Master Thesis

Sébastien Vaucher
sebastien.vaucher@unine.ch

9 March 2016

Contents

1 State of the art

1.1 Articles

1.1.1 XORing Elephants: Novel Erasure Codes for Big Data [1]

The article presents a new family of erasure codes called Locally Repairable Codes (LRCs). These codes enable local repair of faulty data. With traditional erasure codes like Reed-Solomon, the cumulative size of the blocks needed to repair a file has to be bigger or equal than the original size of the file. With LRC, a failure affecting a small number of blocks can be repaired using less clean blocks. The authors implemented their algorithm in Hadoop HDFS and deployed a test to Facebook clusters. They measured that the repair process of LRC uses half the disk and network bandwidth compared to Reed-Solomon, at the expense of 14 % more storage usage.

2 Erasure codes tester

2.1 Summary

The erasure codes tester is a program written in Java that exposes a filesystem to the user. It is particular in the way it stores files data: they are saved in a key-value store after being processed by an erasure coding algorithm. Each of the three components (filesystem interface, key-value store, erasure code) can be replaced to test different implementations against each other.

2.2 Software architecture

This section presents the architecture of the system. It is graphically summarized in Figure 2.1.

The Java program exposes a filesystem by using the Filesystem in Userspace (FUSE) interface. When the user runs the program, the filesystem is mounted under a directory of its choice. The user can then use this directory like it would with any other one. The tester will intercept system calls by making use of the *fuse-jna* [2] Java library. Read and write calls will be passed through to the encoder/decoder layer.

The encoder/decoder layer will handle the complicated task of correctly chunking the data. Its role includes the handling of aligning read and writes operations to the correct

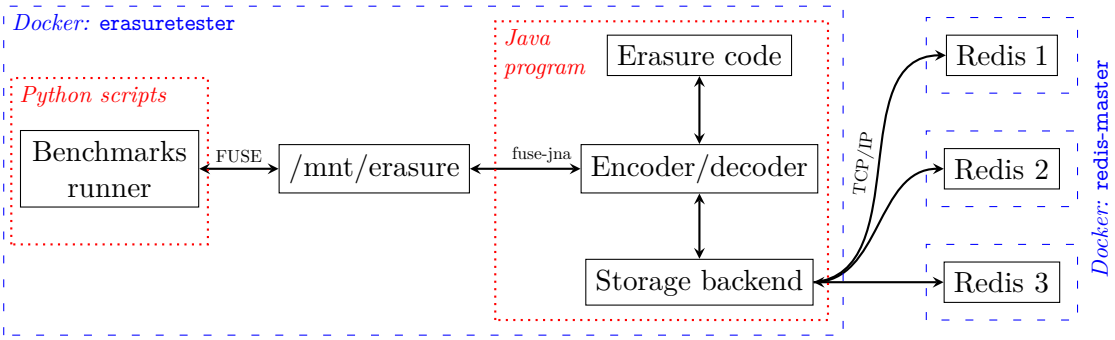

Figure 2.1: Graphical representation of the components of the system

boundaries. Once the data is chunked in blocks of the correct size, it is passed to the erasure coding algorithm, which will add redundancy blocks. The data and parity blocks will then be passed to the storage backend.

The storage backend is the component that interfaces the block storage primitives of our program to operations on a key-value store.

2.2.1 Available implementations

As stated in section 2.2, each component of our tester can have several implementations. The frontend has one available implementation: a FUSE filesystem interface backed by *fuse-jna* [2]. The encoder/decoder layer is not modular because it is sufficient to have one correct way of handling that operation. There are three erasure coding algorithms bundled with the program. They come from [1], and were adapted to free them from dependencies on any *Hadoop* component. A forth algorithm is provided, in the name of the *Null* encoder. As its name suggests, its *modus operandi* consists in simply forwarding data blocks without any added redundancy. Our program is compatible with one distributed key-value store: *Redis*. The compatibility is possible through two libraries: *Redisson* and *Jedis*. The implementation using *Redisson* is simpler in terms of coding, but is slower. It is therefore recommended to use the *Jedis* implementation. For testing purposes, a storage backend using a single Java Map is also available.

2.2.2 Blocks storage

For performance reasons[1], there is an additional layer before the storage of blocks in the key-value store. As the size of each block is 1 byte, performing an operation on the key-value store over the network for each block is very costly. For this reason, the role of this intermediate layer is to aggregate multiple operations at once. Instead of storing one block per key in the key-value store, 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.

An Least Recently Used (LRU) cache optimizes the retrieval of multiple individuals blocks. This way, when reading a file sequentially, each blocks aggregation is only retrieved once from the key-value store.

2.3 Surrounding components

Running the tester requires multiple independent services. They need to be launched simultaneously and need to be bootstrapped to work together. On top of these services,

[1] This additional component provides a considerable speed-up comprised between 100× and 1000×

we want to perform measurements on the performance of different erasure codes. The solution we adopted is the containerization of each service. The tester along with supporting Python scripts are bundled together in a Docker image. When an experimenter wants to start the benchmarks with, say, a Redis cluster as storage backend, he only needs to ask Docker Compose to scale the number of running containers. The supporting Python scripts will configure the Redis cluster and then run the benchmarks and collect the results.

Thanks to the use of Docker Swarm, the setup to run the benchmarks on a local machine or on a cluster of machines is similar. The system can use any reasonable number of Redis servers.

Bibliography

[1] M. Sathiamoorthy *et al.*, "Xoring elephants: novel erasure codes for big data", in *Proceedings of the VLDB Endowment*, VLDB Endowment, vol. 6, 2013, pp. 325–336.

[2] E. Perot. (6 Mar. 2016). Fuse-jna, [Online]. Available: `https://github.com/EtiennePerot/fuse-jna` (visited on 08/03/2016).