Design and Implementation of LH*$_{RS}$: a Highly Available Distributed Data Storage System

Rim Moussa

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

Thomas J.E. Schwarz, S.J.

Santa Clara University
USA
TSchwarz@calprov.org

Abstract

The ideal storage system is always available and is incrementally expandable. Existing storage systems are far from this ideal. Affordable computers and high-speed networks allow us to investigate storage architectures that bring us closer to the ideal storage system. We describe a prototype implementation of the highly available scalable distributed data structure LH*$_{RS}$. The scheme allows to recover from a multiple unavailability using a variant of Reed Solomon erasure correcting code. We present the system architecture and experimental performance measurements.

Keywords: High availability, Scalable and Distributed Data Structures, Reed Solomon Codes, Erasure-resilient systems.

1 Introduction

The Scalable and Distributed Data Structures [SDDS] are being developed for computers over fast networks, usually local networks, i.e. for the *multicomputers*. This new hardware architecture is promising and becomes highly popular. In spite of the advantages given by the data distribution layout, vulnerability to failures remains the arena, and accentuates with the increase of the number of machines in the network. Many approaches to build highly available, i.e., fault tolerating, distributed data storage systems have been proposed. They generally fall into the two categories of (*i*) data mirroring, and (*ii*) parity information. In the former approach the storage overhead is prohibitive. The latter approach uses erasure-correcting codes to guard against failures. The simplest codes, e.g. in RAID systems [PGK88], use XOR calculus for the tolerance of a single site failure. For multiple failures more complex codes are needed. These can be the binary codes [H94] for double or triple failure, or character codes. Examples of character codes are: the array codes as the EVENODD code [BB94], the X-code [XB99] or the Reed Solomon codes. The latter appear best at present to deal with multiple failures [R89, BK95, P97, LS00, M00, ML02]. Theoretical proofs demonstrating superiority of erasure resilient systems to replicated systems can be found in [S02, WK02].

Below, Section 2 recalls the LH*$_{RS}$ file structure. Section 3 overviews a proposed architecture for LH*$_{RS}$. Performance results are given in section 4. Finally, section 5 concludes the article.

2 LH*$_{RS}$ Scheme

LH*$_{RS}$ scheme is described with details in [LS00, S02, ML02]. An LH*$_{RS}$ file is subdivided into groups. Each group is composed of *m* Data Buckets and *k* Parity Buckets. The data buckets store the data records, same for parity buckets. Every data record fills a rank *r* in its data bucket. 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 (see Fig. 1(a)). The record grouping has an impact on the data structure of a parity record. The latter keeps track of the data records it is computed from. Fig. 1(b-c) shows each of the structure of a data record and a parity record. The parity calculus is done using Reed Solomon codes.

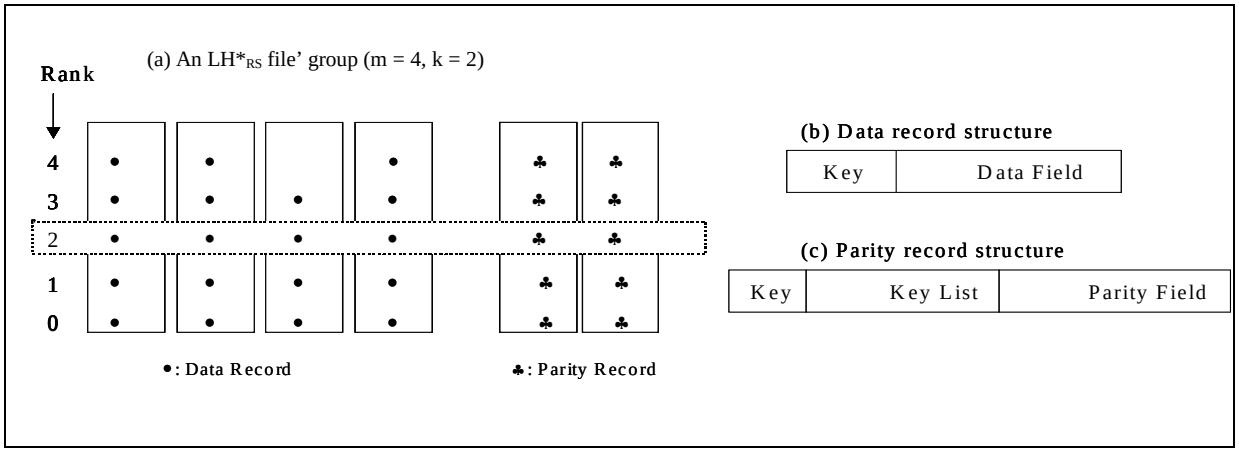


Figure 1: LH*$_{RS}$ file Structure

In the scenarios described below, all the buffers are sent through TCP/IP for performance and reliability concerns demonstrated in [M00, ML02], where we compared TCP/IP-based scenarios to UDP-based scenarios.

2.1 Data Bucket split Scenario

The file starts with one data bucket and *k* parity bucket. It scales up through data buckets' splits, as the data buckets get overloaded. 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 adverts a special entity coordinating the splits, which is the coordinator. The latter designates a data bucket to split.
During a data bucket split process, half of the splitting data bucket contents move to a new created data bucket. As a consequence to the data records' transfer, the data records remaining in the splitting data bucket and those moving get new ranks. So, the parity buckets belonging to (i) *g1:* the splitting data bucket's group, and (ii) *g2:* the new data bucket's group, have to be updated. In that way during the split process, two update buffers are filled respectfully at the splitting data bucket and the new data bucket, and sent to update the parity buckets of each group.

2.2 High availability Scenario

The high availability scenario ensures the increase of the availability of a group, just by adding a parity bucket. The new parity bucket executes at most $x \leq m$ stages, such that *x* is the number of not dummy data buckets in the group *g*. At each stage, the parity bucket updates its contents with respect to each data bucket contents. Each data bucket of the group adds the new parity bucket to the list of its group reliability, and will be able to reflect the client's manipulations on this parity bucket.

2.3 Bucket recovery Scenario

The data buckets' recovery scenario starts at the coordinator level. The coordinator probes all buckets belonging to the group, waits a time-out, then either notifies the application of the impossibility of doing recovery, due to the lack of surviving buckets; or assigns to the first parity bucket replying to the probe the task of failed buckets' recovery. In case of possible recovery, the elected parity bucket is adverted of the bucket's states and addresses. It chooses *m* buckets among surviving one's, in preference parity buckets. That is to let the data buckets attend to application requests.
At each iteration, the recovery manager asks participating buckets to search *slice* records, corresponding to ranks in range [*r, …, r*+*slice*-1]. Then, *r* is incremented of slice. The buckets reply in the limit of the number of records they hold. And, on the receipt of the buffers, data records having same rank are retrieved from the buffers and from the local data structure, to compute missing records. Finally, the recovery manager sends to each spare data bucket its partial recovered contents.

3 System Architecture

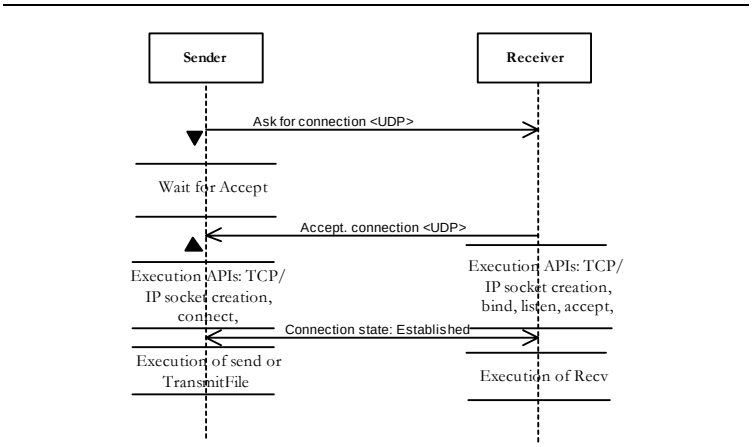
In [M00][ML02], we have implemented our scenarios related to file creation –data bucket split, high availability and buckets recovery, on the top of SDDS2000 architecture [D01]. SDDS2000 architecture was proposed by F. Bennour and A. W. Diène, as an architecture for LH* and RP*. In order to have better performance results, we embedded to SDDS2000 other components, namely: (*i*) an efficient TCP/IP connections handler, (*ii*) flow control and acknowledgements strategy, and (*iii*) a dynamic addressing structure.

To M. Welch and al. [WGBC00] classification of server architectures, our architecture is an hybrid architecture. Indeed, it falls in the spectrum between multithreaded architectures and event-driven architectures, since it combines multithreaded architectures and queues. The threads communicate through events and queues, and concurrent threads are synchronized using common concurrency-programming tools.

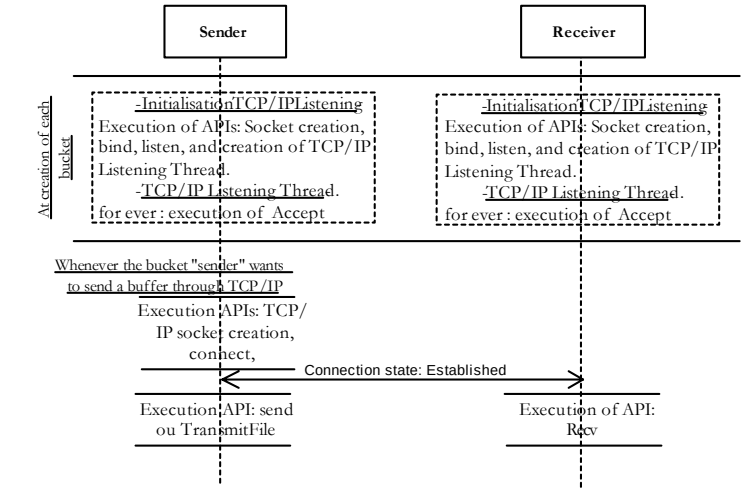
3.1 TCP/IP connections handler

In our last implementation of LH*$_{RS}$ scenarios mapped to SDDS2000 architecture [ML02], the communication time dominates the total time; especially for TCP-based scenarios, i.e., the parity bucket creation scenario and the buckets recovery scenario. To improve the performance results, we have enriched SDDS2000 architecture with an efficient TCP/IP connections handler, and mapped our scenarios to the new architecture.

According to RFC 793 [ISI81] and [MB00], we can open TCP/IP connections in a passive OPEN way, i.e, a process will accept and queue incoming connection requests. The *backlog* parameter designates the number of pending TCP/IP connections. The Windows Sockets 1.1 specification indicates that the maximum allowable value for a backlog is 5; however, Windows 2000 Server accepts a backlog of 200, and Windows 2000 Professional accepts a backlog of 5.
Figure 2, details the way we establish a TCP/IP connection in both of SDDS2000 architecture and the new devised architecture. In SDDS2000, in order to establish a TCP/IP connection, first two messages sent through UDP are exchanged between the two peers. Added to that overhead, the delay underwent to establish the connection while each peer executes appropriate APIs. In our architecture, a TCP listening thread is instantiated on the bucket creation, and handles any incoming connection. Likewise, we don't need to synchronize the peers to establish a TCP/IP connection between them, since in both sides TCP/IP connections are passive OPEN, and the 'sender' peer executes appropriate APIs, without asking the 'receiver' peer to get ready.



(*a*) TCP/IP Connection in SDDS2000



(*b*) TCP/IP Connections Handler

Figure 2: TCP/IP connection handler.

3.2 Flow control and acknowledgement strategy

Diéne [D02] proposed a flow control and acknowledgement strategy, to prevent messages losses under UDP protocol. With respect to his strategy, we designed our flow control and acknowledgement strategy that deploys only one additional thread, while Diéne's strategy deploys *window size+2* threads. The *Window size* parameter is the number of messages that could be sent without acknowledgements, such that each thread handles one message at a time.

For that purpose, each peer has a *Sending Credit* (or *Window size*), that when it reaches zero, the sender stops sending messages, else the sending process pulls a free position from a FIFO managed *Free Positions Queue*, adds the message to *Not Yet Acquitted Messages List*. The message is obviously removed when the corresponding acknowledgement is received; consequently a new position is signaled free, and queued to *Free Positions Queue*. The *Acknowledgement manager Thread* scans periodically the list, checks sending time of each message, and re-sends if necessary the message whenever the maximum number of re-sends is not exceeded. In the last case, the message is removed, and two cases are considered. Indeed, either the '*sender'* peer commits an addressing error or the '*receiver'* peer is failed. In all cases, the coordinator is informed.

3.3 Dynamic addressing structure

In our first implementation, a static table containing the IP addresses of the different involved peers: clients, data/ parity buckets, is used for addressing purpose. We proposed a new scenario to add a data/parity bucket to the file on-line, so that the addressing table evolves accordingly to the file and is not user-fixed.

For that purpose, a bucket depending on its type data or parity bucket, is either connected to the *data buckets multicast group* or the *parity buckets multicast group*. It starts with the *multicast listening thread* and the *multicast working thread*. The former listens to a fixed data or parity multicast port, and queues multicast messages, and the latter processes queued multicast messages. When the bucket receives a multicast message 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 confirmation receipt, disconnects from its multicast group, while the non-selected buckets cancel the instantiation process, and can commit to other invitations.

The new bucket selection scenario, has without doubt changed our architecture, and is applied to each of data bucket split scenario, high availability scenario and bucket recovery.

3.4 Architecture

Hereafter, we briefly describe each functional thread, the overall bucket architecture is illustrated in Figure 3.

- *Multicast Listening Thread*: is a temporary thread. The thread listens to a fixed data or parity multicast port, and queues multicast messages.
- *Multicast Working Thread*: is also a temporary thread. It processes queued multicast messages.
- *UDP Listening Thread*: listens to a fixed UDP port. The latter is deduced from the bucket number.
- *Working Threads*: a working thread processes queued UDP messages.
- *TCP Listening Thread*: accepts and handles multiple TCP/IP connections.
- *Acknowledgement manager Thread*: scans the *Not Yet Acquitted Messages List*, checks *sending time* of each message, and re-sends if necessary the message whenever the maximum number of re-sends is not exceeded, or deletes the message other way.

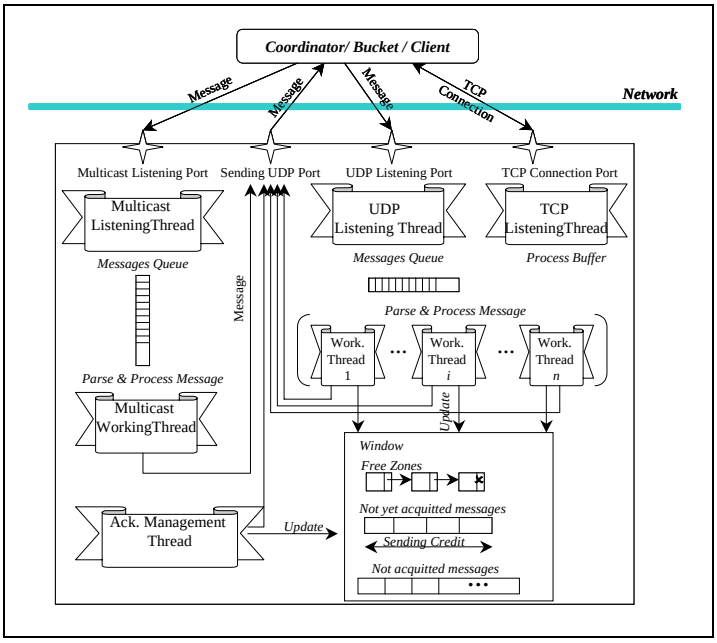


Figure 3: Bucket Architecture

4 Performance Results

The goal of the prototype is to tune and experimentally determine the LH*$_{RS}$ performance characteristics. Our LH*$_{RS}$ implementation improves that presented in [ML02]. It uses distributed RAM memory, and includes a data and parity storage manager, and a basic query manager. Other functionalities are implemented as key-based search queries, data update queries and propagation of updates to parity buckets, record's recovery using UDP, buckets' recovery through UDP, display bucket contents and statistics, etc.

The hardware testbed consists of six machines; each one has 512 MB of RAM, with a 1.8GHz Pentium processor, and runs Windows 2K Server. All the machines are connected to a regular Ethernet configuration with a max bandwidth of 1Gbps. For the experimental set up, the *Record Size* is set to 100 bytes and the *Group Size* is set to 4 buckets. Performance results are expected to degrade for higher values of *Record Size* and *Group Size*. The best obtained performance results are using Reed Solomon codes with encoding and decoding in Galois Field: GF(2^{16}). Experiments details and a full comparison between different architectures and configurations can be found in [M03].
Our Generator matrix is constructed such that its first column is filled with '1's, this reduces Galois field multiplication to simple XOR calculus. Consequently, (*i*) the first parity bucket of each group is XOR-encoded, and (*ii*) in case of one data bucket failure and the first parity bucket is alive, the bucket is recovered using XOR-decoding.

4.1 File creation

Along a synchronous inserts, where the client waits for a reply before issuing the next insert query, the time to create an LH*$_{RS}$ file of 25000 records is 7.896 sec for k = 0, 9.990 sec for k = 1 and 10.963 sec for k = 2. We get better performance results, with the new flow control and acknowledgement strategy described in §3.2. Indeed, for *Window Size* fixed to 5, the time to create an LH*$_{RS}$ file of 25000 records is 4.484 sec for k = 0, 6.969 sec for k = 1 and 8.109 sec for k = 2.

4.2 Search Performances

The key search time is the basic referential of access performance of the prototype, since it does not involve the parity calculus for $k > 0$. We have measured the time to perform random *individual* (synchronous) and *bulk* (asynchronous) successful key searches. For synchronized searches, the client waits for a reply before issuing another search query. While in asynchronous searches, the client sends a flow of search queries to four data buckets. All measures were at the client side.
We have measured the search times in a file of 125000 records, distributed over four buckets. The average individual and bulk search times were 0.2419 ms and 0.0563 ms respectively. Thus the former is about 40 times faster than a disk key search. The latter reaches the speed-up of almost 200 times. The former was bound basically by the server processing speed, while the latter by the client speed.

4.3 Data Record Recovery

The record recovery manager is located at one of the parity buckets. First, it looks for the data record key inside the parity bucket structure, sends search queries to alive buckets, then waits until receipt of replies to compute the missing record, finally it sends the recovered record to the client.
We have measured the recovery times in a file of 125000 records, distributed over four buckets. The timing is measured at the parity bucket and starts when the bucket gets the message from the coordinator, until the recovery of the record. The average data record recovery time is 1.30 ms using XOR decoding and 1.32 using RS decoding. Notice that, the average time to scan our parity bucket to locate the key *c* of the data record was measured to be 0.822 ms. This is the dominant part of the total time as it represents 62% and 64% respectively. If one seeks for faster record recovery, or buckets are much larger, the additional already mentioned index (*c*, *r*) per data bucket at the parity bucket should help. Notice finally that even the basic record recovery times remain significantly faster than for a disk file.

4.4 High Availability

To measure the recovery performance, we create an LH*$_{RS}$ group with 4 data buckets, the group contained 125 000 = 4 * 31 250 data records. Table 1 presents parity bucket (PB) creation times.

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

Table 1: Parity bucket creation times in seconds.

Notice that, (*i*) the time to create the first parity bucket (PB-XOR), using XORing only, is faster than for the other buckets (PB-RS), using the RS calculus, and (*ii*) the communication time represents almost 15% of the total time, while in [M00][M02] it represents 60% of the total time.

4.5 Buckets Recovery

To measure the recovery performance, we create 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 *Slice* parameter varies in the set {1250, 3125, 6250, 15625, 31250}, being respectively {4%, 10%, 20%, 50%, 100%} of a bucket's size.
The recovery of a single data bucket (DB), can use the first parity bucket and consequently the XOR decoding only. The 1st line of Table 2 presents this case. Alternatively, the recovery can use another parity bucket, applying the RS decoding). The 2nd line of the table shows the measures of 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 2: Data bucket recovery times in seconds.

During experiments, in the set of *Slice* parameter values {1250, 3125, 6250, 15625, 31250}, the recovery performances are almost equal. Table 2 reports the average times, for experiments details refer to [M03].

5 Conclusion

We have evaluated an LH*$_{RS}$ file creation time, parity bucket creation, data retrieval in both normal mode and degraded mode, and finally the recovery of more than one data bucket, with respect to the new bucket architecture. Thanks to the TCP/IP connections handler component, communication times are improved of 80% [M03], and no more dominates total times for TCP/IP based scenarios. Experiments prove the efficiency of the proposed scenarios to LH*$_{RS}$ scheme and validate our devised architecture.

Our architectural proposals and scenarios are independent of the erasure resilient code used and data distribution scheme. Further work concerns implementation of other encoding and decoding techniques.

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.

[D01] A W. Diène, *Contribution à la Gestion de Structures Distribuées et Scalables*, PhD thesis in French, Paris Dauphine University, 2002.

[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.

[ISI81] Information Sciences Institute, RFC 793: *Transmission Control Protocol (TCP) – Specification*, Sept. 1981, http://www.faqs.org/rfcs/rfc793.html.

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

[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.

[M00] R. Moussa, *Implantation partielle & Mesures de performances de LH*$*_{RS}$*, Université Paris Dauphine, MSc Report in French, October 2000, http://ceria.dauphine.fr/Rim/dea.pdf.

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

[MB00] D. MacDonal, W. Barkley, *MS Windows 2000 TCP/IP Implementation Details*, http://secinf.net/info/nt/2000ip/tcpipimp.html.

[ML02] R. Moussa & W. Litwin, *Experimental Performance Management of LH*$*_{RS}$ *Parity Management*, WDAS 2002 proceedings, pp. 87-98.

[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*, WDAS 2002 proceedings, pp. 75-86.
[SDDS] http://ceria.dauphine.fr/SDDS-bibliographie.html
[WGBC00] M. Welch, S. D. Gribble, E. A. Brewer, D. Culler, *A Design Framework for Highly Concurrent Systems*, UC Berkeley Tech. Report UCB/CSD-00-1108, April 2000.
[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.
[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.