High availability in cheap distributed key value storage

Thomas Kim
Carnegie Mellon Univ

Daniel Lin-Kit Wong
Carnegie Mellon Univ

Gregory R. Ganger
Carnegie Mellon Univ

Michael Kaminsky
Carnegie Mellon Univ, BrdgAI

David G. Andersen
Carnegie Mellon Univ, BrdgAI

Abstract

Memory-based storage currently offers the highest-performance distributed storage, keeping the primary copy of all data in DRAM. Recent advances in non-volatile main memory (NVMM) technologies promise latency similar to DRAM at reduced cost and energy, but will make providing high availability more challenging. Previous approaches to failure recovery involve maintaining multiple identical replicas or relying on fast offline restoration of data from backup replicas stored on SSD. Unfortunately, NVMM's combination of lower write throughput and increased storage density means that offline restoration can no longer provide sufficiently fast recovery, and maintaining multiple identical replicas is generally cost prohibitive.

CANDStore is a strongly consistent, distributed, replicated key-value store that uses a new fast crash recovery protocol. As a result, CANDStore can use NVMM and NVMe SSD technology to provide low-latency distributed storage that is cheaper and higher-availability than existing main memory-based distributed storage. Our evaluation shows that CANDStore's recovery protocol enables the system to restore performance and meet SLOs after the failure of a primary node 4.5–10.5x faster than offline recovery.

CCS Concepts

• **Computer systems organization** → **Availability**; **Redundancy**.

Keywords

persistent memory, distributed storage, fault tolerance

ACM Reference Format:
Thomas Kim, Daniel Lin-Kit Wong, Gregory R. Ganger, Michael Kaminsky, and David G. Andersen. 2020. **High availability in cheap distributed key value storage**. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3419111.3421290

1 Introduction

Distributed storage systems that provide high performance, fault tolerance, and strong consistency are at the core of today's large-scale Internet services [11, 13, 15, 32]. Non-volatile main memories (NVMMs) offer an enticing design option for building these high performance storage systems. One might hope that they could offer the performance of DRAM-based systems such as RAMCloud [29] and FaRM [13], while providing durability and having lower total cost. Such hopes are not fanciful: compared to DRAM, one can pack about 4x more of Intel's Optane [20], for example, into a single machine at approximately 1/7th the cost and on average an order of magnitude less energy [4] [5] [6]. Unfortunately, today's NVMMs come with drawbacks that complicate the design of failure recovery in distributed storage systems. Overcoming these drawbacks at the design level is the focus of this paper. The combination of substantially lower write bandwidth (7x) [7] and higher density (4x) compared to DRAM means that the time to recover from a failed node is up to 28x longer than a DRAM-based system with a similar number of total DIMMs. Using traditional mechanisms that temporarily halt serving requests during recovery, each machine failure would take 28x longer to recover when using NVMM.

Our main contribution is enabling high availability when using NVMM, through our fast online recovery protocol, which we demonstrate through CANDStore, a consistent, available, non-volatile, distributed store, that leverages NVMM to provide low cost distributed key-value (KV) storage. We achieve this by leveraging temporal locality in datacenter workloads to enable online recovery. To implement online recovery with strong consistency, we design a Raft-based protocol that incorporates ideas from Cheap Paxos [23].

In CANDStore, we forego standard *offline* recovery approaches and focus on techniques to make *online* recovery feasible and performant. We leverage two observations that allow us to achieve this goal: first, real-world workloads have skewed request distributions. This property allows CANDStore to guide the online recovery process in a manner that

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10.
https://doi.org/10.1145/3419111.3421290

Storage type	**Cost per gigabyte (USD)**
DRAM	35.16 [1]
Intel Optane NVMM	4.51 [1]
SSD	0.32 [7]

Figure 1: Cost of different types of storage.

enables the majority of requests to be served at *near-peak* performance before all of the data is copied from a backup node's SSD to the new primary's main memory.

Second, we observe that the availability of a distributed storage system is defined by its ability to achieve performance service level objectives (SLOs). This allows CANDStore to tolerate some performance degradation as long as it meets SLOs. We use this performance leeway to, with degraded performance, serve requests for keys that are not yet populated on the new primary via point queries to one of the backups in the cluster, described in Section 5.3.

To provide consistency and fast failure recovery, CANDStore uses a decentralized distributed consistency protocol based on a modified version of Raft [27]. This protocol ensures that key-value updates are replicated consistently and durably, and that stale or inconsistent reads are never returned to the client. The protocol is designed to enable CANDStore to use a heterogeneous layout of nodes, described in Section 3 and Figure 2, including a backup that stores all keys and values on NVMe SSD to reduce the cost of replication. We add additional phases to the failure detection and leader election parts of the protocol aimed at minimizing the performance degradation experienced following primary failure, which we describe in Section 4. We discuss the design of our protocol for consistently handling non-primary failure in Section 6.

We evaluate the performance of our recovery protocol and find that it is possible to recover from primary node failure 4.5–10.5x faster than offline recovery approaches running on the same cluster configuration.

2 Background and challenges

CANDStore aims to provide low-cost, strongly-consistent storage with high availability. In Section 2.1, we precisely define availability in terms of meeting SLOs, and describe how that can simplify the problem of achieving high availability. Section 2.2 discusses how our system design and crash recovery protocol compare to standard approaches, and what advantages CANDStore provides. Finally, we describe our assumptions about the workload (Section 2.3) and data model (Section 2.4), and discuss how these characteristics enable our system to achieve high availability at low cost.

2.1 Availability and SLOs

Two important metrics, *mean time to failure* (MTTF) and *mean time to repair* (MTTR) [16, 31], help quantify a system's availability. MTTF measures how long, in expectation, a system operates normally before experiencing downtime. MTTR describes how long it takes to restore the system to its previous level of operation after a failure. The *availability* of the system is then $\frac{\text{MTTF}}{\text{MTTF+MTTR}}$. In our work, given that *time to failure* (TTF) is determined primarily by events we cannot control, such as unexpected hardware, OS, or software crashes, we design our online recovery protocol to minimize *time to repair* (TTR) in order to increase availability.

We use *Service Level Objectives* (SLOs) to provide a concrete and quantitative description of whether a system is down or not. SLOs describe guarantees about the performance of a storage system, which users depend on when building applications on top of distributed storage. SLOs describe the baseline performance that can be expected of the system, and violating these SLOs means that the storage system can no longer serve the function it was intended to – in other words, it becomes functionally unavailable. A system can be considered "down" if it is unable to meet its SLOs, and can be considered "repaired" once it has restored its ability to serve its SLOs. The TTR of a system is the duration of this downtime.

To provide high availability despite the various challenges posed by CANDStore's cost-saving design, which we describe in Section 3, we design a primary crash recovery protocol that focuses on restoring the system's ability to meet SLOs as quickly as possible. This is in contrast to typical crash recovery protocols for high performance distributed storage that instead aim to minimize the time needed to restore *full performance* to the system [29].

2.2 Limitations of offline recovery

A common approach to providing high-throughput, low-latency, and fault-tolerant KV storage is to use primary-backup replication. Modern primary-backup systems typically involve a single primary that replicates updates to one or more backups. In the event of node failure, typically an external authority (e.g., a configuration management system like Zookeeper [19]) alters the cluster's configuration and instructs backup nodes to undergo whatever steps are necessary to restore availability to the system. This typically involves reconstructing a new primary from the backups, or promoting one of the backups to be the primary. Current state-of-the-art recovery for primary-backup replication involves fast, parallel log ingest and replay [28]. While this is feasible for recovering relatively small datasets stored in DRAM, as we mentioned in Section 1, challenges introduced when using NVMM necessitate a different approach to recovery.

Primary-backup replication is appealing because it offers high performance and simplicity, and as a result is used in many distributed storage systems such as RAMCloud and FaRM. RAMCloud uses primary backup with the primary storing data on DRAM and backups storing data on SSD. Leveraging DRAM in this way enables RAMCloud to achieve high throughput and low latency distributed storage. Our system uses a similar node layout to RAMCloud, with our primary storing data on NVMM, and backups storing data on SSDs.

FaRM, on the other hand, uses a homogeneous node layout where primary nodes and backup nodes both store their data in non-volatile DRAM. In FaRM, primary crash recovery involves identifying and consistently executing transactions which were interrupted by the primary crash, then shifting load to one of the backups. This design enables FaRM to seamlessly resume operations following the failure of a primary, but carries with it the extra cost of maintaining backups that use the same expensive high performance storage as the primary.

In contrast to primary backup, distributed consensus protocols such as Paxos [22] and Raft [27] can provide consistency without a centralized configuration coordinator.

The increased density and decreased write throughput of NVMM means traditional offline approaches to crash recovery that read logs from backups and write them to a new primary can no longer provide a low TTR. Meanwhile, maintaining multiple replicas with expensive high performance storage can be prohibitively expensive. In CANDStore, we seek to achieve the best of both worlds – low TTR *and* low cost. To achieve this high availability without incurring extra cost in an NVMM-based distributed key-value store, it is necessary to rethink traditional approaches to primary crash recovery.

In our system, we design a protocol which allows for online recovery of after primary failure, solving the problem of slow offline recovery. We achieve this with lower resource use, and consequently cost, compared to standard primary-backup style approaches. One of the principal challenges in designing this protocol was in meeting SLOs during the live recovery. Our protocol ensures linearizability by leveraging properties of the Raft protocol upon which our protocol is based.

2.3 Workload assumptions

Real world workloads often exhibit a high degree of temporal (and spatial) locality. Recent work shows that even YCSB's zipf distribution, a commonly used distribution when generating workloads to benchmark systems, exhibits less spatial locality than modern datacenter workloads [9]. We design our system to target these skewed workloads to best mirror the characteristics of real datacenter workloads. Section 5.1 describes how we take advantage of workload skew to enable online recovery.

2.4 Data model

The data model we consider in this work is a simple KV interface composed of single-key Gets and Puts. For a detailed description of the KV operations supported, see Figure 5. We believe that our techniques to improve crash recovery can be extended to data models involving more complex KV operations and transactions, but we leave the design of these to future work.

In this paper, we use "keys" to refer to key-value pairs, except in the context of witnesses (Figure 2) and popularity sampling (Section 3.3).

3 Steady state operation

CANDStore is a distributed KV store designed to provide high availability at low cost in the face of primary failure, despite the infeasibility of fast offline recovery. In this section, we introduce CANDStore and describe the steady state operation of a single shard of a sharded datastore. The system builds knowledge about workload skew during steady state operation, as uses this to speed up the recovery process, detailed in Section 3.2.

We begin in Section 3.1 by describing how Gets and Puts are handled and replicated in the context of a simplified three-node cluster consisting of a primary node, a backup node, and a witness node (Figure 2). The witness node behaves similarly to an auxiliary node from Cheap Paxos [23], and also serves a similar purpose to hot swap space in a RAMCloud cluster. It is responsible for enabling consistency without paying the full cost of a new primary or backup node, as well as serving as the new primary in the event the primary fails. By including the witness node in the consensus protocol, rather than allocating it on-demand after primary failure, we can take advantage of its knowledge about which local keys are up-to-date when handling requests during live recovery (Section 5.3). However, to realize this cost-effective heterogeneous node layout while maintaining consistency, we needed to make several modifications to the Raft protocol, which we explain in Sections 3.1 and 4. In Section 3.2 we describe the optimizations we used to improve the performance of the backup node. Finally, we describe how our system identifies distribution skew in client requests (Section 3.3) and how we took advantage of this skew to improve the TTR of the system following primary failure (Section 4).

3.1 Gets and Puts in steady state

We use a heterogeneous node layout in CANDStore to provide cost savings compared to a homogeneous node layout like traditional Raft/Paxos formulations and FaRM [13]. In this section, we outline the modifications we made to the Raft protocol to provide consistency when handling client requests in the absence of failures.

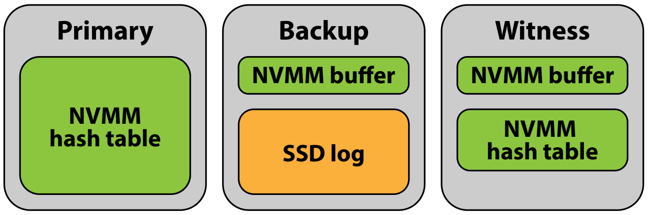


Figure 2: Node types in CANDStore.

Client Get requests, shown in Figure 3, can be handled in 1 round trip by the primary using leases [26]. The client sends Get(key) to the primary, and the primary responds with the value corresponding to key.

Client Put requests, also shown in Figure 3, require the primary to replicate the request to a quorum of nodes in the cluster before committing and responding to the client. Similarly to Raft, this replication is handled by AppendEntries RPCs. However, because the witness nodes behave similarly to Cheap Paxos and do not store values for committed updates, we impose the additional constraint that the quorum for client updates contains all voting backup nodes in the current cluster configuration. In the event of a backup node failure, the system must execute a viewchange to remove the failed node from the cluster configuration and, by extension, the quorum.

Including all backup nodes in the quorum for configuration change creates a situation where it is impossible to remove a backup node from the configuration. We solve this problem by using a mechanism similar to Cheap Paxos. If a backup node crashes and must be removed from the configuration, we allow witnesses to step in and facilitate the configuration change by replicating and committing the new configuration to stable storage.

Upon receiving an AppendEntries request, the backup persists it to a small NVMM buffer, then responds to the primary. These updates are batched and periodically written out to SSD by a dedicated writeback thread. This design has two main advantages: backup nodes can replicate keys at NVMM latency (rather than SSD latency), and batching writes to an SSD improves its throughput (Section 7.6).

A witness node receives only the key and a logical timestamp composed of the Raft term and index, in a similar manner to Cheap Paxos. This creates an issue during leader election which we address in Section 4.2.

Each node commits updates in a different way. The primary node commits updates by writing the key, value and logical timestamp to its local NVMM-backed hash table. The backup commits updates by serializing the key, value, and logical timestamp to SSD, then updating its DRAM index to reflect the location and timestamp of the latest update of the key. The witness commits updates by writing the key and logical timestamp to its underlying hashtable.

3.2 Improving backup performance

As we will discuss in Section 5, the performance of the backup node is the most important factor in quickly restoring the system to in-SLO operation. In this section, we discuss the techniques we use to optimize the operation of the backup node to facilitate faster crash recovery.

Figure 4 describes the internal design of a backup node. To improve its throughput and latency, each backup node uses multiple *workers*, each maintaining exclusive write access to their own NVMM-resident buffers and SSD-resident logs. Each worker maintains two sets of logically-separate NVMM buffers and SSD log files, one for popular (i.e., "hot") keys and one for unpopular (i.e., "cold") keys. We discuss the purpose of separating hot and cold keys in Section 3.3. Each worker has three threads, one to handle RPCs (not explicitly pictured in Figure 4), one to write back batches of updates from the NVMM buffer to the SSD log, and one to garbage-collect (GC) outdated log entries.

Upon receiving a key update from the primary, the RPC handler thread checks whether the key is in the hot or cold set, then writes it to the appropriate NVMM buffer before sending a success response to the primary.

The writeback thread sequentially iterates through all of the updates written to the NVMM buffer, waiting for each entry to be committed. Once enough entries on the NVMM buffer have been committed to fill a batch, the batch is then serialized out to SSD. Batching in this way enables the SSD to achieve maximum write throughput. The writeback thread also updates the shared DRAM index after writing a batch to the SSD log, discards the NVMM-resident copy of the batch, and updates the DRAM index to reflect the latest logical timestamp for each batch entry. The DRAM index is only updated if the entry has an equal or greater logical timestamp to the timestamp stored in the index.

The garbage-collection thread periodically reads a batch of entries from the tail of the log, discarding stale entries and writing non-stale entries back onto the corresponding "hot" or "cold" NVM buffer. The staleness determination is handled by reading the latest logical timestamp from the DRAM index, and the determination of which buffer the entry should be written to is handled in the same way as the RPC handler thread. This garbage collection process serves two main purposes: First, it reduces the log's footprint on SSD, and second, it enables keys to move between the hot and cold sections of the log even if they are never updated by the client.

3.3 Proactively identifying popular keys

To speed up the recovery process (detailed in section 4), we separate the backup logs into two separate contiguous regions on SSD: one region for the popular or hot keys, and one region for less popular or cold keys. The placement of a key into one

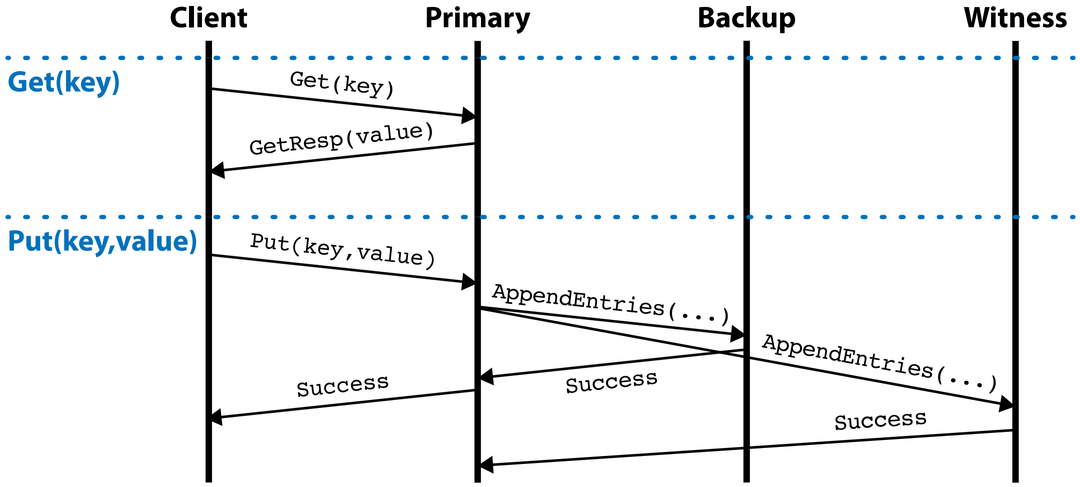


Figure 3: Steady state RPC behavior.

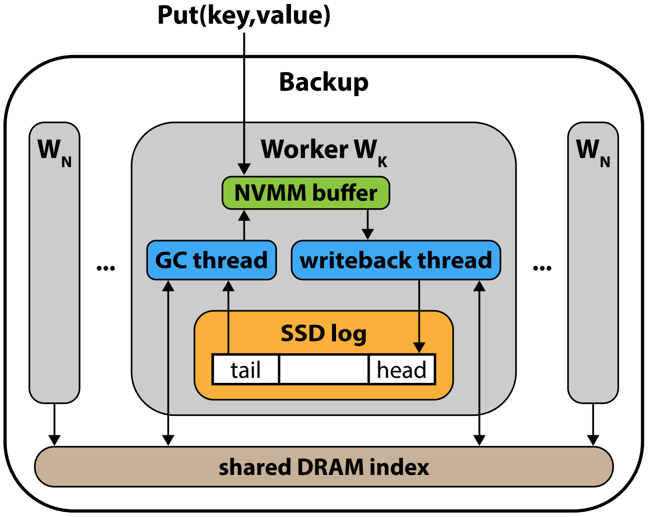


Figure 4: Layout of a single backup worker.

of these regions is decided by the writeback threads, and is based on a continuous sampling of the client requests by the primary.

The sampling mechanism we use is similar to that applied by cache admission policies [8]. Upon receiving a client request, the primary, with some low probability (i.e., 0.001), keeps track of the key for that particular request. Periodically, the list of keys that were tracked by the primary is sent to the replica, which maintains a fixed-size list of the most popular keys (using an LRU eviction policy). This list of popular keys is then used by the write-back threads to make a placement decision when updates are committed to the on-SSD log.

CANDStore's garbage collection mechanism, as described in the preceeding subsection, ensures that "cold" keys which were, by chance, placed into the "hot" log will eventually be written to the "cold" log, and vice versa.

The simplicity of the garbage collection mechanism does result in some degree of write amplification, potentially affecting the lifespan of the SSDs. Smarter policies for determining when garbage collection should be executed would reduce this problem: for example, only garbage-collecting batches with a large percentage of invalid entries, or reducing the frequency of garbage collection during periods when the hot/cold sets remain stable.

3.4 Steady state overhead

During steady state operation, the majority of protocol-related messages are piggybacked on client requests. Heartbeats are implicitly piggybacked on AppendEntries RPCs when replicating client Puts, and in the absence of client Puts, the primary sends heartbeats to every other node in the cluster at a rate that does not burden the network.

The CANDStore protocol does incur overhead in the key popularity propagation, where the primary periodically notifies the backups about frequently read keys. However, this is not a significant overhead, as we expect to send these updates infrequently, perhaps once every few seconds.

The largest overhead incurred by the CANDStore protocol are the additional messages that must be sent to the witness nodes when replicating client Puts, with N additional messages required for each Put where N is the number of witnesses. In cases where the value is comparable in size to the key, this overhead can be up to 33% increase in communication overhead for Put-only workloads. In cases where the value is larger than the key, this overhead is lower since CANDStore does not transmit values to the witness during steady state operation.

Figure 5: CANDStore RPCs

RPC type	Sender	Recipient	Contents	Response contents
AppendEntries	primary	backup, witness	term, leaderid, prevlogindex, prevlogterm, entries	term, success
RequestVote	primary, witness	backup, witness	term, candidateId, lastLogIndex, lastLogTerm	term, lastLogIdx voteGranted
Get	client	primary	key	value
Put	client	primary	key, value	success
BatchPull	witness, primary	backup	lastLogTerm	large batch of entries
PriorityPull	primary	backup	key	4k batch of entries
RequestLog	witness	backup	term, index	entry

4 Handling failures

In the event of primary failure, there are four differences between our protocol and the standard Raft protocol. In this section, we explain these differences and prove that our protocol maintains both consistency and liveness. A timeline of our recovery protocol is shown in Figure 6.

4.1 Leader election

The leader election process in CANDStore is similar to that of Raft, with the additional constraint that only witness and primary nodes can send RequestVote RPCs. This means that Backup nodes cannot become the new leader of the cluster — this design decision ensures that the new primary (i.e., recovery primary) is not coincident with a backup node. The benefit of this is that the backup, which is the most performance-sensitive node in the recovery process, does not experience any contention for resources.

Because the witnesses and primary node constitute at least $F + 1$ nodes in the cluster, our failure assumption guarantees that there is always at least one live witness or primary. Similar to Cheap Paxos, it is possible for the system to lose liveness if the set of non-faulty nodes shifts too quickly. However, unlike Cheap Paxos, which only allows non-witnesses to become the leader, our system only allows voting witnesses to become the leader. We explain in Section 5 how a witness transitions from not having complete information about committed KV updates into becoming a fully functioning primary.

Additionally, it is possible for all voting witnesses to be unaware of a committable or committed update, as the quorum for KV updates need only contain the primary and all backups in the current view.

To fix this problem, we modify the leader election protocol in the following ways. First, we include the most recent log index in the RequestVote response. If a witness receives a RequestVote response from one of the replicas with `voteGranted = 0`, then it saves its local `lastLogIndex` as `lastLogIndexOld` and updates its `lastLogIndex` to the index received in the RequestVote response. This guarantees that leader election will eventually terminate and choose a new leader.

Upon election, the new leader first compares its `lastLogIndex` with `lastLogIndexOld`. If they are equal, then leader election is complete. However, if they are not equal, then the new leader enters the Reconciliation phase, described in the next section.

4.2 Log reconciliation

The purpose of the log reconciliation phase is to allow the new primary to fast forward itself to a sufficiently up-to-date state such that it is possible to handle client Puts. We call the primary during this phase the "provisional leader" of the Raft cluster. During the reconciliation phase, the primary node cannot commit any new log entries. However, it is still able to serve client Get requests using the mechanisms outlined in Section 5.3.

The provisional leader must first bring itself up to date by querying the backup(s) for the log entries from `lastLogIndexOld` to `lastLogIndex`. It achieves this by sending a RequestLog RPC to the backup(s) for each log that it suspects it is missing. Because of the Cheap Paxos–style quorum, it is acceptable to issue these requests in parallel to different backups.

Log reconciliation ends when one of two conditions is met: First, if the primary receives a response for each log entry that it is missing, then it can temporarily treat these logs as uncommitted and resume accepting client Put requests, replicating and committing these log entries based on the normal operation of the Raft protocol. Second, in the event that one of the backups returns a null entry for index i, then it is acceptable to discard responses for log entries corresponding to index i and above. The modified quorum ensures that any log entries that are not present on all voting backups cannot have been committed.

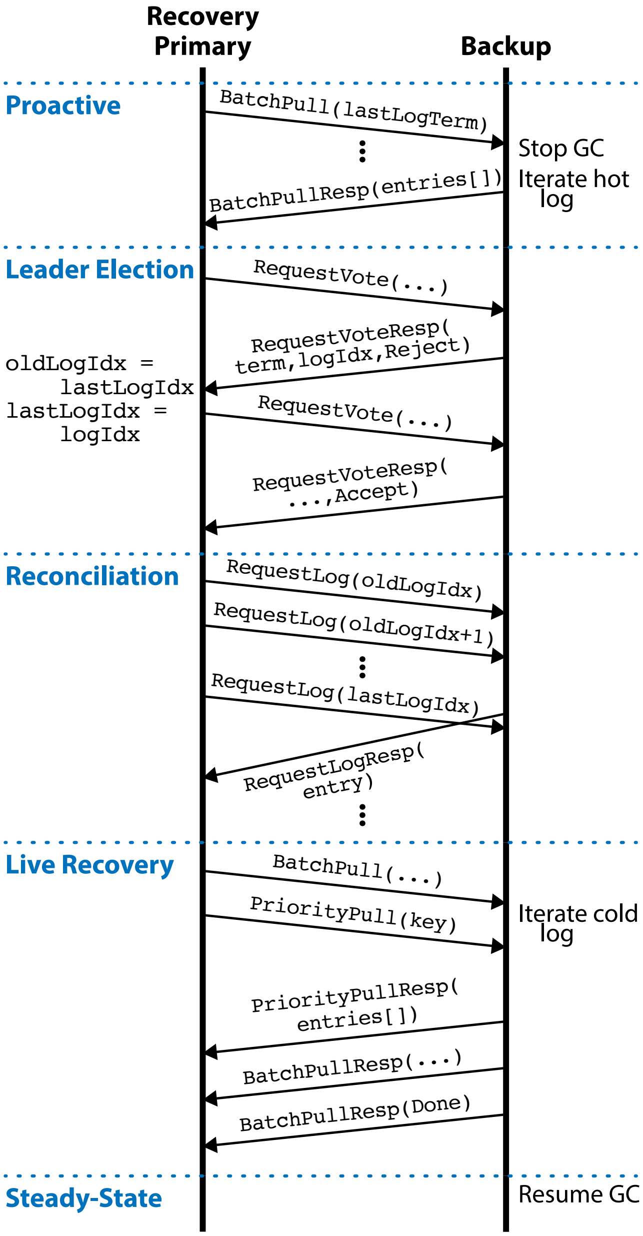


Figure 6: Recovery timeline.

Proof of modified quorum correctness Suppose that there exists a voting backup B which does not have knowledge of a committed update U. This backup must have been admitted to the cluster as a non-voting node by some log entry L_0 and admitted to the cluster as a voting node by some log entry L_1. By the Raft log prefix invariant and the CANDStore modified quorum, U must have been committed after L_1. This is because after L_0 is committed, all updates must be replicated to B before being committed, as part of Raft's joint consensus state, so if B is a voting node, then L_1 must have been committed, which implies that B has knowledge of L_1. However, due to CANDStore's modified quorum, any updates which are committed after L_1 must be replicated to a quorum including all backups in the current view. Therefore, U must have been replicated to B before being committed.

5 Primary failure

In this section, we describe how our system handles the failure of a primary. We discuss the recovery protocol, and describe how the system serves client requests during the period of degraded performance while the system is repairing. We briefly discuss how the system handles witness and backup failures in Section 6.

5.1 Proactive recovery

In CANDStore, we begin the recovery process with what we call "proactive" recovery. This phase is called "proactive" because we begin the process before primary failure is detected, whenever we suspect that there may be a problem with the primary. This phase of recovery involves preemptively copying popular KVs to a witness, in anticipation of that witness becoming the new primary.

In addition to standard heartbeats ("hard" timeouts), we introduce a more aggressive (i.e., earlier) "soft" timeout. This soft timeout triggers the first phase of CANDStore's recovery protocol, which we call proactive recovery. Soft timeouts act as a failure detector with a moderate false positive rate; CANDStore aborts the proactive recovery if it receives a heartbeat before the hard timeout elapses.

During proactive recovery, the witness node sends BatchPull RPCs to the backup. Upon receiving a BatchPull request, the backup pauses its garbage collection threads, begins processing its "hot" log and sends the contents to the witness one batch at a time. If at any point during this process the witness receives a heartbeat from the primary, then the witness sends a BatchPull RPC to the backup with the invalid `lastLogTerm` value of 0, which causes the replica to stop iterating through its log and resume garbage collection. Additionally, if the witness sending BatchPull RPCs is removed from the configuration or becomes part of C_{old} in Raft's joint configuration state, the replica stops iterating its log, resumes garbage collection, and rejects BatchPull requests until the witness is restored to voting status and the configuration of the system returns to a non-joint configuration. Informally, if the cluster appears to be reconfiguring to exclude the witness, then the backup ignores RPCs from that witness and returns to steady state operation.

5.2 Setting timeouts

Setting the soft timeout to be too short can reduce steady state performance, as false positives incur network, disk, and compute costs. Setting the soft timeout and hard timeout too close together effectively eliminates the proactive recovery stage entirely, thus reducing the efficacy of CANDStore's recovery protocol. The longer the interval between soft and hard timeout, the longer the proactive recovery stage is able to cache hot keys on the recovery primary.

The optimal timeouts depend on the network topology, workload, and failure detector used in the actual deployment. One principled way to set the timeouts is by using the Φ accrual failure detector [18] and choosing values of Φ for the soft and hard timeouts such that they fit the desired detection interval and false-positive rate, illustrated in Figures 8 and 9 of Hayashibara et al. [18].

5.3 Live recovery

After the hard timeout elapses, the system begins leader election, as described in Section 4.1. Immediately following the election of a new primary, which we refer to as the recovery primary, we enter the live recovery portion of CANDStore's recovery protocol.

During live recovery, the recovery primary continues to send BatchPull RPCs to the backup node, which, after completing its iteration of the hot log, begins iterating through the cold log and sending cold keys back to the recovery primary. At the same time, the recovery primary begins handling client requests. One key difference between steady state operation and live recovery is that tasks such as popularity sampling and garbage collection are temporarily paused to reduce resource use (especially SSD bandwidth) on the backup.

Client Put requests are still handled in the same way as during normal operation of the system. The recovery primary may not be aware of the values for all of the preceding updates, but has still "committed" those updates and is able to commit new client Put requests.

Client Get requests are handled in one of two ways: If the key is present locally on the recovery primary, then it can be served directly, identically to the non-failure case. However, if the key is not present locally, then the recovery primary issues a point query for that key to the backup (Figure 7). We call these point queries PriorityPulls, a name taken from a similar mechanism in Rocksteady [21]. The backup uses its DRAM index to determine the SSD page(s) containing the most up-to-date value corresponding to the requested key, and sends the value, along with any other keys stored on the same page, to the primary. In our implementation, this was done to take advantage of bytes read from the SSD regardless of whether we wanted or not, but for certain workloads with spatial locality (e.g., clicking one button to add a set of 5 items to the cart), this design could take advantage of that spatial locality. Recent research from Facebook indicates that this sort of spatial locality may be quite common [9].

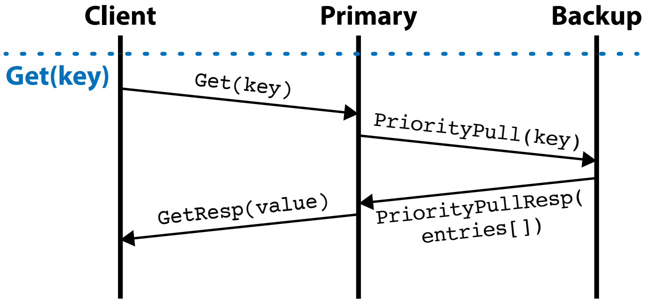


Figure 7: Remote Get.

Once the backup has determined that it has sent all of its local records to the recovery primary, it notifies the recovery primary via a BatchPull response with 0 entries. Similarly to tracking the progress of migrating tuples in Squall [14], the backup determines when all local records have been successfully transferred to the recovery primary. The recovery primary can then transition into steady state operation as a normal primary.

5.4 Recovery scaling

When scaling CANDStore, it is important to take into account that the primary bottleneck during recovery is the load-sensitive SSD. As such, increasing the replication factor or sharding backups across multiple physical servers/SSDs in a manner similar to RAMCloud creates opportunities for SSD load balancing. The fewer PriorityPulls issued against a given SSD, the better the tail latency for reads from that SSD, which will result in better recovery performance as the cluster gets larger.

However, a larger cluster can result in more communication overhead during critical operations, such as leader election or log reconciliation. CANDStore's proactive recovery masks these delays, enabling the system to continue to execute bulk-recovery despite the absence of a leader.

CANDStore does incur overheads from its use of horizontal Paxos rather than vertical Paxos, particularly during configuration change and leader election. Compared to primary-backup systems that rely on a global configuration master, and K nodes in the view, CANDStore requires K messages compared to 1 message to complete leader election (in the best case for both), and $2K$ messages compared to K messages to complete configuration change.

6 Other failure modes

Section 4 discussed what happens in the case of primary failure in a CANDStore shard, which is the most complex type of failure. In this section, we discuss the behavior of the system in the event of backup or witness failure. We discuss this in the context of N backup nodes and N witness nodes, where N is tuned by the operator based on the degree of replication desired and the rate of failure of the servers in the cluster. We believe $N = 2$ (triplication) is a reasonable configuration.

6.1 Backup failure

In the event of backup failure, it is necessary for the system to undergo a configuration change to exclude this backup from the cluster and, by extension, the write quorum. However, in-flight updates that have been replicated to some subset of the backup nodes effectively "block" this configuration change from being committed.

We solve this by temporarily changing the write quorum to exclude the suspected "failed" node, and to include a single witness node in its place. This idea is taken from Cheap Paxos [23], where a witness node is able to help commit log updates and execute a view change in the event of a main processor failure.

It may be desirable to add an additional backup to the cluster to restore the desired level of fault tolerance. This can be done easily and without interruption of availability, despite the constraint of the write quorum including all voting backup nodes. The node can be added to the cluster using first as a non-voting node, and only after it is up to date will it be promoted to a voting node and join the write quorum.

6.2 Witness failure

If a witness fails, it is not necessary for the system to remove the failed witness from the cluster before making additional progress. To restore the desired level of failure tolerance, it may be desirable to allocate and add a new witness node to the cluster.

7 Evaluation

This section describes how we evaluated CANDStore's crash recovery mechanism compared to the baseline approach of copying the entire set of keys/values from the replica to the recovery primary. We show that our approach achieves higher availability than traditional primary-backup style approaches while maintaining cost efficiency and high performance.

Due to constraints on available hardware, we did not shard the recovery primary or backups. However, the techniques we use and the resulting relative improvements in availability are still applicable, and our system design benefits from the parallelism afforded by optimizations such as sharded backups and multiple/striped SSDs. In particular, our recovery protocol benefits greatly from spreading load across multiple SSDs, as the main cause of SLO violations during online recovery is high SSD load.

7.1 Testbed setup

Our evaluation testbed is a 3-node cluster. Each machine has two Intel Cascade Lake processors with 24 physical cores clocked at 2.2 GHz with hyper-threading enabled. The platform has 192 GB of DDR4-2666 DRAM and 768 GB of NVMM (Intel Optane DC 2666 Mhz QS [20]) per socket for a total of 384 GB of DRAM (12 × 32 GB) and 1.5 TB of NVMM (12 × 128 GB) per server. One 375 GB Intel P4800X NVMe SSD [2] was installed on each server. Each node also had a 1-port Mellanox ConnectX-3 NIC [25], and the servers were interconnected by a 56 Gb/s InfiniBand switch.

For all experiments, we set up a single region consisting of the 768 GB of NVMM located on NUMA node 0 in AppDirect mode. We format the region into an ext4 filesystem and use libpmem [3] to facilitate reading and persistently writing to NVMM.

We were unable to test a configuration with multiple backups or multiple SSDs to exploit parallelism during the recovery process, as we did not have access to more than 3 servers and 1 SSD per server. As a result, we compare our performance against a baseline of how offline recovery would perform on our testbed configuration. Our performance would be at least proportionally improved by the use of cluster-level parallelism or multiple SSDs on the backup. In particular, tail latency during the live recovery phase would be improved by having multiple SSDs per backup node, thus reducing the per-SSD load and driving down tail latency.

7.2 Client design

In our evaluation, we used a closed-loop client with largely independent threads. Each thread sends one or more streams of requests to each server thread, and the next request in the stream is not sent until a response is received for the preceding request.

7.3 Workloads

We evaluate our system using YCSB [10] workloads B (95% GET, 5% UPDATE) and C (100% GET) using YCSB's zipfian request distribution. In addition to using YCSB's default parameterization of the zipf distribution ($\theta = 0.99$), we also explore less-skewed and more-skewed workloads ($\theta \in \{0.9, 1.1\}$). We omit workload A (update heavy) due to issues outlined in the next section.

Due to resource constraints, we benchmark the system with a 128 GB shard. We believe that shards of real distributed storage systems using NVMM will be larger, but the relative benefits of our recovery approach will still apply.

7.4 Steady state performance

As the focus of this project was on improving performance during recovery, we did not do a comprehensive evaluation of the steady state performance of our system. For YCSB workload C with the default zipfian distribution and 512 B key-value pairs (16 B key, 496 B value), and a latency-sensitive configuration of 8 server threads handling requests from 8 client threads with each client thread sending a single request at a time, our system can serve at a throughput of 1.5 GB/s while maintaining a median latency of 7 μs and a 99%ile tail latency of 51 μs. For the equivalent setup running on YCSB workload B, our system achieves 1.0 GB/s with a median latency of 7 μs and a 99%ile tail latency of about 73 μs.

When run in a throughput-oriented configuration of 8 client threads requesting a window of 64 concurrent requests to 8 server threads, we can achieve approximately 4.5 GB/s in workload C and 3.9 GB/s in workload B.

The decreased performance on workload B is a result of our system being poorly optimized for Puts. During live recovery, the behavior of Puts does not change, but Gets can incur in an additional RTT for a PriorityPull. As a result, we focused mainly on optimizing Gets, but we believe that with additional optimizations and larger NVMe SSDs (large enough to comfortably pre-allocate spare log space without filling up the SSD) the performance of Puts in our system will increase.

Our goal is to have engineered enough of the system such that our evaluation of the recovery protocol is informative; as such, this performance is reasonable — it achieves 65% the throughput of its InfiniBand link, and has a median latency of 7 μs which is comparable to the 4.7μs read latency that RAMCloud achieved in its original evaluation environment [30].

7.5 Metrics

We compare the time to repair (TTR) of our approach with traditional log-replay style crash recovery protocols. We measure the time elapsed between primary failure and when the system achieves and maintains a particular latency SLO. For each workload type and distribution, we measure the TTR for a range of SLOs.

We also measure the penalty our approach incurs with regard to the amount of time it takes to fully copy the entire key space from the replica to the primary. This is the primary tradeoff of using CANDStore, as it is necessary to slow down the rate of BatchPulls to avoid increasing the latency of PriorityPulls, i.e. we trade increased total recovery time for the ability to do *online* recovery.

In all of our experiments, tail/median latency is measured on a 100 ms non-overlapping sliding window, and TTR is calculated to be the end of the first window that achieves latency within the SLO such that all following windows do not have latency higher than the SLO.

Block size (KB)	**Read tput (KB/s)**	**Write tput (KB/s)**
0.5	132,647	16,703
1	286,196	38,715
2	614,236	110,490
4	1,668,613	1,627,756
8	2,625,276	2,148,087
16	2,647,880	2,204,804
32	2,651,732	2,212,097
64	2,652,257	2,212,228
128	2,652,623	2,211,742

Figure 8: Block size vs SSD throughput.

7.6 SSD block size effects throughput

SSDs can achieve higher throughput when the block size for IO requests is sufficiently large. We verify this is the case for our SSD, and determined the optimal block size for reading and writing, shown in Figure 8. We find that for our disk, a block size of 32 KB maximizes throughput for both sequential reads and sequential writes. The throughput at a block size of 32 k matches the advertised throughput of this drive [2]. We use this optimal block size to determine the batch size used for reading and writing to the log (to maximize throughput).

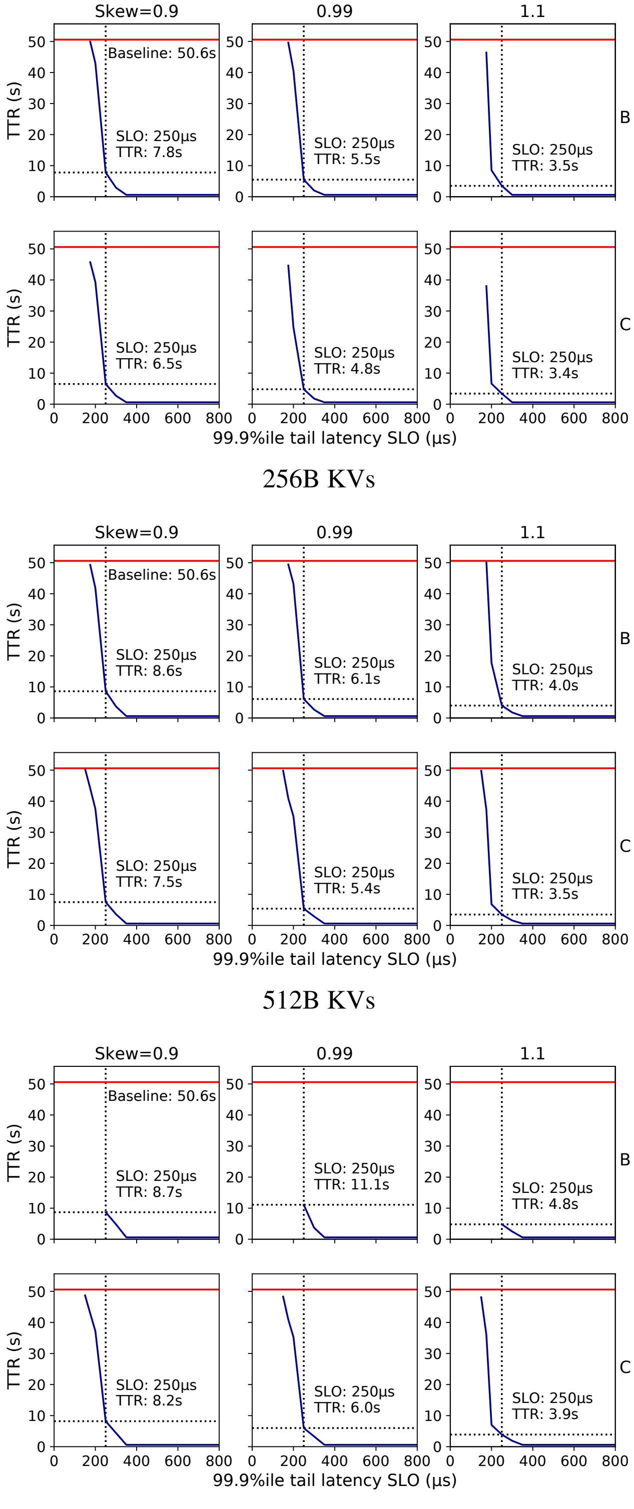
7.7 Tail latency

We evaluate the availability of our system in terms of median and 99.9%ile tail latency of client requests. These experiments were run in a latency-sensitive configuration of 8 client threads, each sending one stream of requests to one of 8 server threads. The backup uses 2 threads to ingest the log.

To provide a fair point of comparison against an equivalent offline recovery approach, we calculated the minimum time necessary to read the entire shard (128 GB) from SSD at the maximum read throughput of the SSD (as measured in Section 7.6). For our drive and a 128 GB shard, the baseline for offline recovery is 50.6 seconds.

For our tail latency–sensitive configuration, it takes 201 seconds (4x longer) to completely copy all data from the backup to the new primary due to rate-limited BatchPulls.

In Figures 9 and 10, the TTR for each SLO shown is the median of 3 trials. Figure 9 shows the TTR of our system compared with the baseline offline recovery approach. This figure is organized into 3 blocks of 6 graphs each, each corresponding to a different key-value pair size indicated by the label below the block. Within each block, there are 2 rows and 3 columns; each row corresponds to a different YCSB workload, indicated by the label to the right of the row, and each column corresponds to a different request distribution skew, indicated by the label at the top of each column. For



256B KVs

512B KVs

1KB KVs

Figure 9: TTR for 99.9% tail latency SLO (lower is better).

example, the top left graph of each block corresponds to an experiment run with YCSB workload B with a skew of $\theta = 0.9$.

Each graph shows the TTR (y-axis) for a particular latency SLO (x-axis). Lower TTR is better, and larger SLOs have lower TTR. For example, looking at the top left graph, the TTR for a 99.9%ile tail latency SLO of 250 μs is 7.8 s (labeled), and the 99.9%ile tail latency for an SLO of 300 μs is 2.9 s (not labeled).

As the tail latency SLO approaches (left on the x-axis) the steady state tail latency of the system, the TTR of our online recovery approach increases, getting worse. As we mentioned in Section 1, our recovery approach is built with the assumption that there will be some performance leeway in the system, with a gap between the steady state performance of the system and the SLO. As the tail latency SLO decreases, the degree of performance leeway decreases, and at a certain point it is better to do offline recovery. However, tail latency–sensitive storage systems will often have resources overprovisioned [12], creating the headroom necessary for our approach to be effective.

For the standard YCSB zipfian distribution skew of $\theta = 0.99$, our results show that with a 99.9%ile tail latency SLO of 250 μs, CANDStore achieves a TTR of 4.8–11.1 s, which is 4.5–10.5x lower than the baseline offline approach.

7.8 Median latency

Figure 10 shows how long it takes our system to restore a certain level of median latency. It takes only 0.6–2.4 seconds to restore the system to 5 μs median latency (steady state latency) for workloads with a skew of 0.99.

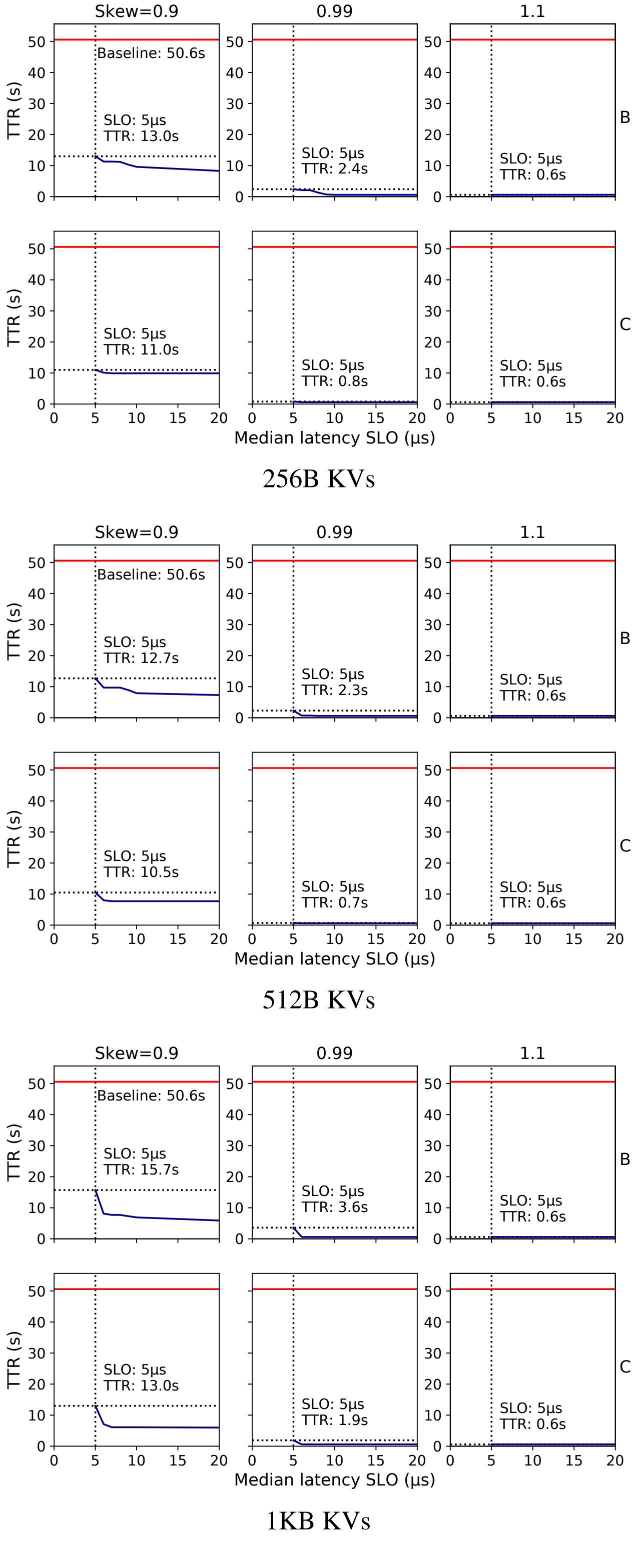
8 Related work

We briefly explore related work on high performance storage systems.

KVell [24] describes the design of a fast, persistent KV store leveraging NVMe SSDs. The simple design of an unstructured SSD-resident log paired with an in-memory index is similar to the design of our backup node. KVell shows that it is possible to get sub-millisecond latency for random reads on modern NVMe SSDs, but that too many simultaneous requests can result in much higher latency. We leverage this in conjunction with workload skew to enable PriorityPulls to be served at low latency.

Unlike CANDStore, KVell handles crash recovery only in the context of recovering local state from persistent storage after a crash, rather than failing over to a replica.

Rocksteady and Squall (H-store) Rocksteady [21] and Squall [14] are both systems that enable reconfiguration of in-memory datastores. Both focus on reconfiguring in a way that minimizes the performance impact during the reconfiguration process.



256B KVs

512B KVs

1KB KVs

Figure 10: TTR for median tail latency SLO (lower is better).

Both Squall and Rocksteady solve some similar problems to CANDStore, but in the context of database reconfiguration, copying tuples from an in-memory primary partition to another in-memory primary partition using a combination of large asynchronous batched copying and fine-granularity on-demand fetching. In database reconfiguration, there is flexibility in the rate at which the tuples are migrated — this contrasts with primary crash recovery where restoring in-SLO performance is time-critical. Unlike main memory, increasing queue depth for I/O operations to SSD quickly increases latency. In CANDStore, we solve the problem of primary crash recovery, with source tuples stored on SSD, meaning more pressure to copy tuples quickly, and less performance headroom to do so without affecting latency. In CANDStore, this necessitated design choices such as hot/cold tiering of the backup, and aggressive timeouts for proactive recovery.

RAMCloud As discussed in Section 2.2, our system closely resembles RAMCloud [28], with primary nodes serving KVs using fast main-memory based storage, and backups storing updates on SSD in log format. However, in our system, to maintain high performance and availability when using NVMM instead of DRAM we use a different crash recovery protocol, described in Section 4.

FaRM [13] is another high performance in-memory datastore, which uses capacitor-backed DRAM to provide persistence. FaRM provides much richer transactional semantics and much higher performance, particularly for writes. However, FaRM maintains multiple copies of the dataset in DRAM, incurring a much larger cost compared to CANDStore, which stores backups on SSD.

FaRM includes a phase of primary crash recovery which involves identifying tuples involved in transactions which were interrupted by the primary failure. We drew inspiration from this design, identifying log updates which were “interrupted” before they could be replicated to the witness node in our modified leader election and log reconciliation phase of recovery (Sections 4.1 & 4.2).

Gemini [17] is a system for fast crash recovery of persistent caches. The focus of Gemini is to enable a recovering primary to immediately serve client reads and writes at high performance without violating consistency. Similar to CANDStore, Gemini prioritizes copying popular cache entries to ensure that a large proportion of requests can be served by a recovering cache server.

Unlike our system, the primary concern when recovering a primary is load balancing, as tuples can be offloaded to other servers in the system immediately following primary failure. Gemini is a cache, so it only provides RAW consistency for single keys, rather than linearizability.

9 Conclusion

This paper introduces a new approach to primary crash recovery for replicated key value stores in NVMM. To overcome the challenges in implementing high performance distributed storage in NVMM rather than DRAM, we present a crash recovery protocol that takes advantage of workload characteristics and modern NVMe SSD technology to enable *online* crash recovery. We show that traditional offline recovery approaches are insufficient to take advantage of new cost-effective NVMM technologies, and how online recovery can be used to reduce time to repair after primary failure by up to 4.5–10.5x. Our system, CANDStore, shows that it is possible to achieve high performance, high availability, consistency, and low cost using NVMM-based distributed storage.

Acknowledgements. We thank Intel for generously providing access to the testbed used in our evaluation. We thank the members and companies of the PDL Consortium (Alibaba, Amazon, Datrium, Facebook, Google, HPE, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Pure, Salesforce, Samsung, Seagate, Two Sigma, and Western Digital) and VMwware for their interest, insights, feedback, and support.

References

[1] [n.d.]. Intel Optane DC persistent DIMM prices listed: $842 for 128gb, $2,668 for 256gb. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[2] [n.d.]. Intel® Optane™ SSD DC P4800X Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-ssd-series/optane-dc-p4800x-series/p4800x-375gb-aic-20nm.html.

[3] [n.d.]. PMDK - libpmem. https://pmem.io/pmdk/libpmem/.

[4] 2015. 8Gb: x4, x8, x16 DDR4 SDRAM Features. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf.

[5] 2017. TN-40-07: Calculating Memory Power for DDR4 SDRAM Introduction. https://media-www.micron.com/-/media/client/global/documents/products/power-calculator/ddr4_power_calc.xlsm?la=en&rev=5e97be39078d4a1b8619cb85c96bbe63.

[6] 2020. Intel® Optane™ Persistent Memory 200 Series. https://ark.intel.com/content/www/us/en/ark/products/203880/intel-optane-persistent-memory-200-series-512gb-pmem-module.html.

[7] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. 2019. Assise: Performance and Availability via NVM Colocation in a Distributed File System. (2019). arXiv:1910.05106 [cs.DC]

[8] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. 2017. Adaptsize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 483–498.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *FAST*.

[10] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*. Indianapolis, IN.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *Proc. 10th USENIX OSDI* (Hollywood, CA). USENIX.

[12] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA.

[14] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 299–313. https://doi.org/10.1145/2723372.2723726

[15] Facebook. 2015. RocksDB. http://rocksdb.org/.

[16] Daniel Ford, Francois Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie

Grimes, and Sean Quinlan. 2010. Availability in Globally Distributed Storage Systems. In *Proc. 9th USENIX OSDI*. Vancouver, Canada.

[17] Shahram Ghandeharizadeh and Haoyu Huang. 2018. Gemini: A Distributed Crash Recovery Protocol for Persistent Caches. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. Association for Computing Machinery, New York, NY, USA, 134–145. https://doi.org/10.1145/3274808.3274819

[18] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. 2004. The *Phi* accrual failure detector.

[19] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX Annual Technical Conference*. Boston, MA.

[20] Intel Optane DC Persistent Memory [n.d.]. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[21] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2017. Rocksteady: Fast Migration for Low-Latency In-Memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 390–405. https://doi.org/10.1145/3132747.3132784

[22] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[23] Leslie Lamport and Mike Massa. 2004. Cheap Paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN '04)*. IEEE Computer Society, Washington, DC, USA, 307–.

[24] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 447–461. https://doi.org/10.1145/3341301.3359628

[25] Mellanox ConnectX-3 Product Brief [n.d.]. http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN_Card.pdf.

[26] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proc. 5th ACM Symposium on Cloud Computing (SOCC)*. Seattle, WA.

[27] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proc. USENIX Annual Technical Conference*. Philadelphia, PA.

[28] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal.

[29] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2010. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. In *Operating Systems Review*, Vol. 43. 92–105.

[30] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM TOCS* (2015).

[31] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM SIGMOD*. Chicago, IL.

[32] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101