considered a centralized service - the boogieman of cryptocurrency). The GDS has two primary roles: registering a member onto the network and locating and describing counter parties in a compliance exchange. Registration and amending directory records (e.g. the write workload of the system) happens rarely, usually once a year, and involves the coordination of only two parties: the Virtual Asset Service Provider ("VASP") itself and the TRISA administrators. Locating and describing counter parties in a compliance exchange is the vast majority of the GDS workload and involves several different kinds of reads: searches, index-based look ups, as well as full scans of the directory listing, shown in Figure 2. Because of this, we generally describe GDS as a read-heavy workload that can tolerate stale reads but must repair conflicting writes as soon as possible. While a stale read might lead to a retry of a travel rule information exchange, an invalid write could lead to a compliance failure; the former is more routine while the latter would be very harmful.

The GDS service is a gRPC API that exchanges data in the protocol buffer format [1, 3]. Because the directory service is primarily about looking up and returning specific records, the easiest optimization in the data store was to reduce the amount of data processing required and simply store the protocol buffer records in the database. The GDS therefore requires a fairly basic key/value document store with economical global replication and data storage, and this is what pushed us towards implementing our own data storage layer, an adapted version of leveldb [12, 20] that includes an indexing side car to support more flexible search. The result is a hybrid between a traditional key-value store and a document store implemented in the Go programming language.

The system is currently deployed as stateful sets in three independent Kubernetes clusters hosted by Google Kubernetes engine in Germany, Singapore, and the United States. Communication between replicas and from clients is secured via mTLS with no additional specialized networking. The pods themselves are fairly basic - each Pod requests 2.0 CPUs to support concurrent requests in multiple go routines as well as a background anti-entropy process. Memory and disk requests are 2Gi and 500GiB respectively. The primary variable cost is the networking cost of communicating between regions, which we've stabilized via tuning anti-entropy parameters, as described in this paper. In total, we're operating The GDS for several hundred dollars a month instead of several thousand dollars if we'd used alternative geo-replicated data stores.

2.1 Bilateral Anti-Entropy

An anti-entropy process similar to the mechanism described in [22] facilitates periodic data synchronization between peers in the GDS system. At some configurable interval, a replica (the "initiator") will select a peer (the "remote") in the system at random and initiate an anti-entropy session. Bilateral anti-entropy guarantees us eventual consistency [23] via a "latest writer wins" policy which ensures that both replicas are synchronized to the latest versions on either the initiator or the remote.

Bilateral anti-entropy is critical for cost effectiveness because it halves the number of object versions that have to be transmitted to repair both replicas, particularly when writes are balanced across the system. It also enables additional cost-saving techniques that are discussed in §4.

From the perspective of the initiator, anti-entropy occurs in two phases; first, in the "push" phase, the initiator sends a message to the remote for each object in its local data store. Each message contains the version vector [6, 19] of the object, which includes the object's version number, its parent, as well as the initiator's global process id. During the push phase, messages consist only of metadata and do not

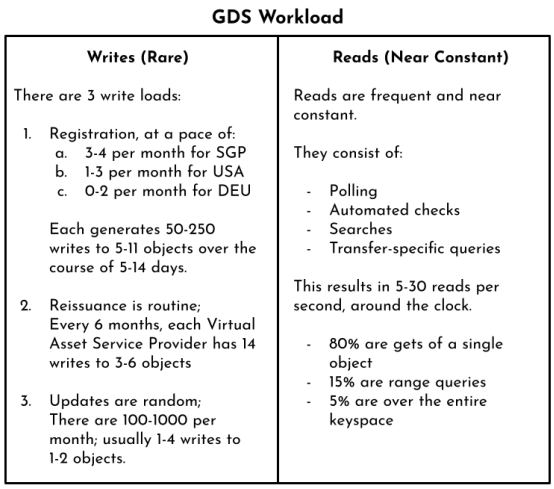

Figure 2. The GDS workload is read-heavy; write activity is bursty but relatively rare.

include object data, a cost-saving mechanism which reduces data egress. After the initiator has finished the push phase, it sends a completion notification to the remote peer, which signals the beginning of the "pull" phase. In this second phase, the initiator receives version vectors and data from the remote for any object that requires local repairs at the initiator.

From the perspective of the remote peer, anti-entropy also occurs in multiple phases. First, the remote will begin receiving a series of messages from the initiator containing version vectors. The remote will use these to determine which objects to later request from the initiator. Any object for which the initiator has a later version than the remote peer will be answered with a request from the remote to the initiator for a "repair": a subsequent message containing the object data as well as the metadata so that the remote may update it's local data store. Once notified that the initiator's push phase is complete, the remote can determine any objects it has locally that the initiator does not have, and will send these to the initiator to allow it to make local repairs.

When comparing version vectors, if a concurrent change has been applied to an object, meaning that it has the same version but different values from the perspective of the originator and the remote, the global process id of the two peers will be used as a tie-breaker, where the value supplied by the peer with the lower of the two ids is given precedence (this is known as a "stomp" and will be discussed later). Global process ids are assigned as new peers come online and guaranteed to be unique.

In addition to gossiping about object data, peers also replicate network information during anti-entropy sessions, including all known peers and their global process ids. Thus, in order to begin the anti-entropy process, at least one peer in the network must be made aware of other peers with which it can gossip.

3 Conflicts

The detection and analysis of conflict is a particular focus in this paper as they provide a means of reasoning about consistency in the GDS.

From the perspective of two users engaging with GDS from opposite sides of the world, conflict is most likely to manifest as a stale read for one or both. For instance, a member lookup by the first user might suggest the second is not yet a certified member of the system, even though the second user's certificates have been issued. Such errors interfere with the trust principles of the system, so it is important to be able to detect conflicts and understand their causes so that the anti-entropy process can be optimized. In particular, we tune the time between anti-entropy sessions to ensure the expected duration [8] of a stale read is less than the retry duration typical to GDS clients. This practical application of probabilistically bounded staleness has allowed us to apply a reasonable engineering semantic (e.g. retry after 30 seconds) that is directly tied to our consistency model without exposing the details of the data store to the client.

The most problematic though also most rare conflict is a fork created by concurrent writes to the same object. We consider each object to have its own version history such that every write to an object creates a new conflict-free, monotonically increasing version number [6, 19] that references the previous local object's version as the parent. The ideal version history is a linear one - e.g. every version is the parent to at most one child version. A concurrent write occurs when writes are applied to two different replicas between anti-entropy sessions, resulting in a parent version that has two children – the linear history forks. The more replicas writing concurrently, the greater the number of competing versions. As the write throughput increases, forks can quickly turn into branches.

Note that overlapping read/write quorums are not the most effective solution to handling forks in our network. We require reads to happen in the geographic region of the client, which means we can have at most a read quorum of 3 (e.g. the primary and secondary nodes in that region). However, this requires us to have a write quorum size that will cover all geographic regions, a configuration that will only tolerate single failures and will not tolerate partitions.

3.1 Conflict Detection

In the context of the GDS, we define two types of conflicts that are easily detectable and a good operational measure of the consistency of the system.

First, a "stomp" is when the remote replica identifies that it and its initiating peer have differing values for the same key, but the same version number for the object. This suggests

that the two replicas have concurrently incremented their versions of the same object, but at the behest of two different and conflicting write requests. In this case, the unique global process id (PID) of each replica must be used as a tie-breaker; the replica with the lower of the two PIDs wins and the value it has for the key replaces the value previously stored at the losing replica's corresponding key. We see stomps as a useful way of understanding which if any objects are being written to concurrently by geographically distributed users (such objects should be rare). We are also interested in understanding how frequently PIDs must be used to break ties. PIDs are assigned to peers when they join the system and are not expressly designed to encode precedence, though stomp behavior might help to uncover unintentional partisanship in the network's behavior. Later as you'll see, we exploit this property to gain an application-level semantic that allows us to give precedence to system administrators.

The second type of conflict we track in the GDS is a "skip"; a skip is counted any time either replica must update an object's version in an increment greater than 1. For example, if Replica Alpha initiates replication with Bravo, and Bravo receives from Alpha the version 5 for a given key that locally is stored at version 3, in determining that it must perform a local repair, we also log a skip conflict because Bravo will have skipped version 4 altogether. Skip conflicts can be a means of identifying peers or groups of peers that are updating and synchronizing objects much more quickly or slowly than the rest of the network.

3.2 Conflict Resolution

Using a policy of "latest writer wins", we are guaranteed eventual consistency – one of the concurrent versions will be selected as the latest version using the precedence of the replica that wrote to it. Unfortunately, this semantic tends to be rather indiscriminate and must be used with care because it can allow single writes to select between two longer branches or create version thrashing where the replicas keep switching between two branches as writes continue. While we'd prefer to keep "the longest branch" or "the most replicated version" – these semantics require global knowledge that is not present at synchronization time; therefore we've elected to use a conflict resolution method based on the access patterns of the GDS.

Our system has two types of users who write to directory records: members who create and edit their directory record and administrators who verify members and issue certificates. Of these types of writes, the more significant is the verification and certificate issuance write; which will more directly impact reads in normal GDS operations. Our administrators are all located in North America, therefore as a first step, we keep our highest precedence replicas in North America to help ensure that administrator writes take priority. More generally, we do store geographic provenance information with objects, and only allow certain operations to occur in the region where the object was created. Note that we greatly benefit from working time zones in this scenario as well; writes by members in Singapore are likely to be at least 4 hours ahead of writes created by administrators and vice versa, minimizing the likelihood of forks and making us more confident in using an eventually consistent system.

When a conflict does occur, we mark the version as requiring resolution. When the conflict is a skip, the latest version is marked as pending until the versions between the latest local version and the updated version are populated and the presence of a stomp can be verified. When a stomp is detected, the versions from the forked parent are all marked as in conflict. We have introduced a special write operation, *merge* that identifies multiple parent versions. When the merged version is replicated, all prior versions through to the forked parent are marked as resolved. Merged versions are replicated similarly to other writes, meaning that the recency bias could cause merged version to be stomped, and concurrent merges and writes that require additional downstream merges, but eventually merges will repair the version history into a linear sequence. Right now the merge operation is manual, meaning that in a write-heavy workload we would quickly fall behind, but we hope to introduce automatic document merging soon and explore how merge replication influences consistency and the branching factor of versions.

4 Cost Effectiveness

Another key focus of our research on the GDS has been to track sufficient data so that anti-entropy is a cost-effective solution. For example, making replication bilateral rather than simply unidirectional is not necessary for successful anti-entropy replication, but has been shown to substantially improve consistency by reducing visibility latency [10]. The introduction of the "pull" phase is intended to improve the consistency of the system by making each anti-entropy session more productive. However, there is a cost associated with each session, one which we would like to be able to control.

Ideally the system can be carefully tuned to provide as much consistency as is necessary for its effective operation, while minimizing its operating budget. In practice, the best means of reducing costs derives from reducing egress by exchanging only metadata during the "push" phase, exchanging objects more selectively (discussed in §4.1), and by minimizing unnecessary anti-entropy sessions (discussed in §4.2).

There are two types of monthly costs in running the GDS system: fixed costs in the form of Kubernetes managed resources such as nodes and disks; and variable costs that are largely related to the amount of network traffic. We consider disk space a fixed cost in our system even though we've implemented a data store because the amount of disk space required per month is easy to estimate for our read-heavy

workload. Because the fixed costs are easy to estimate, we consider them a baseline and attempt to minimize them with respect to normal operation.

The variable networking costs are where the real costs associated with geographically-replicated data systems come in - and they can be tougher to estimate. For Google Cloud Platform, network egress between regions can range between USD $0.01 to $0.15 per GB. The version metadata that we maintain is on average 832 bytes. In a naive implementation of anti-entropy, a data store with 100,000 objects would transmit at least 0.0832GB in the absence of writes. For a 10 node system, operating with an anti-entropy interval of 5 minutes, this represents an estimated price of USD $580 per month in egress costs when no actual data is being replicated. Moreover, we know that the anti-entropy interval is directly related to the consistency of the system – a 5 minute interval dramatically increases the probability of inconsistencies; to achieve SLA targets, an anti-entropy interval of 15 seconds is most appropriate, a naive baseline cost of over USD $11,000 per month in a stable state! These figures helped us understand why commercial solutions for geo-replicated databases are so expensive, particularly when they implement strong consistency semantics - every access incurs egress costs. In order to make our system cost-effective, we needed to find ways to modify replication to reduce the amount of messaging while still maintaining our consistency semantics.

Note that we have chosen to replicate individual objects (documents) in our system rather than the pages of our underlying LSM-Tree database. Although this results in the exchange of more object metadata, it helps us control exactly what data we're sending across regions (to minimize the amount of data transfer) and ensures that replication is not influenced by background operations such as compaction.

4.1 Object Sampling

In the push phase, the initiating replica must iterate through all objects in its local database and send CHECK requests to the remote replica to pull later versions from the remote and push local versions if required. However, in the common case, an object has not been modified since the last anti-entropy request, meaning that the large majority of messages sent result in no improvement to consistency. We have implemented the optimization of having the initiating replica send only version vectors in the push phase, which saves on sending all objects thereby reducing the volume of message traffic. However, we have also explored several further improvements.

One option might be for each peer to track changes since their last anti-entropy session and only send those changes during the next session; however this would prevent changes on the remote from being easily sent back to the initiator, which is necessary for bilateral anti-entropy.

Instead, we have implemented an object sampling mechanism that leverages a key observation about the replication behavior of the GDS. If an object has been recently modified, it has a very high likelihood of being synchronized during the next anti-entropy session. Over time, that probability decreases until the object is modified again. This means we can detect objects whose version vectors should be sent based on a decreasing probability relative to the time since the object's last modified timestamp. This mechanism works during both the push and pull phase iterations over the database.

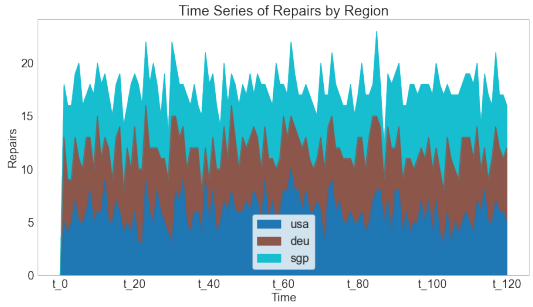

Figure 3. Repairs over time demonstrate natural fluctuations in write activity in GDS, resulting in short periods of staleness experienced in each region.

4.2 Adaptive Anti-Entropy

In previous work, we explored the use of reinforcement learning to replace uniform random selection of anti-entropy peers with a multi-armed bandit strategy of selection [10]. Using an adaptive cost function allowed us to further mitigate cross-region costs by allowing a network to emerge where some peers preferred cross-region connections, other preferred local connections, and replication tended to spread through close connections rather than make long distance hops. We are currently in the process of collecting observations about the replication behavior of the GDS to define an application-specific cost function to incorporate this work into the GDS. We also plan to include network segment weighting to further reduce costs, taking advantage of the Google egress pricing model between regions.

5 Observations

A great deal of behavior of the GDS is accessible via logging and added data collection tools. The GDS uses both the zerolog library [4] as well as a Prometheus [2] client that allows for the close tracking of the system's replication behavior. Of particular interest to this report are metrics on observed latencies for reads and writes, the yield of anti-entropy sessions, and the number of type of conflicts detected during synchronization.

5.1 Read and Write Latency

Reads, writes, deletes, and iterations over GDS data are each tracked at each peer, disaggregated by the namespace (i.e.

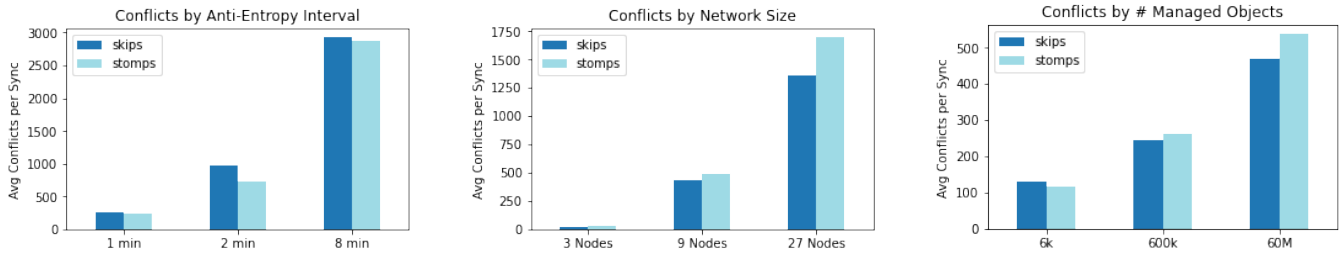

Figure 4. The anti-entropy process in the GDS is configurable. Changes to the anti-entropy synchronization interval, size of the network, and number of managed objects impact the frequency of skips and stomps – a proxy for measuring visibility latency because a conflict implies that the object was not fully replicated before being accessed.

index) of the object. Total objects per namespace are also tracked, as well as tombstones – objects that have been deleted at a particular peer. The latency of each operation is also measured, such that it is possible to evaluate the time taken for each successful RPC call to complete, disaggregated by the type of call.

Tracking read and write latency has allowed us to carefully study the impact of changing the anti-entropy interval (the frequency with which peers synchronize) on GET and PUT operations.

5.2 Anti-Entropy Yield

Anti-entropy behavior is also tracked in the GDS, including counts of each anti-entropy session by peer and by region (there is often more than one replica supporting each region) and the time taken for anti-entropy sessions to complete (disaggregated by peer). In Figure 3 we see repairs as a time series, aggregating peers to their region. Repairs happen when a peer discovers that its anti-entropy partner has a more up-to-date version of an object and requests the updated object from its partner. Repair data enables us to analyze how productive synchronization is in terms of the amount of new information shared during each session. It can also allow us to detect patterns between regions and certain peers that can help uncover topologies of staleness with the network.

It is also useful to be able to gauge the productivity of anti-entropy during each phase. Recall that traditional anti-entropy replication does not necessitate a second phase at all, requiring only the initial push phase. The second phase is an addition targeted at improving consistency, but comes with additional latency and the cost of more messages sent across the network. To this end, GDS tracks pushes (e.g. objects that exist locally and which are sent as repairs) and pulls (e.g. objects that are locally repaired during anti-entropy) separately to allow us to investigate the yield of each phase independently. As with the other anti-entropy metrics, these are disaggregated by peer and region.

5.3 Stomps and Skips

Finally, the GDS tracks numbers of versions per peer (which may indeed be a proxy for anti-entropy sessions from the perspective of an individual peer in the network), as well as counts of stomped and skipped versions, disaggregated by peer and region.

A peer logs a stomp every time it makes a local repair that requires it to overwrite an object version and value because its PID was of lower precedence compared to its anti-entropy partner. A skip is logged every time a peer's local repair requires it to advance the version of an object in an increment greater than one, meaning it has missed at least one iteration in the global version history.

Stomps and skips are important for local reasoning about replica consistency so that replicas can provide users timely information. In Figure 4 we can see the impact on visibility latency of tuning different parameters of the system and its anti-entropy process such as the anti-entropy synchronization interval, the size of the network, and the number of managed objects in the GDS. It could be possible to try to replicate a global version history, which could then be used to more completely reason about visibility latency, branches, and stomps. However, even if we were to replicate the version history without data, there would be no guarantee at any given point in time that a replica knows the full state of the system, as this version history would also be eventually consistent (and in the case of failures, may never be fully replicated).

6 Future Work

The requirements for global coverage of a read-heavy workload may suggest a number of commercially available databases well suited to the task. However, the pricing of these solutions expect large scale deployments of "traditional" geo-replicated systems, making them impractical for a context such as TRISA's volunteer-led professional working group. Configuration of an open source or community version is a good, economical alternative, but frankly still much larger and much more complex than our use case required.

Architecting the GDS required us to reason about consistency in a way that applications developers generally do not, and we discovered many more variables to consistency than the relationship between access throughput and the synchronization interval. We anticipate that future work will yield a tunable model of eventual consistency to enable globally replicated data stores that prefer cost-effectiveness over strong consistency semantics. Even better, a probabilistic model would allow developers to directly modify system behavior in a way that was right for the application.

This work may also lead to new ways of thinking about composed operations and the tolerance that different types of operations have for the risk of consistency failure. Exposing conflict probability to the application layer (e.g. via thresholded transactions) might help create different operation-oriented policies that would be easy for an application developer to reason about.

Much of the consistency literature we have found concerns maintaining consistency in the face of faults. In practice, the administration of a geo-replicated database is complex and difficult, and there are many different potential causes for inconsistencies that should not be taken for granted. As such, we are enthusiastic to share our data and findings with the systems community and hope that it will encourage further research on eventual consistency. In particular, we believe algorithmic solutions that include configuration, adaptation, variable load, and endurance will make a distinct impact on how applications developers orchestrate the complex interactions of modern and future systems.

References

[1] 2011. *Protocol Buffers: Google's Data Interchange Format.* https://developers.google.com/protocol-buffers/

[2] 2014. *Prometheus / Prometheus.io.* https://prometheus.io/

[3] 2018. *Grpc / Grpc.Io.* https://grpc.io/

[4] 2021. *Zero Allocation JSON Logger.* https://pkg.go.dev/github.com/rs/zerolog

[5] 2022. *Mutual Authentication.* https://en.wikipedia.org/wiki/Mutual_authentication

[6] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. 2002. Version Stamps-Decentralized Version Vectors. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference On.* IEEE, 544–551. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1022304

[7] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2012. Probabilistically Bounded Staleness for Practical Partial Quorums. 5, 8 (2012), 776–787. http://dl.acm.org/citation.cfm?id=2212359

[8] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2014. Quantifying Eventual Consistency with PBS. 23, 2 (2014), 279–302. http://link.springer.com/article/10.1007/s00778-013-0330-1

[9] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. 2006. PRACTI Replication.. In *NSDI*, Vol. 6. 5–5.

[10] Benjamin Bengfort, Konstantinos Xirogiannopoulos, and Pete Keleher. 2018-07-05. Anti-Entropy Bandits for Geo-Replicated Consistency. In *Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS)* (Vienna, Austria). IEEE Computer Society Press.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's Globally Distributed Database. 31, 3 (2013), 8. http://dl.acm.org/citation.cfm?id=2491245

[12] Sanjay Ghemawat and Jeff Dean. 2014. *LevelDB, A Fast and Lightweight Key/Value Database Library by Google.*

[13] Bernhard Haeupler. 2015. Simple, Fast and Deterministic Gossip and Rumor Spreading. 62, 6 (2015), 47. http://dl.acm.org/citation.cfm?id=2767126

[14] Pat Helland. 2015. Immutability Changes Everything. 13, 9 (2015), 40. http://dl.acm.org/citation.cfm?id=2884038

[15] Dave Jevans, Thomas Hardjono, Jelle Vink, Frank Steegmans, John Jefferies, and Aanchal Malhotra. 2020. *TRISA Whitepaper Version 8.* Technical Report. https://trisa.io/trisa-whitepaper/

[16] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. 2000. Randomized Rumor Spreading. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium On.* IEEE, 565–574. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=892324

[17] Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. 28, 10 (2017), 2733–2746.

[18] Yamir Moreno, Maziar Nekovee, and Amalio F. Pacheco. 2004. Dynamics of Rumor Spreading in Complex Networks. 69, 6 (2004), 066130. http://journals.aps.org/pre/abstract/10.1103/PhysRevE.69.066130

[19] D. Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. 1983. Detection of Mutual Inconsistency in Distributed Systems. 3 (1983), 240–247. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1703051

[20] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles.* ACM, 497–514.

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Symposium on Self-Stabilizing Systems.* Springer, 386–400.

[22] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference On.* IEEE, 140–149. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=331722

[23] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. 29 (1995).

[24] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* ACM, 1–12.

[25] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data.* ACM, 1041–1052.