Sift: Achieving Resource-Efficient Consensus with RDMA

by

Mikhail Kazhamiaka

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Masters of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

© Mikhail Kazhamiaka 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Sift is a new consensus protocol for replicating state machines. It disaggregates CPU and memory consumption by creating a novel system architecture enabled by one-sided RDMA operations. We show that this system architecture allows us to develop a consensus protocol which centralizes the replication logic. The result is a simplified protocol design with less complex interactions between the participants of the consensus group compared to traditional protocols. The dissaggregated design also enables Sift to reduce deployment costs by sharing backup computational nodes across consensus groups deployed within the same cloud environment. The required storage resources can be further reduced by integrating erasure codes without making significant changes to our protocol. Evaluation results show that in a cloud environment with 100 groups where each group can support up to 2 simultaneous failures, Sift can reduce the cost by 56% compared to an RDMA-based Raft deployment.

Acknowledgements

I would like to thank Professors Bernard Wong and Khuzaima Daudjee for their support and guidance throughout my graduate research and in the creation of this thesis. I would also like to thank Babar Memon, Chathura Kankanamge, Siddhartha Sahu, Sajjad Rizvi, and Dora Hu for their help throughout this project.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

2 Background and Related Work 5

2.1 State Machine Replication . . . 5

2.2 RDMA . . . 6

2.3 Resource Disaggregation . . . 6

2.4 Erasure Codes . . . 7

2.5 Consensus as a Service . . . 8

3 Replicated Memory 9

3.1 Architecture . . . 9

3.2 Coordinator Election . . . 10

3.3 Normal Operation . . . 14

3.3.1 Read Requests . . . 14

3.3.2 Write Requests . . . 14

3.4 Fault Recovery . . . 16

3.4.1 Coordinator Failure . . . 16

3.4.2 Memory Node Failures . . . 17

3.4.3 Persistence . . . 17

4 Key-value Store 19
4.1 Architecture . . . 19
4.2 Put/Get Requests . . . 20
4.3 Failure Handling . . . 20
4.4 Optimizations . . . 21
4.4.1 LRU Cache . . . 21
4.4.2 Batching . . . 21

5 Resource Reduction 22
5.1 Erasure Codes . . . 22
5.2 Shared Backup Nodes . . . 24

6 Evaluation 25
6.1 Implementation . . . 25
6.2 Experimental Setup . . . 27
6.3 Key-Value Store . . . 27
6.3.1 Other Systems . . . 27
6.3.2 Throughput . . . 28
6.3.3 Latency . . . 30
6.4 Cost Analysis . . . 30
6.4.1 Normalized Performance . . . 30
6.4.2 Shared Backup Nodes . . . 33
6.4.3 Costs . . . 33
6.5 Failure Recovery . . . 37
6.6 Zookeeper . . . 40

7 Conclusion and Future Work 43

References 45

List of Tables

6.1 Machine configurations for each system normalized for performance. 34

List of Figures

3.1 Sift architecture. 11

3.2 Example of a coordinator election scenario. Two backup CPU nodes compete to become the new coordinator once their *election timeout* expires. The CPU node which wins the election begins performing heartbeat writes while the other detects the new coordinator and reverts back to the `follower` state. 13

3.3 Messages exchanged for read and write requests. 15

5.1 Replication process with erasure codes. The original data is divided into $F_m + 1$ chunks and F_m redundant chunks are generated. Each memory node receives one of these chunks. 23

6.1 Performance comparison with Sift's key-value store and an RDMA-based implementation of Raft. 26

6.2 Latencies at low load (1 client) and 90% load. 29

6.3 Performance of Sift and Raft-R with a varied number of cores on all nodes (excluding Sift memory nodes, which use no CPU), for F = 1 and 2. These results show us how machines should be provisioned to achieve equivalent performance. 31

6.4 Results of a simulation over a Google cluster trace [43] of machine failures. Estimates how many backup nodes are needed to prevent additional recovery time due to VM provisioning. 32

6.5 Costs of deploying Sift configurations relative to the cost of Raft-R. Machines provisioned for F=1. 35

6.6 Costs of deploying Sift configurations relative to the cost of Raft-R. Machines provisioned for F=2. 36

6.7 Read-heavy workload throughput during a memory node failure. 38
6.8 Read-heavy workload throughput during a coordinator failure. 39
6.9 Throughput vs. latency curves of ZooKeeper and SiftZK, which uses Sift for replication. 41

Chapter 1

Introduction

State Machine Replication (SMR) enables the construction of reliable services by ordering events and ensuring these events are recorded by a majority of participants before they are applied to the state machine. This allows services to remain available and maintain a consistent state in the presence of failures, and provides a mechanism to recover from partial failures. Paxos [22] is the most widely used protocol for implementing SMR. It is efficient and has proven safety and liveness properties. However, like all consensus protocols for asynchronous systems, Paxos must replicate the application state to a large number of nodes, requiring $2F+1$ replicas to support F simultaneous fail-stop errors, and has limited throughput due to this requirement. Providing high throughput requires partitioning the state machine across multiple consensus groups. As a result, the cost of implementing SMR can be prohibitive for applications with large state machines or high throughput requirements, such as databases and distributed storage systems.

The high cost of existing consensus protocols is also in part due to nodes needing to be provisioned with the same compute and memory resources. Non-leader nodes are responsible for applying requests passed by the leader to their local state machine and participating in the election process, where they may be elected as the new leader. However, this results in computing resources on non-leader nodes being underutilized and performing redundant work to apply updates. The effect is that achieving consensus is made more expensive by provisioning nodes with underutilized computing resources. In practice, applications are often configured with low replication factors to satisfy cost constraints.

For applications running in the cloud, one potential approach to reduce the costs of consensus is by having the cloud provider offer a consensus service that can satisfy the requests of multiple applications using a single group. This allows sharing across appli-

cations which can reduce the cost of using the service. However, providing performance isolation between applications can be challenging. Effective performance isolation requires both accurate modeling of different application workloads, and careful scheduling of requests to reduce load imbalance. The failure of a single consensus group can also lead to multiple application failures. Such failures are especially problematic when requests from applications that should fail independently are assigned to the same group. Furthermore, this approach does not reduce the amount of state being replicated which is often the dominant cost in providing a consensus service.

Previous work [31] has explored using erasure codes to reduce storage requirements. Instead of storing a full replica at every node, each node only stores either a portion of the data or parity information that can be used to reconstruct the data. Although erasure codes have previously been used in consensus protocols, they have introduced significant complexity to an already complex system. This makes it more difficult to reason about the correctness of an implementation. More importantly, to reduce complexity, these systems cannot provide the same degree of fault tolerance and require more than $2F + 1$ nodes to tolerate F failures.

In this thesis, we introduce Sift, a resource-disaggregated consensus protocol in which request processing and state storage are logically separated to simplify the protocol and reduce the cost of deployment. To support this disaggregated design, a Sift deployment consists of two types of nodes: (i) stateless CPU nodes that store only soft (non-persistent) state; and (ii) passive memory nodes that store the consensus log and the state machine. One of the CPU nodes is elected as a coordinator, which handles all requests from clients by both logging the requests and updating the state machine on the memory nodes. Similar to other protocols that solve non-Byzantine consensus, Sift requires $2F + 1$ memory nodes to handle F memory node failures. However, only $F + 1$ CPU nodes are required. The required number of active CPU nodes is even lower when compute resources are shared across consensus groups.

Our design borrows ideas from Disk Paxos [14] to separate processing from storage. However, unlike Disk Paxos, our memory nodes store both the consensus log and a representation of the state machine, allowing coordinator changes without any state reconstruction. One of the key enabling technologies for Sift's design is one-sided Remote Direct Memory Access (RDMA). One-sided RDMA enables the coordinator to read and write to predefined memory regions on the memory nodes without involving the CPUs of the memory nodes. As a result, the memory nodes can be completely passive and can be provisioned with minimal CPU resources. By contrast, in other consensus protocols, the followers – nodes that do not respond to client requests – must actively handle log replication requests and monitor the status of the coordinator to detect possible coordinator

failure. By allowing the coordinator to directly interact with the memory of the memory nodes as though it was stored locally, we are able to greatly simplify the design of the coordinator, and our memory nodes need to process only new coordinator connection requests. In addition, by leveraging Sift's centralized replication logic, the system can easily support erasure codes, further reducing its memory footprint without the complexities and reduced fault tolerance that other systems would experience.

In Sift, a coordinator is elected when a candidate successfully acquires an exclusive lock on a majority of the memory nodes. Instead of direct communication between coordinator candidates, each candidate uses an RDMA compare-and-swap operation to write their `server_id` and `term_id` on the memory nodes. As a result of this design, we require only $F+1$ CPU nodes to handle F CPU node failures, since only one non-faulty CPU node is needed to successfully acquire an exclusive lock on a majority of the memory nodes. This coordinator election design is simpler to implement than other election protocols as there is no direct communication between candidates. By using the one-sided RDMA abstraction, the protocol simply resembles acquiring and releasing local locks.

An additional benefit of CPU and memory disaggregation comes from running in a shared environment, such as the cloud, in which CPU and memory nodes are virtual instances. The amount of resources provisioned to each instance can correspond to its required function. Memory nodes can run on instances with minimal CPU resources and CPU nodes can run on instances with minimal memory resources. Disaggregation also enables a single group of CPU nodes to detect failures and replace failed coordinators across multiple consensus groups. The sharing of backup CPU nodes is made practical by our architecture due to the stateless nature of CPU nodes.

This thesis makes three main contributions:

- It introduces an RDMA-based resource-disaggregated consensus protocol and describes the design of a replicated memory system which utilizes it.
- It shows how the proposed replicated memory interface is leveraged to implement erasure coding with limited additional complexity.
- It shows how the cost of deploying SMR in the cloud is reduced by sharing backup coordinators across consensus groups.

The rest of this thesis is organized as follows. Chapter 2 surveys related work, providing background information for, and contrasting, similar systems. Chapter 3 describes Sift's replicated memory system, including replication, leader election, and fault recovery mechanisms. Chapter 4 outlines the key-value store built on top of Sift's replicated memory

abstraction. Chapter 5 describes how erasure codes and shared backup nodes are used in Sift. Chapter 6 evaluates our key-value store against similar systems in terms of performance and cost, and shows the effects of Sift's recovery mechanisms. Chapter 7 concludes the thesis and outlines future directions for this work.

Chapter 2

Background and Related Work

In this chapter, we describe State Machine Replication (SMR) protocols and provide an overview of Remote Direct Memory Access (RDMA). We also discuss past work on resource disaggregation and other systems that propose consensus as a service.

2.1 State Machine Replication

SMR is used to build fault-tolerant services. Fault-tolerance is achieved by ordering events and ensuring that events are recorded on a majority of replicas before they are committed and applied to the state machine. To make progress in the presence of F fail-stop failures in an asynchronous environment, at least $2F + 1$ replicas are required.

Most SMR protocols such as Paxos [22], Raft [32], and DARE [33] are leader-based, where a single leader is elected and is responsible for coordinating the operations of the protocol. In these protocols, the non-leaders (followers) receive event requests from the leader, write the events to their local logs, apply committed events to their local state machine, and monitor the status of the leader. In the event of a leader failure, followers are responsible for detecting the failure and must collectively elect a new leader. Because a follower may later become the leader, each follower must be provisioned with the same resources as the leader node. However, followers perform only a fraction of the work of a leader. As a result, the resources of the followers may be heavily under-utilized during normal operations.

EPaxos [30] and Mencius [28] are leaderless consensus protocols, meaning that every node can service requests from clients. This property is especially useful over wide-area

networks. In these protocols, every node must be provisioned with sufficient memory and compute resources to handle client requests. Unlike in leader-based protocols, the resources are better utilized since every node acts as a leader. However, these protocols introduce additional overhead, such as the need to include dependency information with each update in EPaxos. As such, overall resource costs are not reduced.

2.2 RDMA

Remote Direct Memory Access (RDMA) [11] is an interface that allows a server to communicate with another server without involving the remote machine's kernel. Additionally, RDMA transfers data from one user-space buffer to another, eliminating the need to copy data between network layers. One-sided RDMA can enable remote memory access without any involvement from the remote CPU. The remote access is performed entirely by the remote NIC without any kernel interaction on either server. Sift utilizes RDMA read, write, and compare-and-swap (CAS) operations in its protocol. Additionally, Sift makes use of the reliable variant of RDMA that returns an acknowledgement when an operation has succeeded.

DARE [33] and APUS [42] are RDMA-based consensus protocols. In DARE, the consensus leader directly reads and writes to the logs of its followers. Followers monitor their local log and perform the logged operations on their local state machine. DARE's leader election protocol closely resembles traditional leader election protocols and requires direct communication between election candidates. APUS transparently optimizes the Multi-Paxos [23] approach to resolving consensus by using RDMA primitives over `LD_PRELOAD`. However, unlike Sift, followers in both systems are still active participants and track the leader state using heartbeat messages. Furthermore, in both systems any node can potentially become leader, which requires equal resource provisioning across nodes and ultimately results in under-utilized resources on non-leader nodes.

2.3 Resource Disaggregation

Disk Paxos [14] proposes a consensus protocol where processes communicate by reading and writing to predefined regions on disks in a Storage Area Network environment. A key argument in Disk Paxos is that replacing computers with commodity disks for redundancy offers a cheaper alternative to state machine replication. Sift revisits this idea from a modern perspective, making use of RDMA to achieve the same goals. Additionally, Sift

extends Disk Paxos by proposing a system that stores materialized state on the passive participants of the protocol.

Past work has proposed data center designs in which resources are disaggregated to different nodes across racks in the same data center and connected over a high-speed network [17, 24, 25, 37, 8]. This design allows for fine-grained resource provisioning [41, 1] that is tailored to the resource requirements of a workload. Sift follows a resource disaggregation design where the participants are divided into CPU and memory nodes.

Several systems [44, 5] disaggregate a state machine replication system into an agreement cluster and an execution cluster. However, the execution cluster nodes are active participants in ensuring that requests are executed in the agreed order, and provide no gains in terms of simplification or resource minimization.

Aurora [39] and Snowflake [9] are databases that decouple their compute and storage into separate layers, allowing each layer to scale independently. Aurora, designed for OLTP workloads, aims to overcome network constraints by sending only log entries across the network and performing updates at the storage layer, utilizing only a buffer cache at the database instances. Snowflake offloads storage to a separate service to be able to scale compute instances independently. Unlike traditional shared-nothing data warehouse architectures where increasing storage capacity also increases computing resources, Snowflake is able to provide elastic compute resources which scale with query load. The ability to scale compute and storage separately through the use of RDMA is the key idea behind Sift's architecture.

2.4 Erasure Codes

In general, erasure codes encode a message of length k into a longer message of length n, of which any subset of size k can be used to rebuild the original message. Sift utilizes a variant of Cauchy Reed-Solomon codes [38] which generates blocks of redundant data that can be used to replace missing blocks of the original data when decoding to recover the original data.

RS-Paxos [31] proposes a variant of Paxos that makes use of erasure codes for reducing the amount of state that needs to be transferred across the network and stored at each replica. They show that an intuitive implementation which simply stores a share of the original data on $F+1$ nodes and redundant shares on the remaining F nodes is no longer able to withstand F failures. As a result, a new protocol is proposed with a more complex replication strategy which requires more than $2F+1$ nodes to tolerate F failures.

Sift's architecture and centralized protocol make the intuitive erasure coding approach possible, with only minor modifications to Sift's protocol. By incorporating erasure codes into Sift, we are able to greatly reduce the memory footprint of replication while maintaining the same fault tolerance guarantees with $2F + 1$ nodes.

2.5 Consensus as a Service

Filo [29] proposes a cloud service which utilizes the same consensus group across multiple tenants while providing service level agreements. Filo aims to better utilize the provisioned resources of a consensus group, which are often underutilized by a single tenant. However, this approach relies on the leader node being under-utilized, which results in reduced effectiveness when tenants require higher throughput and cannot be co-located. Filo's current implementation also uses chain replication, which results in higher latency that may not be acceptable for some applications.

In contrast, Sift focuses on improving utilization by limiting the provisioning of resources in a consensus group which are typically idle. Additionally, when Sift is used as a service in a shared environment, redundant compute resources can be shared across multiple consensus groups. Shared backups allow Sift groups to be provisioned with even fewer compute resources.

For cases where consensus systems are used off the critical path and require relatively low performance, Filo and Sift can both provide significant cost savings. This includes cases such as node discovery and leader election, where ZooKeeper [20] is often used. However, in the case of highly consistent storage services like Spanner [7], where many consensus groups are used, Filo will struggle to share consensus groups due to high load tenants. Sift's sharing approach is workload-agnostic; each tenant receives their own consensus group. Cost savings are achieved by sharing backup nodes which are only responsible for detecting leader failures.

Chapter 3

Replicated Memory

Sift's design is implemented as two independent layers: a replicated memory layer and a key-value layer. The replicated memory layer is accessible through logical addresses and the key-value layer, described in detail in Chapter 4, uses this memory layer as though it was its own local memory. In this chapter, we describe the structure of CPU and memory nodes, how requests are processed, and the failure detection and recovery mechanisms.

3.1 Architecture

Sift is a leader-based disaggregated consensus protocol that is deployed on two types of nodes: CPU nodes and memory nodes. CPU nodes are provisioned with minimal memory resources and memory nodes are provisioned with minimal CPU resources. CPU nodes compete to become the coordinator of the Sift group, while memory nodes are passive participants in the consensus protocol, serving only to replicate state and facilitate communication between CPU nodes. This is accomplished through the use of RDMA, where the coordinator is allowed to directly access and modify predefined memory regions of the memory nodes. Memory nodes need to be actively involved only in establishing the initial connections to the CPU nodes. For this reason, memory nodes need minimal CPU regardless of the size or desired performance of the group. Figure 3.1 illustrates our system's architecture.

Without direct communication between CPU nodes, only a single non-faulty CPU node is required to make progress, as agreement is achieved through writes to the memory nodes. Therefore, to handle F_c CPU node failures, we need only $F_c + 1$ CPU nodes. Similarly to

other consensus protocols, state must be replicated to $2F_m + 1$ memory nodes to handle F_m memory node failures. This is because a majority of memory nodes must be present to reach consensus and guarantee consistency. As a result of Sift's stateless CPU nodes, the number of CPU nodes active at any time is flexible. For example, if minimizing recovery time is not a priority, we can further reduce resource requirements by only instantiating backup CPU nodes on demand if multiple consecutive client requests to the coordinator fail. Further optimizations are discussed in Section 5.2.

A coordinator is elected from the $F_c + 1$ CPU nodes and is responsible for serving client requests and replicating the log and state machine. The coordinator maintains only soft state information as shown in Figure 3.1. The `term_id`, `node_id`, and `timestamp` fields are used in the heartbeat and coordinator election mechanisms, which we discuss further in Section 3.2. The `term_id` and `node_id` fields are each 16 bits, while the `timestamp` field is 32 bits.

The internal structure of a memory node consists of two distinct memory regions: the administrative region and the replicated memory region. The administrative region contains a memory block holding `term_id`, `node_id`, and `timestamp` data that is used by CPU nodes to exchange heartbeat reads/writes. The replicated memory region consists of a write-ahead log and the replicated memory. The write-ahead log allows multiple writes to be committed in parallel using a single RDMA operation, while updates are applied to the replicated memory in the background. The log is also used to facilitate coordinator recovery, described in Section 3.4.1. The replicated memory is a contiguous block of memory that clients interact with through logical addresses.

3.2 Coordinator Election

In traditional consensus protocols, the follower nodes rely on heartbeats from the leader to detect server failures and initiate a new leader election. In Sift, the memory nodes are completely passive, and backup CPU nodes do not directly communicate with each other. Instead, the coordinator periodically sends a heartbeat to the administrative memory region within each memory node using an RDMA CAS operation. This heartbeat message contains the coordinator's `server_id`, `term_id`, and a monotonically increasing `timestamp`. Backup CPU nodes in turn read the heartbeats from this memory region. After reading a heartbeat, each follower compares the value of the `timestamp` to the value it obtained previously. In the event that the heartbeat has not been updated, the follower becomes a candidate and triggers an election. The frequency of heartbeat reads is based on an *election timeout* parameter. This timeout directly determines coordinator failure detection

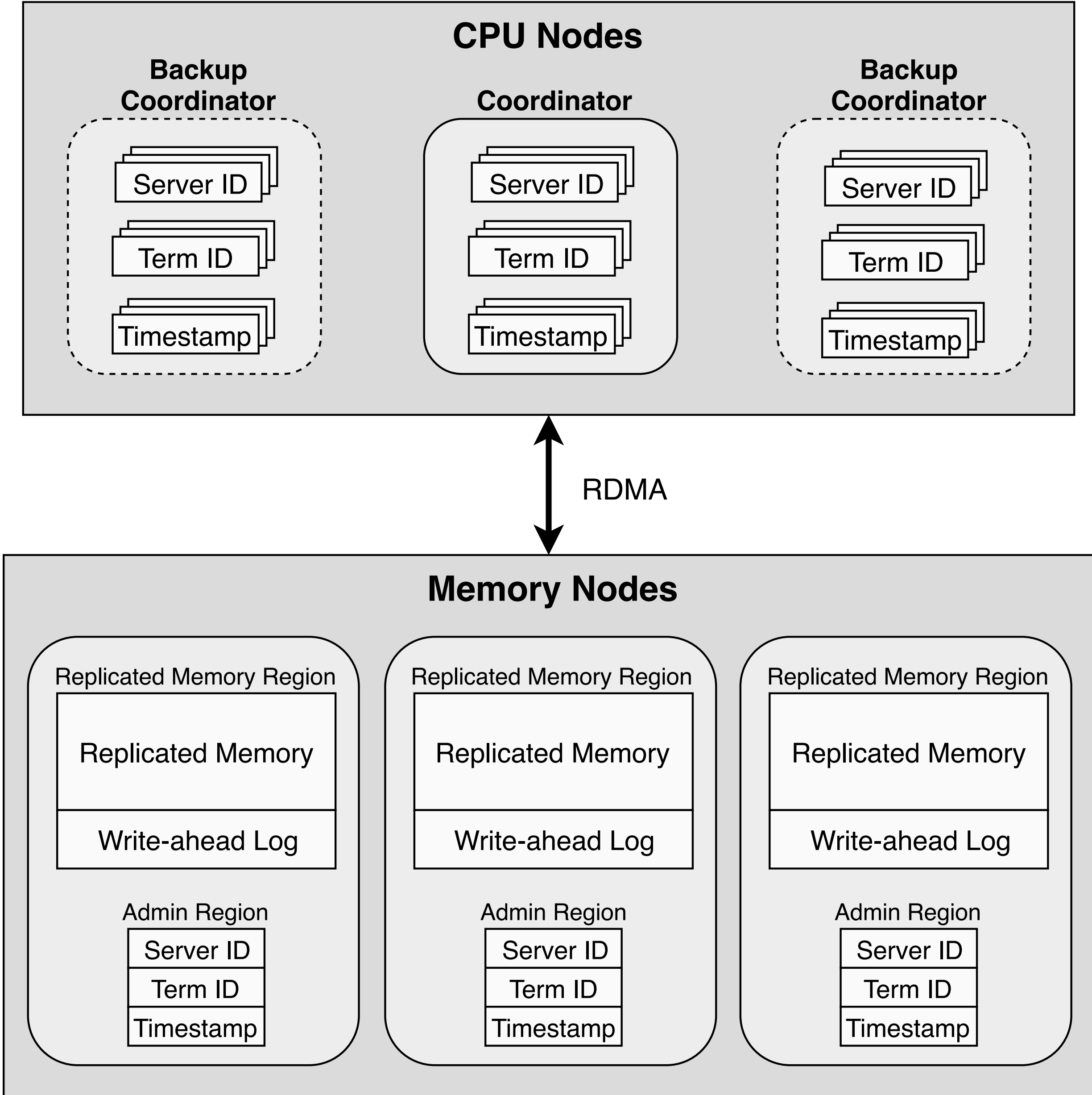

Figure 3.1: Sift architecture.

time. The frequency of heartbeat reads and writes can be configured to allow multiple missed heartbeats before triggering an election.

This heartbeat system works similarly to leases [15] used by other systems to increase read performance by avoiding the need to achieve consensus on each read operation. In Sift, leases can be viewed as follows: the length of a lease is determined by the number of allowed missed heartbeats, and a lease is renewed on each heartbeat write.

Sift has a protocol similar to Raft [32] for electing a coordinator. The CPU nodes initially start in a follower state and a single node is elected as the coordinator. Each follower performs periodic heartbeat reads from the memory nodes. If the follower's *election timeout* period expires without a coordinator performing a heartbeat write, it transitions to the candidate state and initiates a coordinator election. It increments its local `term_id` by 1 and attempts an RDMA CAS operation on all memory nodes using its `node_id` and `term_id`. The follower has the previous `node_id`, `term_id` values needed for the CAS operation stored from previous heartbeat reads.

Multiple candidate nodes compete to perform the CAS operation but only one node successfully sets the `node_id` and `term_id` fields to its own values on any given memory node. This process closely resembles the locking of spinlocks. After the CAS operation succeeds on a majority of the memory nodes, the candidate becomes the coordinator. The rest of the candidate nodes see that another node has successfully written to a majority from the return value of the CAS operation. In this case, the candidates fall back to the follower state, restarting their *election timeout* timer. In the event of an unsuccessful election where no coordinator is elected, each candidate executes a random back-off period before restarting the CAS operation, using the value returned by their unsuccessful CAS operation in their next attempt. Candidates increment their `term_id` for each round of elections. This restricts a candidate from acquiring a memory node using CAS values from older rounds. Since another candidate can acquire the same memory node in successive rounds, we require that the CAS values from the most recent round are used.

Figure 3.2 shows an example execution of the heartbeat mechanism where a coordinator fails. The failure is detected because the coordinator no longer updates the administrative regions of the memory nodes. After a number of missed heartbeats (only one in this example, to conserve space), the backup coordinators reach a timeout and enter the election phase. The first backup coordinator to have their CAS operation succeed on a majority of memory nodes becomes the new coordinator, and begins sending periodic heartbeat writes.

A coordinator detects that it has been replaced when it fails to perform a heartbeat write on a majority of memory nodes. This occurs because another candidate has overwritten their `server_id` and `term_id` during an election, causing subsequent heartbeat

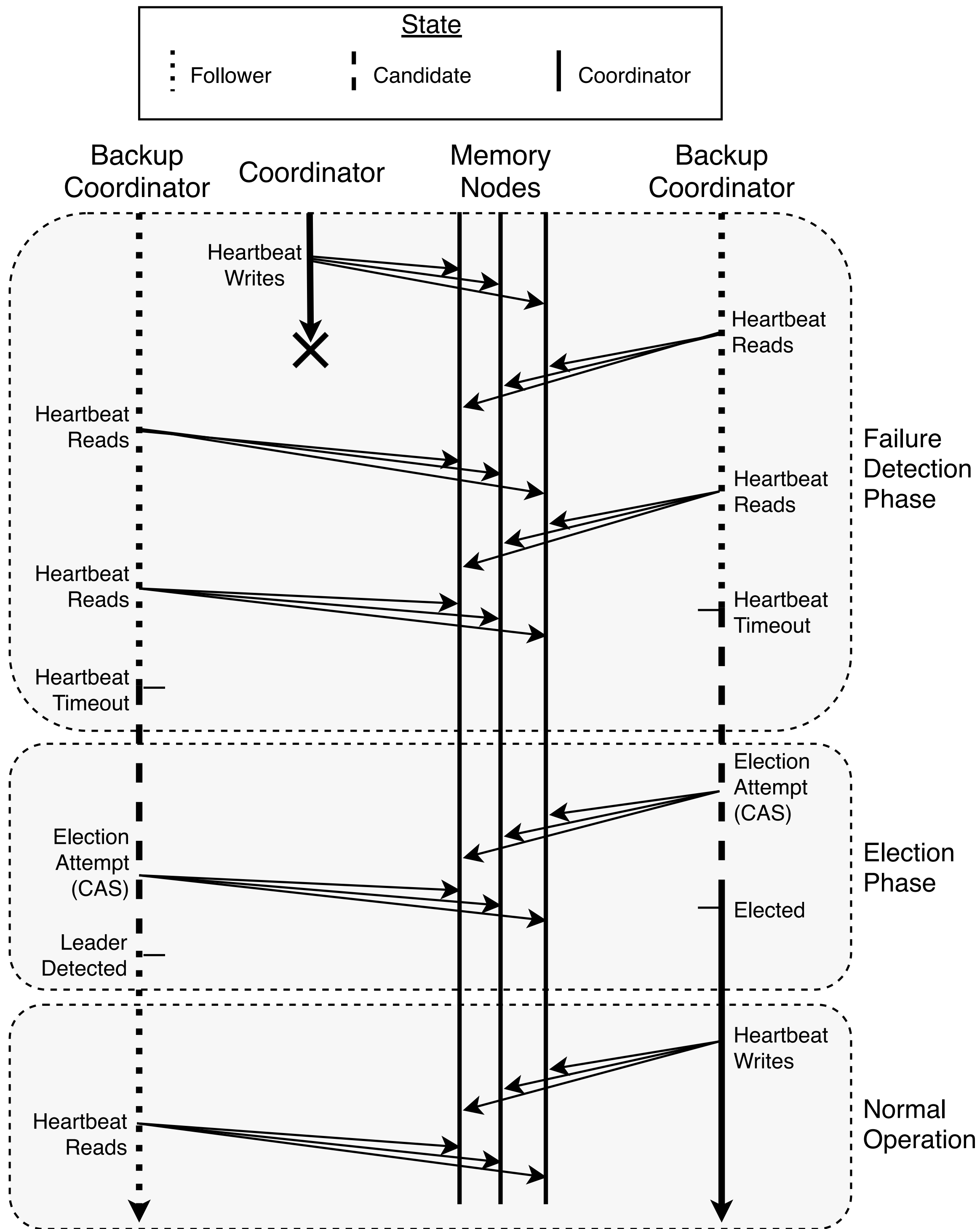

Figure 3.2: Example of a coordinator election scenario. Two backup CPU nodes compete to become the new coordinator once their *election timeout* expires. The CPU node which wins the election begins performing heartbeat writes while the other detects the new coordinator and reverts back to the `follower` state.

writes (CAS) to fail. In this case, the old coordinator reverts back to a follower state. By configuring an *election timeout* that tolerates multiple missed heartbeat writes and assuming bounded clock skew, a dethroned coordinator will be aware of its status before a new coordinator is elected and begins to process requests, thereby avoiding stale reads.

To maintain safety during network partitions where delayed messages arrive after a new coordinator is elected, we implement *at-most-one-connection* semantics to the replicated memory region of the memory nodes. Only the most recently elected coordinator nodes connect to the replicated memory region of a memory node. If a new connection is made to this region, any previous connections are disconnected. This ensures that only the most recently elected coordinator can modify the replicated memory, while delayed messages from past coordinators will be automatically dropped upon arrival by the hardware.

3.3 Normal Operation

In this section, we describe how these operations are performed and how consistency is maintained. Figure 3.3 provides a visual example of the messages exchanged during read and write operations.

3.3.1 Read Requests

Clients issue read requests to the coordinator for a particular address and size in the replicated memory. The coordinator acquires a local read lock for the corresponding blocks, reads the value from a memory node using one-sided RDMA, and returns the read value to the client. It is not necessary to reach a quorum because all requests are processed by the coordinator, which effectively maintains a read lease on the entire replicated memory.

3.3.2 Write Requests

The coordinator performs write requests by acquiring a local write lock for each affected block and appending the update to the write-ahead log on the memory nodes using one-sided RDMA. By using a reliable variant of the RDMA protocol, the coordinator receives an acknowledgment for each write operation to a memory node. Once the append is successful on a majority of the memory nodes, the write has been safely committed and is recoverable in case of failures, thus making it safe for the coordinator to reply to the client.

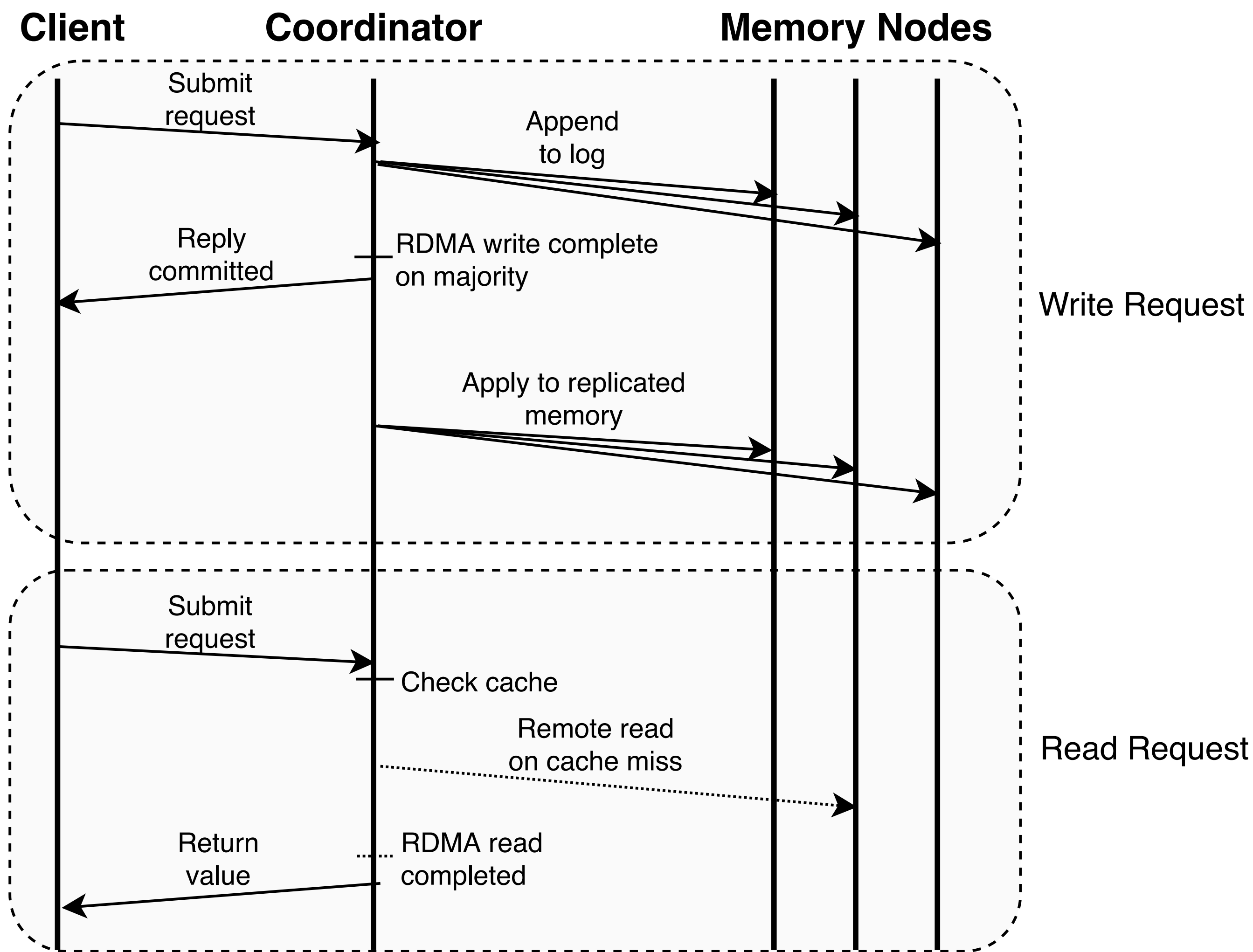

Figure 3.3: Messages exchanged for read and write requests.

The coordinator then updates the replicated memory on memory nodes in the background. We use RDMA's ordering guarantees to maintain consistent state at all times; locks are only released once the operation to update replicated memory has been submitted, so that no reads return stale data after an update has been committed.

To allow for more complex applications to be built on top of the replicated memory, we expose an interface that accepts a list of write requests to be committed together without interleaving with other conflicting write requests. This operation ensures that all included write operations have been logged before replying to the client. We also allow for regions of replicated memory to be written to directly, without being logged. This is useful for applications that manage the conflict resolution and fault tolerance of sections of memory themselves, as with our key-value store's log described in Section 4.1.

3.4 Fault Recovery

Sift is designed to provide fault-tolerance for fail-stop failures. As with all consensus protocols, it cannot guarantee liveness in an asynchronous environment [13]. However, by using random backoffs in its coordinator election algorithm, Sift ensures with high probability that a single coordinator will eventually be elected if there are at most F_c CPU node failures and F_m memory node failures. As a result, with high probability, writes will eventually be committed at every non-faulty memory node when these failure conditions are not exceeded.

3.4.1 Coordinator Failure

In the event of a coordinator failure, a backup CPU node detects the failure through the heartbeat mechanism and initiates a coordinator election. The newly elected coordinator performs log recovery by reading the circular logs from all memory nodes and constructing a consistent, up-to-date version of the log. For memory nodes whose log differs from the majority, the coordinator updates their logs with the missing entries. These updates ensure that the logs on all memory nodes are consistent. The coordinator then replays the log on all memory nodes. Each log entry has its log index embedded within it; these indices are used to obtain the order in which the circular log should be replayed. After replaying the log, all previously committed entries from the log have been applied to the replicated memory, so that future read requests reflect the correct value. At this point the new coordinator can begin processing client requests.

3.4.2 Memory Node Failures

The coordinator keeps track of the live memory nodes in the system. If a memory node becomes unreachable, the coordinator removes it from the list of live memory nodes. A background recovery thread periodically polls all failed memory nodes. When a connection is reestablished, the recovery process begins.

To bring a memory node back into the system, the coordinator must copy all of the data from the replicated memory region to the node. This is achieved by incrementally read-locking regions of memory and copying over the blocks that are in use. By holding a read lock on a region of memory we ensure that no updates can be applied to it, but allow reads to go through. This allows us to continue to process writes at a lower write throughput level while leaving read performance unaffected. Once all of the used blocks within the replicated memory region have been copied over, all locks are released and the new memory node rejoins the system.

Unlike approaches taken by other systems, we do not rely on snapshots for this recovery process as these approaches generally require twice the amount of memory (see [33]). Our approach prioritizes reducing memory requirements while minimizing the performance impact of recovery. In addition, memory nodes that provide persistence (described in Section 3.4.3) would rejoin the system with an older version of the state machine. Recovery for persistent memory nodes can be expedited by performing partial recovery, which can reduce the amount of data that needs to be transferred. A simple method for achieving this would require that the coordinator tracks blocks that have been modified since a memory node failed. This method would work as long as the coordinator does not fail in the meantime. Alternatively, the write-ahead log can be used to determine whether a partial recovery is possible, by looking at the indices of log entries.

3.4.3 Persistence

By default, Sift does not provide persistence in the event of all memory nodes failing, due to all data being stored in volatile memory. However, Sift's architecture does not restrict which storage medium is used by memory nodes, as long as remote access is permitted. With current work on NVMe-over-Fabrics [16, 40], applying new storage mediums such as flash storage to memory nodes is an increasingly feasible approach. The introduction of persistent memory also offers interesting configuration options, with various persistence and cost effects. One example of this is a deployment with a majority of memory nodes being provisioned with volatile memory, while the remainder are given persistent memory. Such a scenario could provide a lower-cost deployment with tunable amounts of data loss.

Alternatively, the coordinator can persist logs onto a remotely mounted Storage Area Network (SAN) device, such as EBS on Amazon EC2, using a write-ahead logging strategy. If a SAN is not available, committed writes can be persisted at just the coordinator. We have implemented such a design using RocksDB, where all updates are synchronously written to the persistent database by a background thread. By limiting the number of outstanding writes to be the size of the log, this design also allows for an alternative to memory node recovery by using snapshots of the database to repopulate the state machine of the new memory node. The trade-off with this approach is requiring a much larger write-ahead log size to reduce the frequency of snapshots and increasing the allowed number of outstanding requests.

Chapter 4

Key-value Store

We build a recoverable key-value store on top of the replicated memory layer described in Chapter 3 to provide a more common and usable abstraction. In our implementation, the process that manages the key-value store is co-located with the Sift coordinator process. However, this is a design decision and not a requirement. The key-value store interacts with the replicated memory layer as it would with its local memory, without having to directly communicate with memory nodes for replication or coordinator election.

4.1 Architecture

The key-value store is designed as a hash table that uses hashing with chaining for simplicity. It consists of four key data structures: an array of data blocks, an index table that holds pointers to data blocks, a bitmap for available data blocks, and a write-ahead log (separate from the log used by the replicated memory system). The data blocks are all of a predefined size; this is a simplifying design decision rather than a limitation of the replicated memory layer.

All of these structures exist within the replicated memory at predefined locations. The index table and bitmap are cached at the coordinator to improve write performance by eliminating up to two remote reads per write request. Read performance is improved by maintaining an LRU cache of key-value pairs at the coordinator. This cache reduces the number of reads sent to the replicated memory, which require a remote read from memory nodes.

4.2 Put/Get Requests

Put requests are processed by appending the update to the key-value store's write-ahead log. This write-ahead log lies in a portion of replicated memory that supports direct writes (without being logged at the replicated memory layer), which allows write requests to be committed in a single RDMA roundtrip. Once the log write has completed, meaning the update has been committed at the replicated memory layer, a reply to the client is sent. In the background, we apply the logged updates to the local and replicated data structures. An update can only be appended to the log if there is space; the number of outstanding (logged but not applied) updates is bounded by the size of the log.

To apply an update, a lookup in the index table is performed using a hash of the key. If there is no entry in the index table, a new block is allocated using the bitmap and its pointer is inserted into the index table. Otherwise, the chain at the index table entry's pointer is traversed using a *next pointer* located in each allocated block. If a block with the same key is found, the value is updated and written back to the replicated memory. If the chain does not contain the key, a new block is allocated and added to the chain, with its pointer being written to the previous block's *next pointer*. Note that updates can be applied concurrently to multiple keys with appropriate locking of the local index table and bitmap structures.

Get requests are processed by first checking the cache. If the key does not exist in the cache, the value is retrieved from replicated memory and, space permitting (see Section 4.4.1), placed in the cache. Otherwise, the value from the cache is returned.

4.3 Failure Handling

The key-value layer needs to be able to recover from only the failure of the process that manages the key-value store; memory node and coordinator failures are dealt with by the replicated memory layer. To recover from a failure, the new key-value process first needs to load the index table and bitmap from replicated memory. With these structures in memory, it can read and replay the contents of the write-ahead log. Once the log has been replayed, the process is able to begin processing client requests.

4.4 Optimizations

4.4.1 LRU Cache

Our LRU cache was extended to help provide linearizability in our system. Before a write request is added to the write-ahead log, a local write lock for that key is obtained and the value is inserted into an LRU cache with a flag marking it as dirty. The write lock ensures that read operations do not see an uncommitted value, while the flag ensures that the entry is not evicted. When a write is successfully appended to the log, the write lock is released, but the flag in the cache remains set. To prevent stale reads, the flag is cleared only when the write to the key has been applied to the data table. This modification makes our cache operate similarly to a buffer cache. It allows reads to be serviced while the more expensive process of applying writes of the same key to the data table in the background completes.

4.4.2 Batching

We utilize batching in our key-value store in two ways. The first is by combining multiple appends to the write-ahead log into a single append. By combining log writes, we reduce the number of RDMA writes that are being issued, and therefore the load on the system. Our second batching technique is collecting writes to the same key in the background data table update phase. If there are multiple logged write requests to the same key, we take the value from the request with the highest log index and perform only one update on that key.

Chapter 5

Resource Reduction

5.1 Erasure Codes

The centralized nature of the replicated memory layer makes it an excellent candidate for further reducing memory requirements through erasure coding. Most protocols already require complex interactions between components and the introduction of erasure codes complicates them further, making it harder to both implement and reason about its correctness. It can also reduce the fault tolerance of the system, requiring more than $2F+1$ nodes to tolerate F failures [31]. In Sift, erasure codes fit the architecture naturally, requiring minimal changes to the protocol and maintaining the same level of fault tolerance.

We modify our replicated memory structure by splitting each block of size B into $F_m + 1$ chunks of size C and use a variant of Cauchy Reed-Solomon codes [38] to generate F_m redundant chunks of size C. Figure 5.1 shows an example of this process on an incoming write request. Using this encoding, the original block can be rebuilt using any subset of size $F_m + 1$ of these chunks. This allows us to tolerate F_m failures while storing only $(2F_m + 1) \times C$ bytes compared to storing $(2F_m + 1) \times B$ bytes with traditional replication – a reduction in memory usage by a factor of $F_m + 1$.

The coordinator is responsible for generating the $2F_m + 1$ chunks before distributing them across the memory nodes. This incurs a computational cost, but also results in fewer bytes being sent across the network compared to standard replication. While a write can still be committed using a quorum of $F_m + 1$ nodes, there is a potential for data loss if we experience a coordinator failure along with the failure of a memory node that was part of the quorum. If none of the other F_m memory nodes received their block before the

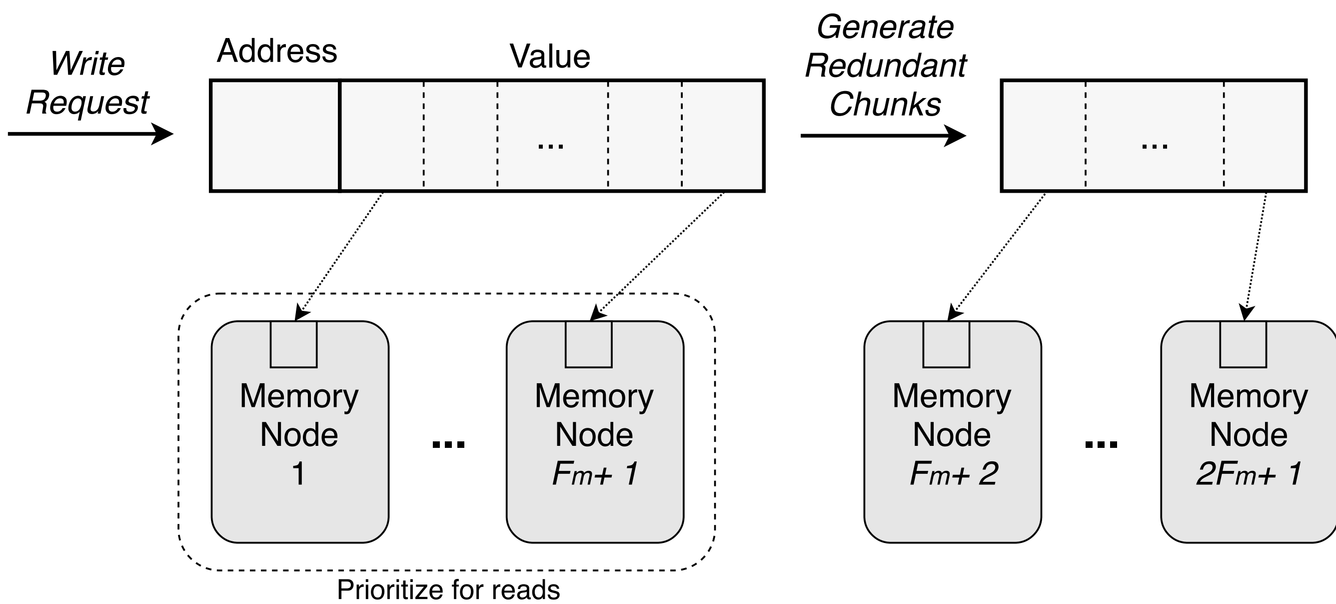

Figure 5.1: Replication process with erasure codes. The original data is divided into F_m+1 chunks and F_m redundant chunks are generated. Each memory node receives one of these chunks.

coordinator failed, the original value would be unrecoverable. To deal with this scenario, we modify our protocol by storing the write-ahead log in non-encoded form. This modification allows Sift to handle failures and tolerate slow memory nodes without impacting common case performance. The write-ahead log provides us with the original data for recovery and generally uses only a small fraction of the total memory. When processing a read request, the coordinator must read from a majority of nodes to rebuild a block, but still transfers a total of only B bytes across the network. We can prioritize reading from memory nodes which store non-parity data to avoid the decoding cost. For memory node recovery, since nodes no longer store a full copy of the data, the coordinator rebuilds each block and encodes it to generate the missing chunks.

5.2 Shared Backup Nodes

One of the key benefits of decoupling compute and storage is the ability to scale these resources independently. Provisioning enough memory for $2F + 1$ replicas is unavoidable in any fail-stop consensus system, but Sift's stateless CPU nodes give us flexibility in how many compute resources are provisioned. By storing only soft state, CPU nodes are not necessarily tied to any given Sift group. This observation leads to a concept that is rarely feasible in consensus systems with coupled resources: shared backup nodes across multiple Sift groups. CPU nodes can be provisioned and added to the backup pool, allowing us to reduce the $F + 1$ CPU node requirement of a single Sift group, which significantly reduces the cost of deployment. This idea is especially attractive in a cloud environment, where consensus can be offered as a service by a cloud provider and backup nodes can be shared across multiple tenants.

There is a trade-off between recovery time and cost savings when choosing the number of backup nodes to provision; a Sift group waiting for a new CPU node to be provisioned is unable to make progress. We investigate the relationship between the number of backup nodes and recovery time using a Google cluster trace, and evaluate the cost effects of this approach, in Section 6.4.2.

Chapter 6

Evaluation

We compare the performance of Sift's key-value store with a custom, RDMA-based, simplified implementation of Raft [32] as well as EPaxos [30]. Sift is evaluated in two configurations: with and without erasure codes, the former of which is named Sift EC.

Our evaluation shows that despite marginally lower performance while running on the same hardware, at normalized performance levels, Sift provides substantial cost savings of over 35% when $F = 1$ and 50% when $F = 2$. These savings are made possible by utilizing erasure codes and shared backup CPU nodes. These optimizations cannot be fully utilized by other consensus systems due to their coupling of compute and storage resources, and the resulting complexity.

6.1 Implementation

Our implementation consists of the core features of the Sift protocol such as replication on memory nodes, maintaining coordinator liveness, coordinator election, as well as recovery from coordinator, memory node, and key-value server failures. The optimizations discussed in Section 4.4 have also been implemented, in addition to the erasure coding and RocksDB configurations discussed in Section 5.1 and Section 3.4.3. The system was written in C++ and consists of approximately 9,000 lines of code. We plan to open-source our implementation.

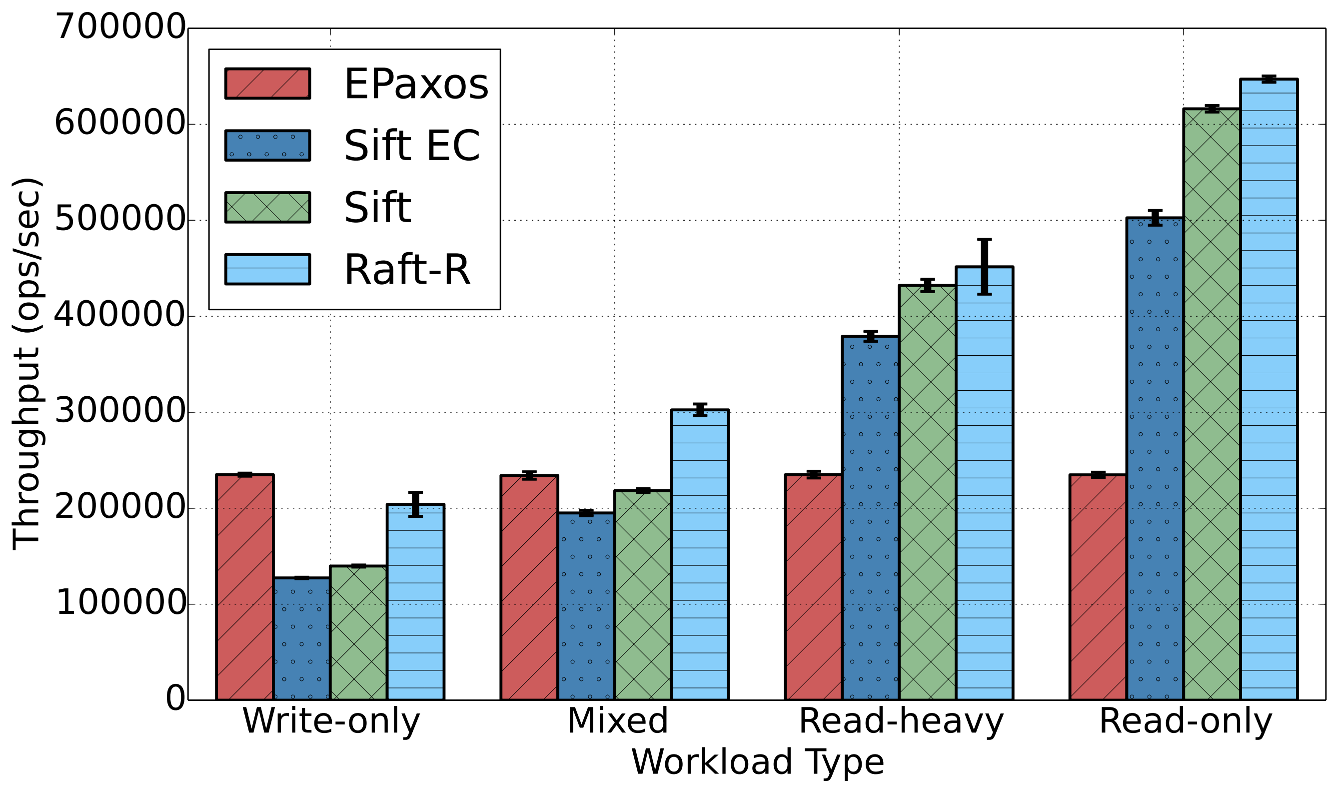

Figure 6.1: Performance comparison with Sift's key-value store and an RDMA-based implementation of Raft.

6.2 Experimental Setup

All experiments are run on a cluster where each machine contains one Mellanox 10GbE SFP port, 64GB of RAM, and two Intel E5-2620v2 CPUs.

Sift's replicated memory system is configured to have a write-ahead log that holds 32k entries. The key-value store is configured to hold 1 million keys, with a maximum key size of 32 bytes and a maximum value size of 992 bytes. The cache is set to hold up to 50% of the key-value pairs and the index table has a maximum load factor of 12.5%. The key-value store's circular write-ahead log can hold up to 64k entries.

All systems we have implemented use the same custom select-based RPC over TCP library for communication between clients and servers.

Experiments run with a fault tolerance level of $F = 1$ correspond to a Sift deployment consisting of three memory nodes and two CPU nodes, and a three-node Raft/Paxos group. Likewise, experiments with $F = 2$ correspond to a Sift deployment with five memory nodes and three CPU nodes, and a five-node Raft/Paxos group. Our experiments use four workload types: write-only, mixed, read-heavy, and read-only. A mixed workload consists of 50% reads and writes, while a read-heavy workload consists of 90% reads and 10% writes. We utilize a Zipfian distribution with a parameter of 0.99 to generate a skewed workload unless otherwise noted.

Each system is pre-populated with all of the keys at the start of each experiment, followed by a 10 second warm-up period. Each experiment lasts for 50 seconds and is repeated 5-8 times. 95% confidence intervals are included when they exceed 5% of the mean.

6.3 Key-Value Store

We evaluate our key-value store that uses Sift and compare it to other consensus-backed key-value stores. All systems in this evaluation store their state machine in memory.

6.3.1 Other Systems

Although DARE [33] implements an RDMA-based consensus protocol, our hardware does not support RDMA multicast operations which are required to run the system. We omit

performance results for DARE but note that it requires fewer RDMA operations, which would likely result in improved performance, at the cost of a coupled architecture.

We were unable to find a popular Raft implementation that allows for an in-memory state machine out-of-the-box, and running popular systems such as LogCabin [26] on a RAM disk introduced bottlenecks in other parts of the systems resulting in an unfair comparison. Thus, we instead built a basic Raft-like system using RDMA send/recv verbs, which allows us to compare the protocols without the performance impact of RDMA.

This Raft-like key-value store, which we call Raft-R, maintains a complete replica on the leader. Write requests are replicated to a majority of nodes (including the leader) before they are committed. Read requests are serviced locally from the leader's replica. It uses a partitioned map with 1000 partitions to reduce contention and read/write locks to provide strong consistency.

We also include performance results for EPaxos [30], a Paxos variant that uses a leaderless approach to consensus to achieve higher throughput. To make it more suitable for a LAN deployment, we have changed the batching parameter from 5ms to 100μs or 100 requests, whichever comes first.

6.3.2 Throughput

Figure 6.1 shows the performance of Sift compared to Raft-R and EPaxos when $F = 1$. Sift's write throughput is lower than a Raft-R due to the larger amount of work being performed in the background to apply writes to the state machine. Similarly, due to stateless CPU nodes, a portion of read requests result in remote reads. The effect of remote reads is magnified in Sift EC because of the higher number of RDMA reads that need to be performed. We limit the effect of remote reads through the cache, resulting in read throughput similar to Raft-R.

The performance of EPaxos is independent of the workload type. This is primarily because both reads and writes require network operations. Because of the need for network operations for reads, the read throughput for EPaxos is significantly lower than the other systems. The write-only performance is better than both Raft-R and Sift. This is likely due to the leaderless design; clients were configured to be evenly distributed across the EPaxos nodes.

In summary, the performance of both Sift and Raft-R is far higher than a state-of-the-art, non-RDMA consensus protocol for read operations. Both Sift and Raft-R perform fewer remote operations with more read-heavy workloads. With a write-only workload, EPaxos performs better than the leader and RDMA-based systems.

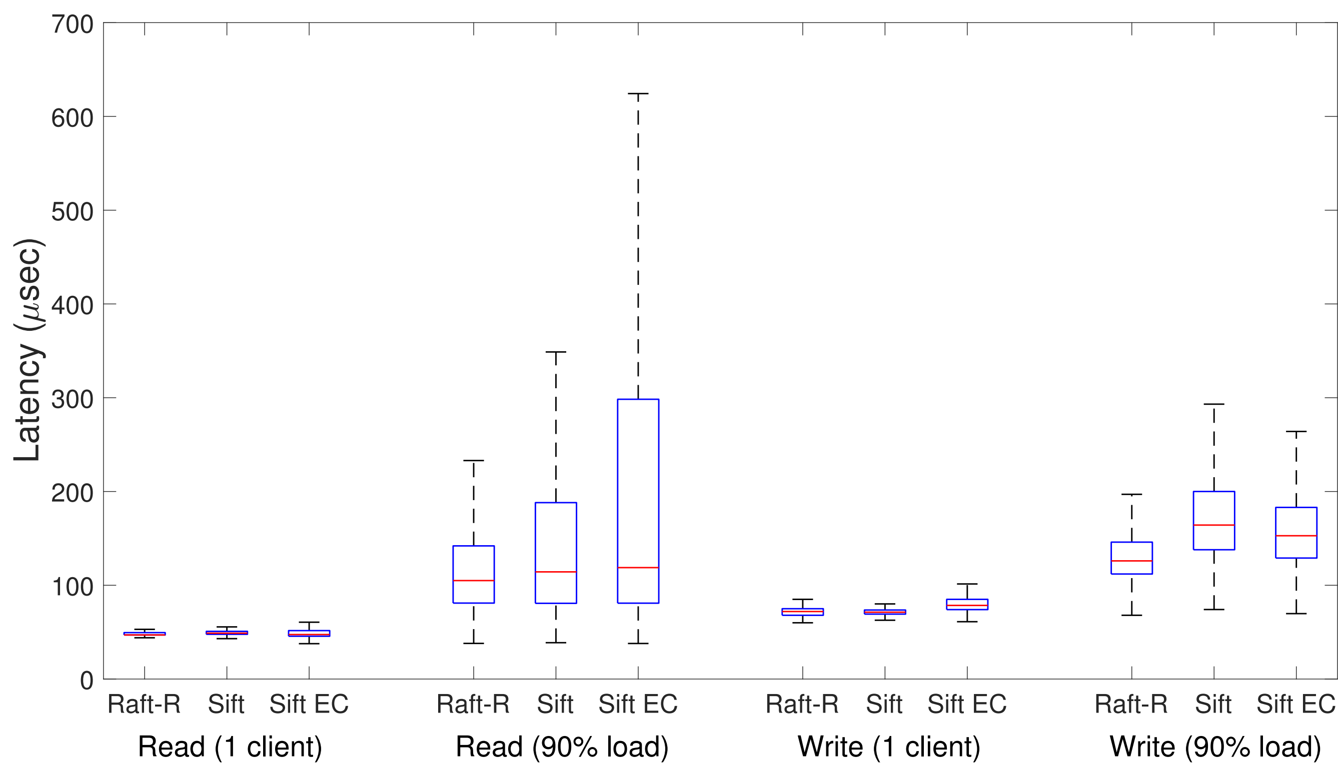

Figure 6.2: Latencies at low load (1 client) and 90% load.

6.3.3 Latency

Figure 6.2 compares the latencies of Sift and Raft-R at different load levels: at low load, with at most one request in the system at a time, and at 90% of peak throughput.

EPaxos achieves a median latency of 94μs and a 95th percentile latency of 140μs for both reads and writes at low load. Latencies for reads and writes at low load are equivalent because there are no conflicts between requests, resulting in the same number of roundtrips to commit a request. At 90% load, read and write latencies exceed 1.3ms and 95th percentile latencies approach 2ms. To improve the clarity of Figure 6.2, we do not include the latencies for EPaxos.

At low load, the cost of writes is similar for all systems since they both wait for one RDMA roundtrip to replicate the state update. Sift EC experiences a slightly higher latency due to the encoding process. At higher load, the increased load from background threads applying writes results in more contention and consequently, higher latencies for Sift and Sift EC.

Read latencies at low loads are also similar for all RDMA-based systems. This is mainly due to the cache being able to service the majority of requests in Sift and Sift EC. At higher loads, while the median stays relatively similar, Sift experiences higher variance due to an increase in the number of remote reads. Sift EC is especially sensitive to this as it requires sending multiple RDMA reads to decode the data for each cache miss.

For both Sift and Raft-R, approximately 50μs of latency is attributed to the RPC layer used for communication between clients and the coordinator.

6.4 Cost Analysis

6.4.1 Normalized Performance

To analyze the cost of deploying each of these systems, we first normalize the performance. We extend the performance evaluation in Section 6.3.2 and consider deployments with varying CPU resources, shown in Figure 6.3. These results are used to determine how each system's machines should be provisioned to fairly evaluate costs.

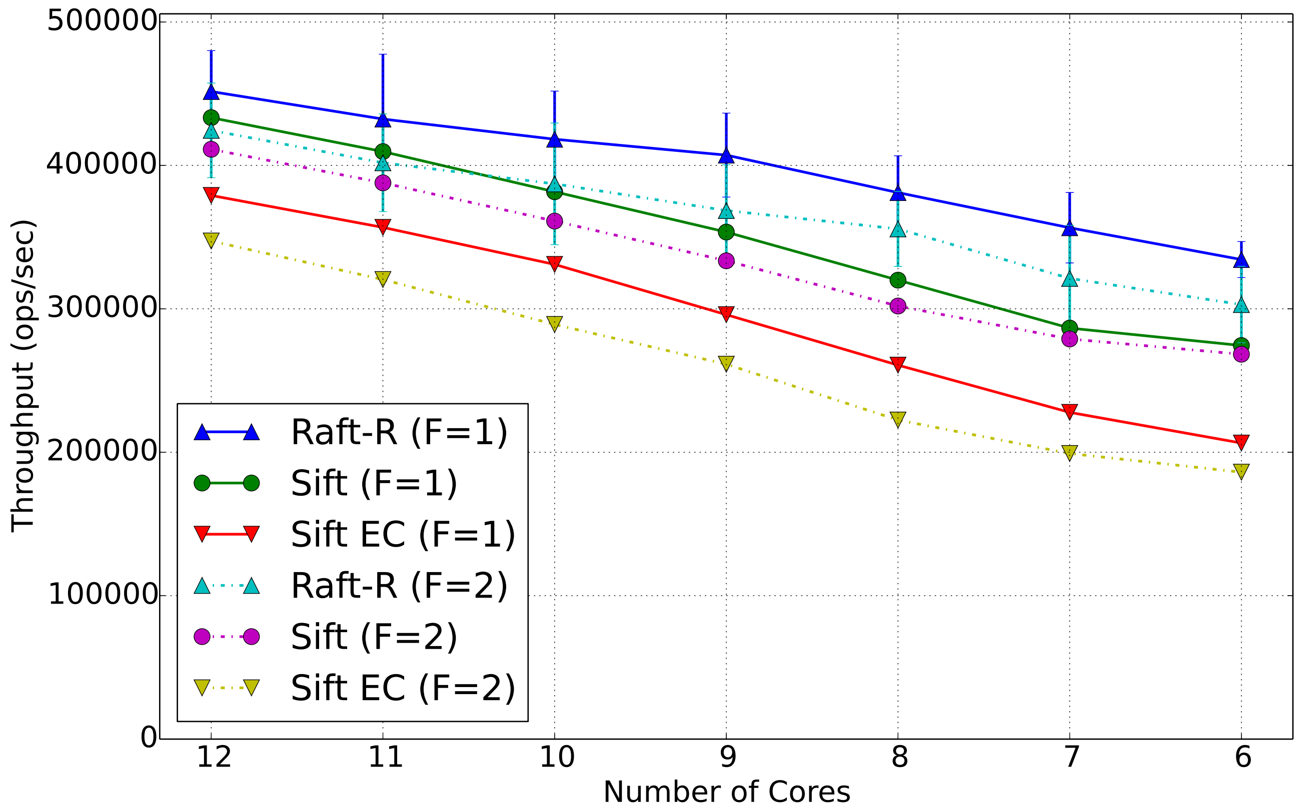

Figure 6.3: Performance of Sift and Raft-R with a varied number of cores on all nodes (excluding Sift memory nodes, which use no CPU), for F = 1 and 2. These results show us how machines should be provisioned to achieve equivalent performance.

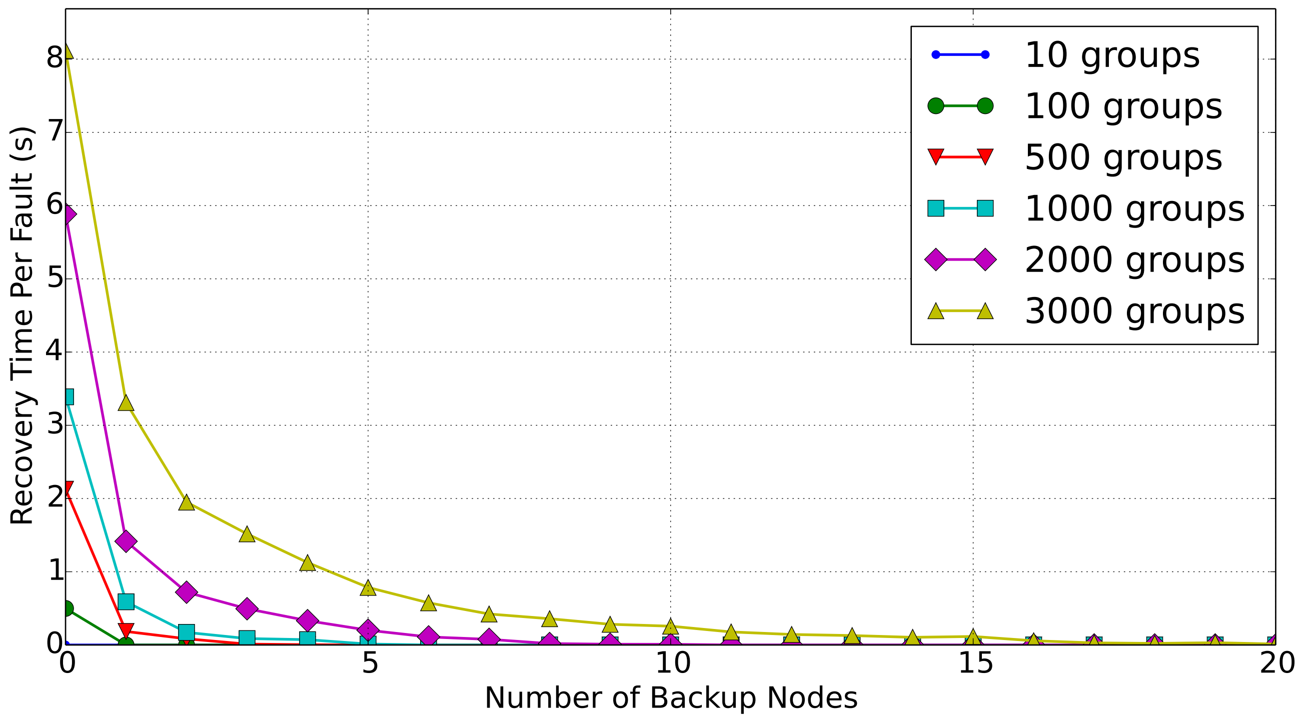

Figure 6.4: Results of a simulation over a Google cluster trace [43] of machine failures. Estimates how many backup nodes are needed to prevent additional recovery time due to VM provisioning.

6.4.2 Shared Backup Nodes

To better understand how backup nodes influence recovery time, we use a Google cluster trace [43] which provides a 29 day trace of cluster information, including failure events. The cluster consists of approximately 12500 machines. This trace is used to run simulations over several numbers of Sift groups to determine the average recovery time per fault with varied numbers of backup nodes. Each simulation was run by randomly assigning machines to Sift groups and observing the additional recovery time incurred by a lack of backup nodes. When a node experienced a failure, it was assumed that it would take 100 seconds to provision a replacement – the average time to start up a Linux VM in EC2 [27].

For every combination of group size and number of backup nodes, the simulation was repeated 50 times. Each group was configured for $F = 1$, resulting in 3 memory nodes and 1 CPU node per group. Figure 6.4 shows the average recovery time per fault of these simulations. We are only interested in the additional recovery time due to VM startup, so Sift coordinator recovery time is not included, leading to a best-case recovery time of 0. These results show that even for a relatively large number of groups (1000), maintaining a pool of 6 backup nodes is enough to ensure no additional recovery time. For a much larger number of groups (3000), 20 backup nodes were needed. By contrast, a deployment without shared backup nodes would require 1 extra CPU node per group.

6.4.3 Costs

We compare the projected costs of deploying Sift (in various configurations) and Raft-R in a cloud environment. While RDMA has increasingly many use cases, it remains difficult to purchase machines equipped with it from cloud providers. Furthermore, there are no machines that provide a hardware configuration that complements one-sided RDMA and by extension, our design. We instead use the currently available machine pricing from AWS[3] compute and memory-optimized instances to calculate the marginal costs of CPU and memory, assuming RDMA capabilities. These computations give us a price of \$0.033/core/hour and \$0.00275/GB/hour for memory, which we use to provision custom machines for each system. We believe that this is a reasonably accurate portrayal of hardware configurations that cloud providers could offer. Regardless of current offerings, cloud providers who offer consensus as a service are able to provision machines for themselves as needed.

We use Sift and Raft-R deployments that achieve the same fault tolerance guarantees and performance. Using the results from Figure 6.3 to determine resource requirements,

	F=1		F=2	
	Cores	Memory (GB)	Cores	Memory (GB)
Raft-R Node	8	64	8	64
Sift CPU Node	10	32	10	32
Sift Memory Node	1	64	1	64
Sift EC CPU Node	12	32	12	32
Sift EC Memory Node	1	32	1	22

Table 6.1: Machine configurations for each system normalized for performance.

we set a target throughput for a read-heavy workload of 380k ops/sec for $F = 1$ and 350k ops/sec for $F = 2$. The machine configurations used are shown in Table 6.1. The memory requirement for CPU nodes is calculated from the soft state needed in our key-value store, using the configuration from Section 6.2. A Sift EC memory node uses a factor of $F + 1$ less memory.

Figures 6.5 and 6.6 show the costs of these deployments. For $F = 1$, a single Sift and Sift EC group requires marginally higher costs than a Raft-R group. This is due to the performance differences between the systems, which require Sift and Sift EC to be provisioned with more compute resources. However, once we introduce shared backup nodes and erasure codes, we see a cost reduction of up to 35%. This decrease in cost is the result of removing the redundant CPU node from each group and maintaining only a small group of backup nodes. The backup pool sizes are selected by analyzing the results of the simulation over the Google cluster trace (Figure 6.4) and finding sizes that result in no additional recovery time for each number of groups.

Sift costs decrease across all configurations when F is increased to 2. These reductions in cost are due to the increased number of replicas in each group, resulting in higher memory usage which begins to dominate the costs. Sift EC receives an additional benefit from erasure codes due to the $F + 1$ reduction factor in state replication. A single Sift EC group now costs about 13% less than a Raft-R group. When both erasure codes and shared backup nodes are used, a cost reduction of up to 56% is achieved.

In summary, we see larger cost reductions using erasure codes than with standard Sift. These cost reductions are significantly increased when shared backup nodes are used. With increasing values of F, all Sift configurations improve relative to Raft-R despite Sift maintaining a large cache, due to Sift's disaggregated architecture which allows us to scale compute and memory resources independently.

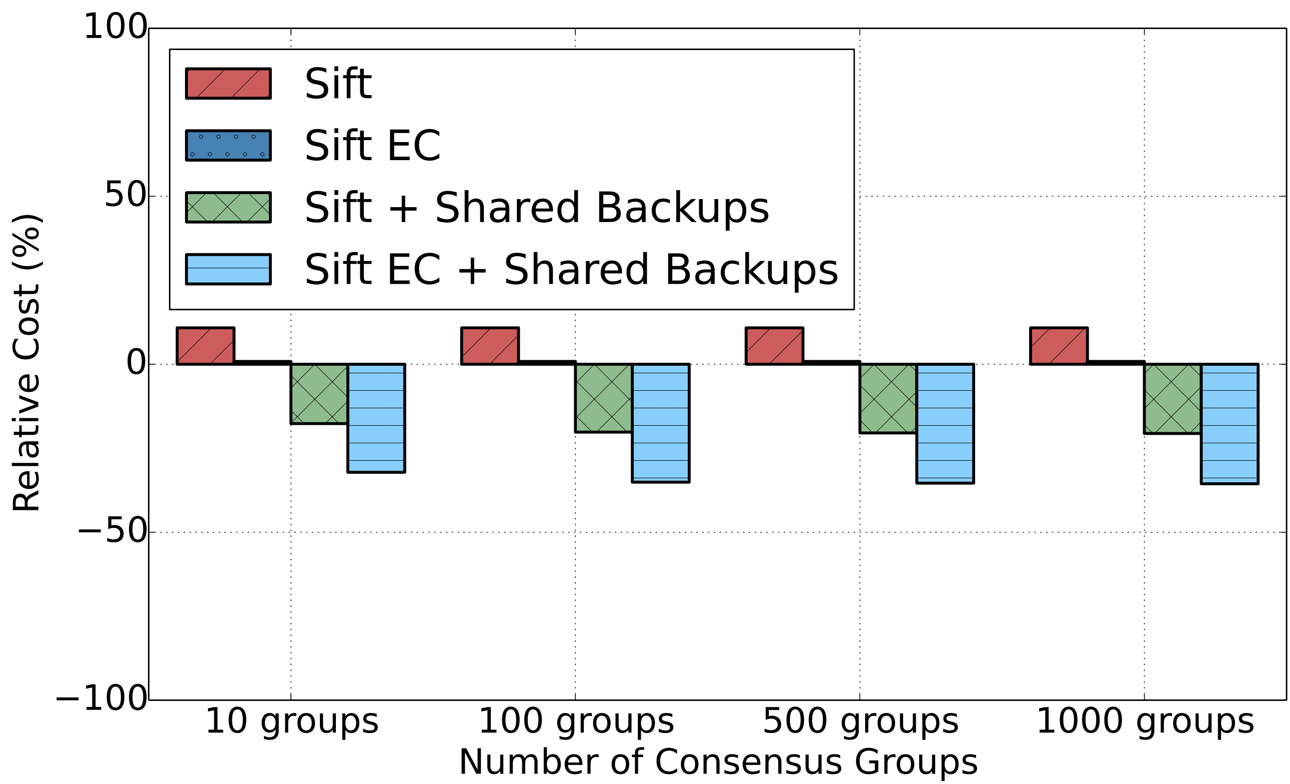

Figure 6.5: Costs of deploying Sift configurations relative to the cost of Raft-R. Machines provisioned for F=1.

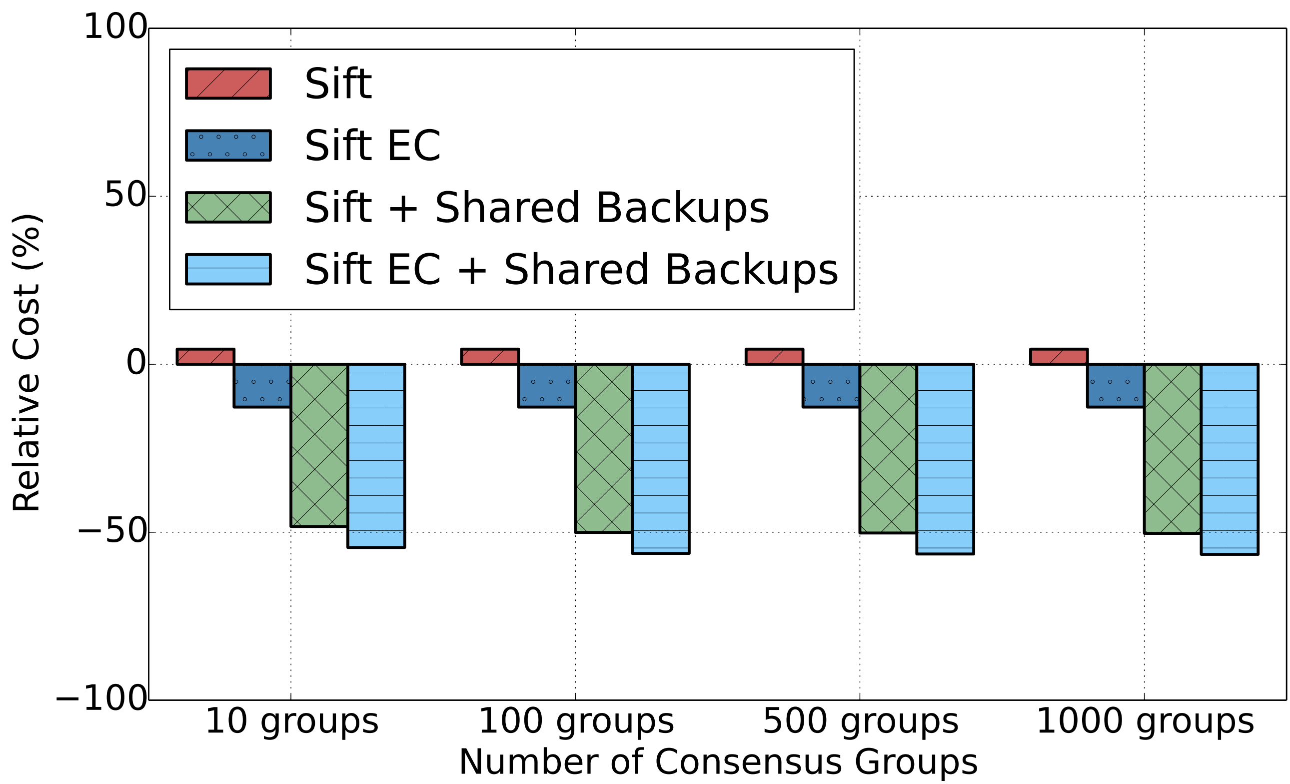

Figure 6.6: Costs of deploying Sift configurations relative to the cost of Raft-R. Machines provisioned for F=2.

6.5 Failure Recovery

We measure the effects of failures in Sift by killing a node in the system and recording the throughput. We use a read-heavy throughput with a skewed workload and measure in 100ms intervals.

Figure 6.7 shows how a memory node failure affects throughput. Throughput drops as regions of memory are copied over for memory node recovery. Our experimental setup has the most popular keys stored at lower memory addresses, so we see the maximum effects instantly. If the workload was uniformly distributed, the throughput would not drop more gradually. Similarly, if the popular keys were stored at higher addresses, the system would sustain a higher throughput for the majority of the recovery period. In this experiment, the recovery lasts approximately 6 seconds, after which the system returns to its pre-failure throughput level.

The current memory node recovery approach aggressively copies data to the new memory node to bring it back into the system as quickly as possible. This approach is flexible as it is possible to prolong the recovery time to maintain a steadier degradation of throughput. The coordinator can also wait for a period of reduced load to begin the recovery process. A more complex recovery approach could identify the most popular memory blocks and copy them in order of increasing popularity to reduce the effective performance impact.

A coordinator failure causes the system to pause processing client requests until the system has been brought to a consistent state, as shown in Figure 6.8. The recovery time consists of three parts: detecting a failed coordinator through heartbeats, recovering and replaying the replicated memory log, caching key-value structures, and replaying the key-value log. Recovery time is largely determined by the size of the write-ahead log in both the key-value and replicated memory layers. In general, smaller log sizes reduce recovery time but can potentially degrade system performance under high load by reducing the number of in-flight (committed but not applied) write requests.

In the experiment shown by Figure 6.8, recovery took approximately 6 seconds. With a heartbeat read interval of 7ms and a tolerance of three missed heartbeats, failure detection took approximately 21ms. Recovering the replicated memory state took approximately 1s. The remaining 5s were spent loading the index table and bitmap into memory and replaying the log. Note that while the log is being replayed, the cache is populated in parallel, which allows our system to process requests at a faster rate when it resumes than with a cold start. Similarly, a burst in throughput is experienced following recovery due to buffers being empty.

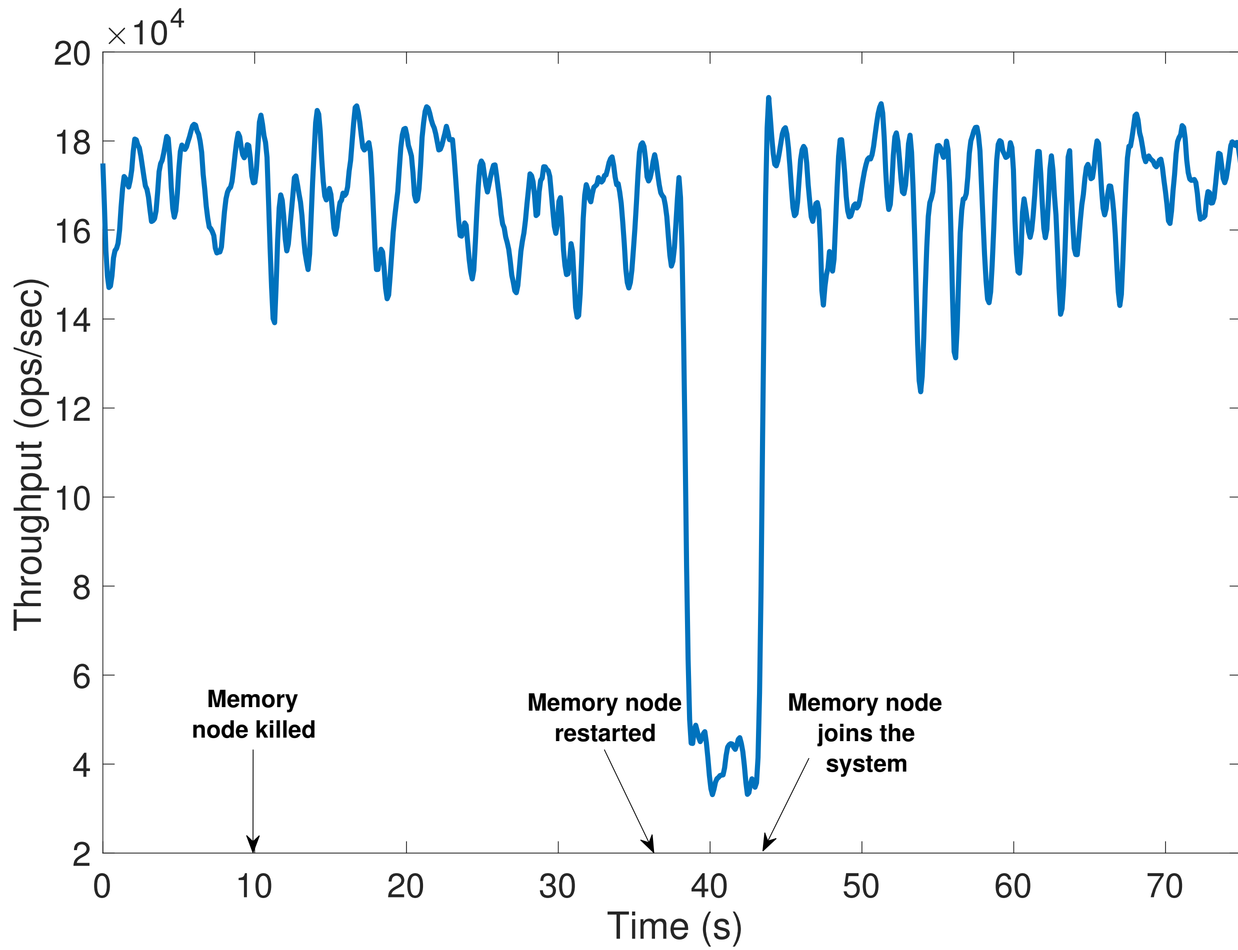

Figure 6.7: Read-heavy workload throughput during a memory node failure.

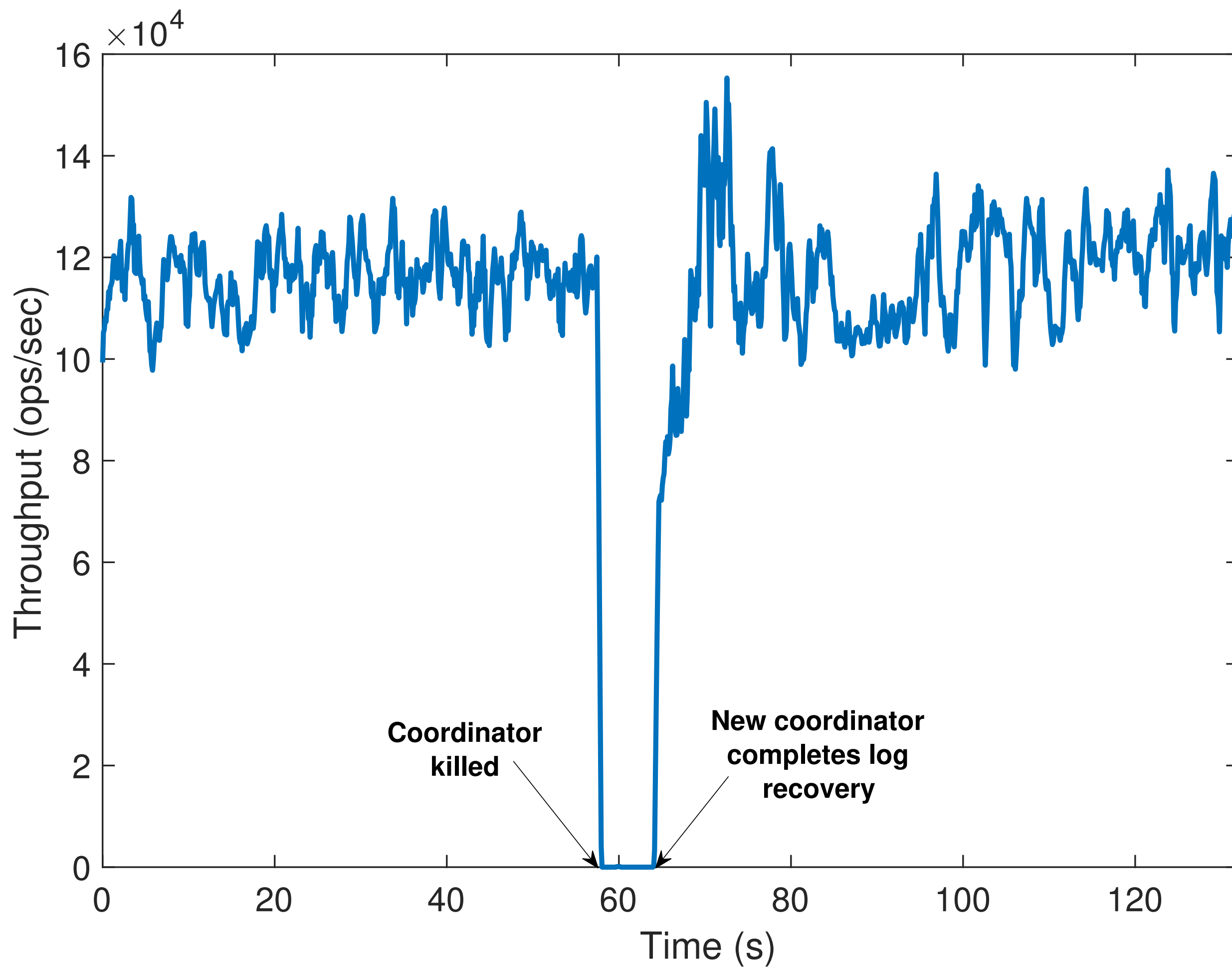

Figure 6.8: Read-heavy workload throughput during a coordinator failure.

6.6 Zookeeper

In this section, we compare the performance of Sift and ZooKeeper's atomic broadcast protocol by integrating Sift into ZooKeeper, which we call SiftZK. The integration was done by replacing ZooKeeper's protocol with 12 Sift client threads, which communicate over TCP to a Sift coordinator. For these experiments, the Sift coordinator was located on the same machine as the Zookeeper leader.

Both ZooKeeper and SiftZK are set up with a 3-node system ($F = 1$). For SiftZK, this means using a Sift group with 3 memory nodes. ZooKeeper was configured to store its data in a *tmpfs* partition to compare against Sift's in-memory storage.

For this experiment, SiftZK does not have fault recovery implemented, so a single SiftZK server is used that acts as the leader. To enable fault recovery, backup CPU nodes would simply act as passive SiftZK servers until a leader failure is detected. Since SiftZK servers are now stateless, recovery is minimal and mostly handled by Sift, which also allows for the sharing of backup CPU nodes. Note that backup CPU nodes can load any program upon failure detection, so consensus groups need not be running the same application to share backups.

To measure the throughput and latency of requests, we modify the clients from the ZooKeeper smoketest [19] to send requests with varied batch sizes, ranging from 1 to 100 requests sent at a time per client. Clients are distributed across 7 machines, with up to 60 sessions being created. In ZooKeeper, clients are able to send requests to any of the 3 nodes. Since SiftZK only has 1 active server, all requests go through a single node.

This experiment uses 50000 znodes (keys), which are created prior to running the experiments. Each request consists of an 8-byte key and 25-byte value. Key selection follows a uniform distribution.

Figure 6.9 shows the throughput and latency curves of both systems under read-only and write-only workloads. Both ZooKeeper and SiftZK achieve similar read performance. While ZooKeeper services all read requests from local state on each node, Sift does not maintain a replica at the coordinator, so some read requests will require reading from memory nodes. Additionally, SiftZK must communicate with the coordinator over TCP. These two factors result in marginally lower peak throughput for SiftZK.

Write performance varies to a greater extent. In ZooKeeper, all write requests are forwarded to the leader. From there, the leader is responsible for coordinating and committing state changes. Additionally, all communication is done over TCP. The result is that ZooKeeper's protocol results in higher overhead compared to Sift, in which a write request is committed after a single RDMA write to a write-ahead log.

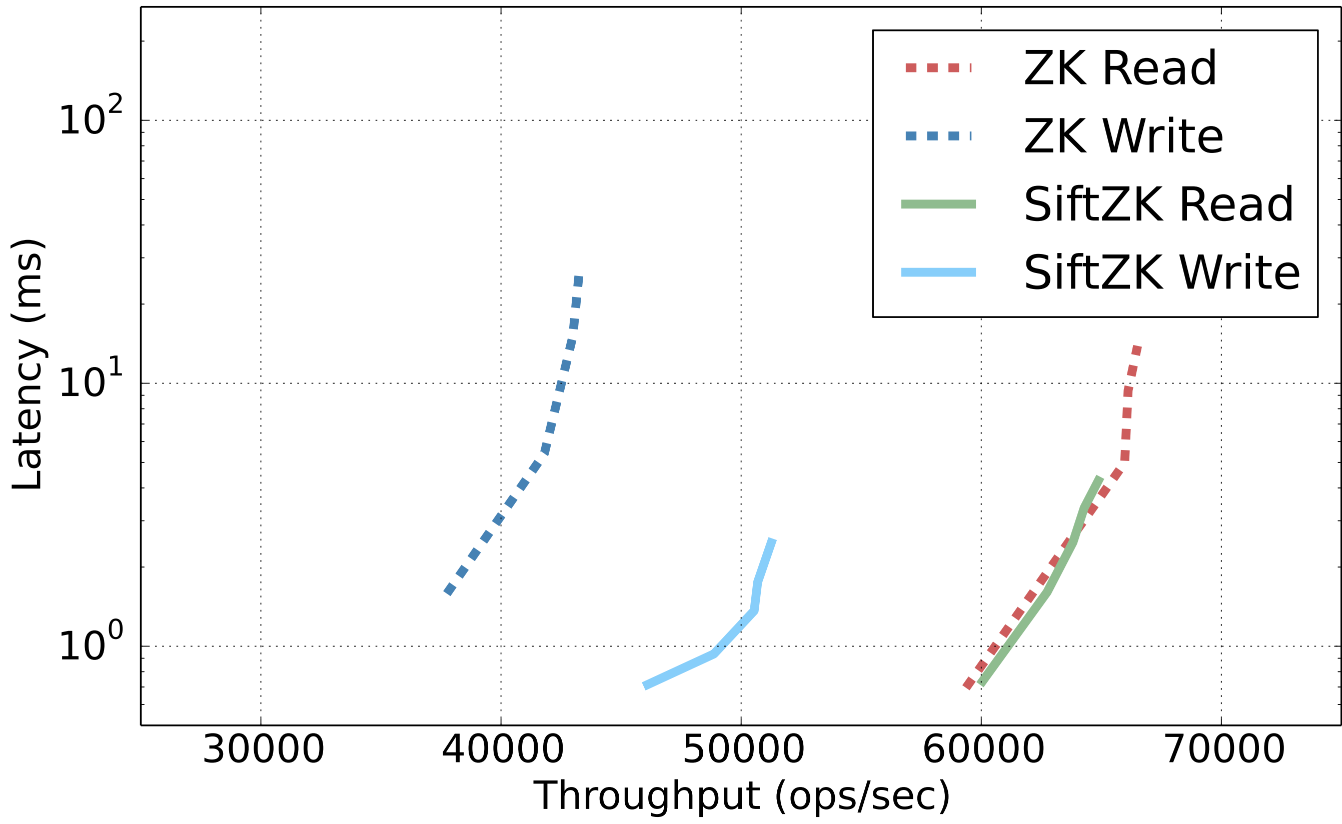

Figure 6.9: Throughput vs. latency curves of ZooKeeper and SiftZK, which uses Sift for replication.

The integration of Sift into ZooKeeper requires minimal work and results in a simplified replication protocol. Applications that currently use ZooKeeper could utilize SiftZK to achieve higher throughput and lower latency writes. With some additional modifications to ZooKeeper that allow for shared backup nodes, these applications can also make use of a Sift consensus-as-a-service offering to significantly reduce their deployment costs, as shown in Figures 6.5 and 6.6.

Chapter 7

Conclusion and Future Work

In this thesis, we present Sift, a consensus protocol that uses one-sided RDMA to disaggregate compute and memory resources. Our consensus protocol is used to build a replicated memory system, upon which we build a fault-tolerant key-value store. Our evaluation shows that even when accounting for slight throughput and latency overhead from our disaggregated design, the introduction of erasure codes and shared backup nodes can significantly reduce deployment cost. After normalizing for performance, for $F = 1$ and 100 consensus groups, Sift is able to achieve up to a 35% cost savings compared to an RDMA-based Raft implementation. Cost savings improve with higher values of F. When $F = 2$, Sift increases its cost savings to 56%.

Although all systems evaluated used volatile storage, persistence remains a desirable option for many applications. Sift's architecture allows for interesting opportunities with respect to persistence. Since memory nodes are passive participants in the system, it becomes possible to vary the storage devices within a single group, which changes the fault tolerance characteristics of the system. For example, a group with up to a minority of non-volatile memory nodes can achieve a configurable amount of data loss in the event of a catastrophic failure, while still being able to achieve consensus through presumably faster volatile memory nodes. The usage of non-volatile memory nodes can also reduce the deployment cost of a system, since non-volatile storage tends to be cheaper.

Another direction for consensus-as-a-service with Sift that we did not explore entails the sharing of memory nodes across Sift groups. This approach is similar to the one taken by Filo [29]. A memory node is able to register multiple isolated memory regions, creating a virtual memory node for each group. Such an approach is subject to issues regarding resource-sharing and failure effects, as described in Section 2.5, but could be a useful

extension to Sift that enhances resource sharing, further reducing costs.

Finally, our decision to implement Sift as a key-value store was made primarily for comparison purposes. It allowed us to include existing systems in our evaluation such as ZooKeeper, which required minimal work for integrating our key-value store. However, Sift's replicated memory system can be used as a standalone service. It closely resembles distributed shared memory and can be used to build arbitrary fault-tolerant services atop.

References

[1] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The resource-as-a-service (raas) cloud. In *USENIX conference on Hot Topics in Cloud Ccomputing*, pages 12–12, 2012.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[3] Amazon AWS Pricing. https://aws.amazon.com/ec2/spot/pricing.

[4] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Symposium on Operating Systems Design and Implementation*, pages 335–350. USENIX Association, 2006.

[5] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *ACM Symposium on Operating Systems Principles*, pages 277–290, 2009.

[6] David Cohen, Thomas Talpey, Arkady Kanevsky, Uri Cummings, Michael Krause, Renato Recio, Diego Crupnicoff, Lloyd Dickman, and Paul Grun. Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options. In *IEEE Symposium on High Performance Interconnects*, pages 123–130, 2009.

[7] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[8] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. *ACM SIGCOMM Computer Communication Review*, 45(4):551–564, 2015.

[9] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226. ACM, 2016.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles*, pages 205–220, 2007.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. Farm: Fast remote memory. In *USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *ACM Symposium on Operating Systems Principles*, pages 54–70, 2015.

[13] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[14] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[15] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.*, 23(5):202–210, November 1989.

[16] Zvika Guz, Harry Huan Li, Anahita Shayesteh, and Vijay Balakrishnan. Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, page 16. ACM, 2017.

[17] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *ACM Workshop on Hot Topics in Networks*, pages 10:1–10:7, 2013.

[18] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of big data on cloud computing: Review and open research issues. *Information Systems*, 47:98–115, 2015.

[19] Patrick Hunt. Zookeeper smoketest. https://github.com/phunt/zk-smoketest.

[20] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *USENIX Annual Technical Conference*, pages 437–450, 2016.

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

[23] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):51–58, 2001.

[24] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture*, pages 267–278, 2009.

[25] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *International Symposium on High-Performance Computer Architecture*, pages 1–12, 2012.

[26] LogCabin. https://github.com/logcabin/logcabin.

[27] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430. IEEE, 2012.

[28] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[29] Parisa Jalili Marandi, Christos Gkantsidis, Flavio Junqueira, and Dushyanth Narayanan. Filo: Consolidated consensus as a cloud service. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 237–249, Denver, CO, 2016. USENIX Association.

[30] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on*

Operating Systems Principles, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[31] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 61–72. ACM, 2014.

[32] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–320, 2014.

[33] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118, 2015.

[34] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[35] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *High Performance Computing, Networking, Storage and Analysis*, pages 19:1–19:10, 2012.

[36] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[37] Kshitij Sudan, Saisanthosh Balakrishnan, Sean Lie, Min Xu, Dhiraj Mallick, Gary Lauterbach, and Rajeev Balasubramonian. A novel system architecture for web scale applications using lightweight cpus and virtualized i/o. In *International Symposium on High Performance Computer Architecture*, pages 167–178, 2013.

[38] Christopher Taylor. cm256. https://github.com/catid/cm256.

[39] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM, 2017.

[40] Benjamin Walker. Spdk: Building blocks for scalable, high performance storage applications. In *Storage Developer Conference. SNIA*, 2016.

[41] C. Wang, A. Gupta, and B. Urgaonkar. Fine-grained resource scaling in a public cloud: A tenant's perspective. In *IEEE International Conference on Cloud Computing*, pages 124–131, 2016.

[42] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Symposium on Cloud Computing*, pages 94–107, 2017.

[43] John Wilkes. More google cluster data. https://ai.googleblog.com/2011/11/more-google-cluster-data.html, 2011.

[44] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *ACM Symposium on Operating Systems Principles*, pages 253–267, 2003.

[45] Wenbing Zhao. Fast paxos made easy: Theory and implementation. *International Journal of Distributed Systems and Technologies*, 6(1):15–33, 2015.