Fast and strongly-consistent per-item resilience in key-value stores

Konstantin Taranov, Gustavo Alonso, Torsten Hoefler
Systems Group, Department of Computer Science, ETH Zurich
{firstname.lastname}@inf.ethz.ch

ABSTRACT

In-memory key-value stores (KVSs) provide different forms of resilience through basic r-way replication and complex erasure codes such as Reed-Solomon. Each storage scheme exhibits different trade-offs in terms of reliability and resources used (memory, network load, latency, storage required, etc.). Unfortunately, most KVSs support only a single such storage scheme, forcing designers to employ different KVSs for different applications. To address this problem, we have designed a strongly consistent in-memory KVS, Ring, that empowers its users to set the level of resilience on a KV pair basis while still maintaining overall consistency and without compromising efficiency. At the heart of Ring lies a novel encoding scheme, Stretched Reed-Solomon coding, that combines hash key distributions of heterogeneous replication and erasure coding schemes. Ring utilizes RDMA to ensure low latencies and offload communication tasks. Its latency, bandwidth, and throughput are comparable to state-of-the-art systems that do not support changing resilience and, thus, have much higher memory overheads. We show use cases that demonstrate significant memory savings and discuss trade-offs between reliability, performance, and cost. Our work demonstrates how future applications that consciously manage resilience of KV pairs can reduce the overall operational cost and significantly improve the performance of KVS deployments.

CCS CONCEPTS

• **Information systems** → **Distributed storage**; • **Computer systems organization** → **Reliability**; • **Applied computing** → Enterprise data management;

KEYWORDS

Key-value store, Resilience management, Reed-Solomon, Replication

ACM Reference Format:
Konstantin Taranov, Gustavo Alonso, Torsten Hoefler. 2018. Fast and strongly-consistent per-item resilience in key-value stores. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3190508.3190536

1 MOTIVATION

Key-value stores (KVSs) were designed as an alternative to conventional database engines to bypass the cost of imposing a schema and the scalability limitations inherent in the transactional and relational models used in database engines. KVSs can achieve outstanding performance and scalability while providing resilience through different storage schemes. One serious downside of existing KVSs, however, is that the degree of resilience is typically fixed per engine and/or volume. Many features may affect the choice of a particular KVS, such as consistency, performance, reliability, or memory cost. These features are usually determined by the underlying storage schemes employed by the KVS [12]. As a result, each application needs to use its own KVS to match its requirements, leading to a proliferation of engines and a significant deployment and maintenance complexity in real settings. The following table illustrates the trade-offs between performance, reliability, and storage overheads for three schemes: Simple storage (no replication), three-fold replication, and a Reed-Solomon coding scheme.

Scheme	Reliability	Put Latency	Put Throughput	Storage Cost
Simple	None	1x	1x	1x
Rep(3)	2 failures	2x	0.5x	3x
RS(3,2)	2 failures	3.4x	0.31x	1.66x

Databases have addressed parts of this problem by offering different consistency guarantees at the SQL level, allowing each application to determine the degree of consistency it wants to achieve, while the engine still preserves the correctness of the data and mediates among all applications to provide strong consistency. In the cloud, several approaches have been proposed to classify data according to its importance and assign different levels of consistency to each one of them [16, 17, 34], showing the importance of having a flexible approach to deciding how much replication is needed for each data item. With the exception of some initial ideas from research [23], we are not aware of any KVS offering similar functionality.

In this paper, we explore the design and architecture of Ring, a KVS allowing users to set the level of resilience on a KV pair basis while still maintaining overall consistency and without compromising efficiency. An additional benefit of Ring is a more efficient use of resources such as memory, network traffic, and storage space.

Challenges. At a first glance, it might seem that implementing Ring requires nothing but just adding various storage schemes to a known KVS. However, this seemingly easy step entails subtle technical challenges. The first one is to ensure strong consistency across storage schemes, so that updating the storage scheme of a key remains consistent and atomic. By strong consistency we mean that updates occur atomically and requests need to be seen by all clients in the same order regardless of failures in the system. However, naively combining different KVSs makes key updates in different storages either slow or does not guarantee strong (sequential) consistency.

The second challenge is to guarantee the highest possible performance for finding keys with an unknown storage scheme. This goal requires minimizing communication during the lookup phase. Ring achieves that with a fully decentralized design. As opposed to traditional stores [4, 10], it does not rely on a central server to manage the location of key items. Instead, it relies on a novel allocation scheme that guarantees a stable key-to-server mapping regardless of the storage schemes that are used.

The last challenge is to support memory efficient erasure coding schemes while maintaining strong consistency. Ring utilizes a combination of replication and optimal erasure codes to allow users to choose the best reliability-overhead-performance trade-off for every single data item.

Overview of Ring. Ring's storage abstraction relies on *memgests*, which are different storage schemes with various resilience and performance properties. The name *memgest* is derived from the Latin term *gestus*, which can be roughly translated as *carry or bear*. Ring's memgests range from Reed-Solomon (*RS*) codes to flexible forms of full replication.

Conceptual Contributions. We design a new storage encoding scheme called *Stretched Reed-Solomon* (*SRS*) to maintain a stable key-to-node mapping regardless of resilience levels used. *SRS* redistributes the data chunks of optimal *RS* codes and simple replication schemes among nodes such that distinct interleaved layers of erasure coding and replication always share the same distribution for key hashes. In this way, Ring allows users to change storage schemes (memgests) transparently and independently from the client. *SRS* coding is our key tool to address all three challenges outlined above, and we show how to combine it with versioning to guarantee strong consistency.

Technical Contributions. In addition to the *SRS* scheme itself, we implement Ring as a fully-functional resilient high-performance in-memory KVS. We utilize Remote Direct Memory Access (RDMA) networking to provide low latencies and offload communication tasks from the CPUs. Our extensive experimental analysis shows that Ring reaches a low remote read latency of $5\mu s$ and an aggregate throughput of more than 1.5M put requests/sec for unreliably stored 1KiB KV pairs. For reliably stored 1KiB KV pairs, Ring achieves 800K put requests/sec for three-fold replication and more than 300K put requests/sec for $RS(3,2)$ Reed-Solomon coding.

In summary, *Ring empowers its users to explicitly manage storage schemes for each KV pair, in the same way that they would manage conventional system resources such as memory or processor time.* The key feature of Ring is that all keys live in the same *strongly consistent* namespace, and a user does not need to specify the storage scheme when looking up a key. Users can update a key's storage scheme during the key insertion or at arbitrary points during execution and still be strongly consistent.

2 RING APPLICATIONS

There are a wide range of scenarios where per-key management of the storage trade-offs provides a powerful abstraction enabling a more efficient utilization of expensive DRAM memory. To fully utilize the overall system, Ring users can flexibly change the storage scheme for each key at any time during operation. Changing the storage scheme influences both the network and server CPU loads, which determine the overall performance of the KVS, an aspect that we will study in the paper through four use cases.

Transparent multi-temperature data management. Data in warehouses is often classified according to its *temperature*. Frequently accessed data is considered *hot* and must be available at the highest performance. Rarely accessed *cold* data permits higher response times. Ring can transparently place hot data in high-performance replicated storage while keeping cold data in low-overhead erasure-coded storage using standard temperature-tracking schemes [13]. Ring's *SRS* storage scheme enables flexible temperature management by moving data from cold storage to hot storage fully transparently to users while ensuring strong consistency. It can lead to significant cost savings while maintaining the highest performance.

Heavy updates. KVSs often experience highly varying load over time. For example, items in online auctions or limited sales become very popular during the final stages of the sale. The last seconds of an auction are usually the ones of highest interest for a bidder and the system may receive millions of requests per second. A crash and the corresponding period of unavailability may be disastrous, even if the data is stored reliably. The designer of an online auction with heavy updates can use Ring to move items to high-performance, less reliable storage when a high workload is observed to increase throughput. Even if it seems counter-intuitive, the overall reliability is not reduced, for two reasons: First, the data is only stored less reliably for a short period of time, thus reducing the probability of data loss. Second, Ring supports *versioning*, allowing each key to have multiple versions in different storage schemes, preserving previous reliable copies of the data.

Importance of the data. The importance of data may change over time according to the intrinsic nature of the data. For instance, in iterative algorithms such as PageRank, the time to recover data increases as the computation progresses because losing data at a late stage requires expensive recomputation from the start. In other words, the intermediate page ranks at iteration $i + 1$ are more important than the ones at iteration i. In general, Ring can be useful for algorithms where the temporal data has to become persistent, since it dynamically increases the reliability of given KV pairs.

Temporary blob storage. Our last use case concerns typical cloud storage schemes providing write-commit or write-modify-commit patterns. Such patterns are typical in block blobs on Azure Storage and others, where users may upload blobs and after that decide on whether to store them persistently or not [19]. For example, many services for uploading pictures allow users to apply filters and then either commit or discard the changes. Blobs are deleted by the session management if they have not been committed within a predefined time. Objects should be stored in less reliable, high-performance memgests before a final decision is made on their persistence. This use case generalizes to other user-facing storage systems, with Ring providing a convenient interface to manage them.

3 STORAGE SCHEMES

We now explain how Stretched Reed-Solomon (*SRS*) codes are built and their advantages over classical *RS* codes. First, we briefly summarize key aspects of the replication (*Rep*) and Reed-Solomon (*RS*) schemes required to build *SRS* codes. In all our analyses, we assume a standard fail-stop failure model [30].

3.1 Replication

Replication is the simplest and most widely used approach for fault tolerance [36]. Numerous methods for data replication such as primary-backup replication [3], quorum-based replication [2, 14], and chain replication [35] exist. Primary r-fold replication provides simple reliability and availability, as any copy of the data can be read independently, but causes an $(r-1)$-fold increase in memory overhead. Strong consistency is often provided by a distinguished leader that is responsible for serving client requests. To commit a put request, the leader has to replicate the request to a majority of nodes. Therefore, we consider that availability and reliability of the r-fold quorum-based replication are guaranteed when less than or equal to $\left\lfloor \frac{r-1}{2} \right\rfloor$ nodes are faulty. Conversely, basic fully synchronous replication can tolerate $r-1$ failures, but the unavailability in case of failures is higher because of the synchronous communication with worker nodes.

3.2 Reed-Solomon coding

The alternative, (k, m) partial replication through erasure coding (e.g., Reed-Solomon), uses m additional parity blocks to secure k data blocks on different servers [26]. Maximum distance separable (MDS) erasure codes can tolerate up to m simultaneous failures in a group of $k+m$ blocks, which is the theoretically optimal storage overhead [26]. *RS* codes achieve the MDS property and provide a flexible choice of parameters k and m. The memory overhead is only proportional to the expected number of failures m, but requires accessing at least k data blocks during recovery. Thus, the performance of erasure coding is affected by faults, which always trigger recovery operations during which the data under recovery cannot be accessed. Therefore, systems using erasure codes are less available than ones using replication schemes in the presence of failures.

A common way to use *RS* schemes is to split data of size C into k blocks of size C/k. According to $RS(k, m)$ encoding scheme, m additional parity blocks are calculated from the k original data blocks. These blocks are stored on separate nodes and are grouped to form a stripe with k data blocks (denoted by $[D_1, ..., D_k]$) and m parity blocks (denoted by $[P_1, ..., P_m]$). We refer to nodes which store parity blocks as parity nodes, and nodes which store data blocks as data nodes. We also refer to data on data nodes as primary data, and data on parity nodes as parity data. This arrangement allows recovery from any combination of up to m simultaneous failures. The choice of the parameters k and m influences the fault tolerance, memory overhead, and recovery time. In the case of failures, the decoding operation reads any k out of the $k+m$ blocks to recover the lost blocks. When failures are frequent, the system performance degrades dramatically due to data recovery.

An *RS* encoding operation can be represented as a matrix-vector multiplication where the vector of k data blocks is multiplied by a particular matrix $H = \left[\begin{array}{c} I \\ \hdashline G \end{array}\right]$ of size $(k+m) \times k$, (see Eqn. (1)). Here, I is the identity matrix and G is called the generator matrix, and yields the MDS property [6].

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ g_{11} & g_{12} & g_{13} & \cdots & g_{1k} \\ g_{21} & g_{22} & g_{23} & \cdots & g_{2k} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ g_{m1} & g_{m2} & g_{m3} & \cdots & g_{mk} \end{bmatrix}}_{H:k+m\times k} \times \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \\ P_1 \\ P_2 \\ \vdots \\ P_m \end{bmatrix} \tag{1}$$

The matrix G can be constructed from a Vandermonde matrix ($g_{ij} = j^{i-1}$), where the elements are calculated according to Galois Field (GF) arithmetic [8]. In the GF with 2^n elements, where n is a positive integer, addition is equivalent to a bitwise XOR operation. Multiplying blocks by a scalar constant (such as the elements of H) is equivalent to multiplying each GF word component by that constant. The matrix G ensures the main property of the matrix H: any set of k rows of the matrix H is linearly independent. It means that the data can be recovered if at least k blocks out of $k+m$ data and parity blocks are available.

Recovery. Lost data blocks can be reconstructed by solving the reduced system of linear equations obtained by removing the rows corresponding to lost blocks in Eqn. (1). Reconstruction of a parity block is the same as an encoding operation, and involves multiplying the corresponding row of H by the data vector. Data block reconstruction takes two steps. The first step is to calculate a decoding matrix. The decoding matrix is built choosing any k linearly independent surviving rows of H, and then taking the inverse of them. The second step involves multiplication of the previously selected combination of data and parity blocks by the corresponding row of the decoding matrix to the missing block.

Update. If data on any server is updated, all corresponding parity blocks must be updated as well. Fortunately, it is not necessary to recalculate the whole data in parity blocks, as only recomputation of the concerned pieces of information is required. An update operation first calculates the difference between the old and the new data items, then replicates the update operation to parity nodes. Finally, at the parity nodes, the stored parity block is XORed with the update operation multiplied by a corresponding coefficient from the encoding matrix H.

3.3 Stretched Reed-Solomon coding

In this section, we introduce our Stretched Reed-Solomon (*SRS*) codes, which are based on *RS* codes. A common way of load balancing $RS(k, m)$ codes among k data nodes is to put an object with a key to data node $i = (h(key) \mod k)$, where $h(key)$ is a hash function. The major problem in this mapping, and other distribution schemes, is the coupling between the hash key distribution and the number of data blocks k. For instance, a storage system based on $RS(2, 1)$ has 2 primary data nodes, and thus has 2 key shards, whereas $RS(3, 1)$ includes 3 data nodes and 3 key shards. As a result, they cannot both be accessed with the same key-to-node mapping. Even worse, when the storage scheme is changed to a different k, the keys need to be remapped and migrated. Hence a new erasure code is needed to avoid key remapping when keys are moved across storage schemes.

The main idea behind *SRS* codes is to ensure the same key-to-node mapping for a range of *RS* codes with different k. This is a key feature of Ring that enables efficiently locating a node responsible for any key with the unified mapping, irrespective of the storage scheme (memgest) chosen for the data.

Derivation. *SRS* codes are defined by parameters k, m, and s. The parameters k and m are inherited from *RS* codes such that $SRS(k,m,s)$ codes apply a $RS(k,m)$ coding algorithm to the data. Yet, instead of storing k data blocks on k nodes, the blocks are spread or stretched over s machines ($s \geq k$). As a result, we have s data nodes and m parity nodes, but the data on them is encoded according to $RS(k,m)$. Note that $SRS(k,m,k)$ is identical to $RS(k,m)$.

Having $s \geq k$ data nodes for all *RS* codes enables them to share identical key-to-node mappings regardless of the erasure codes. For instance, if we stretch $RS(2,1)$ over 3 data nodes and obtain $SRS(2,1,3)$ as in Figure 1, then it will have the same number of data nodes as $RS(3,1)$. Hence, $SRS(2,1,3)$ and $RS(3,1)$ can share a key-to-node mapping, and their data nodes can be stored on the same physical machines. When resilience requirements for a key is updated from $SRS(2,1,3)$ to $RS(3,1)$, the key can be moved locally from one coding scheme to the other, since the masters of two schemes reside on the same physical node.

We build a family of $SRS(k,m,s)$ codes as follows:

(1) Encode data according to $RS(k,m)$ coding
(2) Compute the least common multiple of k and s.
$l = lcm(k,s)$
(3) Divide the original data into l blocks: $\left[\tilde{D}_1, ..., \tilde{D}_l\right]$.
(4) Distribute l data blocks over s data nodes such that each data node stores l/s blocks
(5) Parity blocks are stored on m nodes as in $RS(k,m)$

The encoding matrix H for $RS(k,m)$ from Eqn. (1) can be expanded to a stretched matrix H_{exp} of size $l + \frac{lm}{k} \times l$:

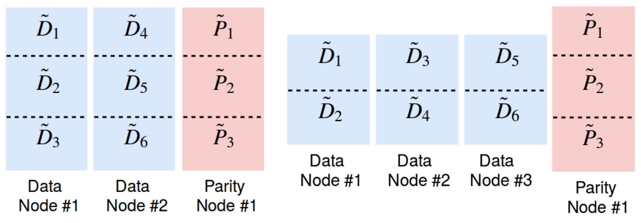
$$\underbrace{\begin{bmatrix} I_{\frac{l}{k}} & 0_{\frac{l}{k}} & 0_{\frac{l}{k}} & \dots & 0_{\frac{l}{k}} \\ 0_{\frac{l}{k}} & I_{\frac{l}{k}} & 0_{\frac{l}{k}} & \dots & 0_{\frac{l}{k}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0_{\frac{l}{k}} & 0_{\frac{l}{k}} & 0_{\frac{l}{k}} & \dots & I_{\frac{l}{k}} \\ g_{11}I_{\frac{l}{k}} & g_{12}I_{\frac{l}{k}} & g_{13}I_{\frac{l}{k}} & \dots & g_{1k}I_{\frac{l}{k}} \\ g_{21}I_{\frac{l}{k}} & g_{22}I_{\frac{l}{k}} & g_{23}I_{\frac{l}{k}} & \dots & g_{2k}I_{\frac{l}{k}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ g_{m1}I_{\frac{l}{k}} & g_{m2}I_{\frac{l}{k}} & g_{m3}I_{\frac{l}{k}} & \dots & g_{mk}I_{\frac{l}{k}} \end{bmatrix}}_{H_{\text{exp}}: l+\frac{lm}{k}\times l} \times \begin{bmatrix} \tilde{D}_1 \\ \tilde{D}_2 \\ \vdots \\ \tilde{D}_l \end{bmatrix} = \begin{bmatrix} \tilde{D}_1 \\ \tilde{D}_2 \\ \vdots \\ \tilde{D}_l \\ \tilde{P}_1 \\ \tilde{P}_2 \\ \vdots \\ \tilde{P}_{\frac{lm}{k}} \end{bmatrix} \qquad (2)$$

where each element is a matrix: I_n is the identity matrix of size n, and 0_n is the zero matrix of size n.

For $RS(k,m)$, H_{exp} is a coding matrix corresponding to encoding l data blocks using an $RS(k,m)$ code. It distributes l data blocks among k data nodes so that each node stores l/k data blocks. It also calculates lm/k parity blocks and distributes them among m parity nodes so that each node stores l/k parity blocks each. The matrices H and H_{exp} are equivalent in terms of data encoding, since they produce the same output. The matrix H_{exp} can also be calculated as entry-wise product of H and an expansion matrix E:

$$H_{\text{exp}} = H \circ E = E \circ H, \qquad (3)$$

where $E_{ij} = I_{\frac{l}{k}}$, H and E are of the same dimensions.



(a) $RS(2,1)$ **(b)** $SRS(2,1,3)$

Figure 1: Block distribution for $RS(2,1)$ and $SRS(2,1,3)$

According to classical Reed-Solomon the i-th data node D_i and j-th parity node P_j are comprised of the following blocks from Eqn. (2):

$$D_i = \left[\tilde{D}_{\frac{(i-1)l}{k}+1}, ..., \tilde{D}_{\frac{il}{k}}\right], \; P_j = \left[\tilde{P}_{\frac{(j-1)l}{k}+1}, ..., \tilde{P}_{\frac{jl}{k}}\right]$$

To obtain the *SRS* code, we reassign data blocks to s nodes instead of k. Thus, every data node is responsible for l/s instead of l/k chunks of data, whereas parity nodes are not involved in stretching and are kept the same. Therefore, nodes of $SRS(k,m,s)$ store data as follows:

$$D_i = \left[\tilde{D}_{\frac{(i-1)l}{s}+1}, ..., \tilde{D}_{\frac{il}{s}}\right], \; P_j = \left[\tilde{P}_{\frac{(j-1)l}{k}+1}, ..., \tilde{P}_{\frac{jl}{k}}\right]$$

Example of building SRS(2, 1, 3). In this paragraph we introduce an example of creating Stretched Reed-Solomon over three data nodes. The coding matrix H for $RS(2,1)$ is presented in Eqn. (5). The least common multiple of 2 and 3 is 6, so the data is divided into 6 blocks in order to spread it over 3 nodes. Afterwards, we expand the coding matrix of $RS(2,1)$ to 6 blocks by multiplying it by E according to Eqn. (3), where $E_{ij} = I_3$ (see Eqn. (5)). As a result, the data is encoded in Figure 1(a) as follows:

$$\tilde{P}_1 = \tilde{D}_1 \oplus \tilde{D}_4 \qquad \tilde{P}_2 = \tilde{D}_2 \oplus \tilde{D}_5 \qquad \tilde{P}_3 = \tilde{D}_3 \oplus \tilde{D}_6 \qquad (4)$$

In $RS(2,1)$ every server is responsible for $l/k = 6/2 = 3$ blocks as shown in Figure 1(a). In $SRS(2,1,3)$, we assign $l/s = 6/3 = 2$ data blocks for every data node (Figure 1(b)), and the data on them is encoded with the matrix from Eqn. (5) as in Eqn. (2).

$$H_{\text{exp}} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}}_{H} \circ \underbrace{\begin{bmatrix} I_3 & I_3 \\ I_3 & I_3 \\ I_3 & I_3 \end{bmatrix}}_{E} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \qquad (5)$$

Properties of Stretched Reed-Solomon codes. An important feature of $SRS(k,m,s)$ is that it preserves the coding properties of the original $RS(k,m)$. First, restoring a lost block still requires collecting k corresponding blocks over $s+m$ nodes, since data is still encoded according to $RS(k,m)$. Second, when a data server receives a put request, the request has to be propagated to m parity nodes. It however leads to memory imbalance as a parity server is responsible for more data than a data node. Furthermore, it can be observed that $SRS(k,m,s)$ can tolerate at least m and sometimes more simultaneous failures. For instance, $SRS(2,1,4)$, can tolerate

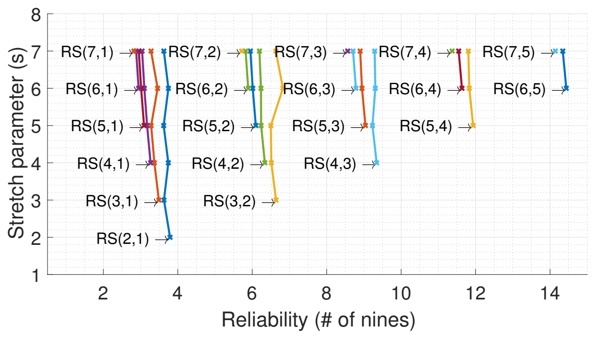


Figure 2: Reliability of Stretched Reed-Solomon coding with different parameters.

two simultaneous failures when two independent data servers are failed.

The main benefit of *SRS* codes is that we can combine many distinct storage schemes that fit the node layout, and access them with a unified key-to-node map. Keys can hence be transparently moved from one scheme to another, since all primary data of all schemes are present at each node. For example, with $s = 4$, Ring can support the following families of *SRS* coding schemes: $SRS(2, m, 4)$, $SRS(3, m, 4)$, and $SRS(4, m, 4)$, where m is an arbitrary integer greater than one. In practice, the number of parity nodes in *RS* schemes is bounded by the number of data nodes, i.e., $m < k$. Taking that into account, the total number of different erasure coded storage schemes with given s equals to $\frac{s(s-1)}{2}$. Furthermore, we can include replication schemes by partitioning them into s shards in order to have the same key hash distribution as $SRS(k, m, s)$. It enables, even for moderate s, a very large number of possible storage schemes (memgests) in a single KVS.

Reliability of SRS codes. In this paragraph we show that $RS(k, m)$ and $SRS(k, m, s)$ provide a comparable level of resilience. $SRS(k, m, s)$ distributes the data across more nodes and thus seems more liable to failures. However, the sparser distribution leads to a lower data loss after a node failure. We investigate the overall reliability using Markov models for reliability and availability. We explain the details in Appendix A and focus on the results here. Figure 2 indicates reliabilities of *RS* codes from which we derive stretched versions. The diagram shows a vertical line for each stretched code. The lowest point with the label is $RS(k, m) = SRS(k, m, k)$ and the connected points above are different stretching factors. The results show that stretching maintains approximately the same level of reliability. For example, the family of $SRS(3, 1, s)$ codes provides reliability around 3.5 nines for $s \in \{3, 4, 5, 6, 7\}$.

An interesting observation is that the reliability sometimes increases when the data is stretched. For instance, $SRS(3, 2, 6)$ is more reliable than $RS(3, 2)$. One reason for that is that each data node of a stretched version stores less data than a data node of the original one, which results in less data requiring recovery in the event of data node failure. For example, if the system stores 600GiB, in case of a data node failure $RS(3, 2)$ loses 200GiB and $SRS(3, 2, 6)$ loses only 100GiB. Faster recovery increases reliability because the system can tolerate more failures if the data has been recovered before the next failure. In addition, $SRS(k, m, s)$ is sometimes able to tolerate more than m simultaneous failures. It happens when failed data nodes store independent data blocks, still allowing the system of equations in Eqn. (2) to be solved without these nodes.

The availability behaves similarly and the results are presented in Appendix A.

4 SYSTEM ARCHITECTURE

After establishing the concepts of Stretched Reed-Solomon coding, we proceed to describe the architecture and the key components of Ring. We will discuss how Ring combines *SRS* coding to unify a flurry of *RS* coding and replication modes into a high-performance, fault-tolerant, strongly-consistent in-memory KVS.

5 API

Ring allows clients to access objects identified by a key through the three standard KV operations `get`, `put`, and `delete`. The `put(key, object, memgestID)` operation determines where the object should be placed based on the associated `key`, and writes the object to the memgest of Ring that determines the storage mode. Ring also supports the standard `put(key, object)` call which stores the data in a configurable default memgest. The `get(key)` operation locates the object master associated with the `key` in the storage system and returns the object. The `delete(key)` operation deletes the object associated with a given `key`. Ring also supports `move`, which provides additional flexibility in storing the data by allowing objects to be moved between memgests.

To manage memgests, clients can dynamically add and remove memgests from Ring. Clients can create a memgest by sending a `createMemgest(descriptor)` request. The `descriptor` contains parameters of the storage scheme. It adds additional flexibility to Ring, since users may tune the KVS by deploying different resilience levels. Memgests can be removed with the `deleteMemgest` command. Finally, the default storage scheme for new keys can be specified with `setDefaultMemgest(memgestID)`.

```
/* Conventional KVS requests */
object_t* get(const key_t)
int put(const key_t, const object_t*)
int delete(const key_t)
/* Resilience management */
int put(const key_t, const object_t*, const id_t)
int move(const key_t, const id_t)
/* Storage scheme management */
id_t createMemgest(const descriptor_t*)
int deleteMemgest(const id_t)
int setDefaultMemgest(const id_t)
descriptor_t* getMemgestDescriptor(const id_t)
```

5.1 Data layout, memgests

Ring's storage abstraction relies on memgests, which are different storage schemes with various resilience, overhead, and performance properties. Each memgest corresponds to either erasure coded $SRS(k, m, s)$ or replicated $Rep(r, s)$ schemes. An $SRS(k, m, s)$ memgest contains s data nodes that handle requests to data blocks and m parity nodes that receive update requests from the data nodes. A $Rep(r, s)$ memgest contains s data partitions which are replicated r times (Figure 3). Ring also supports memgests that are not fault-tolerant and have no additional storage overheads ($Rep(1, s)$). They

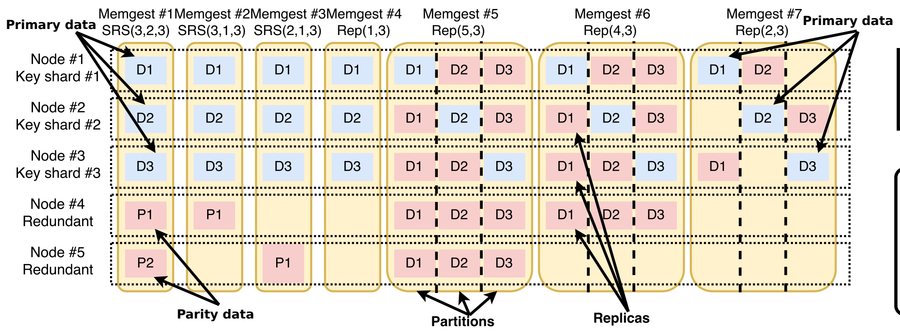


Figure 3: Various memgests for a 5 node system with 3 coordinators and 2 redundant nodes. The group provides 7 resilience levels: $SRS(3,2,3)$, $SRS(3,1,3)$, $SRS(2,1,3)$, $Rep(1,3)$, $Rep(5,3)$, $Rep(4,3)$, and $Rep(2,3)$. Note that stretching only affects the (blue) data blocks and parity or replica blocks can be allocated arbitrarily.

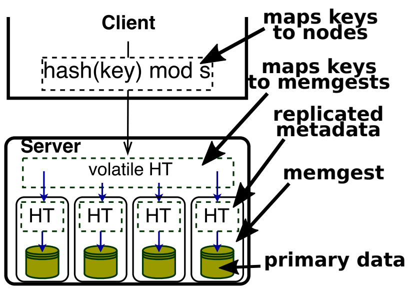


Figure 4: Request direction mechanism. The figure depicts only the server's coordinator-side data, i.e. without parity and replica related data.

can be used for storing temporal or recomputable data. The unreliable memgest offers highest update performance because it does not replicate `put` requests and can immediately acknowledge them.

A set of memgests which share the same number of key shards s constitutes a memgest group with s coordinator nodes and d redundant nodes (Figure 3). We refer to a server with an assigned shard as a coordinator node, since it coordinates the shard and all memgests sharing the shard. The parameter d is also an upper bound on the number of parities m in $SRS(k,m,s)$ memgest, and $s+d$ is an upper bound on the replication factor r in $Rep(r,s)$ memgest. Ring is configured for particular s and d parameters and does not allow having families with other parameters. However, our SRS codes allow transforming any storage scheme to have s data shards and build the memgest group from any number of unified storage schemes.

Ring has a distinguished leader, which is responsible for processing `createMemgest` requests. It decides how to distribute primary and parity data across nodes, and then inform other nodes about its decision.

We use a simple key-to-node mapping $i = (h(key) \mod s)$ to partition the whole key range into s shards, where each coordinator node handles one key shard within a memgest group. Thus, on a single node, *memgests share the same key shard*, but with different resilience and performance requirements (blue rectangles in Figure 3). As a result, our design allows `get` and `put` requests to a certain key to be performed by a single node only. The key thus suffices to directly retrieve data despite the fact that data can be stored in one of multiple memgests. Moreover, our storage design avoids the need for distributed transactions when the data moved across memgests since all required data is stored locally. In Ring every object has a version (see Section 5.2), which is incremented when the object is modified or moved across memgests. It helps to design a strongly consistent KVS, where only one instance of the key of a certain version exists across all memgests.

When a client sends a KVS request, it first applies $i = (h(key) \mod s)$ to map a key to node i, which is responsible for storing the key as in Figure 4. Afterwards, the request is performed locally at the requested node. For example, Figure 4 depicts a data server that supports 4 different resilience levels. When a server receives a `get` request, it first looks the requested key up from a volatile hashtable, which maps the key to the list of pairs $\langle version, memgestID\rangle$. Each coordinator node only has keys in its volatile hashtable that are mapped to it by $(h(key) \mod s)$ mapping. The volatile hashtable is used to quickly retrieve the *memgestID* of the memgest that stores the highest *version* of the object. Then the requested object is looked up from that memgest using $\langle key, version\rangle$ pair.

Querying of a memgest is done through its metadata hashtable. The metadata hashtable is a part of each memgest, and is replicated to survive failures (except the unreliable memgest). In contrast, the volatile hashtable that solely acts as an interface to the memgests is not replicated at all. It can be reconstructed by combining metadata hashtables of all local memgests. During normal operations the volatile hashtable is kept consistent with the memgests' hashtables.

5.2 Strong consistency

Strong consistency requires Ring to employ a range of techniques to ensure existence of only one instance of a key of a certain version across all memgests. First of all, the volatile hashtable and all metadata hashtables are write-ahead on the master node, that is all modifications are written to them before they are committed. The write-ahead approach does not violate consistency since, in case of a data node failure, only committed entries will be recovered. We also postpone requests that read uncommitted objects. Second, Ring exploits *versioning* and increments the key version when a key migrates across memgests. Ring retrieves the highest version of the key (even uncommitted) and continues writing with a higher version. Old versions are removed from the system periodically. It can be tuned to trigger removing of old versions of a key after every committed put request to it. It is the main mechanism for strong consistency in case of failures. It prevents having two distinct copies of the key with the same version number after failures, which would lead to an inconsistent state.

Write-ahead and versioning approaches allow serving requests to distinct memgests independently without waiting. For instance, if multiple clients want to put new values to the same key simultaneously, then their requests can be served together and independently from each other. An interesting case is when two clients put values to the same key but to different resilience levels. Since Ring allows committing the different versions of keys independently, then higher

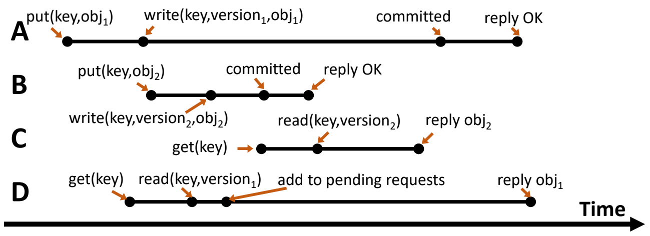


Figure 5: Put/get scenario with multiple clients.

versions may be committed earlier than lower versions. It can happen when one client puts to a fast storage scheme, while another client is writing the same key to slower storage scheme. The diagram below illustrates this case for clients **A** and **B**. It also shows that client **C** gets the highest value committed by **B** regardless the success of client **A**'s request. In other words, the highest version depends on the last writer only.

A request becomes committed when it is replicated within the requested memgest; hence puts to unreliable memgests are always committed immediately. Each memgest has a special replicated log to propagate updates generated from client requests within itself. To provide strong consistency, it is allowed to get data from committed entries only. Thus, the response to the client will be postponed until the requested entry is committed, as with client **D** in Figure 5. When the request from **D** was received, the highest version of the requested key was 1. Ring read the uncommitted version of the object and prepared a reply to the client **D**. The reply was sent once the version 1 had become committed. To ensure the proper order, the metadata hashtables store a list of pending get requests and *committed* flag for each object.

$$key, version \rightarrow data, length, \underbrace{committed, requests}_{volatile}$$

This information is volatile and can be lost in case of failures. When an entry becomes committed, the flag is flipped and all pending replies are sent. It ensures that a get request returns the version (even uncommitted at that moment) that was the highest when the request was received by Ring. Ring handles `move(key, memgestID)` requests similarly because the object has to be moved from the memgest with the highest version. Therefore, the `move` request will also be postponed if the requested object is not durable.

When clients send a `move(key, memgestID)` request, the key has to be moved from one memgest and written to another atomically, so partial updates have to be prevented. Here we benefit from a feature of Ring's design: the coordinator of memgests which are responsible for the same key is situated on a single physical machine due to the *SRS* coding. It allows us to avoid expensive locking and distributed transactions. In case of failures, Ring recovers all recoverable versions of keys from multiple memgests. Ultimately, several resilience requirements can be satisfied at almost no cost, since all storage management can be performed locally.

5.3 Erasure coded and replicated memgests

All memgests share a similar structure: they have a replicated log to replicate updates, replicated metadata hashtable, and the actual data, which is stored separately and treated according to its storage scheme.

Separating metadata from actual data provides a wide range of advantages. Firstly, the metadata suffices to serve `delete` requests. Secondly, it allows memgests to recover in two steps: metadata recovery, data recovery. What is more, data recovery can be postponed and only recovered on demand which is quite important for expensive erasure codes. Zhang et al. [39] proposed a similar approach for erasure codes. They replicated metadata of keys with a primary-backup replication scheme while actual values are erasure coded.

The `put` operation induces an update of the data of a memgest on the coordinator and redundant nodes. The coordinator thus replicates requests within the memgest. Coordinators of erasure coded memgests generate special parity updates and replicate them to m parity nodes to commit the operation, whereas the special processing of the request is not required for replication scheme, and can be replicated immediately. For replication, we implement quorum-based replication [27], where requests are replicated on the majority of nodes within a memgest to ensure their durability. The remaining nodes in replication scheme are updated asynchronously. Once the entry is properly replicated to redundant nodes, the put is committed, and the coordinator replies to the client with an acknowledgment.

Get requests do not alter the data, so they should not be replicated. To ensure that gets do not return stale data, the coordinator node has to periodically verify its role in the system by reading the configuration from a replicated state machine [27].

5.4 Balancing

One issue caused by our design is that nodes within a memgest group occupy different amounts of main memory (see Figure 3). Unfilled rectangles represent unbalance in memory usage. It is mainly caused by the design of *SRS* memgests, where data is stretched over many servers. Additionally, every parity node stores metadata of all data nodes from the same *SRS* coding stripe. Therefore, a parity node stores more metadata than a data node. In our system, parity nodes are also responsible for recovering lost data and parity blocks and therefore require more memory to store blocks under recovery and recovery metadata. Regarding workload, as parity nodes only need to participate in put operations, they may become idle for get-mostly workloads. In contrast, for put-mostly workloads, the parity nodes may become busy and may become a bottleneck of the KVS. Finally, during node failures, parity nodes are overloaded by data recovering processes.

To resolve these issues, we can create many memgest groups and assign them round-robin to fixed nodes, as has been done in [39]. It requires creating $s + d$ memgest groups, where s is the number of shards and d is the number of redundant nodes, since there are $s + d$ different ways of rotating the single memgest group. As we mentioned before, we use sharding to distribute keys among coordinators. Thus to support multiple memgest groups we also partition shards into $s+d$ memgest groups. It allows balancing workload and memory on each node. The shards on one node belong to different memgest groups, therefore a single node failure leads to a data loss on each memgest. However, since parity nodes are spread evenly across machines, the recovery workload also will be uniformly balanced.

5.5 Membership and handling failures

To deploy Ring, at least $s + d$ nodes are required, where s and d are coordinators and redundant nodes, respectively. Nonetheless, the overall system is comprised of $s + d + n$ machines, where n stands for *spare* nodes as in Figure 6. Thus, there are two types of nodes in the system: some that take part in serving requests and some that do not. Spare nodes are always ready to replace a failed node and immediately handle the requests for the node. The system has a leader, which is responsible for membership of nodes and, in case of failures, for reassigning the roles of failed nodes to healthy spare ones. The leader is elected according to a leader election protocol [27].

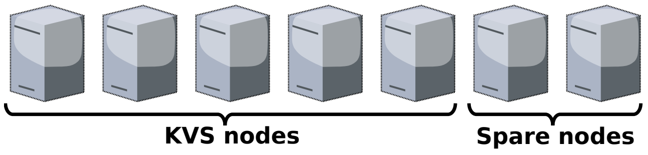


Figure 6: The roles inside the cluster. The system consists of spare nodes and KVS nodes.

While in spare mode, spare nodes incur minimal memory and CPU load. They use memory only for sending and receiving heartbeats to check membership and the log that replicates requests for changing roles. Once the leader recognizes that a node crashed, it replicates an entry over the log, which consists of the new responsibilities for all of the nodes. Therefore, all servers know about all changes in the system.

To change the role and start processing requests, a spare node reads all necessary metadata from alive nodes. Once the metadata on the node is recovered and the volatile hashtable built, it starts providing services while performing data recovery in the background. If the requested data is lost, it will be recovered with an on the fly recovery algorithm with high priority. For replicated memgest, it will request a copy of the requested data from any available replica. For erasure coded memgest, it will start an online decoding algorithm similar to one introduced in [39]. Data node sends a recovery request to the parity node responsible for the lost block. Then the parity node starts block recovery by collecting k available corresponding blocks from a coding stripe according to $RS(k, m)$ and decoding them. Finally, the parity node sends the recovered block to the data node initiated the recovery.

Clients access the data by sending requests using a hash function to determine the required node. If a data node has failed and a request is not answered in a predefined period of time, clients re-send the request through multicast. The request will be serviced only by the node that is responsible for the requested key. The clients will then communicate with the new data node directly.

6 EVALUATION

In this section, we evaluate the performance of replication and erasure coding approaches on RDMA networks. We use a 12-node InfiniBand cluster: each node has an Intel E5-2609 CPU clocked at 2.40GHz and WDC WD5003ABYX-01WERA1 internal hard drives. The cluster is connected with a single switch using a single Mellanox QDR NIC (MT27500) at each node. The nodes are running Linux, kernel version 3.18.14. Replication and erasure coding are implemented in C and rely on the following libraries: libibverbs, an implementation of the RDMA verbs for InfiniBand, and libev, a high-performance event loop; Jerasure [24], a library in C that supports erasure coding in storage applications; and GF-Complete [25], a library for Galois Field arithmetic. Each server is single-threaded, but can be potentially multi-threaded, e.g., by partitioning keys and assigning threads to partitions.

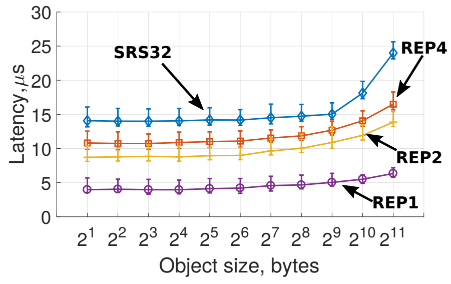
Remote direct memory access (RDMA) is an interface that enables direct access to memory located in the user-space of remote machines on a cluster computer [29]. RDMA allows global computations without the involvement of CPUs of the machines owning the memory. It enables those machines to continue unobstructed and improves overall system performance. The remote access is performed entirely by the hardware (using a reliable transport channel). An advantage of implementing erasure coding over RDMA is that CPUs on redundant nodes are not involved in receiving messages. They can perform other operations, such as data recovery and coding.

6.1 Latency

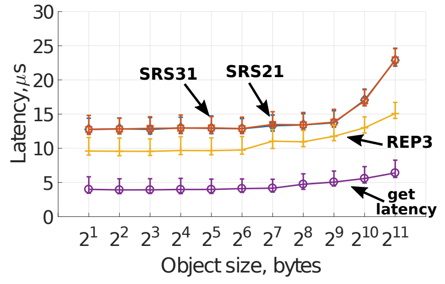
Ring is designed as a low latency KVS. Figure 7 shows the latency of put and get requests (split into two figures for readability). We deployed Ring on 5 nodes with 7 memgests: $SRS(3, 2, 3)$, $SRS(3, 1, 3)$, $SRS(2, 1, 3)$, $Rep(4, 3)$, $Rep(3, 3)$, $Rep(2, 3)$, and unreliable $Rep(1, 3)$. Since they all share the parameter s, which is equal to 3, we label them on graphs as SRS32, SRS31, SRS21, REP4, REP3, REP2, and REP1, respectively. In the benchmark, a single client gets and puts objects of varying size to/from the system. Each measurement is repeated 5,000 times, the figure reports the median and the 90th percentile. According to our implementation the get latencies of all memgests are the same, therefore we plot only one line in the figure for all studied schemes. It can be explained by all memgests using the same algorithm for retrieving data: They first ensure that they are allowed to reply to get requests by periodically checking the current node configuration, then respond to the client.

The main difference lies in put requests: The latency for $SRS(2, 1, 3)$ is the same as for $SRS(3, 1, 3)$, regardless of whether they share the same number of coordinator nodes s in storage scheme. The reason is that they have to replicate update to one parity node only. $SRS(3, 2, 3)$ has the highest latency among the system because the data nodes are responsible for calculating updates and replicating them to two parity nodes, whereas other storage schemes are less compute and network intensive. The rationale is that an unreliable memgest writes directly to main memory, but erasure coded memgests have to read memory first to apply XOR operations to the data to build special updates and replicate them to all parity nodes. The size of the parity update is larger than the actual request, since the metadata must be replicated along with the update. Therefore, writing to replicated memgests is faster than to erasure coded ones, and the lowest put latency can be observed for the unreliable memgest $Rep(1, 3)$.

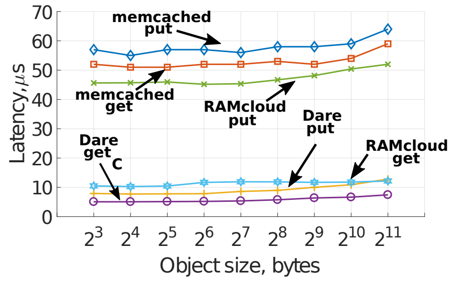
We compare Ring with several state-of-the-art KVSs: the single-threaded caching KVS memcached [9], erasure coded Cocytus KVS [39]; a strongly-consistent RDMA KVS Dare with in-memory replication [27]; and a strongly-consistent RDMA KVS RAMCloud with disk-backed replication [22]. We were not able to reproduce experiments for Cocytus, hence we used data from their paper [39],



(a) Put latency of SRS32, REP4, REP2, and REP1.

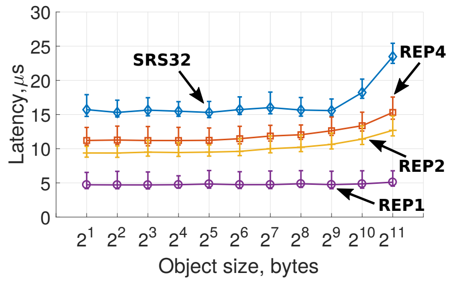


(b) Put latency of SRS31, SRS21, REP3, and get.

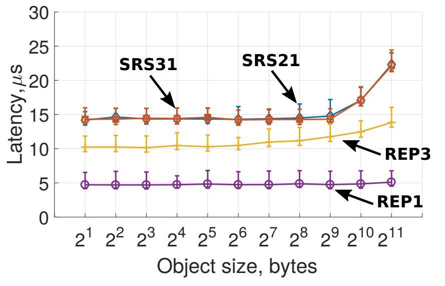


(c) Put and get latency baselines.

Figure 7: Latency of put and get requests for Ring and other systems.



(a) Move latency to SRS32, REP4, REP2, and REP1.



(b) Move latency to SRS31, SRS21, REP3, and REP1.

Figure 8: Latency of moves.

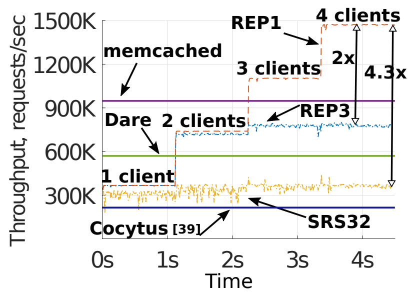


Figure 9: Put throughputs of memgests with 1KiB value size.

evaluated on the cluster hardware comparable to ours. Both CPUs belong to Intel Xeon E5 family and have the same characteristics except the number of cores. However, both KVSs are single-threaded and it should not influence overall performance. Networks bandwidths are different: they used 10Gb/s NICs whereas our cluster is armed with 40Gb/s NICs. However, it should not influence the evaluations either, since *RS* codes are compute-bound rather than network-bound. Cocytus achieves throughput of 2.4 Gb/s, which is less than 10Gb/s.

Memcached does not utilize RDMA to communicate with clients, therefore it provides higher get and put latencies at about 55μs which is 10x higher than the $REP1$ memgest. Cocytus KVS with $RS(3, 2)$ coding scheme for 1KiB values has get latency 500μs which is 100x slower than Ring implementation. Also Ring put latencies are 30x lower than Cocytus' ones for 1KiB objects for the same coding scheme. Dare KVS with replication factor 3 provides the same get latency as Ring, which is not a surprise since they utilize RDMA for communication with clients. Ring also provides approximately the same put latency as Dare for comparable $REP3$ memgest. Another baseline is RAMCloud with 2 backup stores, which provides median 45 μs latency of putting objects up to 512 bytes using an unloaded server with a single client. The high latency is resulting from the fact that our cluster equipped with HDDs instead of SSDs. RAMCloud replicates a put request 2 times, therefore it corresponds to our $REP3$ scheme. RAMCloud has lower main memory storage overhead since it flushes data to disk on backup nodes, whereas Ring stores all data in main memory. Hence, Ring has lower put latency, and also is less subject to tail latency, which is typical for disk-backed systems. Potentially, Ring can also support disk-backed replication and still be strongly consistent.

6.2 Move requests

In Figure 8, we can see the results of our benchmarks regarding move operations. We split the data into two subfigures to improve readability. The graph shows only destination storage schemes because the source storage scheme does not impact performance due to the local availability of the data. An interesting observation is that moving the object to unreliable scheme $REP(1, 3)$ has about the same latency for all object sizes. The reason is that the client does not send the object again and it is copied from high-bandwidth main memory. We can observe a similar pattern for other schemes since the time of moving the object from the unreliable memgest to the reliable is lower than putting directly to the second one. However, put operations are still dominated by building update requests and replicating them.

Benefits of using move requests. Move requests enable explicit resilience management and have a range of use cases discussed in Section 2. Next we outline storage benefits of employing move requests using the example of utilizing the unreliable memgest for the ☁ *Blob storage*. Let us estimate the footprint of an object in memory before it is committed. By commit operation we mean the decision of storing the object persistently. We denote the time between the first write and commit operation as τ. The memory footprint of the object that was written to reliable storage is equal to the size of object S multiplied by storage overhead O and by the duration of time it was stored: $S \cdot O \cdot \tau$. When we write first

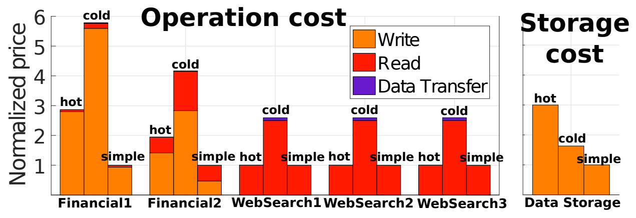


Figure 10: Storage Pricing for different I/O traces.

to the memgest without additional overhead, the footprint is just $S \cdot \tau$. We can thus achieve a relative *memory reduction* of $\frac{1}{O}$ using the unreliable storage scheme. The cost of this significant memory reduction is a single move request, about $5\mu s$.

The unreliable memgest $REP(1,s)$ has also the lowest put latency among schemes under study. Its put latency is 3x times lower than latency of $SRS(3,2,3)$ and 2x lower than latency of $Rep(3,s)$ (see Figure 7). Therefore, Ring can provide significant *latency reduction* by employing its move requests and also reduce the load in the backbone network by decreasing the number of replications required to commit an update. Ring can also store a backup version of a key to withstand failures. This is beneficial in the *dynamic importance* use case.

The next advantage is that the unreliable $REP(1,s)$ memgest has the highest throughput among all coding schemes (see Figure 9). We can thus achieve a *considerable speedup in throughput* by moving objects to $Rep(1,3)$ memgest and performing all put requests there. It corresponds to the *Heavy updates* use case.

Real-world applications show different access patterns: some applications are put-heavy, while others are get-heavy. To demonstrate the influence of storage scheme choice on real-world workloads, we estimated the price of operations from five traces obtained from the Storage Performance Council [33] for three storage schemes: $Rep(3)$ (hot), $SRS(3,2,3)$ (cold), and $Rep(1)$ (simple). The first two traces represent put-heavy OLTP applications running at a large financial institution. The remaining three are get dominant I/O traces from a popular search engine. Operation and storage costs for hot and cold schemes are obtained from Azure Blob Storage Pricing for Central US [18]. Azure Blob Storage does not provide a simple storage scheme, thus its price is assumed to be the same as for $Rep(3)$, but with 3x cheaper puts, as they are not replicated.

Figure 10 shows estimated prices for storing data at a constant capacity and performing traces with hot, cold, and simple storage schemes. The estimated costs are normalized and represent the price relative to no replication. As we can see the choice of storage schemes influences the price dramatically, depending on access pattern of traces and the volume of stored data. For example, cold storage is 5.5x more expensive than simple storage and 2x more than hot storage for the **Financial1** trace.

Ring enables *multi-temperature data management* by moving data across storage schemes. It can significantly reduce financial expenses compared to a KVS with a single storage scheme.

6.3 Throughput

Figure 9 shows throughput traces for $SRS(3,2,3)$, $Rep(1,3)$, and $Rep(3,3)$. In these experiments clients send requests to Ring with the same requests rates of 400K requests/sec. The length of the key is 8B, and value size is 1KiB. Every second a new client is created, which starts sending requests to Ring. Ring achieves put throughput of almost 1.5M requests/sec under the load of 4 clients for 1KiB objects with $Rep(1,3)$. $Rep(3,3)$ processes requests 2x times slower, and $SRS(3,2,3)$ 4.3x slower than $Rep(1,3)$. We also compare Ring throughput with throughputs of memcached, Dare, and Cocytus. According to our experiments, comparable memgests achieve higher throughput than memcached, Dare, and Cocytus.

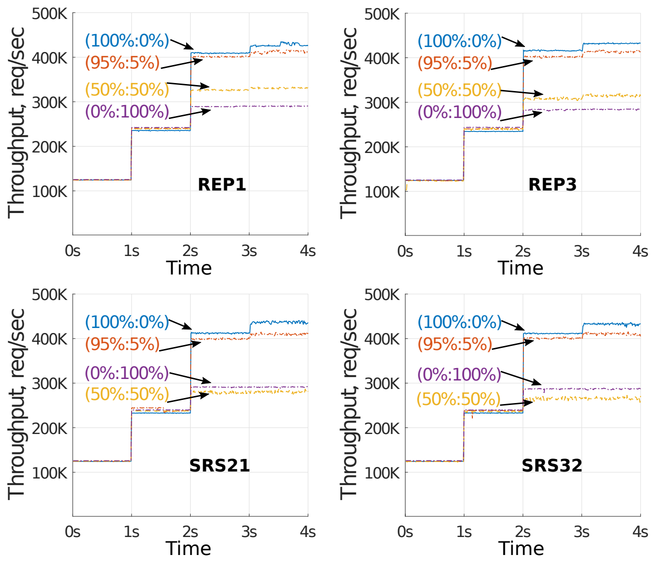


Figure 11: Single client throughputs of memgests under different (get:put) ratios workloads with 1KiB value size.

We also use the YCSB [7] benchmark to generate our workloads for Figure 11. The distribution of the key probability is Zipfian [7], with which some keys are hot and some keys are cold. The length of the key is 8B, and value size is 1KiB. We evaluate the systems with different (get:put) ratios, including equal-shares (50%:50%), get-mostly (95%:5%) and get-only (100%:0%).

Figure 11 shows traces for $SRS(2,1,3)$, $SRS(3,2,3)$, $Rep(1,3)$, and $Rep(3,3)$. In these traces a single client sends requests to Ring with different request rates. Every second the client doubles its request rate from 128K requests/sec until it reaches 1024K requests/sec.

In all experiments the requests were served by 3 coordinator nodes, and the client accesses them in order. As noted earlier, all memgests share the same implementation of how get requests are served, and therefore exhibit the same get throughput of 418K requests/sec. This number drops as the workload's put ratio is increased. Since our implementation is single threaded, we have not noticed a significant difference between different storage schemes. A small drop in throughput of erasure coded schemes for (50%:50%) workloads is due to differences in memory allocation algorithms for replicated and erasure coded memgests.

Figure 11 shows that the highest put throughput is achieved by the unreliable memgest $Rep(1,3)$ at 290K requests/sec. Other schemes achieve a slightly lower throughput of 280K requests/sec, despite the fact that they have to replicate requests. This is due to two effects: the replicated memgests employing quorum-based replication; and

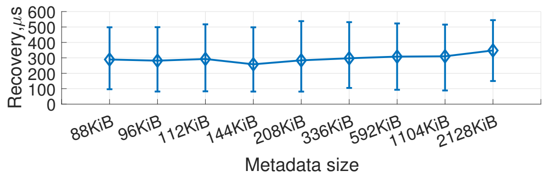


Figure 12: Recovery latencies depending on metadata size.

the current implementation being single-threaded. We believe that a multi-threaded implementation of memgests can achieve higher performance. Finally, Ring's throughput of $SRS(3,2,3)$ memgest is 1.5x-2x faster than a comparable Cocytus configuration, which achieves approximately 220K requests/sec for (100%:0%), (95%:5%), and (50%:50%) workloads [39].

6.4 Failures and recovery

We evaluate the recovery efficiency of Ring depending on recovered metadata size (see Figure 12). Each measurement is repeated 500 times, and the figure reports the median and the 90th percentile. The coordinator node failures are simulated by manually killing processes on the node. Our evaluations show that for all storage schemes the median recovery time is 300μs for recovering the coordinator node after a failure with 1MiB of metadata. Ring has to ensure strong consistency and therefore recover all metadata of all storage schemes within the system before answering client requests. Without this measure, there would be a risk for the system to reply with stale data, since the key with the highest version can be stored in an unrecovered memgest. The complexity of the recovery process results in a high variance of metadata recovery time, which includes:

(1) The leader detects the failure and substitutes the failed node with a spare one.
(2) The leader replicates the new configuration to all nodes.
(3) Once all nodes have received the decision, they start in their new role, i.e., existing nodes connect to the new node.
(4) The new node creates the required empty memgests and connects to alive nodes.
(5) Once nodes are connected, the new node requests the metadata, followed by the logs, which store previous requests from clients to ensure strong consistency.
(6) Once the coordinator nodes have all the metadata, they can rebuild the volatile hashtable. With only the metadata, the new node can serve put and delete requests without actual data. To process get requests, however, the node needs data, which is either copied or recovered depending on the storage scheme (*Rep* or *SRS*, respectively).

Block recovery. We also evaluate the recovery time of erasure coded data blocks for different *SRS* memgests. Figure 13 reports the median and the 90th percentile. Time is measured from receiving a request from the client to when the block is fully recovered. As expected, the time of recovery is correlated to the size of the lost block. We can also see that the latency of recovering a block encoded with $SRS(3,1,3)$ takes longer than $SRS(2,1,3)$, despite the fact that they have the same number of coordinator nodes. As mentioned before, the data in $SRS(2,1,3)$ is encoded according to $RS(2,1)$, and according to $RS(3,1)$ in $SRS(3,1,3)$. It therefore requires collecting 2 blocks for $SRS(2,1,3)$ and 3 blocks for $SRS(3,1,3)$ to recover one lost block. At first glance, it seems that $SRS(2,1,3)$ and $SRS(3,1,3)$ are two identical schemes because they are allocated across 4 nodes and ensure the same throughput and latency. As it can be seen from the experiment, however, they have have different recovery rates and, therefore, different resilience. Since $SRS(2,1,3)$ recovers faster than $SRS(3,1,3)$, it provides higher reliability and availability guarantees.

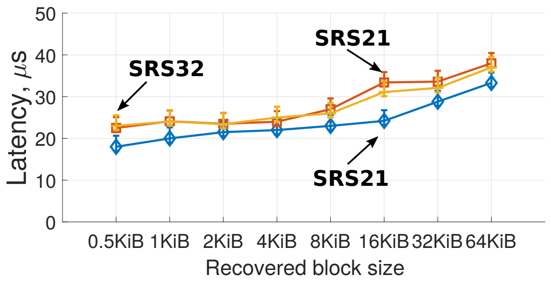


Figure 13: Recovery latencies for storage schemes $SRS(2,1,3)$, $SRS(3,1,3)$, $SRS(3,2,3)$, depending on recovered block size.

Figure 13 shows that $SRS(3,2,3)$ and $SRS(3,1,3)$ have approximately the same latencies for all block sizes. This is because the number of parity nodes affects the number of failures the scheme can tolerate, while computation stays practically the same. $SRS(3,2,3)$ recovers data a little bit faster because the recovery master requires any 3 blocks out of 4 available ones, whereas only 3 blocks are available for reconstruction for $SRS(3,1,3)$. Therefore, $SRS(3,2,3)$ can recover faster in the case of a single failure.

7 RELATED WORK

To address opportunities of exploiting explicit resilience management, we give an overview of the existing technologies allowing users to modify storage schemes. In particular, one of the most well-known KVS, Redis [38], offers the option of determining the degree of replication of each volume, but it does not allow performing the update in a per-key manner. It also does not support erasure codes to reduce memory usage. Redis also does not ensure strong consistency since it employs asynchronous replication.

Erasure codes also support altering resilience of the data. For instance, it is well known that RAID6 can be easily converted to RAID5 and vice versa, but it can take days to decode all the data on the disks [5].

The known method to combine a wide range of storage schemes and be able to change them per object is to maintain a special name node, which stores all metadata and references to requested data [4, 32, 37]. The name node can, however, become a bottleneck and be a single point of failure. In addition, this approach leads to additional hops in data center networks to read and write data.

Another approach of changing storage schemes with low computational overhead is introduced by the BlowFish distributed data store [15]. BlowFish stores data in a compressed format and enables dynamically changing the compression factor. A smaller compression factor indicates higher storage requirements, but also lower latency (and vice versa). However, to ensure resilience the data itself is still replicated, and changes in compression do not influence the reliability of the stored keys. Finally, the granularity for updating the compression factor is a single shard, whereas Ring supports per-key resilience management.

It is worth mentioning that Ring is not the first attempt to create a KVS that supports multiple resilience levels. For instance, Phanishayee et al. [23] suggest supporting multiple replication algorithms with different consistency levels [2, 3, 35]. However, this approach does not support erasure coding and is not strongly-consistent.

8 SUMMARY AND CONCLUSIONS

Modern, in-memory KVSs are widely used because of their performance characteristics and their ability to provide fault tolerance and strong consistency. However, KVSs do not offer users the possibility to select the most suitable trade-off in terms of fault tolerance, performance, and resource usage. In this paper we have presented Ring, a distributed in-memory KVS that allows users to control the level of resilience on a per-key basis and, thus, control the resource usage of the KVS that better matches the application profile. The core of Ring is a novel encoding mechanism, Stretched Reed-Solomon (*SRS*), which enables strongly consistent systems to support different resilience levels on a per-key basis, allows dynamic changes, and does so transparently, without affecting performance or consistency. The experimental evaluation indicates Ring provides significant memory savings and allows choosing the best trade-offs between reliability, performance, and cost.

Acknowledgements. This work was supported by Microsoft Research through its Swiss Joint Research Centre. We thank our shepherd Kimberly Keeton, and the anonymous reviewers for their help in improving the paper.

A FAULT RESILIENCE ANALYSIS

In this section, we illustrate the model used to verify that our *SRS* codes ensure approximately the same level of reliability and availability as *RS* codes. We assume a fail-stop model where nodes only exhibit crash failures, and any correct node in the system can detect that a particular node has failed. Once a node fails, it can no longer influence the operation of other nodes in the system. We also assume that node failures are independent and identically distributed.

A.1 Reliability model for Reed-Solomon

Reliability of a KVS is the ability to perform a required function, under given environmental and operational conditions and for a stated period of time [28]. The reliability function $R(t)$ of a KVS is defined as the probability that the KVS does not lose data in the time interval $(0, t]$. We estimate the reliability of the KVS as annual reliability, i.e., the probability that the data is not lost within one year. For instance, reliability of 99.99% (i.e., four nines) refers to the probability of data loss over one year period should be less than 10^{-2}.

We use Continuous-Time Markov Chain (CTMT) model to estimate reliability $R(t)$ [28]. CTMT models have been extensively used for reliability analysis of erasure coded and replicated storage [11, 31]. The diagram in Figure 14 shows the Markov model for *RS* codes that have k data blocks and m parity blocks. *RS* codes can tolerate up to m simultaneous failures which is reflected in the Markov model by $m + 1$ states. The final state FS stands for the unrecoverable fail state. The KVS comprises $k + m$ nodes with a failure rate of a node equal to λ. Finally, μ_i is a rebuild rate from state i to state $i - 1$, which is considered to be constant μ. The model also assumes that the KVS restores only one node at a time.

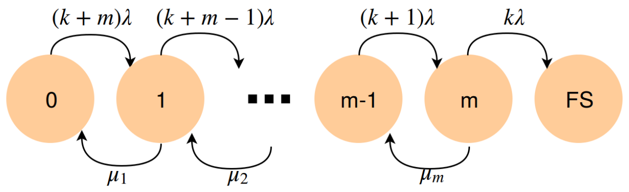


Figure 14: Markov model for *RS* codes with k data blocks and m parity blocks.

The rebuild rate μ can be calculated as $1/T_{\text{reconst}}$ [20], where T_{reconst} is the reconstruction time for $RS(k, m)$ coding calculated as:

$$T_{\text{reconst}} = \frac{C}{B_N} + T_{\text{comp}}(C), \tag{6}$$

where C is the full size of data set, B_N denotes the network bandwidth. T_{comp} is the computation time including the encoding and the decoding time for erasure coding schemes. It can be seen from Eqn. (6) that the reliability of the KVS depends also on the data size and bandwidth.

The Kolmogorov problem for Markov chains is defined as:

$$\begin{cases} \frac{dP(t)}{dt} & = AP(t) \\ P(0) & = P_0 \end{cases} \tag{7}$$

where A is a transition matrix of the Markov model, $P(t)$ is a vector of state probabilities and P_0 is the initial state vector of the system. $P_i(t)$ represents the probability that the system is in the state i at the time t. According to definition of probability $\sum_i P_i(t) = 1$.

Finally, the reliability $R(t)$ is the sum of all probabilities of being in functional states. In the case of $RS(k, m)$, the first $m + 1$ states are considered to be functional. Hence,

$$R(t) = \sum_{i=0}^{m} P_i(t) = 1 - P_{FS}(t),$$

where $P_i(t)$ is a solution of the problem (7). In practice, (7) can be solved with the use of the matrix exponential [1, 21].

The transition matrix of the Markov model is given by

$$\underset{m+2\times m+2}{A} = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0\text{FS}} \\ a_{10} & a_{11} & \cdots & a_{1\text{FS}} \\ \vdots & \vdots & \vdots & \vdots \\ a_{\text{FS}0} & a_{\text{FS}1} & \cdots & a_{\text{FSFS}} \end{bmatrix},$$

where a_{ij} is the transition rate from state i to state j, when $i \neq j$. Diagonal elements can be calculated as

$$a_{ii} = -\sum_{\substack{j=0 \\ i \neq j}}^{m+2} a_{ij}.$$

For instance for $RS(3, 2)$, according to Markov model in Figure 14, the transition matrix has the following form

$$A = \begin{bmatrix} -5\lambda & 5\lambda & 0 & 0 \\ \mu & -4\lambda - \mu & 4\lambda & 0 \\ 0 & \mu & -3\lambda - \mu & 3\lambda \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

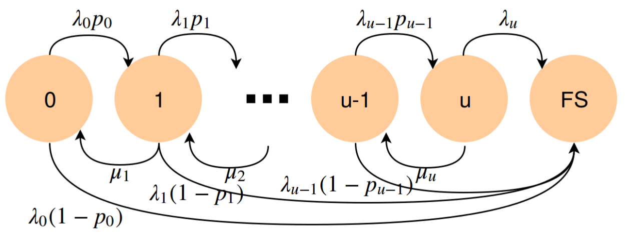


Figure 15: Markov model for *SRS* codes with k data blocks stretched over s nodes and m parity blocks.

A.2 Reliability model for Stretched Reed-Solomon

$SRS(k, m, s)$ is obtained by stretching k data blocks over s nodes. This manipulation has two side effects: first of all, *SRS* data nodes stores less data than *RS* data nodes, and therefore less data needs to be recovered in the case of data node failures. In addition, sometimes $SRS(k, m, s)$ is able to tolerate more than m simultaneous failures. Thus, the model for *RS* is not applicable to *SRS* codes.

Let us define an array f, where f_i stands for the probability that the KVS tolerates $i + 1$ simultaneous node failures. We can always estimate f_i by total enumeration of all possible failure scenarios. Since $SRS(k, m, s)$ can always tolerate loss of up to m nodes, $f_i = 1$ for $i \in [0, m - 1]$. We denote the maximum number of simultaneous failures after which the KVS can be recovered as u. Basically,

$$u = \underset{i}{\operatorname{argmin}}\{f_{i-1} \neq 0 \wedge f_i = 0\}$$

The general Markov model for *SRS* is shown in Figure 15, where $\lambda_i p_i$ stands for the transition to tolerate $i + 1$ failures with the condition that it tolerated i failures and $\lambda_i(1 - p_i)$ reflects opposite situation when the KVS is not able to tolerate an additional node failure. Similarly to Markov model for *RS*, λ_i is equal to the number of alive nodes multiplied by a failure rate of a single node. Thus, $\lambda_i = (s + m - i)\lambda$, where λ is the failure rate of a single node.

In Figure 15, p_i is a conditional probability to tolerate $i+1$ failures when i nodes are lost. p_i can be estimated as

$$p_i = p(i + 1|i) = \frac{p(i + 1, i)}{p(i)} = \frac{p(i + 1)}{p(i)} = \frac{f_{i+1}}{f_i},$$

where $p(i + 1|i)$ stands for conditional probability to survive after $i + 1$ failures when i failures are already tolerated, $p(i + 1, i)$ is a probability to tolerate $i+1$ and i failures. Clearly $p(i+1, i) = p(i+1)$ since if the the KVS tolerated $i + 1$ failures then it also tolerated i.

Recovery flows μ_i represents the average recovery rate from state i to state $i - 1$, i.e.; when there are i nodes down. Here we distinguish between data nodes and parity nodes, because data nodes in $SRS(k, m, s)$ store less data than in original $RS(k, m)$. Basically, when i nodes are lost, then there are up to $i + 1$ different recovery speed options which are represented by the number failed data nodes. Thus,

$$\mu_i = \sum_{j=0}^{i} \mu_{ij} p_{ij},$$

where μ_{ij} is a recovery rate when i nodes are lost and j of them are data nodes and p_{ij} is its probability. μ_{ij} and p_{ij} equal 0 when $(i - j) > m$, since it is impossible to have more than m failed parity nodes.

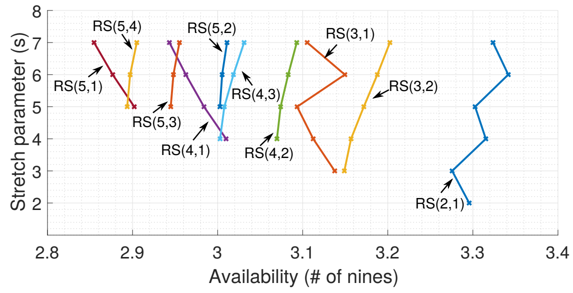


Figure 16: Availability of *SRS* codes with different parameters.

The probability p_{ij} of having j failed data nodes and $i - j$ failed parity nodes obeys the hypergeometric distribution, that describes the probability of drawing i data nodes in $i + j$ draws, without replacement, from a "bag" of size $s + m$ that contains exactly s data nodes.

$$p_{ij} = \begin{cases} \dfrac{\binom{s}{j}\binom{m}{i-j}}{\sum\limits_{\substack{k=0\\ i-k\leq m}}^{i} \binom{s}{k}\binom{m}{i-k}} & \text{if } i - j \leq m, \\ 0 & \text{otherwise,} \end{cases}$$

where $\binom{i}{j}$ is binomial coefficient "i choose j".

The recovery rate μ_{ij} is a recovery rate when i nodes are lost and j of them are data nodes. Thus,

$$\mu_{ij} = \frac{j}{i}\mu_D + \frac{i - j}{i}\mu_P,$$

where μ_D and μ_P are recovery rate of data node and parity node respectively.

Since parity nodes in $SRS(k, m)$ and $RS(k, m)$ are identical, they share the same recovery rate μ. We assume that μ is linear with respect to data size. A data node stores s/k times less data than a parity node, thus $\mu_D = \frac{k}{s}\mu_P = \frac{k}{s}\mu$. Finally,

$$m_{ij} = \frac{j}{i}\frac{k}{s}\mu + \frac{i - j}{i}\mu,$$

where μ is the recovery rate in $RS(k, m)$ model.

As an example let us consider $SRS(2, 1, 4)$ code. It can always tolerate a single failure and with probability $\frac{2}{5}$ overtake the second failure. Thus, the Markov model in Figure 15 is comprised of 4 states, and the transition matrix has the following form

$$A = \begin{bmatrix} -6\lambda & 6\lambda & 0 & 0 \\ \mu_1 & -5\lambda - \mu_1 & 5\lambda\frac{2}{5} & 5\lambda\frac{3}{5} \\ 0 & \mu_2 & -4\lambda - \mu_2 & 4\lambda \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The theoretical findings of reliability analysis of *SRS* codes are presented in Section 3.3.

A.3 Availability model

Availability is the ability of a KVS to provide immediate access to information or resources at a stated instant of time or over a stated period of time. Similar to reliability, availability is estimated as a sum of probabilities being in available states S:

$$A(t) = \sum_{i \in S} P_i(t)$$

In the case of RS and SRS codes, only the 0 state ($S = \{0\}$) is considered to be available, since in other states there exist unrecovered data which cannot be accessed immediately.

Next, we introduce the interval availability in the interval $(0, \tau)$ as

$$A_{av}(\tau) = \frac{1}{\tau} \int_0^{\tau} A(t)dt$$

$A_{av}(\tau)$ is just the average value of the point availability over a specified interval from startup. Finally, availability can be measured as average availability over a year as $A_{av}(1 \text{ year})$.

Figure 16 indicates estimated availabilities of RS and SRS codes. Lines represents different stretching factors of $SRS(k, m, s)$ codes that share the same parent code $RS(k, m)$. It can be observed that all RS schemes and their stretched variations have availability less than 3.4 nines. In addition, the number of nodes in the stripe decreases the availability. Maximal availability has been observed for the family of $SRS(2, 1, s)$ codes and stands at approximately 3.35 nines.

REFERENCES

[1] M. L. Abell and J. P. Braselton. *Differential Equations with Maple V®*. Academic Press, 2014.

[2] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–154, 2011.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[4] S. Chandrashekhara, M. R. Kumar, M. Venkataramaiah, and V. Chaudhary. Cider: A Case for Block Level Variable Redundancy on a Distributed Flash Array. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, July 2017. doi: 10.1109/ICCCN.2017.8038466.

[5] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[6] B. Cipra. The ubiquitous Reed-Solomon codes. *Siam News*, 26(1):1993, 1993.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[8] L. E. Dickson. *Linear groups: With an exposition of the Galois field theory*. Courier Corporation, 2003.

[9] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450.

[11] J. L. Hafner and K. Rao. Notes on reliability models for non-MDS erasure codes. *IBM Res. rep. RJ10391*, 2006.

[12] R. Hecht and S. Jablonski. NoSQL evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341. IEEE, 2011.

[13] Jim Seeger,Naresh Chainani,Aruna De Silva,Karen Mcculloch,Kiran Chinta,Vincent Kulandai Samy,Tom Hart. DB2 V10.1 Multi-temperature Data Management Recommendations, 2012.

[14] R. Jiménez-Peris, M. Patiño Martínez, G. Alonso, and B. Kemme. Are Quorums an Alternative for Data Replication? *ACM Trans. Database Syst.*, 28(3):257–294, Sept. 2003. ISSN 0362-5915. doi: 10.1145/937598.937601.

[15] A. Khandelwal, R. Agarwal, and I. Stoica. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 485–500, Santa Clara, CA, 2016. USENIX Association. ISBN 978-1-931971-29-4.

[16] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay Only when It Matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687657.

[17] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6.

[18] Microsoft Azure. Azure Storage Pricing. https://msdn.microsoft.com/en-us/library/azure/ee691964.aspx. [Online; accessed 18-Feb-2018].

[19] Microsoft Azure. Understanding Block Blobs, Append Blobs, and Page Blobs. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/, 2016. [Online; accessed 18-Feb-2018].

[20] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 30. ACM, 2016.

[21] C. Moler and C. Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM review*, 45(1):3–49, 2003.

[22] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015. ISSN 0734-2071. doi: 10.1145/2806887.

[23] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini. Flex-KV: Enabling high-performance and flexible KV systems. In *Proceedings of the 2012 workshop on Management of big data systems*, pages 19–24. ACM, 2012.

[24] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[25] J. S. Plank, K. Greenan, E. Miller, and W. Houston. GF-Complete: A comprehensive open source library for Galois Field arithmetic. *University of Tennessee, Tech. Rep. UT-CS-13-703*, 2013.

[26] J. S. Plank et al. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Softw., Pract. Exper.*, 27(9):995–1012, 1997.

[27] M. Poke and T. Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118. ACM, 2015.

[28] M. Rausand and A. Høyland. *System reliability theory: models, statistical methods, and applications*, volume 396. John Wiley & Sons, 2004.

[29] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An RDMA protocol specification. Technical report, IETF Internet-draft draft-ietf-rddp-rdmap-03. txt (work in progress), 2005.

[30] F. B. Schneider. Byzantine Generals in Action: Implementing Fail-stop Processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, May 1984. ISSN 0734-2071.

[31] M. L. Shooman. *Reliability of computer systems and networks: fault tolerance, analysis, and design*. John Wiley & Sons, 2003.

[32] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[33] Storage Performance Council. SPC Trace File Format Specification, Revision 1.0.1, 2002.

[34] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522731.

[35] R. Van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, volume 4, pages 91–104, 2004.

[36] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on*, pages 206–215. IEEE, 2000.

[37] M. Xia, M. Saxena, M. Blaum, and D. Pease. A Tale of Two Erasure Codes in HDFS. In *FAST*, pages 213–226, 2015.

[38] J. Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine*, 79, 2009.

[39] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 167–180, 2016.