Sift: Resource-Efficient Consensus with RDMA

Mikhail Kazhamiaka, Babar Memon, Chathura Kankanamge, Siddhartha Sahu, Sajjad Rizvi, Bernard Wong, Khuzaima Daudjee
{mkazhami,bmemon,c2kankan,s3sahu,sm3rizvi,bernard,kdaudjee}@uwaterloo.ca
Cheriton School of Computer Science, University of Waterloo

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.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**.

KEYWORDS

Consensus, Reliability, RDMA, Resource Disaggregation

ACM Reference Format:
Mikhail Kazhamiaka, Babar Memon, Chathura Kankanamge, Siddhartha Sahu, Sajjad Rizvi, Bernard Wong, Khuzaima Daudjee. 2019. Sift: Resource-Efficient Consensus with RDMA. In *CoNEXT '19: International Conference On Emerging Networking Experiments And Technologies, December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3359989.3365437

1 INTRODUCTION

State Machine Replication (SMR) enables the construction of reliable services by ensuring events are recorded in the same order 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. Paxos [13] 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.

Although the performance of consensus protocols can pose a problem for some applications, most that require high throughput, such as databases and storage systems, have states that can be easily partitioned, allowing horizontal scaling across consensus groups. However, horizontal scaling does not address the monetary cost of replicating state to $2F + 1$ replicas. Each replica must also be provisioned with enough compute and memory resources to serve as the elected leader, resulting in significant resource under-utilization on the non-leader nodes. As a result, the cost of providing SMR can be prohibitive for applications that have large state machines and must remain available in the event of multiple simultaneous failures.

For applications running in the cloud, one approach to reduce the costs of consensus is by using a shared consensus service offered by the cloud provider. This allows resource sharing across applications which can reduce the cost of running the service. However, providing performance isolation between applications can be challenging. 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 that is often the dominant cost in providing a consensus service.

Previous work [22] has explored using erasure codes to reduce the storage requirement of SMR. 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. Adding erasure coding introduces significant complexity to an already complex system, making 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 paper, 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) 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 consensus protocols, 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.

Type	Resource Location	Protocol	Erasure Coding	Replication Factor
Sift	Disaggregated	1-sided RDMA	Yes	$2F_m + 1, F_c + 1$
Raft	Coupled	TCP	No	$2F + 1$
DARE	Coupled	1-sided RDMA	No	$2F + 1$
RS-Paxos	Coupled	TCP	Yes	$Q_R + Q_W - X$ [22]
Disk Paxos	Disaggregated*	Unspecified	No	$2F + 1$ disks $+ P + L$

Table 1: Comparison of key consensus protocol characteristics. Sift's compute replication factor can be reduced through the sharing of backup CPU nodes, which we describe in Section 5.2. Note a difference in disaggregated architectures: in Sift, the passive memory nodes store both the log and materialized state, whereas in Disk Paxos the passive nodes are only acceptors. A Disk Paxos deployment requires $2F + 1$ acceptors (disks) and some number of proposers (P) and learners (L). RS-Paxos presents its replication factor in terms of its read quorum size (Q_R), write quorum size (Q_W), and erasure coding factor (X).

Our design borrows ideas from Disk Paxos [7] to separate processing from storage. However, unlike Disk Paxos, our passive storage nodes store both the consensus log and a representation of the state machine, allowing a failed coordinator to be replaced with a different CPU node without requiring 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 allowing the coordinator to directly access the memory of memory nodes as though it was stored locally, we are able to greatly simplify the design of the coordinator In addition, by centralizing the 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, time is divided into terms, and a coordinator is elected when a candidate successfully acquires an exclusive lock on a majority of the memory nodes for the next term. Each candidate uses an RDMA compare-and-swap operation to write their identifier and their proposed term number 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 paper makes three main contributions:

- We present the design of an RDMA-based resource disaggregated consensus protocol.
- We leverage our replicated memory interface to implement erasure coding with limited additional complexity.
- We reduce the cost of deploying SMR in the cloud by sharing backup coordinators across consensus groups.

2 RELATED WORK

In this section, we provide an overview of SMR protocols and Remote Direct Memory Access (RDMA). We also discuss past work on resource disaggregation. Table 1 highlights the key differences between Sift and other relevant consensus protocols.

2.1 State Machine Replication

Most State Machine Replication (SMR) protocols such as Paxos [13], Raft [23], and DARE [24] are leader based, where a single leader is elected and the leader coordinates the operations of the protocol. In these protocols, the followers receive event requests from the leader, write the events to their local logs, update the log commit index, and monitors the status of the leader. In the event of a leader failure, the followers are responsible for detecting the failure and the followers must collectively elect a new leader. Because a follower may later become the leader, each follower must be provisioned the same amount of resources as the leader node. Thus, the resources of the followers may be heavily underutilized during normal operations.

EPaxos [21] and Mencius [19] are leaderless consensus protocols, meaning that every single node can service requests from clients. This means that every node must be provisioned with full memory and compute resources. Unlike leader-based protocols, these resources are not under-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.

2.2 RDMA

Remote Direct Memory Access (RDMA) [5] is an interface that allows servers to directly read and write the memory of a remote server. 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 interactions with the OS of either servers. 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 [24] and APUS [29] are RDMA based consensus protocols. In DARE, the consensus leader directly reads and writes to the logs of its followers. DARE's leader election protocol closely resembles traditional leader election protocols and requires direct communication between election candidates. APUS transparently optimizes the Multi-Paxos [14] 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.

2.3 Resource Disaggregation

Disk Paxos [7] proposes a consensus protocol where processes communicate by reading and writing to predefined regions on disks in a Storage Area Network environment. Similar to Paxos, $2F + 1$ acceptors (disks) are required to handle F faults. 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, resulting in different fault tolerance properties.

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 [4, 11, 15, 16, 25]. This design allows for fine-grained resource provisioning [1, 28] 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 [3, 31] 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.

2.4 Consensus as a Service

Filo [20] 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.

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 Section 4, uses this memory layer as though it was its own local memory. In this section, we describe the structure of CPU and memory nodes, how requests are processed, and the failure detection and recovery mechanisms.

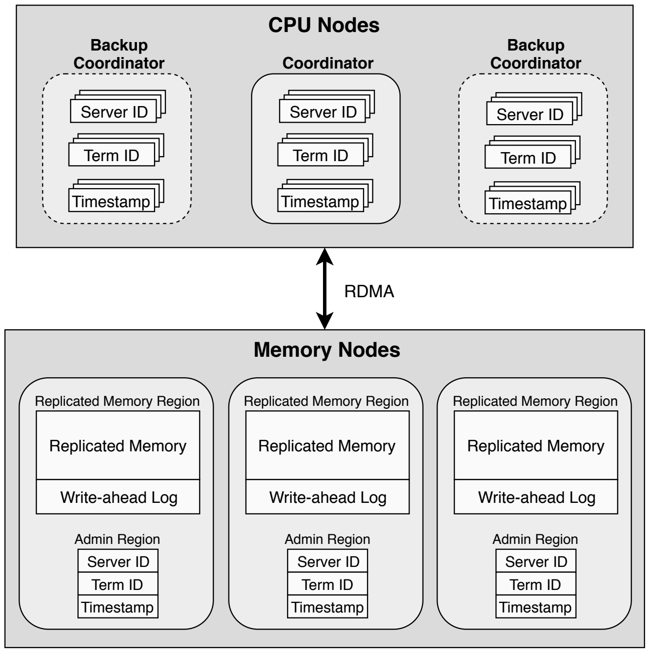

Figure 1: Sift architecture.

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 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, we can further reduce resource requirements by only instantiating backup CPU nodes if multiple consecutive client requests to the coordinator fail, trading off recovery time. Further optimizations are discussed in Section 5.2.

Similar to other consensus protocols, Sift divides time into terms of arbitrary length, and each term has a single coordinator. 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 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 is a memory block which holds term_id, node_id, and timestamp data, 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 of 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 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 [9] 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 [23] for electing a coordinator. The primary difference is that Sift's election does not involve any direct communication between election participants — the election is performed entirely through reads and writes to memory nodes.

The CPU nodes are initially in a follower state and a single node is elected as the coordinator. Each follower performs periodic heartbeat reads from the memory nodes. If a follower's *election timeout* period expires without a coordinator performing a heartbeat write,

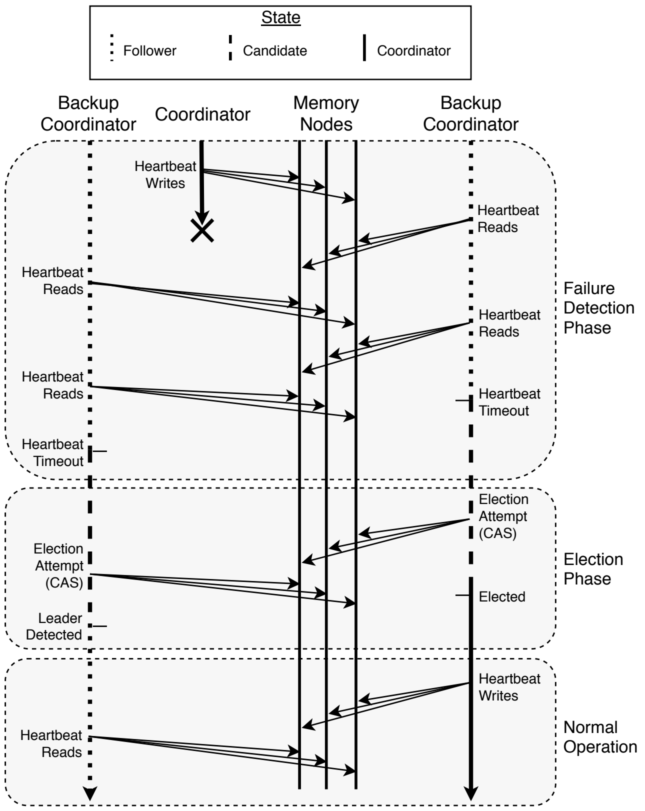

Figure 2: Example of a coordinator election scenario. Two backup CPU nodes compete to become the new coordinator once a coordinator failure is detected.

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 and term_id values 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. If the CAS operation succeeds on a majority of the memory nodes, the candidate becomes the coordinator. The remaining candidates see that another node has successfully written to a majority from their CAS operation's return value. 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.

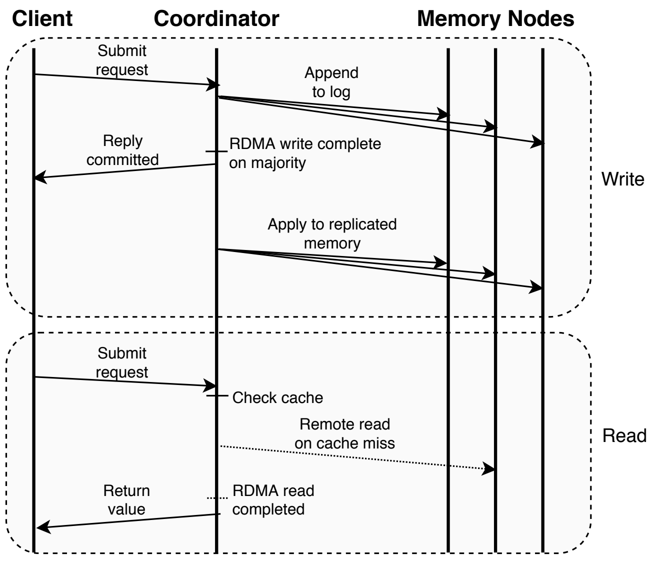

Figure 3: Messages exchanged for reads and writes.

Figure 2 shows an example 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 when a candidate has overwritten the `server_id` and `term_id` during an election, causing subsequent heartbeat 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 heartbeats 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, the memory node disconnects any previously held connections. 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 client requests are processed and how consistency is maintained. Figure 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, making it safe for the coordinator to reply to the client. 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 a replicated memory update has been submitted, to prevent stale reads.

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 conflicts and fault tolerance 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 [6]. 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 determine the circular log order. After replaying the log, all previously committed writes have been applied to the replicated memory. 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 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 data in that region. 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 gradually degrade write performance while leaving read performance unaffected. Once the entire replicated memory region has been copied over, all locks are released and the new memory node joins 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 [24]). Our approach prioritizes reducing memory requirements while still minimizing the performance impact of recovery. In addition, memory nodes that provide persistence (see Section 3.5) 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.

3.5 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 [10, 27], 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.

4 KEY-VALUE STORE

We build a recoverable key-value store on top of the replicated memory layer described in Section 3. We chose a key-value store as it is a common representation of an application's state machine. Other state machine representations can be implemented on top of our memory layer. 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 circular 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

4.2 Put/Get Requests

Put requests are processed by first 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 (no logging), which allows put requests to be committed in a single RDMA roundtrip. Once the log write is complete, meaning the update has been committed at the replicated memory, we reply to the client. Logged put requests are then applied in the background. Since our write-ahead log is circular, 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*. Updates to multiple keys can be applied concurrently through the 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. To ensure consistency, our cache tracks whether entries have been applied yet and does not evict entries which have pending updates.

4.3 Failure Handling

The key-value layer needs 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 loads the index table and bitmap from replicated memory. With these structures in memory, it reads and replays the contents of the write-ahead log. Once the log has been replayed, the process begins processing client requests.

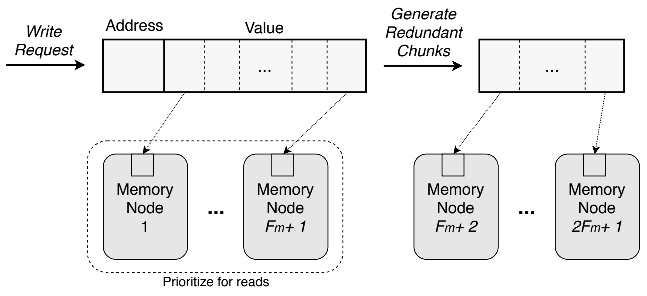

Figure 4: Replication process with erasure codes.

5 RESOURCE REDUCTION

5.1 Erasure Codes

Erasure coding has been used in other consensus protocols [22] to reduce their storage requirements. However, these protocols are more complex and provide weaker fault tolerance guarantees than standard consensus protocols. RS-Paxos, for example, requires more nodes to maintain fault tolerance guarantees; this has implications for cost as well as communication overhead. In Sift, erasure coding fits the architecture naturally, requiring minimal changes to the protocol while maintaining the same level of fault tolerance.

To support erasure coding in Sift, 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 [26] to generate F_m redundant chunks of size C. Figure 4 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 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 reads from a majority of nodes to rebuild a block, but still transfers only B bytes across the network. Note that any coupled architecture would also be subject to this, since reads could no longer be served from local memory. We can prioritize reading from memory nodes which store non-parity data to avoid the decoding cost. For memory node recovery, since nodes do not 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 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. By allocating only a single CPU node per Sift group to act as the coordinator, we can provision a pool of CPU nodes that monitor failures across, and can become coordinators for, all groups. Instead of requiring $(F + 1) \times G$ CPU nodes for G groups, we pool resources to maintain only $(G + B)$ CPU nodes, where B is the size of the backup pool. Note that for sufficiently large G, it will often be the case that $B << (F \times G)$ (see Section 6.4.2 for relevant analysis). This backup pool allows us to significantly reduce the $F + 1$ CPU node requirement of a single Sift group, which greatly 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.

The communication overhead of a backup CPU node being responsible for multiple groups is negligible since heartbeats are reads that rarely occur more frequently than every few milliseconds. However, 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.

6 EVALUATION

We compare the performance of Sift's key-value store with a custom, RDMA-based implementation of Raft [23] as well as EPaxos [21]. 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 RocksDB and erasure coding configurations discussed in Sections 3.5 and 5.1 respectively, have also been implemented. The system was written in C++ and consists of approximately 9,000 lines of code. We have open-sourced our implementation [12].

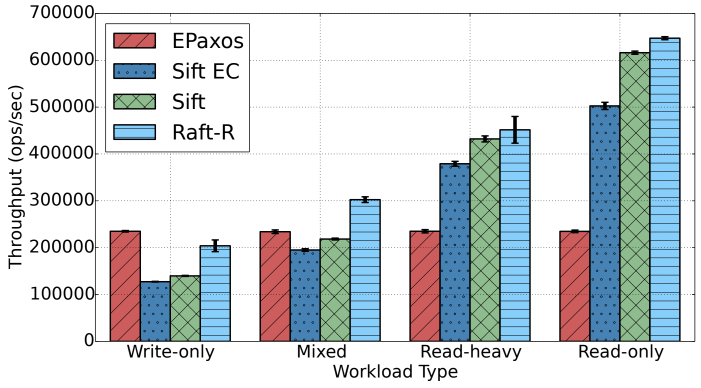

Figure 5: Performance comparison with Sift's key-value store and an RDMA-based Raft implementation.

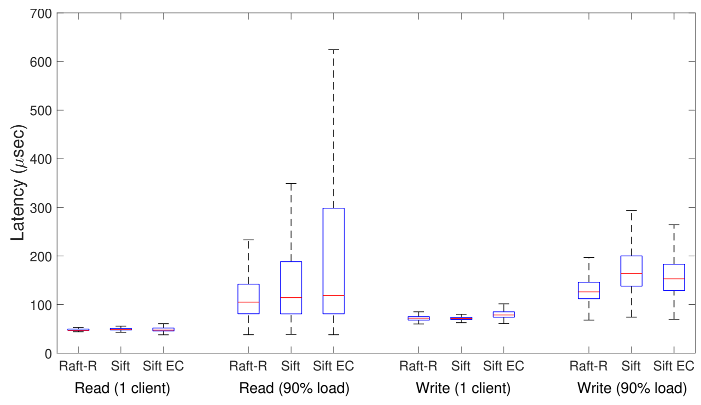

Figure 6: Latencies at low load (1 client) and 90% of peak throughput.

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 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 and are run on the same hardware. In Section 6.4 we analyze the more realistic setting where the systems are running on different hardware in order to meet a fixed performance target, which gives a clearer comparison of the deployment costs of each system.

6.3.1 Other Systems. 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 [17] on a RAM disk introduced bottlenecks in other parts of the systems resulting in an unfair comparison. We instead built a basic Raft-like system using RDMA send/recv verbs, which additionally allows us to compare both protocols using 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 [21], 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.

Although DARE [24] implements an RDMA-based consensus protocol, our hardware does not support RDMA multicast operations which are required to run the system. As a result, we omit performance results for DARE but note that its architecture is similar to that of Raft-R, with the main differences being its use of one-sided RDMA, RDMA multicast, and the addition of several low-level optimizations. Thus the performance of Raft-R serves as a performance lower bound for DARE. Similar to Raft-R, DARE is unable to benefit from Sift's resource-saving optimizations due to its coupled architecture.

Lastly, while Disk Paxos provides a resource-disaggregated protocol, it has different fault recovery properties compared to Sift, making a direct comparison unfair.

6.3.2 Throughput. Figure 5 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

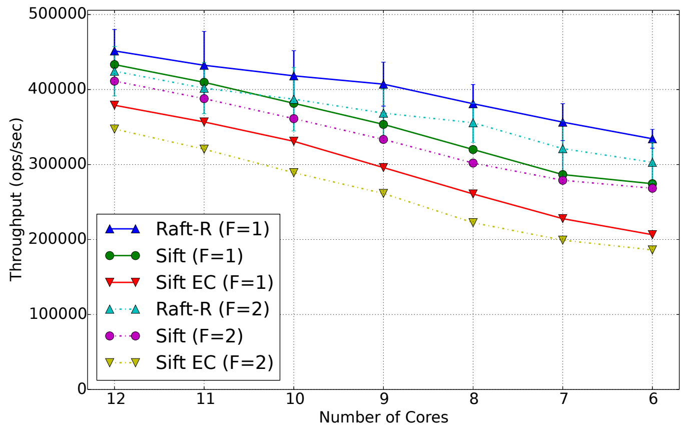

Figure 7: Performance of Sift and Raft-R with a varied number of cores for F = 1 and 2. These results show us how Raft nodes and Sift CPU nodes should be provisioned to achieve equivalent performance.

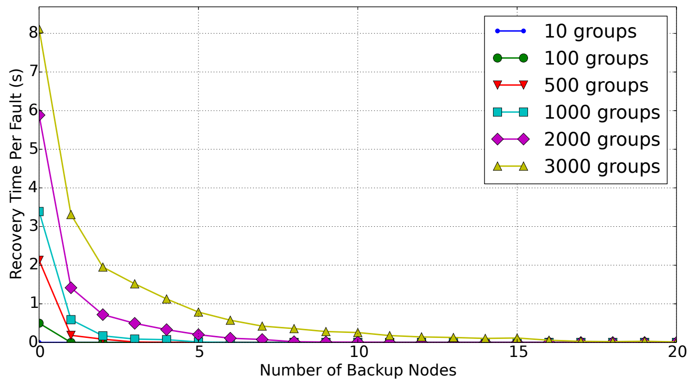

Figure 8: Results of a simulation over a Google cluster trace [30] of machine failures. Estimates how many backup nodes are needed to prevent additional recovery time due to VM provisioning.

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.

	F=1		F=2	
	CPU	MEM	CPU	MEM
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 2: Machine configurations for each system normalized for performance. CPU resources are measured in cores, memory resources are measured in GB.

6.3.3 Latency. Figure 6 compares Sift and Raft-R latencies 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, we omit EPaxos latency results.

At low load, the cost of writes is similar for all systems since they 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

Our performance results from previous sections show how each system performs given the same hardware. However, each system has varied resource requirements: a Sift deployment requires fewer compute resources because of passive followers, while a Raft deployment uses less memory overall due to leaders storing a replica of the state machine. Thus, to analyze the real-world cost of deploying each system, this section evaluates the cost needed to achieve a given throughput level. We use our experimental results and pricing information from cloud providers to determine these costs.

6.4.1 Normalized Performance. We extend the performance evaluation in Section 6.3.2 and consider deployments with varying CPU resources, shown in Figure 7. These results are used to determine how each system's machines should be provisioned to fairly evaluate costs.

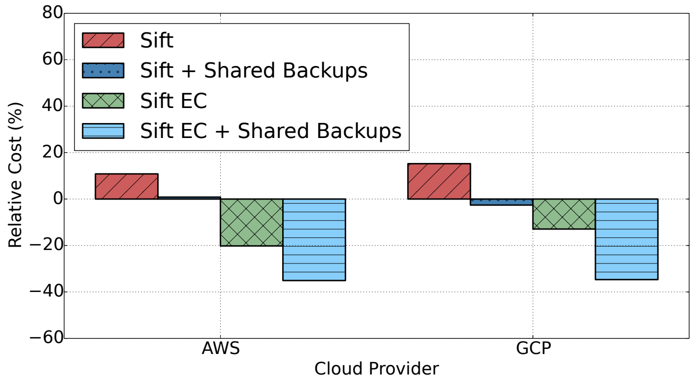

Figure 9: Costs of deploying Sift relative to the cost of Raft-R in AWS and GCP. Machines provisioned for equal performance with F=1.

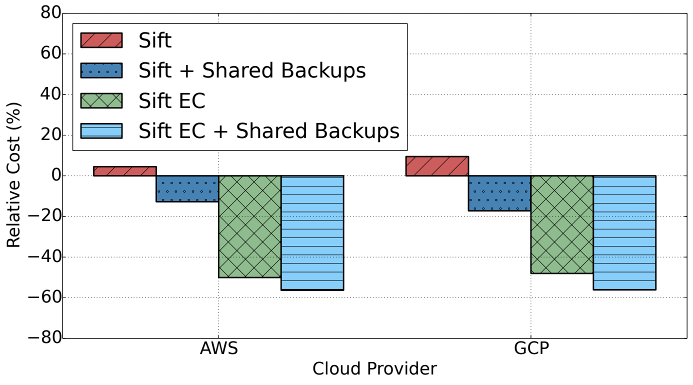

Figure 10: Costs of deploying Sift relative to the cost of Raft-R in AWS and GCP. Machines provisioned for equal performance with F=2.

6.4.2 Shared Backup Nodes. To better understand how backup nodes influence recovery time, we use a Google cluster trace [30] 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 [18].

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 8 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, with high probability, 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, there are currently limited options for RDMA-capable instances available from cloud providers. Instead, we use prices for non-RDMA instances. Given that there is no price premium for RDMA-capable hardware, we believe RDMA will come standard with most instances as the demand continues to increase. Our use of non-RDMA instance prices does not influence our results because we only compare RDMA-based protocols.

For our experiments, we use the currently available machine pricing from Amazon Web Services (AWS) [2] and Google Cloud Platform (GCP) [8]. While GCP provides the capability to provision machines with custom resources, we calculate the marginal costs of CPU and memory in AWS by comparing compute and memory-optimized instances. These pricing models give us a price of \$0.033/core/hr and \$0.00275/GB/hr for memory for AWS, and \$0.033/core/hr and \$0.00445/GB/hr for memory for GCP, which we use to provision custom machines for each system.

We use Sift and Raft-R deployments that achieve the same fault tolerance guarantees and performance. Using the results from Figure 7 to determine resource requirements, 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 2. 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.

Note that for Sift configurations with shared backup nodes, we assumed 100 Sift groups with a backup pool consisting of 2 CPU nodes. This backup pool size was taken from the results of the failure simulation in Section 6.4.2.

Figures 9 and 10 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 8) and finding sizes that result in no additional recovery time for each number of groups.

Sift costs decrease relatively 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.

We used the most common F values in our experiments, but note that the performance and cost trends would be similar as F grows. Sift's communication overhead would grow as it sends roughly twice the number of messages as a protocol such as Raft, but the reduction in compute resources being relative to F would likely overshadow this overhead when analyzing overall cost.

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. While not all applications can tolerate longer recovery times with erasure codes, or potential delays in coordinator recovery if too few backup nodes are provisioned, we believe the trade-offs are outweighed by the cost savings for many use-cases. Additionally, with increasing values of F, all Sift configurations improve relative to Raft-R despite Sift maintaining a large cache. This improvement is due to Sift's disaggregated architecture which enables independent scaling of compute and memory resources. This allows applications to use larger consensus groups for increased fault tolerance levels at a significantly reduced cost.

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 11 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 nearly worst-case effects instantly. If the workload was uniformly distributed, the throughput would 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 efficient 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 12. 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.

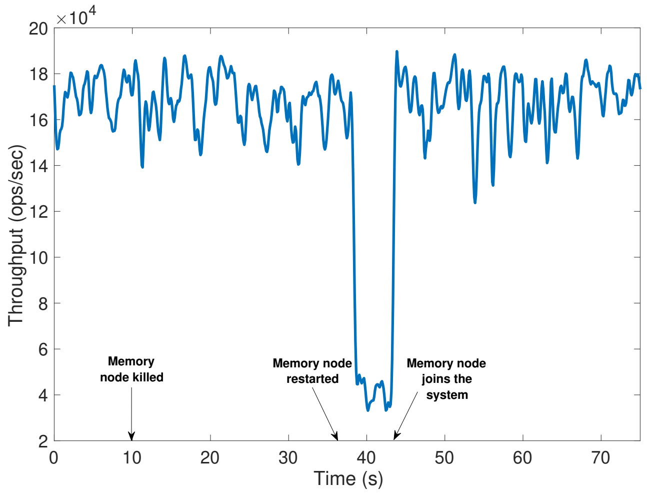

Figure 11: Read-heavy workload throughput during a memory node failure.

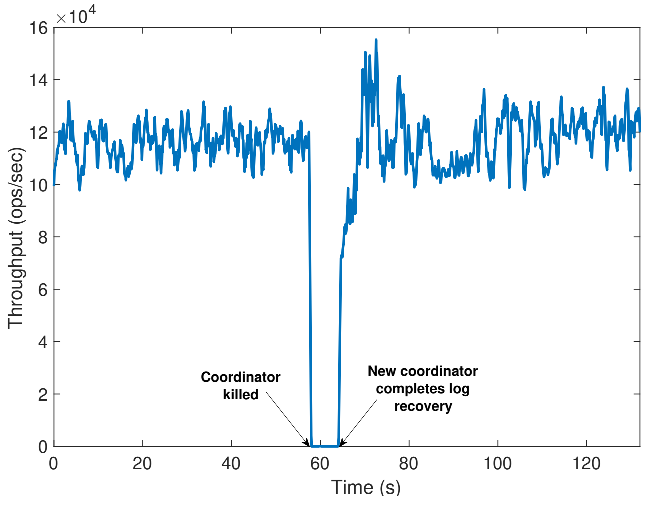

Figure 12: Read-heavy workload throughput during a coordinator failure.

In the experiment shown by Figure 12, 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 key-value 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.

At a high level, Sift's leader election mechanism is very similar to Raft's; the same number of messages are exchanged during an election. For the same experiment, Raft would require approximately the same amount of time to bring up a new coordinator as Sift. The majority of the recovery time in Figure 12 was spent recovering the key-value store's structures, which is largely independent of the consensus layer.

7 CONCLUSION

In this paper, we introduce Sift, a consensus protocol that uses one-sided RDMA to disaggregate compute and memory resources. 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. 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 from 35% to 56%.

These results indicate that while erasure codes alone can provide cost savings over a Raft-like protocol, especially with higher F values, the ability to share backup nodes across groups is where Sift sees the greatest benefit. For applications that make use of several consensus groups, or cloud providers looking to provide consensus-as-a-service, Sift significantly reduces resource consumption and ultimately provides a lower-cost solution for deploying consensus.

Acknowledgements: We would like to thank the anonymous reviewers for their feedback and Gianni Antichi for shepherding the paper. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Canada Foundation for Innovation and Ontario Research Fund.

REFERENCES

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

[2] aws 2019. Amazon AWS Pricing. https://aws.amazon.com/ec2/spot/pricing. Accessed October 2019.

[3] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright Cluster Services. In *ACM Symposium on Operating Systems Principles*. 277–290. http://doi.acm.org/10.1145/1629575.1629602

[4] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. 2015. R2C2: A Network Stack for Rack-scale Computers. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 551–564. http://doi.acm.org/10.1145/2829988.2787492

[5] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *USENIX Conference on Networked Systems Design and Implementation*. 401–414. http://dl.acm.org/citation.cfm?id=2616448.2616486

[6] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. http://doi.acm.org/10.1145/3149.214121

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

[8] gcp 2019. Google Cloud Platform Pricing. https://cloud.google.com/compute/vm-instance-pricing. Accessed October 2019.

[9] C. Gray and D. Cheriton. 1989. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. *SIGOPS Oper. Syst. Rev.* 23, 5 (Nov. 1989), 202–210. https://doi.org/10.1145/74851.74870

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

[11] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. 2013. Network Support for Resource Disaggregation in Next-generation Datacenters. In *ACM Workshop on Hot Topics in Networks*. Article 10, 7 pages. http://doi.acm.org/10.1145/2535771.2535778

[12] Mikhail Kazhamiaka. 2019. Sift. https://github.com/mkazhami/Sift. https://github.com/mkazhami/Sift

[13] Leslie Lamport. 1998. The Part-time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

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

[15] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *International Symposium on Computer Architecture*. 267–278. http://doi.acm.org/10.1145/1555754.1555789

[16] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level Implications of Disaggregated Memory. In *International Symposium on High-Performance Computer Architecture*. 1–12. http://dx.doi.org/10.1109/HPCA.2012.6168955

[17] logCabin 2017. LogCabin. https://github.com/logcabin/logcabin.

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

[19] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 369–384. http://dl.acm.org/citation.cfm?id=1855741.1855767

[20] Parisa Jalili Marandi, Christos Gkantsidis, Flavio Junqueira, and Dushyanth Narayanan. 2016. Filo: Consolidated Consensus as a Cloud Service. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 237–249. https://www.usenix.org/conference/atc16/technical-sessions/presentation/marandi

[21] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[22] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. 2014. 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*. ACM, 61–72.

[23] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*. 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

[24] Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *International Symposium on High-Performance Parallel and Distributed Computing*. 107–118. http://doi.acm.org/10.1145/2749246.2749267

[25] Kshitij Sudan, Saisanthosh Balakrishnan, Sean Lie, Min Xu, Dhiraj Mallick, Gary Lauterbach, and Rajeev Balasubramonian. 2013. A Novel System Architecture for Web Scale Applications Using Lightweight CPUs and Virtualized I/O. In *International Symposium on High Performance Computer Architecture*. 167–178. http://dx.doi.org/10.1109/HPCA.2013.6522316

[26] Christopher Taylor. 2015. cm256. https://github.com/catid/cm256. https://github.com/catid/cm256

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

[28] C. Wang, A. Gupta, and B. Urgaonkar. 2016. Fine-Grained Resource Scaling in a Public Cloud: A Tenant's Perspective. In *IEEE International Conference on Cloud Computing*. 124–131. https://doi.org/10.1109/CLOUD.2016.0026

[29] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: Fast and Scalable Paxos on RDMA. In *Symposium on Cloud Computing*. 94–107. http://doi.acm.org/10.1145/3127479.3128609

[30] John Wilkes. 2011. More Google Cluster Data. https://ai.googleblog.com/2011/11/more-google-cluster-data.html.

[31] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *ACM Symposium on Operating Systems Principles*. 253–267. http://doi.acm.org/10.1145/945445.945470