Have a Seat on the ErasureBench: Easy Evaluation of Erasure Coding Libraries for Distributed Storage Systems

Sébastien Vaucher, Hugues Mercier, Valerio Schiavoni
University of Neuchâtel, Switzerland
first.last@unine.ch

***Abstract*—We present ErasureBench, an open-source framework to test and benchmark erasure coding implementations for distributed storage systems under realistic conditions. ErasureBench automatically instantiates and scales a cluster of storage nodes, and can seamlessly leverage existing failure traces. As a first example, we use ErasureBench to compare three coding implementations: a $(10, 4)$ Reed-Solomon (RS) code, a $(10, 6, 5)$ locally repairable code (LRC), and a partition of the data source in ten pieces without error-correction. Our experiments show that LRC and RS codes require the same repair throughput when used with small storage nodes, since cluster and network management traffic dominate at this regime. With large storage nodes, read and write traffic increases and our experiments confirm the theoretical and practical tradeoffs between the storage overhead and repair bandwidth of RS and LRC codes.**

I. Introduction

The business model of consumer-facing Cloud providers like Dropbox often consists in storing huge volumes of data. Customers expect a high level of reliability, but rarely envisage paying top dollar for such services.[1] The very-large scale of Cloud provider setups implies that failures occur fairly often [1], thus fault-tolerance is intrinsically built into the design of large-scale systems. The result is that irrecoverable user data loss caused by technical failures is rare [2].

In order to offer reliability and availability guarantees towards their customers, Cloud providers need to implement techniques providing redundancy, while keeping costs under control. The long-established way of providing redundancy is replication, which consists in duplicating functionality across components that are unlikely to fail simultaneously. For data storage, this means storing multiple replicas in geographically distinct locations. The cost of storage is multiplied by the amount of redundancy wanted. To store data on a redundant array, the system has to send write requests to each component in the array. When a copy of the data is unavailable, the system can simply query another copy. A system that is configured with a replication factor of f tolerates $f-1$ simultaneous failures.

[1]The hidden costs of these free services are left for another time.

An alternative solution to provide data redundancy is erasure coding. Error-correcting codes can decrease the storage overhead compared to replication, but recovering errors requires decoding. Furthermore, traditional erasure codes, while being storage-efficient, were not designed to handle failures occurring on distributed storage systems. Replacing a failed disk requires the decoding of all the data objects with a block on it. With a $(k, n-k)$ maximum distance separable (MDS) [3] code like a Reed-Solomon code (RS), the required bandwidth and latency to fetch k geographically distributed blocks required for decoding is significant, and using such codes on geographically distributed data centers would saturate the network links. This observation has led to the study of codes that can repair failed blocks by using less information than required to recover the original data. The drawback of these *locally repairable codes* (LRCs) [4] is a lower code rate, thus they are a tradeoff between MDS codes and replication.

This paper introduces ErasureBench, a framework to evaluate erasure coding implementations under real conditions. ErasureBench exposes a familiar hierarchical file system interface leveraging FUSE. The architecture is modular and allows to easily plug and configure different erasure codes. Testing erasure coding implementations in large distributed systems is a challenging endeavor. Big players like Facebook [5] or Microsoft [6] have the capacity to execute benchmarks on full-fledged, production-ready clusters. Alternatively, the usual path to perform these evaluations is to rely on simulations [7].

ErasureBench allows to test real implementations using a cluster of commodity hardware. It can automatically instantiate a cluster of storage nodes and dynamically scale it during execution using Docker containers. Furthermore, it can inject real-world failure trace from the Failure Trace Archive (FTA) [8] to evaluate a system under realistic conditions. ErasureBench is partially inspired by [9], whose framework exposes a REST API through HTTP. It targets the measurement of read/write throughput and data storage overhead of low-level languages coding libraries. As an improvement over [9], ErasureBench offers a file-system interface and the ability to inject failure traces.

The rest of the article is organised as follows. In Section II, we briefly introduce erasure coding concepts

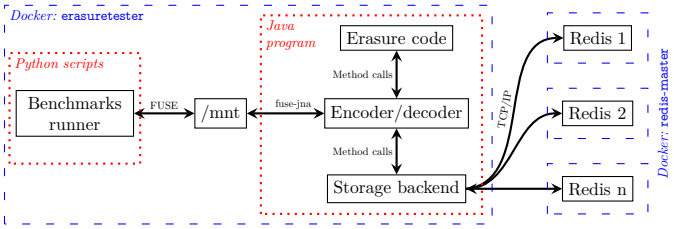

Figure 1. Architecture of ErasureBench.

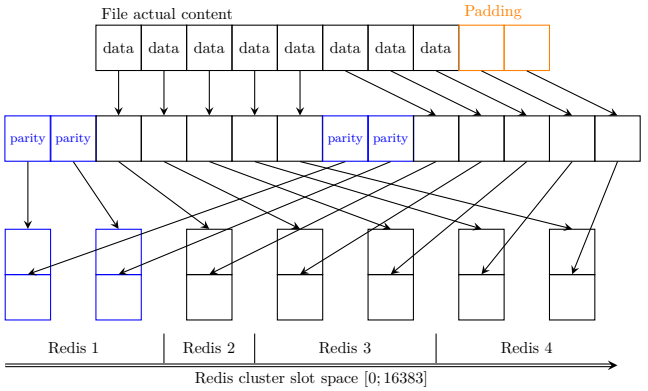

Figure 2. Splitting file contents in blocks, applying erasure coding, and storing data and parity blocks in a Redis cluster.

and the codes used in our evaluation. The architecture of ErasureBench is detailed in Section III. Section IV presents our evaluation, comparing RS and LRC codes. We conclude in Section V.

II. Erasure codes in a nutshell

We write $(k, n-k)$ for the parameters of maximum distance separable (MDS) codes, such as a Reed-Solomon (RS) codes [10], where k is the number of data blocks and $n-k$ is the number of parity blocks. The parameters of Locally Repairable Codes (LRCs) are $(k, n-k, r)$, where r is the block locality. Block locality r means that the block is a function of r other blocks [5]. Correcting a single erased block using a $(k, n-k, r)$ code requires to fetch r (hopefully $< k$) blocks. At both extremes of the spectrum, MDS codes have the worst locality $r = k$ but the best storage overhead, whereas replication has locality $r = 1$ but the worst storage overhead. Since we target distributed storage, we consider systematic erasure-correcting codes. It's trivial to extends it to non-systematic codes.

We emphasize again that we can use ErasureBench with any erasure coding scheme. This being said, to test our platform, we implement and test the following schemes:

- **NC** A $(10, 0)$ *no coding* scheme, with 10 data blocks per stripe, that simply forwards data blocks without redundancy.
- **RS** A $(10, 4)$ RS code. We use the Vandermonde-RS implementation provided in [5].
- **LRC** The $(10, 6, 5)$ LRC code from [5]. The code is based on a $(10, 4)$ RS code, with three extra parity symbols (two explicit and one implicit) providing locality. The storage overhead/locality tradeoff of the code is optimal. We use the open-source implementation provided in [5].

The last two schemes correspond to storage overheads of 1.4 and 1.6, in the sweet spot of their use in large-scale distributed storage systems in the cloud.

III. The ErasureBench Framework

The ErasureBench framework allows researchers to easily evaluate the performances of erasure codes when used in a large-scale storage context. It exposes a filesystem interface to the end-user to facilitate the adoption of well-known file-system benchmarks.[2] Files are split into blocks, processed (encoded), and stored in a key-value store. Thanks to its decoupled architecture, each of the three components (filesystem interface, key-value store, erasure code) can be easily replaced by alternative implementations. In the remainder of this section we provide further details about the software architecture and our implementation choices.

Architecture. Figure 1 depicts the architecture of ErasureBench. At its core, ErasureBench exposes a filesystem interface via a Linux mount point implemented via Filesystem in Userspace (FUSE). Upon its execution, the filesystem is mounted under a regular directory. Low-level IO system calls are intercepted by the *fuse-jna*[3] Java library. The blocks are then sent back and forth between the encoder/decoder layer and the erasure code component. The encoder/decoder first chunks data in blocks of the chosen size and aligns read and write operations to the correct boundaries. Once chunked, blocks are processed by the chosen erasure code to generate the redundancy blocks. Finally, data and parity blocks are passed to the storage backend.

Blocks storage. The storage backend is designed to operate on 1 byte blocks. As usual, storing each block individually in the key-value store is a very slow operation: the metadata is much bigger than the data itself, leading to unworkable overheads. We therefore added an intermediate layer between the encoder/decoder and the storage backend. This layer is transparent to the encoder/decoder which still deals with single 1 byte blocks. Thanks to this new layer, we aggregate multiple write operations on the key-value store, and each key stores an aggregation of multiple blocks. The guarantees offered by the erasure coding process are still kept because one aggregation only stores blocks that belong to the same stripe position. This additional component provides a considerable speed-

[2] Due to the lack of space, we omit experiments using tools such as IOzone or FileBench.

[3] https://github.com/EtienneParot/fuse-jna

up comprised between 100× and 1000×. Figure 2 shows how files are split in blocks, erasure coded, and then aggregated and stored on multiple Redis servers. A Least Recently Used (LRU) cache optimizes the retrieval of multiple individuals blocks. This way, when reading a file sequentially, each block aggregation is only retrieved once from the key-value store.

Metadata management. We keep track of the location of each stored block. Our strategy is to assign a 32-bit key to each block. From each key, we can infer two related identifiers: a Redis key and an offset. The Redis key points to an aggregation of blocks stored in the Redis cluster. We then use the offset to precisely locate the block we want within the aggregation. In the current prototype, we store all metadata in memory. Each file consists in a list of identifiers. We also store the size of the file once decoded to discard padding blocks.

Implementation details. We implement the erasure coding algorithms described in Section II and bundle them in ErasureBench. We extract and adapt the LRC libraries from [5] to remove dependencies on any other *Hadoop* component. We use the Redis distributed key-value store[4] leveraging the battle-tested *Jedis* binding.[5] For testing purposes, an in-memory storage backend is also available. ErasureBench is open-source.[6]

Deployment. The deployment of ErasureBench requires to simultaneously launch multiple independent services and to coordinate their bootstrap. We leverage the Docker[7] technology to facilitate this task. We also exploit Docker Swarm[8] to reduce as much as possible the deployment differences between a local and a remote cluster setup. All the components as well as the supporting Python scripts are packaged as Docker images. The framework allows to easily parametrize the testbed (size of the cluster, erasure code and associated parameters, etc.) by means of configuration files. We further provide Python scripts to setup all the components, execute the benchmarks and collect the resulting logs. We originally used the Redis's own `redis-trib.rb` script[9] to initialize the storage cluster, however we observed severe performance issues with cluster sizes bigger than a few dozens. Hence, we also implemented similar logics in our own Python scripts in order to scale the cluster faster. ErasureBench includes shell scripts that automate the complete pipeline, from the compilation of sources to the execution of benchmarks on a remote Docker Swarm cluster.

Fault injection. ErasureBench supports the injection of synthetic and real failure traces. A synthetic trace is simply a list of system sizes: when a benchmark completes an iteration, we resize the storage cluster using the next value in the list. ErasureBench can inject real failure traces stored in a SQLite database in the format described in [8]. These traces monitor individual nodes of a real-life distributed system and record failure events in a database. ErasureBench can replay these traces to replicate a real-world failure pattern by first creating a storage cluster of the same size as the original system. New nodes are instantiated and old nodes killed at the same rates as reported in the trace. Finally, the framework allows to restrict the replay to a limited time interval of the trace. We use this trace replay feature in the last set of experiments of the next section.

[4]http://redis.io
[5]https://github.com/xetorthio/jedis
[6]https://github.com/safecloud-project/ErasureBench
[7]https://www.docker.com
[8]https://docs.docker.com/swarm/
[9]http://download.redis.io/redis-stable/src/redis-trib.rb

Table I
Workload characteristics and erasure-coding overhead. Sizes are given in MB.

Name	Files	Size	Erasure code	Redis keys	Binary blocks	Base64 blocks
10 B	1000	0.01	NC	10 000	0.12	0.16
			RS	14 000	0.17	0.22
			LRC	16 000	0.19	0.26
bc	94	1.02	NC	1920	4.09	5.45
			RS	2688	5.72	7.63
			LRC	3072	6.54	8.73
httpd	2516	33.02	NC	56 850	132.68	176.99
			RS	79 590	185.75	247.79
			LRC	90 960	212.29	283.18

IV. Evaluation

This section presents the experimental evaluation of the ErasureBench prototype. First, we describe our evaluation settings and workload characteristics. Then we evaluate our system against several metrics: encoding/decoding throughput performance with and without the FUSE layer, network throughput, system scalability and its impact on the request latency. We conclude by evaluating the repairing costs using synthetic and real-world traces.

Evaluation Settings. We deploy our experiments over a cluster of 20 machines interconnected by a 1 Gbit/s switched network. Each host features an 8-core Intel Xeon CPU and 8 GB of RAM. We deploy virtual machines (VMs) on top of the hosts. We use KVM as a hypervisor. To mitigate the performance losses, we expose the physical CPU to guest VMs by mean of the `host-passthrough` option. We leverage the `virtio` module for better I/O performance. All the components of the system are packaged as Docker images. Specifically, we installed Docker (v1.11.2) on each VM, and configured it without any memory restrictions. The deployment and orchestration of the containers rely on Docker Compose (v1.7.1) and Docker Swarm (v1.2.3).

Workloads. In order to evaluate the performance of ErasureBench, we select the source files of two well-known open-source projects: the Apache `httpd`[10] server (v2.4.18)

[10]https://archive.apache.org/dist/httpd/httpd-2.4.18.tar.bz2

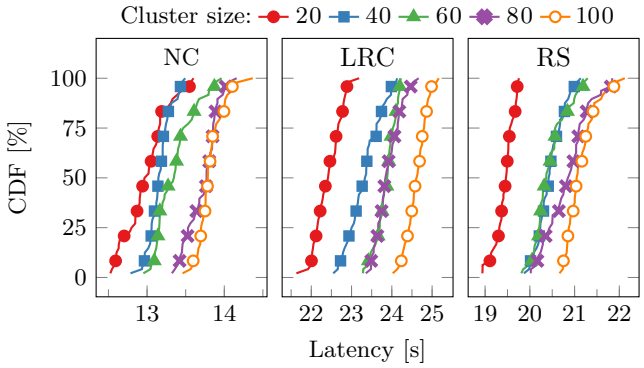

Figure 3. Scalability: write latency for 16 MiB file.

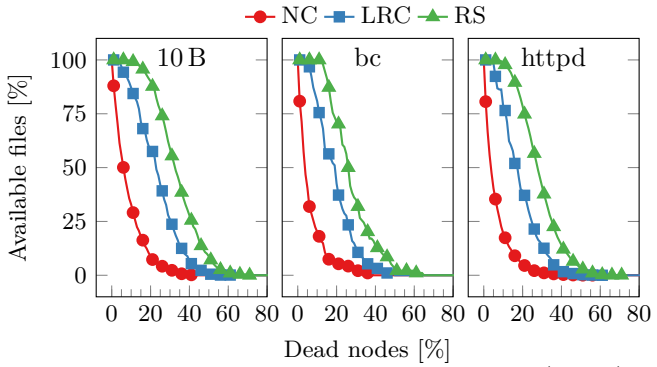

Figure 4. Fault tolerance for different codes: NC, LRC $(10, 6, 5)$ and RS $(10, 4)$. Redis cluster of 100 nodes. Data checked for availability after progressively killing each Redis node.

Table II
ENCODING THROUGHPUT OF THE DIFFERENT ENCODERS IN MB/s

	Block size		
Erasure code	4 MB	16 MB	64 MB
NC	148.1 ± 1.0	148.8 ± 1.1	147.7 ± 7.6
RS	11.7 ± 0.3	12.8 ± 0.1	14.7 ± 0.6
LRC	11.3 ± 0.1	13.7 ± 0.2	15.3 ± 0.4

and GNU `bc`[11] (v1.06). We choose them because they differ in the total number of files and storage requirements. Moreover, we create a synthetic archive containing 1000 files, each consisting of 10 random bytes (10 B in the remainder). We write and read files stored in these archives into/from the ErasureBench partition. Table I details further the workloads. In particular, we highlight 1) the storage overhead induced by the different erasure codes, and 2) the cost of the serialization technique used to save the data in the storage cluster. We indicate the number of Redis keys used to index the blocks of the encoded archives and the size in MB of the binary blocks. Redis can only store strings, hence we apply a Base64 encoding to the blocks, incurring on average a 33 % storage overhead.

Encoding/decoding throughput. We begin our evaluation by measuring the raw encoding throughput of our RS and LRC encoder implementations. We compare them against the NC naive encoder (Section II). Table II presents our results. We observe that the encoding throughput consistently improves with the size of the blocks and achieves up to 15.3MB/s with LRC and 64 MB. When decoding (not shown), we achieve 2.2MB/s and 2.9MB/s, respectively for RS and LRC. This performance is explained by the non-optimized nature of the implementation and the lack of native hardware support, which normally improve the performance by orders of magnitude [9].

Read/write throughput. Next, we measure the read/write performances of the encoders in a more complex scenario, plugging them into the ErasureBench system and user-space file-system. In this experiment, we setup a Redis cluster of 100 nodes. We compare against a modified version of the system (*Direct*) that bypasses the FUSE layer. In both cases, we compare the performances of NC, LRC and RS. Figure 5 presents our results. In all considered scenarios, with the exception of LRC *write*, larger file sizes yield better results. When using bigger files through the FUSE layer, the throughput decreases. We justify this result by the cost of dynamically increasing buffers in the application, as FUSE will only write up to 128 KiB at a time. RS allows faster write operations than LRC, as it always writes 14 instead of 16 blocks per codeword. For read operations, all schemes require ten blocks per codeword, but we observe that LRC is slightly faster than RS. Finally, when compared to NC, we observe slowdown factors between −19.8 % and −41 % for read operations, and from −33.6 % to −84.9 % for write operations, respectively for LRC and RS.

Scalability. To evaluate the scalability of the system, we measure the impact of the Redis cluster size on the latency of write operations. We report the observed latency as computed by the Linux's `time` tool to write 16 MiB into the ErasureBench partition. Figure 3 presents the Cumulative Distribution Function (CDF) of the latencies for different cluster sizes, from 20 up to 100 nodes. We note that the chosen erasure coding technique has a relevant impact on the latency of the operations. Conversely, the size of the cluster does not inflict major penalties to the latency. For instance, using LRC the median (50th) latency is at 22.5s for a 20-nodes cluster, and up to 24.5s for 100-nodes cluster. Note that when using real-world files the performances are much better. For example, when the average size of the files in the `httpd` archive (3 KiB), the median write latency is 0.11s for a 100-nodes cluster.

Bandwidth consumption. Figure 6 illustrates the network impact of RS and LRC codes. First, we observe the network bandwidth consumption while extracting the `httpd` archive into the FUSE partition. LRC writes two more blocks per stripe than RS, thus the write operations are slower (8% slower in our experiments). We then do the reverse operation and read the entire partition, once

[11] https://ftp.gnu.org/gnu/bc/bc-1.06.tar.gz

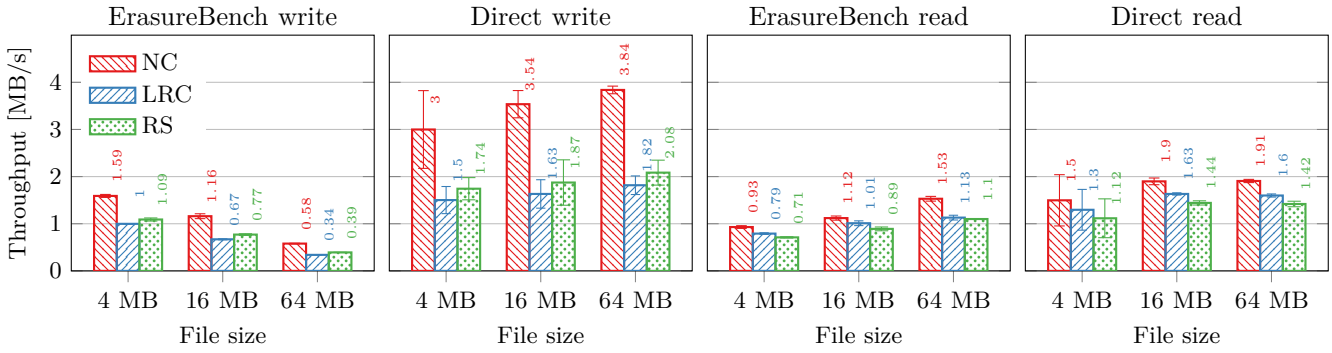

Figure 5. Throughput of NC, RS and LRC for different file sizes. Average (10 runs) and half confidence interval.

with all storage nodes intact and once after killing 5 % of the nodes. We can see that LRC is equivalent to RS when the cluster is intact. However, with degraded reads (that is, when there are missing blocks due to failed nodes), we observe the good locality properties of the LRC, which is much faster than the RS code. In our evaluation, LRC completes a degraded read of the `httpd` archive after 90 seconds, while using RS requires 125 seconds.

Fault-tolerance. An important feature of an erasure code is its resilience to faults, i.e., data stays available when some of the storage nodes crash. To quantify the fault tolerance of RS and LRC, we perform the following experiment. Given a Redis cluster of 100 nodes, we first extract the content of our workload archives. Then, we kill 5% of the Redis nodes uniformly at random and check the integrity of each file in the storage cluster. We repeat the process by killing an additional 5% of the nodes at each subsequent iteration until 80% of the entire cluster is erased. Figure 4 shows the ratio of available files against the ratio of killed nodes. Clearly, RS is more fault-tolerant than LRC for the chosen configurations. This might be surprising because both codes have the same minimum distance and LRC includes RS with extra parity blocks (and a lower code rate), but erasing a constant fraction of the nodes erases more blocks per LRC codeword. In practice, our (10,6,5) LRC has higher availability than our (10,4) RS code because it can repair erased blocks faster in the presence of degraded reads. The figure also shows that when using larger files (as in `httpd`), failures are more likely to occur, as a single damaged stripe within a file will corrupt the entire file.

Network throughput under failures. We conclude our evaluation by leveraging a novel feature of ErasureBench to replay failure traces such as those described in [8]. We evaluate the impact on the network throughput by replaying a portion of the website trace described in [11]. We map each Redis node to a web server and let nodes crash and join following the pattern presented in Figure 7.top. The number of healthy nodes varies between 114 (min) and 118 (max). During the replay, we monitor the network traffic occurring at the storage nodes. First, we extract the `httpd` archive in the ErasureBench partition. When a new node joins, existing blocks are migrated according to a rebalancing policy. Upon departure, we trigger a procedure to repair incomplete stripes. We compare the network throughput required by RS versus LRC, respectively in Figure 7.middle and Figure 7.bottom. We use a stack curve representation to highlight the different traffic types: apart from *Write* and *Read* traffic, we further distinguish between *Check* (to check for damaged strips), *Cluster* (for cluster management), *Other Redis* (for Redis traffic), and *Other* (for application-level traffic). The leftmost peak for both codes occurs during the initial write of the archive into the file system. We achieve the highest throughput for both erasure codes during this initial write: 77.7 MB/s using RS, and 76.3 MB/s using LRC. As expected, we observe the benefit of LRC during read operations, where at most 10.1 MB/s are spent between the 20min-30min range (when 4 nodes leave the network). In comparison, RS requires as much as 15.3 MB/s. We also ran experiments using smaller storage nodes (not shown), where the difference between LRC and RS was negligible because cluster and network management traffic overshadowed read/write traffic.

V. Conclusion

The evaluation of erasure coding libraries in a realistic context is a difficult matter, and for this reason the usual approach explored in literature is to rely on simulations. Unfortunately, simulations sweep important implementation details under the rug, and impacts the outcome and the lessons that practitioners and researchers can learn. In this paper we design, implement and evaluate ErasureBench, a modular framework to overcome these limitations. ErasureBench exposes a canonical file-system interface, allows to easily plug and test different erasure-coding libraries, can inject real-world failure traces, and can be deployed locally or on a cluster without any modifications to the source code.

We test the validity of our prototype by evaluating the cost of two different erasure coding algorithms based on Reed-Solomon and Locally Repairable Codes. Our evaluation confirms well-known theoretical results on the efficiency of LRC codes in large-scale settings.

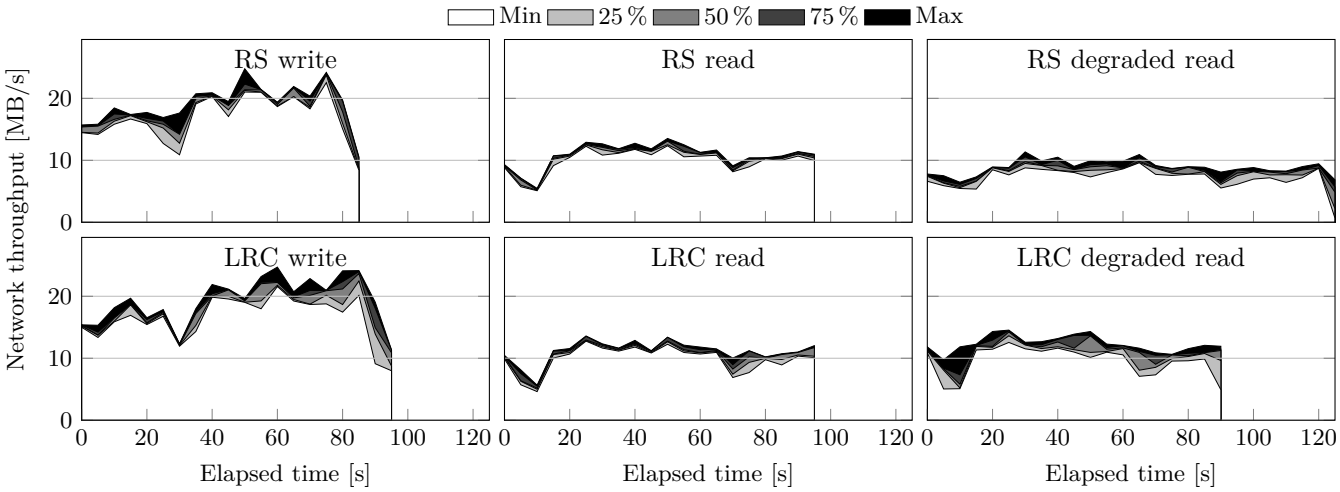

Figure 6. Network throughput of 100 Redis nodes. The `httpd` archive is written and read. Degraded read measured after killing 5% of nodes.

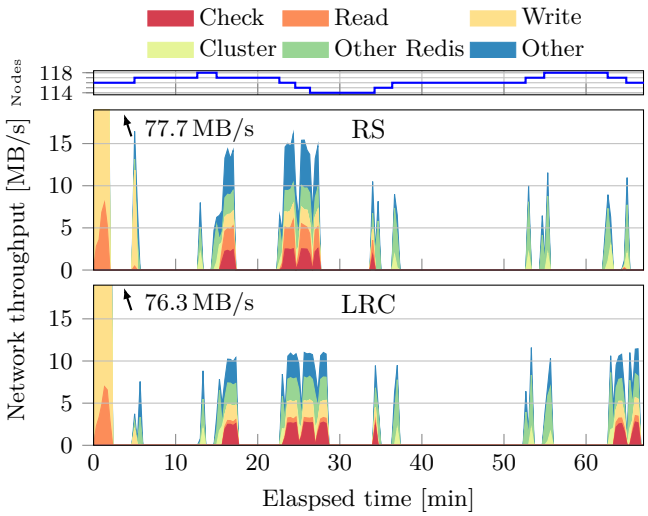

Figure 7. Throughput under faults. Top: failure pattern of nodes. Middle and bottom plots: network traffic writing the *httpd* archive, moving blocks (during repair or rebalance) between Redis nodes.

Our future work includes support of persistent storage for metadata, testing of optimized libraries with hardware support, an important aspect also observed in [9].

VI. Acknowledegements

The research leading to these results was partially supported by the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 653884.

References

[1] B. Schroeder and G. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 10/2010.

[2] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: A Case for Cloud Storage Diversity," in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, Indianapolis, Indiana, USA: ACM, 2010, pp. 229–240.

[3] S. Lin and D. Costello, *Error Control Coding: Fundamentals and Applications*. Pearson-Prentice Hall, 2004.

[4] D. S. Papailiopoulos and A. G. Dimakis, "Locally Repairable Codes," *IEEE Transactions on Information Theory*, vol. 60, no. 10, pp. 5843–5855, 2014.

[5] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing elephants: Novel erasure codes for big data," in *Proceedings of the VLDB Endowment*, vol. 6, 2013, pp. 325–336.

[6] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *USENIX Annual Technical Conference (ATC)*, Boston, MA, USA, 2012, pp. 15–26.

[7] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin, "Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage," in *Proceedings of International Conference on Systems and Storage (SYSTOR)*, Haifa, Israel: ACM, 2014, pp. 1–7.

[8] B. Javadi, D. Kondo, A. Iosup, and D. Epema, "The Failure Trace Archive: enabling the comparison of failure measurements and models of distributed systems," *Journal of Parallel and Distributed Computing*, JPDC, vol. 73, no. 8, pp. 1208–1223, 2013.

[9] D. Burihabwa, P. Felber, H. Mercier, and V. Schiavoni, "A Performance Evaluation of Erasure Coding Libraries for Cloud-Based Data Stores," in *Distributed Applications and Interoperable Systems (DAIS)*. Springer International Publishing, 2016, pp. 160–173.

[10] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, SIAM, vol. 8, no. 2, pp. 300–304, 1960.

[11] M. Bakkaloglu, J. J. Wylie, C. Wang, and G. R. Ganger, "On correlated failures in survivable storage systems," Carnegie Mellon University, Tech. Rep., 05/2002, CMU-CS-02-129.