Teleporting Failed Writes with Cache Augmented Data Stores

Shahram Ghandeharizadeh(✉), Haoyu Huang, and Hieu Nguyen

USC Database Laboratory, Los Angeles, USA
{shahram,haoyuhua,hieun}@usc.edu

Abstract. Cache Augmented Data Stores enhance the performance of workloads that exhibit a high read to write ratio by extending a persistent data store (PStore) with a cache. When the PStore is unavailable, today's systems result in *failed* writes. With the cache available, we propose TARDIS, a family of techniques that teleport failed writes by buffering them in the cache and persisting them once the PStore becomes available. TARDIS preserves consistency of the application reads and writes by processing them in the context of buffered writes. TARDIS family of techniques is differentiated in how they apply buffered writes to PStore once it recovers. Each technique requires a different amount of mapping information for the writes performed while PStore was unavailable. The primary contribution of this study is an overview of TARDIS and its family of techniques.

1 Introduction

Person-to-person cloud service providers such as Facebook challenge today's software and hardware infrastructure [9,24]. Traditional web architectures struggle to process their large volume of requests issued by hundreds of millions of users. In addition to facilitating a near real-time communication, a social network infrastructure must provide an always-on experience in the presence of different forms of failure [9]. These requirements have motivated an architecture that augments a data store with a distributed in-memory cache manager such as memcached and Redis. We term this class of systems Cache Augmented Data Stores, **CADSs**.

Figure 1 shows a CADS consisting of Application Node (AppNode) servers that store and retrieve data from a persistent store (PStore) and use a cache for temporary staging of data [18,24,25]. The cache expedites processing of requests by either using faster storage medium, bringing data closer to the AppNode, or a combination of the two. An example PStore is a document store [10] that is either a solution such as MongoDB or a service such as Amazon DynamoDB [5,11] or MongoDB's Atlas [22]. An example cache is an in-memory key-value store such as memcached or Redis with Amazon ElastiCache [6] as an example service. Both the caching layer and PStore may employ data redundancy techniques to tolerate node failures.

M. Luo and L.-J. Zhang (Eds.): CLOUD 2018, LNCS 10967, pp. 55–68, 2018.
https://doi.org/10.1007/978-3-319-94295-7_4

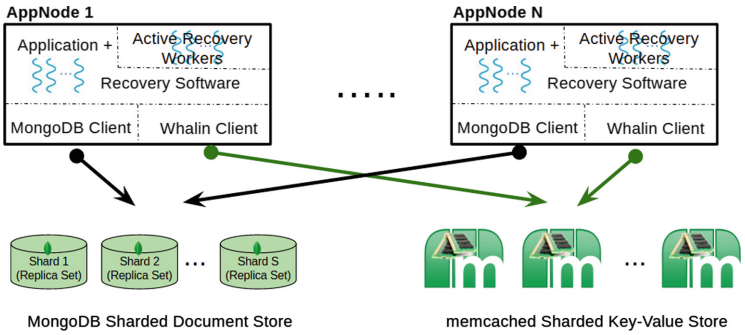

Fig. 1. CADS architecture.

The CADS architecture assumes a software developer provides application specific logic to identify cached keys and how their value is computed using PStore. Read actions look up key-value pairs. In case of a miss, they query PStore, compute the missing key-value pair, and insert it in the cache for future look up. Write actions maintain the key-value pairs consistent with data changes in PStore.

A read or a write request to PStore *fails* when it is not processed in a timely manner. This may result in denial of service for the end user. All requests issued to PStore fail when it is unavailable due to either hardware or software failures, natural disasters, power outages, human errors, and others. It is possible for a subset of PStore requests to fail when the PStore is either sharded or offered as a service. With a sharded PStore, the failed requests may reference shards that are either unavailable or slow due to load imbalance and background tasks such as backup. With PStore as a service, requests fail when the application exhausts its pre-allocated capacity. For example, with the Amazon DynamoDB and Google Cloud Datastore, the application must specify its read and write request rate (i.e., capacity) in advance with writes costing more [5,15]. If the application exhausts its pre-specified write capacity then its writes fail while its reads succeed.

An application may process its failed writes in a variety of ways. Simplest is to report the failures to the end users and system administrators. The obvious drawback of this approach is that the write fails and the end user is made aware of this to try again (or for the system administrator to perform some corrective action). Another variant is for the application to ignore the failed write. To illustrate, consider a failed write pertaining to Member A accepting Member B's friend invitation. The system may simply ignore this write operation and continue to show B's invitation to A to be accepted again. Assuming the failure of the PStore is short-lived then the user's re-try may succeed. This alternative loses those writes that are not recoverable. For example, when Member C invites Member B to be friends, dropping this write silently may not be displayed in an intuitive way for the end user to try again. A more complex approach is

for the application to buffer the failed writes. A system administrator would subsequently process and recover these writes.

This paper describes an automated solution that replaces the administrator with a smart algorithm that buffers failed writes and applies them to PStore at a later time once it is available. This approach constitutes the focus of TARDIS[1], a family of techniques for processing failed writes that are transparent to the user issuing actions.

Table 1. Number of failed writes with different BG workloads and PStore failure durations.

Failure duration	Read to write ratio	
	100:1	1000:1
1 min	5,957	561
5 min	34,019	3,070
10 min	71,359	6,383

TARDIS teleports failed writes by buffering them in the cache and performing them once the PStore is available. Buffered writes stored as key-value pairs in the cache are pinned to prevent the cache from evicting them. To tolerate cache server failures, TARDIS replicates buffered writes across multiple cache servers.

Table 1 shows the number of failed writes with different failure durations using a benchmark for interactive social networking actions named BG [8]. We consider two workloads with different read to write ratios. TARDIS teleports all failed writes and persists them once PStore becomes available.

TARDIS addresses the following challenges:

1. How to process PStore reads with pending buffered writes in the cache?
2. How to apply buffered writes to PStore while servicing end user requests in a timely manner?
3. How to process non-idempotent buffered writes with repeated failures during recovery phase?
4. What techniques to employ to enable TARDIS to scale? TARDIS must distribute load of failed writes evenly across the cache instances. Moreover, its imposed overhead must be minimal and independent of the number of cache servers.

TARDIS preserves consistency guarantees of its target application while teleporting failed writes. This is a significant improvement when compared with today's state of the art that loses failed writes always.

Advantages of TARDIS are two folds. First, it buffers failed writes in the cache when PStore is unavailable and applies them to PStore once it becomes

[1] Time and Relative Dimension in Space, TARDIS, is a fictional time machine and spacecraft that appears in the British science fiction television show Doctor Who.

available. Second, it enhances productivity of application developers and system administrators by providing a universal framework to process failed writes. This saves both time and money by minimizing complexity of the application software.

Assumptions of TARDIS include:

- AppNodes and the cache servers are in the same data center, communicating using a low latency network. This is a reasonable assumption because caches are deployed to enhance AppNode performance.
- The cache is available to an AppNode when PStore writes fail.
- A developer authors software to reconstruct a PStore document using one or more cached key-value pairs. This is the recovery software shown in Fig. 1 used by both the application software and Active Recovery (AR) workers.
- The PStore write operations are at the granularity of a single document and transition its state from one consistent state to another. In essence, the correctness of PStore writes are the responsibility of the application developer.
- There is no dependence between two or more buffered writes applied to *different*[2] documents. This is consistent with the design of a document store such as MongoDB to scale horizontally. Extensions to a relational data model that considers foreign key constraints is a future research direction.

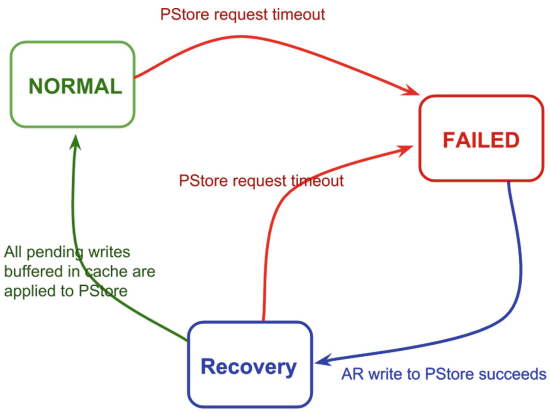

Fig. 2. AppNode state transition diagram.

The rest of this paper is organized as follows. Section 2 presents the design of TARDIS. Section 3 describes how undesirable race conditions may happen and our solution using contextual leases. We survey related work in Sect. 4. Brief future research directions are presented in Sect. 5.

[2] TARDIS preserves the order of two or more writes for the same document.

2 Overview

Each AppNode in TARDIS operates in 3 distinct modes: normal, failed, and recovery. Figure 2 shows the state transition diagram for these three modes. TARDIS operates in normal mode as long as PStore processes writes in a timely manner. A failed PStore write transitions AppNode to failed mode. In this mode, AppNode threads buffer their PStore writes in the cache. They maintain cached key-value pair impacted by the write as in normal mode of operation. Moreover, the AppNode starts one or more Active Recovery (AR) workers to detect when the PStore is available to process writes again. Once PStore processes a write by an AR worker, the AR worker transitions AppNode state to recovery mode.

In recovery mode, TARDIS applies buffered writes to PStore. TARDIS family of techniques is differentiated in how they perform this task. In its simplest form, termed TAR, only AR workers apply buffered writes. The next variant, termed DIS, extends TAR by requiring the write actions of the application to identify pending writes in the cache and apply them to PStore prior to performing their write. With DIS, a write action may merge the pending writes with its own into one PStore operation. With TARD, the developer provides a *mapping* between the read and buffered writes produced by write actions. This mapping enables the application to process reads that observe a cache miss by first applying the pending buffered writes to the PStore. Finally, TARDIS is a hybrid that includes features of both TARD and DIS.

The trade-off with the alternative techniques is as follows. With TAR, the application continues to produce buffered writes even though PStore is available. With both TAR and DIS, a cache miss continues to report a failure (even though PStore is available) until all pending writes are applied to PStore. DIS is different because it stops the application from producing buffered writes, enabling the recovery mode to end sooner. With TARD, a cache miss applies buffered writes to PStore while writes continue to be buffered. TARDIS is most efficient in processing application requests, ending the recovery process fastest. Due to novelty of TARDIS, applicability of TAR, TARD, DIS, and TARDIS in the context of different applications is a future research direction. It may be the case that DIS is applicable to most if not all applications.

When either AppNode or an AR worker incurs a failed write, it switches AppNode mode from recovery to failed. A PStore that processes writes intermittently may cause AppNode to toggle between failed and recovery modes repeatedly, see Fig. 2.

2.1 Failed Mode

In failed mode, a write for D_i generates a change δ_i for this PStore document and appends δ_i to the *buffered write* Δ_i of the document in the cache. In addition, this write appends P_i to the value of a key named Teleported Writes, TeleW. This key-value pair is also stored in the cache. It is used by AR workers to discover documents with pending buffered writes (Table 2).

Table 2. List of terms and their definition.

Term	Definition
PStore	A sharded data store that provides persistence
AR	Active Recovery worker migrates buffered writes to PStore eagerly
D_i	A PStore document identified by a primary key P_i
P_i	Primary key of document D_i. Also referred to as document id
$\{K_j\}$	A set of key-value pairs associated with a document D_i
Δ_i	A key whose value is a set of changes to document D_i
TeleW	A key whose value contains P_i of documents with teleported writes
ω	Number of TeleW keys

Δ_i may be redundant if the application is able to construct document D_i using its representation as a collection of cached key-value pairs. The value of keys $\{K_i\}$ in the cache may be sufficient for the AppNode to reconstruct the document D_i in recovery mode. However, generating a list of changes Δ_i for the document may expedite recovery time if it is faster to read and process than reading the cached value of $\{K_i\}$ to update PStore. An example is a member P_i with 1000 friends. If in failed mode, P_i makes an additional friend P_k, it makes sense for AppNode to both update the cached value and generate the change $\delta_i = \text{push}(P_k,\text{Friends})$. At recovery time, instead of reading an array of 1001 profile ids to apply the write to PStore document D_i, the system reads the change and applies it. Since the change is smaller and its read time is faster, this expedites recovery time.

In failed mode, AR workers try to apply buffered writes to PStore. An AR worker identifies these documents using the value of TeleW key. Each time AR's write to PStore fails, the AR may exponentially back-off before retrying the write with a different document. Once a fixed number of AR writes succeeds, an AR worker transitions the state of AppNode to recovery.

TARDIS prevents contention for TeleW by maintaining ω TeleW key-value pairs. It hash partitions documents across these using their primary key P_i. Moreover, it generates the key of each ω TeleW with the objective to distribute these keys across all cache servers. In failed mode, when the AppNode generates buffered writes for a document D_j, it appends the document to the value of the TeleW key computed using its P_j, see Fig. 3.

Algorithm 1 shows the pseudo-code for the AppNode in failed mode. This pseudo-code is invoked after AppNode executes application specific code to update the cache and generate changes δ_i (if any) for the target document D_i. It requires the AppNode to obtain a lease on its target document D_i. Next, it attempts to mark the document as having buffered writes by generating a key P_i+dirty with value dirty[3]. The memcached `Add` command inserts this key if

[3] Choice of P_i+dirty is arbitrary. The requirement is for the key to be unique. P_i+dirty is a marker and its value may be one byte.

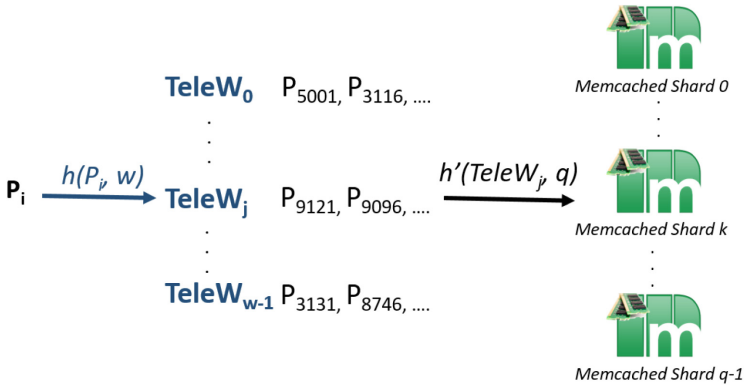

Fig. 3. Documents are partitioned across ω TeleW keys. TeleW keys are sharded across q memcached servers.

it does not exist. Otherwise, it returns NOT_STORED. Hence, the first time a document is inserted, it is appended to one of the ω TeleW keys. Repeated writes of D_i in failed mode do not insert a document D_i in TeleW again.

Algorithm 1. AppNode in Failed Mode (P_i)

1. **acquire lease** P_i
2. InsertAttempt = `Add`(P_i+dirty)
3. **release lease** P_i
4. **if** InsertAttempt is successful {
   TeleW$_i$ = hash(P_i, ω)
   acquire lease TeleW$_i$
   append(P_i, TeleW$_i$)
   release lease TeleW$_i$
   }

2.2 Recovery Mode

To perform an action that references a document D_i, AppNode checks to see if this document has buffered writes. It does so by obtaining a lease on P_i. Once the lease is granted, it looks up Δ_i. If it does not exist then it means an AR worker (or another AppNode thread) competed with it and propagated D_i's changes to PStore. In this case, it releases its lease and proceeds to service user's action. Otherwise, it applies the buffered writes to update D_i in PStore. If this is successful, AppNode deletes Δ_i to prevent an AR worker from applying buffered writes to D_i a second time.

TARDIS employs AR workers to eagerly apply buffered writes to PStore documents identified by TeleW. We minimize the recovery duration by maximizing

the probability of each AR worker to work on a disjoint set of documents. To achieve this, each AR worker randomly pick TeleW key as the starting point and visit other TeleW keys in a round-robin fashion. Also, it only recover α randomly selected documents in a TeleW. With AppNode, each time a user action references a document D_i with buffered writes, the AppNode applies its writes to the PStore prior to servicing the user action. In essence, during recovery, the AppNode stops producing buffered writes and propagates buffered writes to PStore.

Challenges of implementing TARDIS's recovery mode is two folds: (a) correctness: TARDIS must propagate buffered writes to PStore documents in a consistent manner, and (b) performance: TARDIS must process user's actions as fast as normal mode while completing recovery quickly.

We categorize buffered writes into idempotent and non-idempotent. Idempotent writes can be repeated multiple times and produce the same value. Non-idempotent writes lack this behavior. During recovery, multiple AR workers may perform idempotent writes multiple times without compromising correctness. However, this redundant work slows down the PStore for processing regular requests and makes the duration of recovery longer than necessary. Our implementation of recovery uses leases to apply buffered writes of a document to PStore exactly once even in the presence of AR worker or AppNode failures. The details are provided in the full technical report [26].

Algorithm 2 shows each iteration of AR worker in recovery mode. An AR worker picks a TeleW key randomly and looks up its value. This value is a list of documents written in failed mode. From this list, it selects α random documents $\{D\}$. For each document D_i, it looks up P_i+dirty to determine if its buffered writes still exist in the cache. If this key exists then it acquires a lease on D_i, and looks up P_i+dirty a second time. If this key still exists then it proceeds to apply the buffered writes to D_i in PStore. Should this PStore write succeed, the AR worker deletes P_i+dirty and buffered writes from the cache.

The AR worker maintains the primary key of those documents it successfully writes to PStore in the set `R`. It is possible for the AR worker's PStore write to D_i to fail. In this case, the document is not added `R`, leaving its changes in the cache to be applied once PStore is able to do so.

An iteration of the AR worker ends by removing the documents in `R` (if any) from its target TeleW key, Step 6 of Algorithm 2. Duplicate P_is may exist in TeleW$_T$. A race condition involving AppNode and AR worker generates these duplicates: A buffered write is generated between Steps 5 and 6 of Algorithm 2. Step 6 does a multi-get for TeleW$_T$ and the processed P_i+dirty values. For those P_is with a buffered write, it does not remove them from TeleW$_T$ value because they were inserted due to the race condition.

2.3 Cache Server Failures

A cache server failure makes buffered writes unavailable, potentially losing content of volatile memory all together. To maintain availability of buffered writes,

Algorithm 2. Each Iteration of AR Worker in Recovery Mode

```
1. Initialize R = {}
2. T = A random value between 0 and ω
3. V = get(TeleW_T)
4. {D} = Select α random documents from V

5. for each D_i in {D} do {
     V = get(P_i+dirty)
     if V exists {
          acquire lease P_i
          V = get(P_i+dirty)
          if V exists {
               success = Update P_i in PStore with buffered writes
               if success {
                    Delete(P_i+dirty)
                    Delete Δ_i (if any)
                    R = R ∪ P_i
               }
          } else R = R ∪ P_i
          release lease P_i
     } else R = R ∪ P_i
   }

6. if R is not empty {
     acquire lease TeleW_T
     Do a multiget on TeleW_T and all P_i+dirty in R
     V = value fetched for TeleW_T
     For those P_i+dirty with a value, remove them from R
     Remove documents in R from V
     put(TeleW_T,V)
     release lease TeleW_T
   }
```

TARDIS replicates buffered writes across two or more cache servers. Both Redis [1] and memcached [2] support this feature.

A cache server such as Redis may also persist buffered writes to its local storage, disk or flash [1]. This enables buffered writes to survive failures that destroy content of volatile memory. However, if a cache server fails when PStore recovers, its contained buffered writes become unavailable and TARDIS must suspend all reads and writes to preserve consistency. Thus, replication is still required to process reads and writes of an AppNode in recovery mode.

3 TARDIS Consistency

TARDIS consists of a large number of AppNodes. Each AppNode may operate in different modes as described in Sect. 2. While AppNode 1 operates in failed

or recovery mode, AppNode 2 may operate in normal mode. This may happen if AppNode 2 processes reads with 100% cache hit while AppNode 1 observes a failed write. This results in a variety of undesirable read-write and write-write race conditions when user requests for a document are directed to AppNode 1 and AppNode 2.

An example of a write-write race condition is that a user Bob who creates an Album and adds a photo to the Album. Assume Bob's "create Album" is directed to AppNode 1. If AppNode 1 is operating in failed mode then it buffers Bob's Album creation in the cache. Once PStore recovers and prior to propagating Bob's album creation to PStore, Bob's request to add a photo is issued to AppNode 2. If AppNode 2 is in normal mode then it issues this write to an album that does not exist in PStore.

An example of a read-write race condition is that Bob changes the permission on his profile page so that his manager Alice cannot see his status. Subsequently, he updates his profile to show he is looking for a job. Assume Bob's first action is processed by AppNode 1 in failed mode and Bob's profile is missing from the cache. This buffers the write as a change (Δ) in the cache. His second action, update to his profile, is directed to AppNode 2 (in normal mode) and is applied to PStore because it just recovered and became available. Now, before AppNode 1 propagates Bob's permission change to PStore, there is a window of time for Alice to see Bob's updated profile.

We present a solution, contextual leases, that employs stateful leases without requiring consensus among AppNodes. It implements the concept of sessions that supports atomicity across multiple keys along with commit and rollbacks. TARDIS with contextual leases maintains the three mode of operation described in Sect. 2. It incorporates Inhibit (I) and Quarantine (Q) leases of [13] and extends them with a marker.

Table 3. IQ lease compatibility.

		Granted	
		I lease	Q lease
Requesting	I lease	Back-off and retry	Back-off and retry
	Q lease	Grant Q and void I lease	Reject and abort requester

The framework of [13] requires: (1) a cache miss to obtain an I lease on its referenced key prior to querying the PStore to populate the cache, and (2) a cache update to obtain a Q lease on its referenced key prior to performing a Read-Modify-Write (R-M-W) or an incremental update such as `append`. Leases are released once a session either commits or aborts. TARDIS uses the write-through policy of IQ-framework. I leases and Q leases are incompatible with each other, see Table 3. A request for an I lease must back-off and retry if the key is already associated with another I or Q lease. A request for a Q lease may void an existing I lease or is rejected and aborted if the key is already associated with

another Q lease. IQ avoids thundering herds by requiring only one out of many requests that observes a cache miss to query PStore and populate the cache. It also eliminates all read-write and write-write race conditions.

In failed mode, a write action continues to obtain a Q lease on a key prior to performing its write. However, at commit time, a Q lease is converted to a P marker on the key. This marker identifies the key-value pair as having buffered writes. The P marker serves as the context for the I and Q leases granted on a key. It has no impact on their compatibility matrix. When the AppNode requests either an I or a Q lease on a key, if there is a P marker then the AppNode is informed of this marker should the lease be granted. In this case, the AppNode must recover the PStore document prior to processing the action.

In our example undesirable write-write race condition, Bob's Album creation using AppNode 1 generates a P marker for Bob's Album. Bob's addition of a photo to the album (using AppNode 2) must obtain a Q lease on the album that detects the marker and requires the application of buffered writes prior to processing this action. Similarly, with the undesirable read-write race condition, Bob's read (performed by AppNode 2) is a cache miss that must acquire an I lease. This lease request detects the P marker that causes the action to process the buffered writes. In both scenarios, when the write is applied to PStore and once the action commits, the P marker is removed as a part of releasing the Q leases.

4 Related Work

The CAP theorem states that a system designer must choose between strong consistency and availability in the presence of network partitions [21]. TARDIS improves availability while preserving consistency.

A weak form of data consistency known as eventual has multiple meanings in distributed systems [28]. In the context of a system with multiple replicas of a data item, this form of consistency implies writes to one replica will eventually apply to other replicas, and if all replicas receive the same set of writes, they will have the same values for all data. Historically, it renders data available for reads and writes in the presence of network partitions that separate different copies of a data item from one another and cause them to diverge [11]. TARDIS teleports failed writes due to either network partitions, PStore failures, or both while preserving consistency.

A write-back policy buffers writes in the cache and applies them to PStore asynchronously. It may not be possible to implement a write-back policy for all write actions of an application. This is because a write-back policy requires a mapping between reads and writes to be able to process a cache miss by applying the corresponding buffered writes to the PStore prior to querying PStore for the missing cache entry. In these scenarios, with TARDIS, one may use DIS (without TARD) that continues to report a failure for cache misses during recovery mode while applying buffered writes to PStore. A write-back policy does not provide

such flexibility. It also does not consider PStore availability. It improves performance of write-heavy workloads. TARDIS objective is to improve the availability of writes during PStore failure.

Everest [23] is a system designed to improve the performance of overloaded volumes during peak load. Each Everest client has a base volume and a store set. When the base volume is overloaded, the client off-loads writes to its idle stores. When the base volume load is below a threshold, the client uses background threads to reclaim writes. Everest clients do not share store set. With TARDIS, AppNodes share the cache and may have different view of PStore, requiring contextual leases. Moreover, its AppNode may fail during PSTore recovery, requiring conversion of non-idempotent writes into idempotent ones.

The race conditions encountered (see Sect. 3) are similar to those in geo-replicated data distributed across multiple data centers. Techniques such as causal consistency [20] and lazy replication [19] mark writes with their causal dependencies. They wait for those dependencies to be satisfied prior to applying them at a replica, preserving order across two causal writes. TARDIS is different because it uses Δ_i to maintain the order of writes for a document D_i that observes a cache miss and is referenced by one or more failed writes. With cache hits, the latest value of the keys reflects the order in which writes were performed. In recovery mode, TARDIS requires the AppNode to perform a read or a write action by either using the latest value of keys (cache hit) or Δ_i (cache miss) to restore the document in PStore prior to processing the action. Moreover, it employs AR workers to propagate writes to persistent store during recovery. It uses leases to coordinate AppNode and AR workers because its assumed data center setting provides a low latency network.

Numerous studies perform writes with a mobile device that caches data from a database (file) server. (See [27] as two examples.) Similar to TARDIS, these studies enable a write while the mobile device is disconnected from its shared persistent store. However, their architecture is different, making their design decisions inappropriate for our use and vice versa. These assume a mobile device implements an application with a local cache, i.e., AppNode and the cache are in one mobile device. In our environment, the cache is shared among multiple AppNodes and a write performed by one AppNode is visible to a different AppNode - this is not true with multiple mobile devices. Hence, we must use leases to detect and prevent undesirable race conditions between multiple AppNode threads issuing read and write actions to the cache, providing correctness.

Host-side Caches [4,16,17] (HsC) such as Flashcache [3] are application *transparent* caches that stage the frequently referenced disk pages onto NAND flash. These caches may be configured with either write-around, write-through, or write-back policy. They are an intermediary between a data store issuing read and write of blocks to devices managed by the operating system (OS). Application caches such as memcached are different than HsC because they require application specific software to maintain cached key-value pairs. TARDIS is somewhat similar to the write-back policy of HsC because it buffers writes in cache and propagates them to the PStore (HsC's disk) in the background. TARDIS is

different because it applies when PStore writes fail. Elements of TARDIS can be used to implement write-back policy with caches such as memcached, Redis, Google Guava [14], Apache Ignite [7], KOSAR [12], and others.

5 Conclusions and Future Research

TARDIS is a family of techniques designed for applications that must provide an always-on experience with low latency, e.g., social networking. In the presence of short-lived persistent store (PStore) failures, TARDIS teleports failed writes by buffering them in the cache and applying them once PStore is available.

An immediate research direction is to conduct a comprehensive evaluation of TARDIS and its variants (TAR, DIS, TARD) to quantify their tradeoff. This includes an evaluation of replication techniques that enhance availability of buffered writes per discussion of Sect. 2.3.

More longer term, we intend to investigate extensions of TARDIS to those logical data models that result in dependence between buffered writes. To elaborate, rows of tables of a relational data model have dependencies such as foreign key dependency. With these SQL systems, TARDIS must manage the dependence between the buffered writes to ensure they are teleported in the right order.

References

1. Redis. https://redis.io/
2. repcached: Data Replication with Memcached. http://repcached.lab.klab.org/
3. McDipper (2013). https://www.facebook.com/10151347090423920
4. Alabdulkarim, Y., Almaymoni, M., Cao, Z., Ghandeharizadeh, S., Nguyen, H., Song, L.: A comparison of flashcache with IQ-Twemcached. In: ICDE Workshops (2016)
5. Amazon DynamoDB (2016). https://aws.amazon.com/dynamodb/pricing/
6. Amazon ElastiCache (2016). https://aws.amazon.com/elasticache/
7. Apache: Ignite - In-Memory Data Fabric (2016). https://ignite.apache.org/
8. Barahmand, S., Ghandeharizadeh, S.: BG: a benchmark to evaluate interactive social networking actions. In: CIDR, January 2013
9. Bronson, N., Lento, T., Wiener, J.L.: Open data challenges at Facebook. In: ICDE (2015)
10. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Rec. **39**, 12–27 (2011)
11. Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP (2007)
12. Ghandeharizadeh, S., et al.: A Demonstration of KOSAR: an elastic, scalable, highly available SQL middleware. In: ACM Middleware (2014)
13. Ghandeharizadeh, S., Yap, J., Nguyen, H.: Strong consistency in cache augmented SQL systems. In: Middleware, December 2014
14. Google: Guava: Core Libraries for Java (2015). https://github.com/google/guava
15. Google App Engine (2016). https://cloud.google.com/appengine/quotas/

16. Graefe, G.: The five-minute rule twenty years later, and how flash memory changes the rules. In: DaMoN, p. 6 (2007)
17. Holland, D.A., Angelino, E., Wald, G., Seltzer, M.I.: Flash caching on the storage client. In: USENIXATC (2013)
18. Hu, X., Wang, X., Li, Y., Zhou, L., Luo, Y., Ding, C., Jiang, S., Wang, Z.: LAMA: optimized locality-aware memory allocation for key-value cache. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15), July 2015
19. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. **10**(4), 360–391 (1992)
20. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: SOSP (2011)
21. Lynch, N., Gilbert, S.: Brewer's conjecture and the feasibility of consistent, available partition-tolerant web services. ACM SIGACT News **33**, 51–59 (2002)
22. MongoDB Atlas (2016). https://www.mongodb.com/cloud
23. Narayanan, D., Donnelly, A., Thereska, E., Elnikety, S., Rowstron, A.: Everest: scaling down peak loads through I/O Off-loading. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008, Berkeley, CA, USA, pp. 15–28. USENIX Association (2008)
24. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., Venkataramani, V.: Scaling memcache at Facebook. In: NSDI, Berkeley, CA, pp. 385–398. USENIX (2013)
25. Ports, D.R.K., Clements, A.T., Zhang, I., Madden, S., Liskov, B.: Transactional consistency and automatic management in an application data cache. In: OSDI. USENIX, October 2010
26. Shahram Ghandeharizadeh, H.H., Nguyen, H.: TARDIS: teleporting failed writes with cache augmented datastores. Technical report 2017–01, USC Database Laboratory (2017). http://dblab.usc.edu/Users/papers/TARDIS.pdf
27. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: SOSP (1995)
28. Viotti, P., Vukolić, M.: Consistency in non-transactional distributed storage systems. ACM Comput. Surv. **49**(1) (2016)