LH*$_{RS}$: A Highly Available Distributed Data Storage

Witold Litwin   Rim Moussa   Thomas J.E. Schwarz, S.J.

CERIA Lab., Université Paris Dauphine
FRANCE
Witold.Litwin@dauphine.fr Rim.Moussa@dauphine.fr

Santa Clara University
USA
TSchwarz@scu.edu

Abstract

The ideal storage system is always available and incrementally expandable. Existing storage systems fall far from this ideal. Affordable computers and high-speed networks allow us to investigate storage architectures closer to the ideal. Our demo, present a prototype implementation of LH*$_{RS}$: a highly available scalable and distributed data structure.

1. Introduction

Scalable and Distributed Data Structures [SDDS] are intended for computers over fast networks, usually local networks, i.e. for the *multicomputers*. This new hardware architecture is promising and gaining in popularity. In spite of the advantages given by distributing data, vulnerability to failures remains a problem that grows with the number of machines supporting the SDDS.

Many approaches to build highly available, i.e., fault tolerating, distributed data storage systems have been proposed. They generally use either (*i*) data mirroring or (*ii*) parity calculus [WK02]. The latter approach uses erasure-correcting codes. The simplest codes, e.g. in RAID systems [PGK88], use XOR calculus for the tolerance of a single site failure. Multiple failures need more complex codes. These can be the binary codes [H94] for double or triple failure, or character codes, more generally. Examples of character codes are array codes such as the EVENODD code [BB94], the X-code [XB99] or Reed Solomon codes. The latter appear at present to be the best to deal with multiple failures [R89] [BK95][P97] [LS00] [ML02] [S02] [MS04] [LMS04] [M04].

Below, Section 2 recalls the LH*$_{RS}$ file structure. Section 3 overviews our bucket architecture. Section 4 presents the demonstration outline. Finally, performance results are given in section 5.

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

2. LH*$_{RS}$ Scheme

LH*$_{RS}$ scheme is described with details in [LS00] and [LMS04]. An LH*$_{RS}$ file is subdivided into groups. Each group is composed of m Data Buckets and k Parity Buckets. Buckets are basically in distributed RAM, each at a different server node. The data buckets store the data records of the group. These are encoded into the parity records for high availability as follows in the parity buckets. Every data record has a rank r in its data bucket. It receives this rank upon insertion.

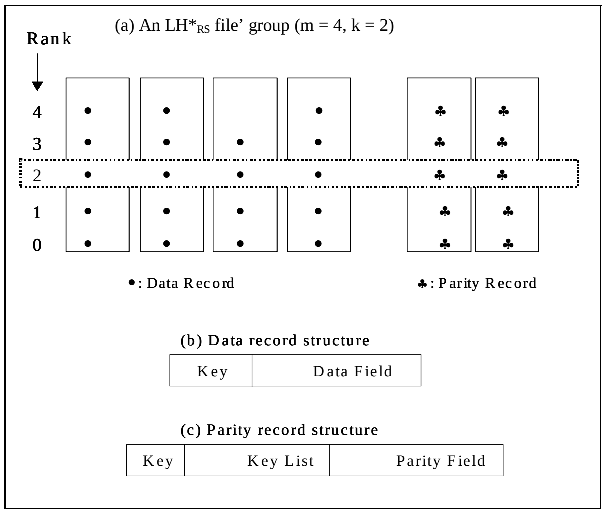
(a) An LH*RS file' group (m = 4, k = 2)
Rank
4
3
2
1
0
•: Data Record
♣: Parity Record
(b) Data record structure
Key
Data Field
(c) Parity record structure
Key
Key List
Parity Field

Figure 1: LH*$_{RS}$ file Structure

A *record group* consists of all records with the same rank in a *bucket group*. We construct parity records from data records having the same rank within data buckets forming a bucket group (Fig. 1(a)). The record grouping has an impact on the data structure of a parity record. Fig. 1(b-c) shows the structure of a data record and a parity record. The *key* field of the parity record is its rank; the *key list* keeps track of the data records in the record group. The parity field contains the actual parity symbols for the record group that we calculate using our version of Reed Solomon codes.

Thanks to the coding scheme, we can calculate the contents of all records in a record group if we have access to m out of the $n = m+k$ records. This provides us with k availability. The actually parity calculation proceeds as follows: When a record is updated (and here an insert counts as a modification of a zero record) we calculate its Δ-record, which is the XOR of the old and the new data field. We send this Δ-record to all parity buckets, who update their parity field by XORing the Δ-record multiplied symbol-wise by a given fixed element, contained in a *parity matrix* **P**. The multiplication is done in a Galois field. According to our experiments we achieve the overall best performance with the Galois field with 2^{16} elements. Since we use the logarithmic method to multiply, we actually store directly the logarithms of **P** in a matrix **Q**. Thanks to an optimization of **P**, the first row and the first column of **P** contain only 1's coefficients. In this case, we obviously can do away with the multiplication. In any case, our experiments show that the processing overhead of Reed-Solomon is small. Table 1 below shows our demo matrices **P** and **Q**.

(P)

0001	0001	0001
0001	eb9b	2284
0001	2284	9e74
0001	9e44	d7f1

(Q)

0000	0000	0000
0000	5ab5	e267
0000	e267	0dce
0000	784d	2b66

Table 1: Our demo matrices P and Q for *m* = 4 and *k* = 3.

In order to reconstruct lost records in a record group, we gather the columns of **P** corresponding to available records in a matrix **H**, invert **H** with the Gaussian algorithm, multiply each available record with a coefficient of **H** and XOR the results together to obtain a missing record. Since the inversion is done once for all records to be reconstructed, it does not constitute a significant overhead.

The file starts with one data bucket and $K \geq 1$ parity buckets. The K value, called Intended Availability Level, is a file parameter. It scales up through data buckets splits, as the data buckets get overloaded. Each bucket group has then the availability level k that is K or $K-1$, [LMS04]. Each data bucket contains a maximum number of b records. The value of b is the bucket capacity. When the number of records within a data bucket exceeds b, the bucket alerts the coordinator, a special entity coordinating splits. The latter designates a data bucket to split.

3. System Architecture

The goal of the prototype is to tune and experimentally determine LH*$_{RS}$ performance. Our current LH*$_{RS}$ implementation is described in depth in [M04]. It completes and improves that in [L00][ML02].

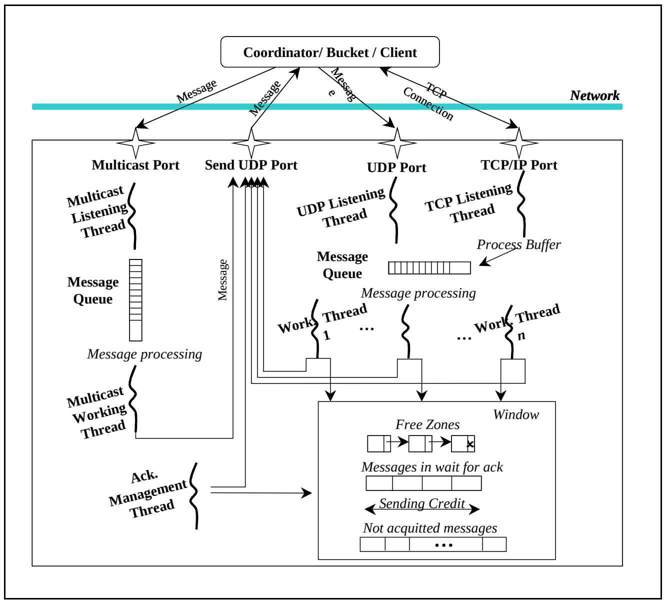

Figure 2: Bucket Architecture.

Figure 2 shows the multithreaded architecture of a bucket. Initially, a bucket is either connected to the *data buckets multicast group* or to the *parity buckets multicast group*. It starts with the multicast listening thread and the multicast working thread. When it receives a multicast group inviting the bucket to be a new or a spare bucket, it instantiates the other threads, responds positively to the coordinator, and waits for the confirmation. A selected bucket, upon receiving the confirmation, disconnects from its multicast group. Non-selected buckets cancel the instantiation process, and can commit to other invitations. Here are the functions of each thread.

- ↻ The *Multicast Listening Thread* is a temporary thread that listens to a fixed data or parity multicast port, and queues multicast messages.
- ↻ The *Multicast Working Thread* is also a temporary thread that processes queued multicast messages.
- ↻ The *UDP Listening Thread* listens to a fixed UDP port, calculated from the bucket number.
- ↻ The *Working Threads*, usually four, process queued UDP messages.
- ↻ The *TCP Listening Thread* accepts and handles multiple TCP/IP connections.
- ↻ The *Acknowledgement Manager Thread*: Each UDP message to be acknowledged is

added to the *messages in wait for the ack. list,* from which it is removed when the acknowledgement is received. This thread scans the list periodically, checks the send time of each message, and resends the message if necessary, provided that the maximum number of resends is not exceeded. In the latter case, it removes the message. Two cases may then happen. Either the sender commits an addressing error or the receiver failed. In both cases, the thread informs the coordinator.

4. Demonstration Outline

We coded our prototype in C. It includes a data and parity storage manager, and a query manager. The demonstration shows the use of LH*$_{RS}$ as a highly available distributed data storage system. The focus is to show how an LH*$_{RS}$ file scales up and how it recovers from a multiple bucket unavailability. We show the following operations.

(a) Creation of a *K*-available LH*$_{RS}$ file. We show how a data bucket is split, and how updates propagate to parity buckets. New data buckets are chosen from data buckets connected to the *data buckets multicast group*.

(b) Increase of the high availability of a group, by adding parity buckets. We show the interactions between the new parity bucket and its data buckets group. Each newly created parity bucket is chosen among the parity buckets connected to *parity buckets multicast group*.

(c) Recovery of *k* buckets in the group, into which we introduce failures.

(d) Key search directed to an unavailable bucket. We show that when search time-out elapses, the client alarms the coordinator. The latter checks if it is an addressing error or if the bucket is unavailable. In the latter case, the coordinator starts the recovery process.

(e) Bucket recovery operation . First, the coordinator designates a parity bucket as a recovery manager. The latter recovers records by *slices* of a given size *s*. It requests *s* successive records from each of the *m* data/parity buckets, and recovers the *s* record groups. Then, it requests the next *s* records from each bucket. While waiting, it sends the recovered slice to the spare(s). We show interaction between the coordinator, the spare data buckets, the recovery manager and the available buckets in the group.

(f) Finally, we issue search queries or display the contents of the recovered buckets, to show the recovery operation.

We also show other functions of the prototype: key search queries in normal mode, update queries, their propagation to parity buckets, record recovery using UDP, bucket recovery through UDP, we display data and parity bucket content, various statistics, etc.

Along the demonstration, we show the actual performance factors. These are basically various execution times proving the rapidity of various manipulations. We now resume those we have measured as the basis, on the original configuration [M03] [LMS04] [M04].

5. Performance Results

The hardware test bed consisted of six machines; each one has 512 MB of RAM, with a 1.8GHz Pentium processor under Windows 2K. All the machines were connected to a regular Ethernet configuration with a max bandwidth of 1 Gbps.

For the experimental set up, the record size was (and is) set to 100 bytes and the group size is set to 4 buckets. Performance results degrade for higher values of record size and group size. The best obtained performance results use Reed Solomon codes over the Galois Field GF(2^{16}). We use this field in our demonstration.

The time to create an LH*$_{RS}$ file of 25000 records was 7.896 sec for k = 0, 9.990 sec for k = 1 and 10.963 sec for k = 2. The related average times per record inserted were, 0.32 ms, 0.41 ms, and 0.44 ms for *k* = 0, 1, 2 respectively.

The average individual and bulk search times were 0.2419 ms and 0.0563 ms respectively.

Table 2 presents creation times for a parity bucket (PB) of 31 250 records.

	Total Time	Processing Time	Communication Time
1PB-XOR	2.062	1.484	0.322
1PB-RS	2.103	1.531	0.322

Table 2: Parity bucket creation times in seconds.

To measure the recovery performance, we simulated the creation of an LH*$_{RS}$ group with 4 data buckets and 1, 2, or 3 parity buckets. The group contained 125 000 = 4 * 31 250 data records. The recovery of a single data bucket (DB) uses the first parity bucket and consequently the XOR decoding only. The first line of Table 3 presents this case. Alternatively, the recovery can use another parity bucket, applying the RS decoding (with XORing and Galois field multiplications). The second line of the table shows the measurements for this case. Our numbers prove the efficiency of the LH*$_{RS}$ bucket recovery mechanism. It takes only 1.555 seconds to recover 9.375 MB of data in three buckets.

	Total Time	Processing Time	Communication Time
1DB-XOR	0.720	0.265	0.414
1DB-RS	0.855	0.380	0.400
2 DBs	1.162	0.600	0.434
3 DBs	1.555	0.911	0.464

Table 3: Data bucket recovery times in seconds.

6. Conclusion

Our demonstration shows the prototype implementation of LH*$_{RS}$: a highly available distributed data structure. We show how it actually functions. The efficient distributed storage system that our prototype constitutes can benefit modern data intensive applications: databases, grids, P2P files… Further work, in progress, concerns various aspects of current implementation, evaluation of other encoding and decoding techniques, and the applications of the prototype.

References

[B00] F. Bennour, *Un Gestionnaire de Structures Distribuées et Scalables pour les multiordinateurs Windows: Fragmentation par Hachage*, PhD thesis in French, Paris Dauphine University, 2000.

[BB94] M. Blaum, J. Brady, J. Bruck & J. Menon, *EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures*, IEEE 1994.

[BK95] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby & D.Zuckerman, *An XOR-Based Erasure-Resilient Coding Scheme*, ICSI Tech. Rep. TR-95-048, 1995.

[H94] L. Hellerstein, G.A. Gibson, R.M. Karp, R.H. Katz & D.A. Patterson, *Coding Techniques for handling Failures in Large Disk Arrays*, Algorithmica, 1994, 12, pp.182-208.

[L00] M. Ljungström, *Implementing LH*$_{RS}$*: a Scalable Distributed Highly-Available Data Structure*, Master Thesis, Feb. 2000, CS Dept., U. Linkoping, Suede.

[LS00] W. Litwin & J.E. Schwarz, LH*$_{RS}$, *A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes*, p.237-248, Proceedings of the ACM SIGMOD 2000.

[LMS04] W. Litwin, R. Moussa & J.E. Schwarz, *LH*$_{RS}$ *– A Highly-Available Scalable Distributed Data Structure*, CERIA Res. Rep. May 2004.

[ML02] R. Moussa & W. Litwin, *Experimental Performance Management of LH*$_{RS}$ *Parity Management*, Distributed Data and Structures, 4, (WDAS02) Carleton Scientific, Waterloo, Ontario, CA. 2003, pp. 87-98.

[M03] R. Moussa, *Experimental Performance Analysis of the new LH*$_{RS}$ *Scenarios and Architecture Design*, CERIA Res. Rep., June 2003,http://ceria.dauphine.fr/Rim/comparison0603.pdf.

[M04] R. Moussa, *Contribution à l'étude des Structures de Données Distribuées et Scalables à Haute disponibilité*, PhD thesis in French, Paris Dauphine University, 2004.

[MS04] R. Moussa & J.E. Schwarz, *Design and Implementation of LH*$_{RS}$*: a Highly Available Distributed Data Storage System*, Workshop on Distributed Data and Structures (WDAS04).

[P97] J. S. Plank, *A Tutorial on Reed-Solomon Coding for fault-Tolerance in RAID-like Systems*, Software – Practise & Experience, 27(9), Sept. 1997, pp 995- 1012.

[PGK88] D. A. Patterson, G. Gibson & R. H. Katz, *A Case for Redundant Arrays of Inexpensive Disks*, Proc. of ACM SIGMOD Conf, pp.109-106, June 1988.

[R89] M. O. Rabin, *Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance*, Journal of ACM, Vol. 26, N° 2, April 1989, pp. 335-348.

[S02] T. J.E. Schwarz S.J., *Reed Solomon Codes for Erasure Correction in SDDS*, Distributed Data and Structures, 4, (WDAS02) Carleton Scientific, Waterloo, Ontario, CA. 2003.

[SDDS] http://ceria.dauphine.fr/SDDS-bibliographie.html

[XB99] L. Xu & J. Bruck, *Highly Available Distributed Storage Systems*, Proceedings of workshop on Distributed High Performance Computing, Lecture notes in Control and Information Sciences, Springer Verlag, 1999.

[WK02] H. Weatherspoon & J. D. Kubiatowicz, *Erasure Coding vs. Replication: A quantitative Comparison*, Proceedings of the 1st International Workshop on Peer-to-Peer Systems, March 2002, p.328-338.

Acnowledgements

This work was partly supported by the European Commission project ICONS (*project no. IST-2001-32429*), Microsoft Research, as well as by a scholarship from the Tunisian government.