Teleporting Failed Writes with Cache Augmented Data Stores

Shahram Ghandeharizadeh, Haoyu Huang, Hieu Nguyen

Database Laboratory Technical Report 2017-01

Computer Science Department, USC

Los Angeles, California 90089-0781

{shahram,haoyuhua,hieun}@usc.edu

May 8, 2017

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. Given a fixed amount of memory, variants of TARDIS may store a different amount of buffered writes. We quantify this using the popular memcached and Redis in-memory key-value stores.

1 Introduction

Person-to-person cloud service providers such as Facebook challenge today's software and hardware infrastructure [36, 12]. 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 [12]. 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, 38, 36, 17, 26]. 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 [14] that is either a solution such as MongoDB or a service such as Amazon DynamoDB [5, 16] or MongoDB's Atlas [35]. 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.

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, 23]. If the application exhausts its pre-specified write capacity then its writes fail while its reads succeed.

A system may address failed writes using either client-side solutions, server-side solutions, or a hybrid of the two. An example client-side solution is to notify[1] the user of the failed write so that she may attempt the action that caused the failed write at a later time. An example server-side solution is to buffer the failed write and apply it at a later time once PStore is available. This second approach constitutes the focus of TARDIS[2], a family of server-side techniques for processing failed writes that are transparent to the user issuing actions.

TARDIS teleports failed writes by buffering them in the cache and performing them once the PStore is available. A system designer may set aside a portion of cache space for buffered writes. Our experimental results using YCSB and BG benchmarks show hundreds of megabytes of memory may accommodate millions of buffered writes; potentially an infinite number with some workloads. 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 [10]. 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?

[1]This notification may be in the form of a pop-up window.

[2]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.

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

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

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 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 Figure 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*[3] 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.

The rest of this paper is organized as follows. Section 2 presents the design of TARDIS using a multi-threaded AppNode. Section 3 extends this discussion to multiple AppNodes and presents two implementations of TARDIS: Recon and Contextual. Section 4 uses YCSB [15] and BG [10] benchmarks to quantify memory requirements of TARDIS, its recovery time, and horizontal scalability characteristics. We survey related work in Section 5. Brief future research directions are presented in Section6.

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

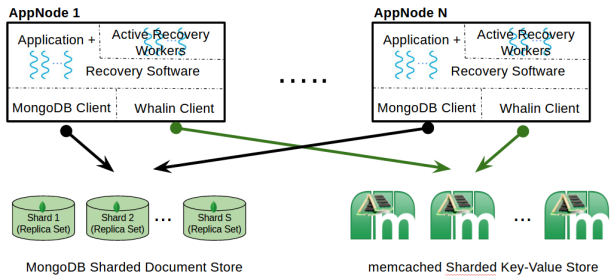


Figure 1: CADS architecture.

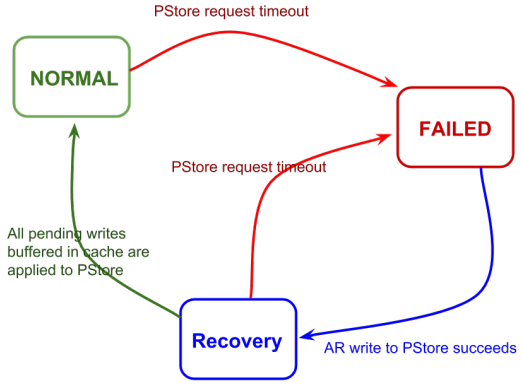


Figure 2: AppNode state transition diagram.

2 TARDIS with 1 AppNode

This section presents TARDIS using a simple configuration consisting of 1 multi-threaded AppNode server, a sharded PStore, and a sharded memcached server. Terms Application thread (App thread) and Active Recovery (AR) worker refer to a thread of the AppNode server (see Figure 1). They implement the application logic or a recovery worker that propogates buffered writes to PStore, respectively. We assume a failed request is due to PStore becoming unavailable completely. Section 2.5 discusses sharded PStores and PStore as a service that may process some PStore reads/writes while resulting in failed writes at the same time.

To enhance performance, TARDIS does not prevent all undesirable race conditions. Instead, it detects their occurrences and resolves them. Pseudo-codes of this section highlight this design decision, e.g., see discussion of Step 6 of Algorithm 3.

Overview: 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 failed writes in a timely manner. A failed PStore write transitions AppNode to failed mode. In this mode, App 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, a request that references a document with buffered writes in the cache causes AppNode to propagate the buffered write to PStore. This lazy technique may result in a long recovery phase when documents with buffered writes are not referenced by a request for a long time. AR workers expedite recovery phase by propagate buffered writes to PStore eagerly. The number of AR workers is a configurable parameter. A large number of AR workers propagate buffered writes to PStore quickly. This may impose a high load on PStore and slow down AppNode's writes and reads (due to cache misses) to PStore. Once the AR workers apply all buffered writes to PStore, an AR worker switches AppNode back to normal mode.

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

Once the AR workers apply all buffered writes to PStore, an AR worker switches AppNode back to normal mode.

TARDIS employs stateless leases such as Redlease[4] [7] to prevent undesirable race conditions between AppNode and AR workers. A lease is granted on a key to provide mutual exclusion for requests that reference this key. A lease has a fixed lifetime. Once a lease on a key is granted to a caller (say AppNode), the request by an AR worker must back off and try again. The duration of backoff is based on an exponential decay. The pending request succeeds once either AppNode releases its lease or the lease expires.

Below, we detail normal, failed, and recovery modes in turn. This discussion uses the terminology of Table 2.

2.1 Normal mode

In normal mode of operation, an AppNode thread processes a read action by looking up the value of one or more keys $\{K_i\}$ in the cache. If it is not found then this action has observed a cache miss. The AppNode thread queries PStore for the document D_i to compute the missing key-value pair(s) and stores them in the cache for future look up. Finally, it provides a response for the action.

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.

Table 2: List of terms and their definition.

[4]Redlease [7] can be implemented by AppNode with no changes to the memcached/Redis server.

With a write-action, we assume a write-through policy where an AppNode thread updates both the impacted PStore document(s) and their corresponding cached key-value pairs.

Concurrent reads and writes of the same key-value pair may suffer from undesirable race conditions, causing the cache to produce stale data. The I and Q leases of [21] prevent these undesirable race conditions. Moreover, they implement the concept of a session that provides atomicity at the granularity of multiple key-value pairs. These leases are detailed in Section 3.2.2.

2.2 Failed mode

In failed mode, an AppNode thread processes a write for a PStore document D_i with primary key P_i by generating a change, δ. The AppNode thread appends δ to the value of a key Δ_i in the cache, creating (Δ_i,δ) if it does not exist. The value of Δ_i is a chronological order of failed writes applied to D_i, $\{\delta_1, \delta_2, ..., \delta_j\}$. This list is a *buffered write*.

In addition, the AppNode thread 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.

Note that 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 and write it to the PStore. However, assuming memory is not a limiting factor, a developer may also generate a list of changes Δ_i for the document to expedite recovery time. This optimization applies when it is faster to read and process Δ_i instead of 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 δ=push(P_k,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. This approach requires more memory due to duplication of changes in the cache.

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 PStore write fails, the AR waits for some time before repeating the write with a different document. This delay may increase exponentially up to a maximum threshold. Once a fixed number of AR writes (say p) succeeds, an AR worker transitions the state of AppNode to recovery.

TARDIS prevents contention for TeleW and one cache server by maintaining ω TeleW key-value pairs. It hash partitions documents across these using their primary key P_i, see Figure 3. 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.

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

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$_i$
   }

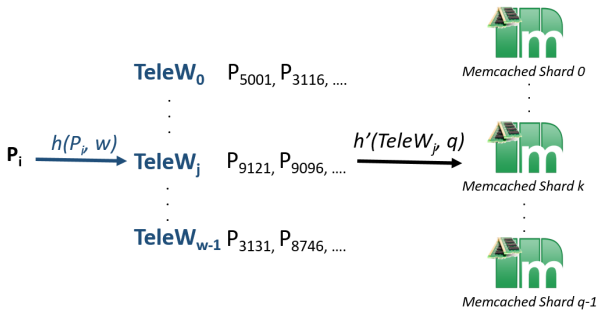


Figure 3: Documents are partitioned across ω TeleW keys. TeleW keys are sharded across q memcached servers.

by generating a key P_i+dirty with value dirty[5]. The memcached `Add` command inserts this key if 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.

2.3 Recovery mode

In recovery mode, TARDIS employs AR workers eagerly and AppNode lazily to apply buffered writes to those PStore documents identified by 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:

1. Correctness: TARDIS must propagate buffered writes to PStore documents in a consistent manner.
2. Performance: TARDIS must process user's actions as fast as normal mode while completing recovery quickly.

Correctness: We categorize buffered writes into idempotent and non-idempotent. Idempotent writes can be repeated multiple times and produce the same value. Non-idempotent

[5]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.

writes lack this behavior. An example idempotent write is to change the value of a property, say name, of a document to a new value, say Bob. A non-idempotent change is to add a photo to Bob's album. Repeating this non-idempotent change N times results in an album with N copies of the same photo. However, repeating the idempotent change N times is side-effect free.

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.

When the AppNode performs a write action using a document with buffered idempotent writes, an AR worker should not apply the same idempotent write again. This undesirable race condition loses the AppNode's write action.

Our implementation of recovery uses leases to apply a buffered write to a PStore document once. Either by an AR worker or the AppNode. We assume the likelihood of a user action referencing a document with buffered writes is small in recovery mode. While our design is functional when this assumption is violated, it maybe possible to develop more efficient designs per discussions at the end of this section.

Algorithm 2 AppNode in Recovery Mode (P_i)

```
1. acquire lease P_i
2. V = get(P_i+dirty)
3. if V exists {
       success = Update D_i in PStore using its buffered writes
       if success {
           delete(P_i+dirty)
           delete Δ_i (if any)
       }
   }
4. release lease P_i
```

In 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, see Algorithm 2. Once the lease is granted, it looks up the key P_i+dirty. 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[6], AppNode deletes P_i+dirty key to prevent an AR worker from applying buffered writes to D_i a second time.

Performance: To enhance performance of user issued requests, AppNode does not update the value of TeleW keys after updating D_i in PStore. This task is left for the AR workers as detailed next.

Algorithm 3 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 sill 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

[6]This fails if PStore is unavailable.

Algorithm 3 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 $\cup$ P_i
         }
      } **else** R = R $\cup$ P_i
      release lease P_i
   } **else** R = R $\cup$ 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$_T$
}

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. Section 4.3 analyzes the value of α and its impact on recovery time.

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 3. Duplicate P_is may exist in TeleW$_T$. A race condition involving AppNode and AR worker generates these duplicates: Algorithm 1 executes between Steps 5 and 6 of Algorithm 3. 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.

2.4 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, TARDIS replicates buffered writes across two or more cache servers. Both Redis [2] and memcached [3] 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, replication is still required to process reads and writes of an AppNode in recovery mode. Otherwise, buffered writes become unavailable while the cache node recovers. During this time, to preserve consistency, an AppNode in recovery mode may not process reads and writes. There is a potential buffered write stored in the failed cache node. This possibility prevents processing of reads and writes even when PStore is available. These requests must employ client side solutions[7] until the cache server recovers and makes its (potential) buffered writes available. This requirement is eliminated by replicating buffered writes across multiple cache servers to render them available in the presence of cache server failures.

2.5 Discussion

In failed mode, an AppNode thread does not perform PStore writes that result in repeated timeouts. Instead, it maintains a volatile boolean variable, PStoreAvailable, that is shared by AppNode threads and AR threads. When PStoreAvailable is true, it means PStore is available and AppNode performs writes using PStore. Otherwise, PStore is unavailable and AppNode generates buffered writes. The AppNode sets PStoreAvailable to false when it detects a failed write. An AR worker sets PStoreAvailable to true when it succeeds recovering a document by writing it to PStore. Use of PStoreAvailable is orders of magnitude faster then timing out by writing to a failed PStore. We do not use atomic variables or protect the value of PStoreAvailable using latches. In the presence of a race condition and in the worst case scenario, the AppNode threads perform a few more failed PStore writes than necessary. In our experiments, implementing PStoreAvailable using Java volatile provides a throughput that is 3% to 12% higher than using Java atomic.

We used P_i+dirty to simplify discussion. One may replace it with Δ_i as follows. When the AppNode buffers a write by updating cached key-value pairs (without proposing a change), the value of Δ_i is set to "dirty". Otherwise, the value of Δ_i reflects a sequence of changes that may grow to be very large depending on the number of updated performed during failed mode. With the latter, use of P_i+dirty may be more efficient as its size is a constant, see Step 2 of Algorithm 2 and Step 5 of Algorithm 3.

Algorithm 2 does not show several important details of the AR. First, in Step 5, every time the request for a lease on P_i backs off, the AR worker abandons P_i and moves to the next document in $\{D\}$. This is because either another AR worker or the AppNode is recovering

[7] Or some other server side solution.

D_i and it is unreasonable for this worker to wait for the lease only to discover that buffered writes for D_i have already been applied (by not finding P_i+dirty in the cache).

Second, every time an AR woker detects an empty TeleW$_T$ key, it fetches the value of TeleW$_{(T+1)\%\omega}$. If all ω TeleW keys have an empty value, the AR worker attempts to acquire a lease on all TeleW keys. If one of its ω leases is denied, the AR worker does not back off and try again. Instead, it releases all its TeleW leases, sleeps for a random amount of time between a minimum and a maximum threshold, and tries to obtain its ω leases from the start. Once the AR worker has its ω leases on ω TeleW keys, it looks up the value of these TeleW keys in the cache. If they have no value or their value is the empty set then the AR worker transitions the system to normal mode of operation by setting the value of PStoreAvailable to true. Next, it releases all its ω leases.

With a sharded PStore, one or more AppNode threads may observe failed writes by referencing a shard that is overloaded. This transitions AppNode to failed mode. However, the AR worker may immediately apply the buffered write to that shard and succeed, transitioning the system to recovery mode and normal mode quickly. This may cause some AppNode threads to operate in different modes; potentially all different threads may exhibit all three modes at the same time. The pseudo-code of Algorithms detailed in this section is robust enough to preserve the correctness property. Moreover, performance observed by AppNode threads in failed mode is faster than normal mode. In recovery mode, an AppNode thread may be slower when it propagates the buffered write to PStore, see Section 4.3.

Note that the failed mode of operation is for PStore writes only. PStore reads are beyond the focus of this study. For our purposes, we assume AppNodes process a cache miss by querying PStore always. Once this fails, the system may exercise alternatives including application specific solutions such as providing stale data or displaying advertisements. (Focus of this paper is on writes.)

With PStore as a service, the writes may fail even though the reads succeed because AppNode has exhausted its provisioned write capacity. In this case, the writes are buffered in the cache and the AR worker succeeds in applying them once the AppNode's write capacity is restored. It is possible to design intelligent techniques that prioritize writes, requiring TARDIS to perform the high priority ones using PStore while buffering the low priority ones. TARDIS may propagate these buffered writes to PStore periodically while ensuring all high priority writes are processed successfully. This technique is a future research direction, see Section 6.

3 TARDIS with N AppNodes

TARDIS of Section 2 supports multiple AppNodes when requests are partitioned such that a document is referenced by one AppNode. Otherwise, it results in undesirable race conditions that compromise application correctness. This happens when AppNodes have different mode of operations for PStore. This section presents the undesirable race conditions and two alternative designs for TARDIS to prevent these race conditions.

3.1 Undesirable race conditions

With multiple AppNodes and intermittent failed PStore writes, different AppNodes may operate in different modes. While AppNode 1 operates in failed or recovery mode, AppNode 2 may operate in normal mode[8]. 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.

To illustrate an undesirable write-write race condition, consider a user Bob who creates an Album and adds a photo to the Album. Assume Bob's "create Album" is directed to AppNode 1 for processing. 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.

This write-write race condition would not have occur if Bob's requests were directed to AppNode 1: AppNode 1 would have recovered Bob's pending Album write prior to performing the photo add operation, see Section 2.

An undesirable read-write race condition is when 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.

The presented read-write race condition requires a cache miss. If Bob's profile was in the cache then it would have been updated by AppNode 1, preventing Alice from seeing his updated status.

3.2 Two Solutions: Recon and Contextual

One approach to address the undesirable race conditions of Section 3.1 is to require all AppNodes to have the same PStore mode of operation. This results in unnecessary complexity and is expensive to realize with a large number of AppNodes. Moreover, it does not work when the PStore is partially available per discussions of Section 1.

We presents two solutions that do not require consensus among AppNodes. The first combines the normal and recovery mode into one, resulting in each AppNode to operate in two possible modes: recon (combined recovery and normal) and failed. The second employs stateful leases. It implements the concept of sessions that supports atomicity across multiple keys along with commit and rollbacks. We describe these techniques in turn.

3.2.1 Solution 1: Recon

Recon combines recovery and normal mode of operation into one by requiring:

[8]A lightly loaded system dominated by reads such that AppNode 2 processes reads with 100% cache hit while AppNode 1 observes a failed write.

- Cache hits to be processed in the same manner as the normal mode of Section 2.1.
- All cache misses and application writes are processed in recovery mode. They must look up P_i+dirty to discover if their referenced document has buffered write. If so, they apply the change to PStore prior to processing the request.

Recon avoids the write-write race condition of Section 3.1 by requiring all writes to operate in recovery mode. In our example, Bob's photo add operation is a write action and it is forced to recover Bob's pending write that creates the album prior to adding the photo.

Similarly, Recon avoids read-write race condition of Section 3.1 by requiring Alice's cache miss for Bob's profile to process pending writes for Bob's document. Once these changes are applied, Alice is not able to see Bob's change of status.

Recon is simple to implement. Its overhead is during normal mode of operation, namely the requirement for all cache misses and write actions to perform additional work than necessary by looking up P_i+dirty. This overhead is marginal for those applications whose workload exhibits a high ratio of reads to writes [12] with a high cache hit rate.

3.2.2 Solution 2: Contextual Leases

TARDIS with contextual leases (Contextual for short) maintains the three mode of operation described in Section 2. It incorporates I and Q leases of [21] and extends them with a marker to prevents the undesirable race conditions of Section 3.1. Below, we provide an overview of the I and Q leases as used during normal mode. Subsequently, we describe our extensions with a marker.

The framework of [21] 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. Given an existing I or Q lease on a key, a request for an I or a Q lease for the same key is as follows:

Requested Lease	Existing Lease	
	I	Q
I	Backoff	Backoff
Q	Void I & grant Q	Deny and abort

If there is a request for an I lease on a key with an existing I lease then the requester must backoff and try again. This avoids thundering herds by requiring only one out of many requests that observes a cache miss to query PStore and populate the cache. Ideally, all other requests should observe a cache hit in their retry.

If there is a request for a Q lease on a key with an existing I lease then there is a potential read-write race condition. The Q lease voids the I lease, preventing the reader from populating the cache with a value. This prevents the reader from inserting a stale value (computed using a the state of PStore prior to write) in the cache.

If there is a request for an I lease on a key with an existing Q lease then, once again, we have a potential read-write race condition. The I lease backoffs and tries again until the writer commits and releases its Q lease.

Finally, if there is a request for a Q lease on a key with an existing Q lease then we have a write-write race condition. In this case, the requester is aborted and restarted, forcing the

R-M-W of one write to observe the R-M-W of the other. It is acceptable for the Q lease requester to back off and try again if a R-M-W action references only one key - however, as noted in Section 2, a write may R-M-W a collection of keys $\{K_i\}$.

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 number of keys with such markers increases monotonically as they are written in failed mode.

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.

To explain why Contextual prevents the undesirable race conditions of Section 3.1, observe: 1) there is at most one lease on a key, either I or Q, 2) an I lease is requested when a read action observes a cache miss, and 3) a Q lease is requested by a write action. When there is a buffered write for a document, either an I or a Q lease on its P_i encounters a P marker that causes the requesting AppNode to propagate the buffered write to the document in PStore. In essence, the mode of operation is at the granularity of a key.

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.

3.3 Non-idempotent Buffered Writes

A buffered write may be a non-idempotent change to a document. For example, it may increment a property or insert a value in an array property of a document. During recovery, when an AppNode thread or an AR worker applies a buffered write to PStore, it may encounter a time out (a failed PStore write) even though the write succeeded. An obvious solution is to repeat the operation. While this is acceptable with idempotent buffered writes, it is not acceptable with non-idempontent buffered writes because applying them two or more times may violate consistency of the database.

One possible approach is to extend the data store with application specific constraints that prevent inconsistent states due to multiple applications of a buffered write. For example, with a change that pushes a value into an array property of MongoDB and advanced knowledge that the array may not contain duplicate values, one may specify a constraint that requires the values in an array property to be unique. This prevents multiple applications of a non-idempotent buffered write from violating database consistency. Such approaches may not be possible with all persistent data types and their use by an application. This section presents SnapShot Recovery (SSR) to address this challenge.

SSR, see Algorithm 4, is used in recovery mode when applying a buffered write to PStore.

Algorithm 4 Snapshot Recovery (P_i)

```
1: acquireLease(P_i)
2: S_i ← Get cached snapshot of D_i using P_i
3: δ_i and uid ← Get Δ_i
4: if S_i ≠ null then
5:     if δ_i = null or S_i.uid ≠ uid then
6:         D_i ← Get document P_i from PStore
7:         if D_i ≠ S_i.data then
8:             if Overwrite D_i with S_i.data = fail then
9:                 releaseLease(P_i)
10:                return fail
11:    Delete S_i
12: if δ_i ≠ null then
13:    D_i ← Get document P_i from PStore
14:    D_i ← Apply δ_i to D_i
15:    S_i ← Init snapshot
16:    S_i.data ← D_i
17:    S_i.uid ← uid
18:    Store S_i in the cache
19:    Delete Δ_i
20:    if Apply δ_i to PStore = fail then
21:        releaseLease(P_i)
22:        return fail
23:    Delete S_i
24: releaseLease(P_i)
25: return success
```

Its essence is to convert a buffered non-idempotent write for D_i into an idempotent buffered write by generating a snapshot of D_i. This snapshot is written to the cache and deleted immediately if the application of buffered write to PStore succeeds. It is used to detect failures during recovery and to apply a non-idempotent write once. Algorithm 4 is robust enough to enable a (different) AppNode to operate in failed mode and generate non-idempotent changes concurrently.

Details of SSR are as follows. The value of Δ_i includes a unique id (uid) generated[9] by AppNode when it creates this key for a failed write of D_i the very first time. In Steps 13, SSR reads the PStore document D_i in memory and applies the non-idempotent changes to it to create the snapshot. In Steps 16-19, SSR initializes the snapshot S_i key-value pair and inserts it in the cache. Next, it applies the buffered write to D_i in PStore, and deletes the snapshot S_i. Note that the snapshot maintains the uid of Δ_i.

When an AppNode thread or an AR worker applies a buffered write to D_i, SSR looks up its snapshot first (see Step 3). If it exists then SSR has detected a failure when a buffered write was being applied to PStore. Hence, it fetches D_i from PStore and compares it with

[9]An integer counter concatenated with the mac address of AppNode server.

S_i. If they are not equal then SSR over-writes D_i with S_i and deletes S_i. Subsequently, it applies new buffered writes (if any).

Note that the writing of D_i to PStore, Steps 8 and 20, may encounter a failed write. Hence, SSR returns an error to the calling thread, either AppNode or AR worker. If it is an AR worker, this thread may invoke SSR immediately or after some delay. Alternatively, it may ignore the failure and let another AR thread try it at a later time. An AppNode thread encounters this failure when performing a read or a write. With a read, it may ignore the failure and consume the value from the cache, deferring to an AR worker to apply the buffered write to PStore. Similarly, with a write, an AppNode thread may defer to an AR worker by generating a buffered write.

The uid is used when there is a failure after Step 19 and before Step 23, e.g., the condition of Step 20 is satisfied. In this case, the buffered write Δ_i is no longer present in the cache. However, its snapshot S_i is available. Should an AppNode thread in failed mode create a buffered write Δ_i then it will a uid different than S_i's uid. Steps 6-11 overwrite the PStore document D_i with S_i. Subsequently, SSR applies the new Δ_i to this D_i.

The *overhead* of SSR is writing and deleting the snapshot of a document impacted by a buffered write prior to its application to PStore, see Steps 18 and 23. This overhead is incurred during recovery mode.

4 Evaluation

We used YCSB [15] and BG [10] to evaluate TARDIS. We focused on YCSB's update heavy workload A consisting of 50% read and 50% update. With BG, we used its moderate read heavy workload of 100 reads for every 1 write. It consists of the following mix of interactive social networking actions: 89% View Profile, 5% List Friends, 5% View Friend Request, 0.4% Invite Friend, 0.2% Accept Friend Request, 0.2% Reject Friend Request, and 0.2% Thaw Friendship.

We evaluated the following cache mangers: Redis 3.2.8, CAMP 1.0, Twemcached 2.5.3, memcached 1.4.36 with and without chunked version configuration. The later supports key-value sizes grater than 1MB. CAMP is a variant of memcached with two changes. First, it uses malloc and free to manage memory space instead of a slab implementation. Second, it uses the CAMP replacement algorithm [20] that is cost aware instead of LRU.

Variants of memcached implement the same commands. Redis provides richer commands and data types along with server scripting, resulting in a different implementation. Given a fixed amount of cache size, different caches may store a different number of buffered writes, see Section 4.2.

Physical representation of buffered writes with YCSB is different than BG because their writes are different in nature. A YCSB update overwrites values of several properties of a document. A BG write action such as "Invite Friend" pushes the member id of the inviter into the invitee's list of pending friend invitations[10]. Section 4.1 details representation of buffered writes with YCSB and BG.

We extended the workload generator of YCSB and BG to implement Recon and Contextual. With both implementations, the same physical YCSB and BG database organizations

[10]This is represented as an array data type.

are used in MongoDB and a cache manager. Both benchmarks databases consist of 100,000 MongoDB documents. Moreover, both are configured to generate a skewed pattern of data access using a Zipfian distribution. In the reported experiments, we emulate a fixed duration of failed writes by terminating the mongod process of MongoDB and restarting it after the specified duration.

In these experiments, a multi-node workload generator emulates multiple AppNodes. These are connected to a server hosting MongoDB version 3.2.9 as our PStore and a second server hosting Twemcache version 2.5.3 as our cache server. Each server is an off-the-shelf PC configured with 4-Core Intel i7-3770 3.40 GHz CPU 16 GB of memory and 1 gigabit per second (Gbps) networking card.

In the following, we detail physical design of buffered writes with YCSB and BG. Subsequently, we report on memory requirement of TARDIS, its recovery time, and horizontal scalability characteristics.

4.1 Physical Design of Buffered Writes

A buffered write with both benchmarks is a key-value pair. The key Δ_i identifies a unique MongoDB document impacted by a write. Its value is a buffered write.

With Redis, a buffered write for YCSB is a hash map consisting of a pairing of the property name with the updated value. Hence, there is a maximum bound on the size of a buffered write with YCSB. It is defined by the number of properties of a document (10). Moreover, generation of a buffered write is one network round-trip that sets the property:value pairings in the hash map.

Variants of memcached do not support a hash map data type. Hence, we represent a YCSB buffered write as a list of property-name:value pairing. Generation of this buffered write performs two network round-trips by implementing a read-modify-write operation.

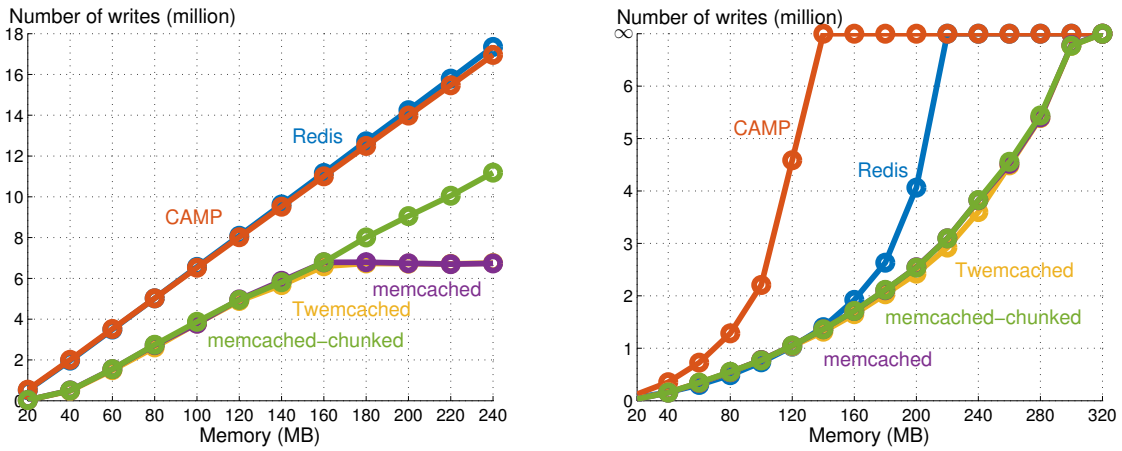
With BG, a buffered write is a list of logical operations such as a push or a pop of a value from an array property of MongoDB document. This implementation is used with all cache managers. We make the logical operations idempotent by configuring the array property of MongoDB documents to contain unique values. Hence, this implementation does not use the SSR technique of Section 3.3.

4.2 Memory Requirement of TARDIS

The memory required for TARDIS to cache buffered writes depends on the characteristics of the workload. In Figure 4, we configure a cache manager with a fixed amount of memory (x-axis) and issue failed writes until the cache manager either refuses to insert the buffered write or evicts a buffered write. The y-axis of this figure is the number of buffered writes.

The YCSB results, Figure 4.b, show all cache managers are able to accommodate an infinite number of buffered writes beyond a certain memory limit. There are two reasons for this. First, the number of documents is fixed at 100,000. Second, the size of a buffered write for a document is fixed and *independent* of the number of failed writes performed on that document. Once all YCSB documents are referenced by a failed write, the memory requirement of TARDIS levels off.

The implementation of a YCSB buffered write with CAMP, Twemcached, and memcached



(a) BG (b) YCSB

Figure 4: Number of writes processed as a function of the amount of available memory

are identical and have the same size. This size is more compact than the hash map of Redis. Both CAMP and Redis use malloc and free to manage the available space. CAMP reaches infinity with a lower memory because of its smaller key-value size. Twemcached and memcached manage memory space using a slab implementation. This implementation suffers from calcification [27, 26], requiring more memory to accommodate an infinite number of failed writes.

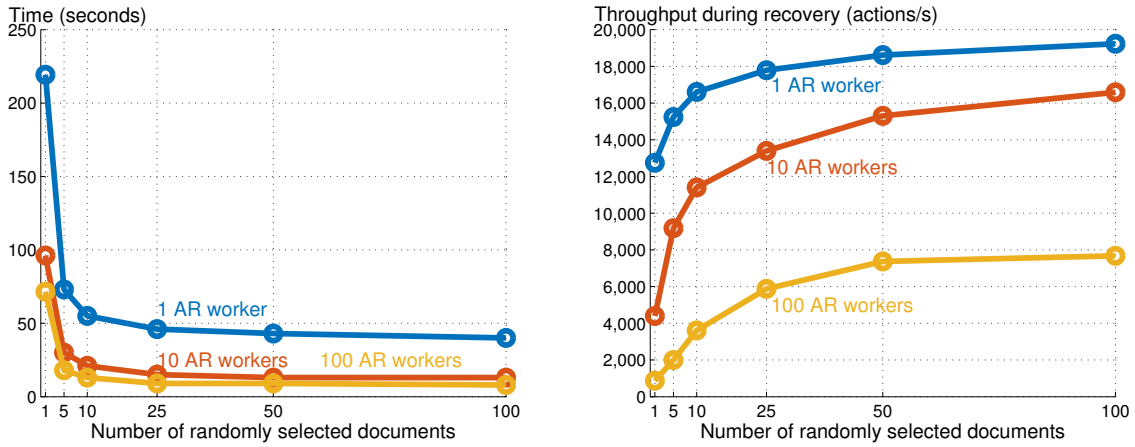
With BG, the size of a buffered write for a document increases monotonically as a function of the number of failed writes referencing the document. There is no memory size that accommodates an infinite number of writes, see Figure 4. Redis and CAMP support approximately the same number of writes because the size of a buffered write is the same with all cache managers. Twemcached and memcached variants continue to suffer from slab calcification. Both Twemcached and memcached cannot support additional writes beyond 160 MB of memory because the maximum key-value size is 1 MB. Chunked variant resolves this limitation to utilize memory capacities higher than 160 MB.

4.3 Recovery Time

We use YCSB and BG to analyze recovery time by varying the number of AR workers and the number of documents selected randomly by each worker, α, see Figure 5.a. In these experiments, the number of TeleW keys is set to 211 and assume a low system load of 20 threads issuing requests during recovery.

We observe that the recovery time reduces an order of magnitude from 1 AR worker to 10 AR workers across all alpha values. A large number of AR workers with a small α value is superior to having a few AR workers with a high α value. 100K buffered documents (due to a 20 minute failure) are recovered in less than 10 seconds with a large number of AR workers (i.e. 10 and 100) and alpha value of 50. Contextual and Recon provide similar performance. This is at the expense of a lower throughput observed with 20 threads issuing requests, see Figure 5.b.

TARDIS's recovery time is faster with BG because its workload is read heavy. Its con-



(a) Recovery duration

(b) Throughput during recovery

Figure 5: Recon Recovery with YCSB as a function of the number of AR workers and the number of randomly selected documents (α) in each iteration.

tention for PStore is lower than YCSB, enabling AR workers to complete recovery faster. However, the observed trends are similar to those of YCSB.

Results of this section highlight the following tradeoff. A faster recovery time is at the expense of a lower observed performance by the foreground requests. The number of AR workers is a knob that enables an application to control this tradeoff.

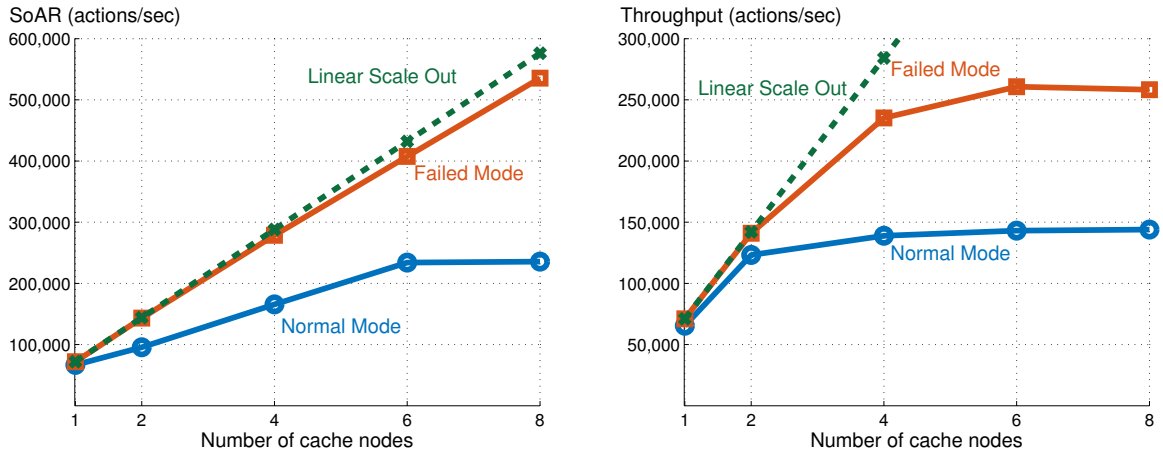
4.4 Horizontal Scalability of TARDIS

A key metric of BG is SoAR, the throughput provided by a system while satisfying a pre-specified service level agreement (SLA). The y-axis of Figure 6a.a shows SoAR of TARDIS as a function of the number of cache servers, x-axis. The SLA requires 95% of actions to observe a response time faster than 100 milliseconds. We show the observed SoAR in both normal mode of operation with Recon and failed mode when TARDIS generates buffered writes. Figure 6a.b shows the scalability of the caching layer relative to the SoAR observed with one cache server.

Obtained results show TARDIS performance and scalability in failed mode is superior to normal mode of operation. In failed mode, the framework scales almost linearly as a function of the number of cache servers. However, the system struggles to scale in normal mode, with performance leveling off beyond 6 servers. In normal mode, Recon performs a write action using both the cache and MongoDB. The server hosting MongoDB is the bottleneck limiting performance. Even with one cache server, the CPU of the server hosting MongoDB spikes from 20% to 100%. Beyond 4 cache servers, the CPU of the server hosting MongoDB becomes 100% utilized. This causes the throughput to level off.

In failed mode, all writes insert buffered writes in the caching layer because MongoDB is not available. These operations are hash partitioned using the document id, enabling the system to scale almost linearly. It is not perfectly linear due to temporary bottlenecks caused by concurrent requests colliding and referencing the same cache server.

Horizontal scalability of the system with YCSB is limited because 50% of requests are



(a) BG (CAMP)

(b) YCSB (Redis)

Figure 6: TARDIS horizontal scalability.

updates, see Figure 6b. These results were obtained with 4 YCSB clients issuing requests. Each client is configured with 200 threads, for a total of 800 threads generating requests concurrently. Prior to updating a key-value pair, the YCSB client obtains a lease on the key-value pair. When two or more concurrent threads try to update the same key-value pair, one succeeds while others back-off and try again. This isolates and serializes concurrent updates. It also causes cache servers to sit idle and wait for work, reducing performance and resulting in sub-linear scalability. The failed mode is more efficient than the normal mode as it generates a buffered write instead of updating MongoDB. By processing a larger number of actions per second, it also observes more than 25% of requests to back-off than normal mode.

5 Related work

The CAP theorem states a system designer must choose between strong consistency and availability in the presence of network partitions [11, 33]. TARDIS improves availability while preserving consistency.

A weak form of data consistency known as eventual has multiple meanings in distributed systems [41]. 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 [16, 42, 9]. TARDIS teleports failed writes due to either network partitions, PStore failures, or both while preserving consistency.

The race conditions encountered during recovery (see Section 3.1) are similar to those in geo-replicated data distributed across multiple data centers. Techniques such as causal consistency [31, 32, 30] and lazy replication [29] mark writes with their causal dependencies. They wait for those dependencies to be satisfied prior to applying them at a replica, pre-

serving 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.

Neumerous studies perform writes with a mobile device that caches data from a database (file) server. (See [40, 37] 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, 28, 13, 25, 24] (HsC) such as Flashcache [34] and bcache [39] 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 propogates 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 [22], Apache Ignite [8], KOSAR [19], and others.

6 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.

We presented alternative implementations of TARDIS with Redis and three variants of memcached. The memcached variant using CAMP [20] is most space efficient. Experimental result using YCSB Workload A show it can cache an infinite number of buffered writes with a few hundred MBs of memory.

TARDIS provides a higher throughput in failed mode than in normal mode. This motivates operating a system in failed mode with AR workers propagating buffered writes to PStore in the background. This is trivial to do with Recon by extending it to perform all

writes in failed mode. The resulting system would be similar to the write-back policy of host side caches [34, 13, 25]. To preserve durability in the presence of cache server failures, the system may replicate buffered writes across caches per discussion of Section 2.4. A comprehensive evaluation and an understanding of tradeoffs associated with such a design decision is a future research direction.

More longer term, we intend to investigate extensions of TARDIS to those logical data models that result in dependence between buffered writes. To elaborate, while there is no dependence between MongoDB documents, the 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 Persistence, https://redis.io/topics/persistence.

[2] Redis Replication, https://redis.io/topics/sentinel.

[3] repcached: Add Data Replication Feature to Memcached 1.2.x, http://repcached.lab.klab.org/.

[4] Alabdulkarim, Y., Almaymoni, M., Cao, Z., Ghandeharizadeh, S., Nguyen, H., and Song, L. A comparison of Flashcache with IQ-Twemcached. In *ICDE Workshops* (2016).

[5] Amazon DynamoDB. https://aws.amazon.com/dynamodb/ and https://aws.amazon.com/dynamodb/pricing/, 2016.

[6] Amazon ElastiCache. https://aws.amazon.com/elasticache/, 2016.

[7] (antirez), S. S., and Kleppmann, M. Redlease and How To do distributed locking. http://redis.io/topics/distlock and http://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html.

[8] Apache. Ignite - In-Memory Data Fabric, https://ignite.apache.org/, 2016.

[9] Bailis, P., Venkataraman, S., Franklin, M. J., Hellerstein, J. M., and Stoica, I. Quantifying eventual consistency with PBS. *Commun. ACM 57*, 8 (2014).

[10] Barahmand, S., and Ghandeharizadeh, S. BG: A Benchmark to Evaluate Interactive Social Networking Actions. *CIDR* (January 2013).

[11] Brewer, E. Towards Robust Distributed Systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (2000).

[12] Bronson, N., Lento, T., and Wiener, J. L. Open Data Challenges at Facebook. In *ICDE* (2015).

[13] Byan, S., Lentini, J., Madan, A., Pabon, L., Condict, M., Kimmel, J., Kleiman, S., Small, C., and Storer, M. Mercury: Host-side Flash Caching for the Data Center. In *MSST* (2012).

[14] Cattell, R. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec. 39* (May 2011), 12–27.

[15] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing* (2010).

[16] 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 *SOSP* (2007).

[17] Fan, B., Andersen, D. G., and Kaminsky, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI* (2013).

[18] Fitzpatrick, B. Distributed Caching with Memcached. *Linux Journal 2004*, 124 (2004).

[19] Ghandeharizadeh, S., and et. al. A Demonstration of KOSAR: An Elastic, Scalable, Highly Available SQL Middleware. In *ACM Middleware* (2014).

[20] Ghandeharizadeh, S., Irani, S., Lam, J., and Yap, J. CAMP: A Cost Adaptive Multi-Queue Eviction Policy for Key-Value Stores. *Middleware* (2014).

[21] Ghandeharizadeh, S., Yap, J., and Nguyen, H. Strong Consistency in Cache Augmented SQL Systems. *Middleware* (December 2014).

[22] Google. Guava: Core Libraries for Java, https://github.com/google/guava, 2015.

[23] Google App Engine Pricing, Apps are free within a usage limit that is reset daily. https://cloud.google.com/appengine/, 2016.

[24] Graefe, G. The Five-Minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. In *DaMoN* (2007), p. 6.

[25] Holland, D. A., Angelino, E., Wald, G., and Seltzer, M. I. Flash Caching on the Storage Client. In *USENIXATC* (2013).

[26] Hu, X., Wang, X., Li, Y., Zhou, L., Luo, Y., Ding, C., Jiang, S., and Wang, Z. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (July 2015).

[27] Irani, S., Lam, J., and Ghandeharizadeh, S. Cache Replacement with Memory Allocation. *ALENEX* (2015).

[28] Kim, H., Koltsidas, I., Ioannou, N., Seshadri, S., Muench, P., Dickey, C., and Chiu, L. Flash-Conscious Cache Population for Enterprise Database Workloads. In *Fifth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures* (2014).

[29] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. Providing high availability using lazy replication. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 360–391.

[30] Lesani, M., Bell, C. J., and Chlipala, A. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *POPL* (2016).

[31] 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 *SOSP* (2011).

[32] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual consistency. *Commun. ACM 57*, 5 (May 2014), 61–68.

[33] Lynch, N., and Gilbert, S. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT New 33* (2002), 51–59.

[34] Mituzas, D. Flashcache at Facebook: From 2010 to 2013 and Beyond, https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920, 2010.

[35] MongoDB Atlas. https://www.mongodb.com/cloud, 2016.

[36] 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., and Venkataramani, V. Scaling Memcache at Facebook. In *NSDI* (Berkeley, CA, 2013), USENIX, pp. 385–398.

[37] Pitoura, E., and Bhargava, B. Maintaining Consistency of Data in Mobile Distributed Environments. In *Proceedings of the 15th International Conference on Distributed Computing Systems* (1995).

[38] Ports, D. R. K., Clements, A. T., Zhang, I., Madden, S., and Liskov, B. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI* (October 2010), USENIX.

[39] Stearns, W., and Overstreet, K. Bcache: Caching Beyond Just RAM. https://lwn.net/Articles/394672/,http://bcache.evilpiepirate.org/, 2010.

[40] 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 *SOSP* (1995).

[41] Viotti, P., and Vukolić, M. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv. 49*, 1 (June 2016).

[42] Vogels, W. Eventually Consistent. *Communications of the ACM, Vol. 52, No. 1* (January 2009), 40–45.