Experimental Performance Analysis of New LH*$_{RS}$ Scenarios and Architecture Design

Rim Moussa

CERIA Lab. Université Paris IX Dauphine.
http://ceria.dauphine.fr/Rim/EvRim/html
Rim.Moussa@dauphine.fr

July 2003

Table of Contents

1 Introduction

We describe the new architecture and report the performance results relative to the following scenarios:

1. Parity bucket creation scenario,
2. Data bucket's recovery scenario,
3. Data record recovery scenario,
4. Search query scenario,
5. File creation scenario.

The experiments are done in the configurations below:

1. First Configuration:
   The 1st configuration is composed of five computers running at 733Mhz, connected to a 100Mbps network.
2. Second Configuration:
   The 2nd configuration is composed of five computers running at 1.8Ghz, connected to a 1Gbps network.

The factor between CPUs speed is equal to 2.45, and the factor between network bandwidth is equal to 10. So, ideally, we'll have a gain performance of 59.18% in process time, and a gain performance of 90% in communication time.

In comparison with [ML02], our LH*$_{RS}$ implementation has mainly changed at the level of use of TCP/IP connections handling. The new architecture has been adopted in all TCP/IP- based scenarios, namely:

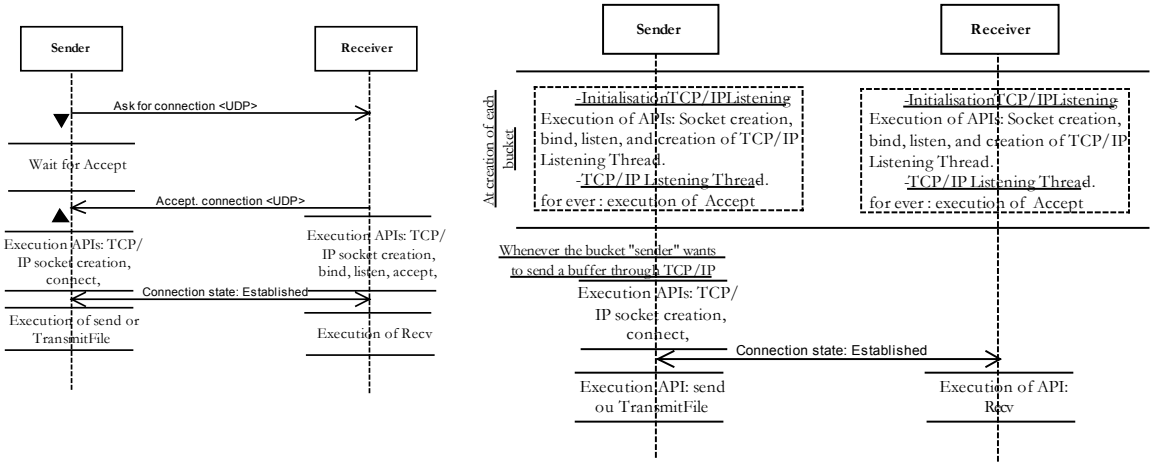
- Data bucket split: move of data records to a new data bucket + parity records update.
- Recovery: receipt of data for reconstruction + sending recovered data.
- High availability: receipt of data for parity calculus.

2 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. 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 an efficient 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 1: TCP/IP connection establishment in both architectures: SDDS2000 and our proposed architecture.

The new architecture is proven efficient through experimentation, and has changed incontestably the performances of three scenarios: data bucket split scenario, bucket recovery scenario, and parity bucket creation scenario.

3 Pre-experimentation

From experiments done in the first configuration, it's turned out that log pre-calculus option and the use of the new generator matrix improve the process time. The main difference between the new generator matrix and the once described in [LS00] is that the 1st column and the 1st line of the new generator matrix are filled with '1's. This should improve parity calculus since Galois field multiplication is replaced with XOR calculus.

3.1 Log pre-calculus

At initialization, a parity bucket performs log pre-calculus. It consists in computing the exponent of each element of the corresponding column in the generator matrix. Log pre-calculus optimizes the multiplication routine, and consequently parity encoding and decoding processes.
To prove the efficiency of the log pre-calculus, we have conducted experiments consisting in the creation of a parity bucket of 31250 records and using RS encoding in GF[2^8]. Under the first configuration, we obtained an average process time of 2.734 sec when log pre-calculus option disabled, and 2.640 sec when log pre-calculus option is enabled. The improvement is of 3.45%, that's why in further experiments the log pre-calculus option is always enabled.

3.2 New Generator Matrix → 1st row of '1's

Parity update occurs either during file creation scenario or parity bucket creation scenario. A parity bucket, processing an update query or any update buffer incoming from the first data bucket of the group, performs XOR calculus instead of multiplication by 1. To valorize this save in parity computation, we proceed to the creation of a parity bucket for different bucket sizes and through two schemes. The first scheme generates a parity bucket using pure RS encoding along the 1st generator matrix, while the second gets profit from the 1st line filled with '1's of the new generator matrix.

The experiments were done under the 1st configuration. We report only the process time to focus on the gain obtained thanks to the line of '1's compared with pure RS calculus.

Bucket Size	*Process time -pure RS calculus (sec)*	*Process time -New Matrix, line of '1's (sec)*	*Improvement (%)*
5000	0,228	0,210	7,895
10000	0,483	0,453	6,211
25000	1,309	1,222	6,646
50000	2,640	2,494	5,530

Table 1: Performance gain in process time thanks to new generator matrix setting, with encoding in GF[2^8].

Bucket Size	*Process time -pure RS calculus (sec)*	*Process time -New Matrix, line of '1's (sec)*	*Improvement (%)*
5000	0,195	0,190	2,564
10000	0,430	0,420	2,326
25000	1,154	1,104	4,333
50000	2,351	2,287	2,722

Table 2: Performance gain in process time thanks to new generator matrix setting, with encoding in GF[2^{16}].

We notice that:

- The average improvement obtained for encoding in GF[2^{16}] is lower than the one corresponding in GF[2^8]. This is due to less number of symbols implied in parity calculus and the fact that XOR is more effective in GF[2^8] than GF[2^{16}].

- The improvement highlighted above and realized when creating a parity bucket from four data buckets group size should decrease when the parameter *group size* increases.

4 Parity Bucket Creation

The high availability scenario ensures the increase of the availability of a group, just by adding a parity bucket. We begin by describing the scenario, and then we report the time to create a parity bucket in different test-beds.

4.1 Parity Bucket Creation Scenario

The skeleton of the algorithm executed by the new parity bucket is showed below at the level of two threads involved in the parity bucket creation scenario,

A working thread……..……
For each not dummy data bucket of my group:
Send a message through UDP "*Would you like to send me your contents* ? "
Wait until all buffers sent by data buckets are received and processed
……

The TCP listeningThread……..
Wait for TCP/IP connections
If there is one, accept it,
Case the content of a data bucket, get the buffer, and process it
If I get all the awaited buffers, Signal the event to the worker thread.
……

We can improve this scenario by relieving the TCP listen thread of the task, consisting in "processing the buffer". But, the parallelism can not be absolute, because the parity bucket data structure should to be under a strong synchronization tool, to not to be updated in the same time by different threads processing the received buffers. In that way, the buffers sent from data buckets have to be processed one after the other. A more sophisticated strategy to get profit from parallelism consists on when one thread finishes the processing of x records having for instance ranks in this range [r, r+x-1]. It allows another thread to process its buffer just for that range. When finishing the last thread will stop waiting for permission, but allows another thread to process the same range. The number of processed before enabling another thread is a parameter in this strategy.

4.2 First configuration

Bucket Size	*Total time (sec)*	*Process time (sec)*	*Com time (sec)*	*Min Com (sec)*	*Max Com (sec)*
5000	1.326	0.172	1.112	0.14	2.324
10000	1.991	0.335	1.59	0.32	2.374

25000	2.598	0.937	1.526	0.731	2.293
50000	3.803	1.905	1.624	1.471	2.092

Table 3: Parity Bucket creation using XOR calculus in GF[2^8].

Bucket Size	*Total time (sec)*	*Process time (sec)*	*Com time (sec)*	*Min Com (sec)*	*Max Com (sec)*
5000	0.589	0.212	0.337	0.130	1.112
10000	1.198	0.434	0.705	0.271	2.363
25000	2.155	1.135	0.875	0.721	1.462
50000	4.200	2.363	1.568	1.493	1.773

Table 4: Parity Bucket creation using the row of '1's calculus in GF[2^8].

In GF[2^8], the performance gains in process time due to the use of the 1st column instead of the other columns of the matrix are of {18.87%, 22.81%, 17.44%, 19.38%} respectively to bucket sizes {5000, 10000, 25000, 50000}

Bucket Size	*Total time (sec)*	*Process time (sec)*	*Com time (sec)*	*Min Com (sec)*	*Max Com (sec)*
5000	0.679	0.186	0.460	0.130	1.753
10000	1.124	0.377	0.687	0.290	2.253
25000	2.121	1.031	0.950	0.721	1.803
50000	3.745	2.055	1.420	1.371	1.473

Table 5: Parity Bucket creation using XOR calculus in GF[2^{16}].

Bucket Size	*Total time (sec)*	*Process time (sec)*	*Com time (sec)*	*Min Com (sec)*	*Max Com (sec)*
5000	0.377	0.202	0.143	0.130	0.161
10000	0.773	0.424	0.291	0.280	0.311
25000	2.177	1.134	0.907	0.731	1.572
50000	3.778	2.107	1.400	1.352	1.472

Table 6: Parity Bucket creation using the row of '1's calculus in GF[2^{16}].

In GF[2^{16}], the performance gains in process time using XOR calculus instead of the other columns of the matrix are of {7.92%, 11.04%, 9.08%, 2.47%} respectively to bucket sizes {5000, 10000, 25000, 50000}

4.3 Second configuration

Bucket Size	*Total time (sec)*	*Process time (sec)*	*Com time (sec)*
5000	0.153	0.109	0.031
10000	0.356	0.256	0.056
25000	0.922	0.638	0.150
50000	1.872	1.316	0.317

Table 7: Parity Bucket creation using XOR calculus in GF[2^8].

Bucket Size	*Total time (sec)*	*Process time(sec)*	*Com time(sec)*
5000	0.184	0.140	0.019
10000	0.426	0.336	0.059
25000	1.085	0.816	0.156

50000	2.228	1.656	0.307

Table 8: Parity Bucket creation using the row of '1's calculus in GF[2^8].

In GF[2^8], the performance gains in process time using XOR calculus instead of the other columns of the matrix are of {22.14%, 23.81%, 21.81%, 20.53%} respectively to bucket sizes {5000, 10000, 25000, 50000}

Bucket Size	*Total time (sec)*	*Process time(sec)*	*Com time(sec)*
5000	0.190	0.140	0.029
10000	0.429	0.304	0.066
25000	1.007	0.738	0.144
50000	2.062	1.484	0.322

Table 9: Parity Bucket creation using XOR calculus in GF[2^{16}].

Bucket Size	*Total time (sec)*	*Process time(sec)*	*Com time(sec)*
5000	0.193	0.149	0.035
10000	0.446	0.328	0.059
25000	1.053	0.766	0.153
50000	2.103	1.531	0.322

Table 10: Parity Bucket creation using the row of '1's calculus in GF[2^{16}].

In GF[2^{16}], the performance gains in process time using XOR calculus instead of the other columns of the matrix are of {6.04%, 7.32%, 3.65%, 3.07%} respectively to bucket sizes {5000, 10000, 25000, 50000}

4.4 Comparison 1st /2nd configuration

We'll compare the obtained results respectively to two components, first CPU component and second network component.

4.4.1 CPU component

Hereafter, we show the improvement in process time for different encoding schemes, elements from GF[2^{16}] or GF[2^8], and 1st column or other columns.

Bucket Size	*1st config.*	*2nd config.*	*Improvement (%)*
5000	0.172	0.109	36.63
10000	0.335	0.256	23.58
25000	0.937	0.638	31.91
50000	1.905	1.316	30.92

Table 11: Comparison between XOR encoding using GF[2^8] in the two configurations.

Bucket Size	*1st config.*	*2nd config.*	*Improvement (%)*
5000	0.186	0.140	24.73
10000	0.377	0.304	19.36
25000	1.031	0.738	28.42

50000	2.055	1.484	27.78

Table 12: Comparison between XOR encoding using GF[2^{16}] in the two configurations.

Bucket Size	*1st config.*	*2nd config.*	*Improvement (%)*
5000	0.212	0.140	33.96
10000	0.434	0.336	22.58
25000	1.135	0.816	28.10
50000	2.363	1.656	29.92

Table 13: Comparison between RS encoding using GF[2^8] in the two configurations.

Bucket Size	*1st config.*	*2nd config.*	*Improvement (%)*
5000	0.202	0.149	26.24
10000	0.424	0.328	22.64
25000	1.134	0.766	32.45
50000	2.107	1.531	27.34

Table 14: Comparison between RS encoding using GF[2^{16}] in the two configurations.

Parity Bucket creation - The 2nd configuration improves the process time needed to create a parity bucket against the 1st configuration by about 27.91%.

4.4.2 Network component

Hereafter, we show the improvement in communication time for different encoding schemes, elements from GF[2^{16}] or GF[2^8], and 1st column or other columns.
The improvement obtained thanks to the 2nd configuration is computed against the minimum communication value we get in the 1st configuration.

----------	*----1st Config ----*	----------	*----- - 2nd config.---*	*Improvement (%)*
Com time(sec)	*Min Com (sec)*	*Max Com (sec)*	*Com. Time (sec)*	
1.112	0.140	2.324	0.031	77.86
1.590	0.320	2.374	0.056	82.50
1.526	0.731	2.293	0.150	79.48
1.624	1.471	2.092	0.317	78.45

Table 15: Comparison between XOR encoding using GF[2^8] in the two configurations.

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {97.21%, 96.48%, 90.17%, 80.48%} for bucket sizes in {5000, 10000, 25000, 50000}.

----------	*----1st Config ----*	----------	*------ 2nd config.---*	*Improvement (%)*
Com time(sec)	*Min Com (sec)*	*Max Com (sec)*	*Com. Time (sec)*	
0.460	0.130	1.753	0.029	77.69
0.687	0.290	2.253	0.066	77.24
0.950	0.721	1.803	0.144	80.00
1.420	1.371	1.473	0.322	76.51

Table 16: Comparison between XOR encoding using GF[2^{16}] in the two configurations.

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {93.70%, 90.39%, 84.84%, 77.32%} for bucket sizes in {5000, 10000, 25000, 50000}.

----------	----1st Config ----	----------	----- - 2nd config.--	Improvement (%)
Com time(sec)	Min Com (sec)	Max Com (sec)	Com. Time (sec)	
0.337	0.130	1.112	0.019	85.38
0.705	0.271	2.363	0.059	78.23
0.875	0.721	1.462	0.156	78.36
1.568	1.493	1.773	0.307	79.44

Table 17: Comparison between RS encoding using GF[2^8] in the two configurations.

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {94.36%, 91.63%, 82.17%, 80.42%} for bucket sizes in {5000, 10000, 25000, 50000}.

----------	----1st Config ----	----------	----- - 2nd config.--	Improvement (%)
Com time(sec)	Min Com (sec)	Max Com (sec)	Com. Time (sec)	
0.143	0.130	0.161	0.035	73.08
0.291	0.280	0.311	0.059	78.93
0.907	0.731	1.572	0.153	79.07
1.400	1.352	1.472	0.322	76.18

Table 18: Comparison between RS encoding using GF[2^{16}] in the two configurations.

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {75.52%, 79.73%, 83.13%, 77%} for bucket sizes in {5000, 10000, 25000, 50000}.

The communication improvements achieved meet our expectations, and realize good performances.

5 Data Buckets' Recovery

5.1 Data Buckets Recovery Scenario

At each iteration, the recovery manager asks participating buckets to search *slice* records, corresponding to ranks: r, …, $r + slice - 1$. 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 slice of recovered contents. All the buffers are sent through a TCP/IP-connection.

To measure the performance results of our scenario, we create a k-available LH*$_{RS}$ file, k = 1,2,3, containing 125000 records. The created file spreads over four data buckets, such

that each bucket holds 31250 records. At each experiment, we simulate the failure of k data buckets, and then we recover them. We have varied the slice size, which is the number of data records recovered per iteration, to determine best performance, $slice \in \{1250, 3125, 6250, 15625, 31250\}$.

5.2 One DB recovery

5.2.1 First Configuration

Slice	Total time (sec)	Process time(sec)	Com Time (ms)	Min Com (sec)	Max Com (sec)
1250	4.654	0.677	3.882	1.631	5.376
3125	5.027	0.701	4.288	1.430	6.649
6250	4.278	0.683	3.575	1.382	5.177
15625	2.498	0.696	1.790	1.422	2.774
31250	2.328	0.683	1.630	1.352	2.473
Avg PT :		0.688			

Table 19: One Data Bucket Recovery using XOR decoding in GF[2^8].

slice	Total time (sec)	Process time(sec)	Com Time (ms)	Min Com (sec)	Max Com (sec)
1250	5.410	0.521	4.438	1.522	7.090
3125	9.238	0.536	7.188	4.797	11.506
6250	4.524	0.536	3.976	1.422	5.037
15625	2.889	0.530	2.351	1.382	2.774
31250	2.591	0.536	2.048	1.352	3.364
Avg PT:		0.532			

Table 20: One Data Bucket Recovery using XOR decoding in GF[2^{16}].

The gain performance in decoding for data bucket recovery using GF[2^{16}] instead of GF[2^8] is equal to 22.67%.

5.2.2 Second Configuration

Slice	Total time (sec)	Process time(sec)	Com Time (ms)
1250	0.746	0.285	0.371
3125	0.707	0.275	0.300
6250	0.664	0.277	0.289
15625	0.739	0.278	0.298
31250	0.738	0.277	0.328
Avg PT:		0.278	

Table 21: One Data Bucket Recovery using XOR decoding in GF[2^8].

Slice	Total time (sec)	Process time(sec)	Com Time (ms)
1250	0,688	0,257	0,391
3125	0,590	0,249	0,305
6250	0,645	0,250	0,293

15625	0,707	0,246	0,305
31250	0,707	0,246	0,335
Avg PT:		0,250	

Table 22: One Data Bucket Recovery using XOR decoding in GF[2^{16}].

The gain performance in decoding done for one data bucket recovery using GF[2^{16}] instead of GF[2^8] is equal to 10.07%.

5.2.3 Comparison 1st /2nd configuration

We'll compare the obtained results respectively to two components, first CPU component and second network component.

5.2.3.1 CPU component

1st config.	*2nd config.*	*Improvement (%)*
0.688	0.278	59.59

Table 23: Comparison of process time of one DB recovery through XOR decoding in GF[2^8].

1st config.	*2nd config.*	*Improvement (%)*
0.532	0,250	53.01

Table 24: Comparison of process time of one DB recovery through XOR decoding in GF[2^{16}].

The 2nd configuration improves the process time needed for one DB recovery (through XOR decoding), against the 1st configuration by 59.59% using GF[2^8], and by 53.01% using GF[2^{16}].

We notice also that in the 2nd configuration the improvement realized using GF[2^{16}] instead of GF[2^8], and performing XOR decoding is of 10.07%. The latter is lower than the corresponding improvement obtained in the 1st configuration (22.67%).

5.2.3.2 Network component

----------	*----1st Config ----*	*----------*	*------ 2nd config.--*	*Improvement (%)*
Com. Time (sec)	*Min Com (sec)*	*Max Com (sec)*	*Com. Time (sec)*	
3.882	1.631	5.376	0.371	77.25
4.288	1.430	6.649	0.300	79.02
3.575	1.382	5.177	0.289	79.09
1.790	1.422	2.774	0.298	79.04
1.630	1.352	2.473	0.328	75.74

Table 25: Comparison of communication time of one DB recovery through XOR decoding in GF[2^8].

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {90.44%, 93%, 91.92%, 79.88%} for slices in {1250, 3125, 6250, 15625, 31250}.

----------	----1st Config ----	----------	------ 2nd config.--	Improvement (%)
Com. Time (sec)	Min Com (sec)	Max Com (sec)	Com. Time (sec)	
4.438	1.522	7.090	0,391	74.31
7.188	4.797	11.506	0,305	93.64
3.976	1.422	5.037	0,293	79.39
2.351	1.382	2.774	0,305	77.93
2.048	1.352	3.364	0,335	75.22

Table 26: Comparison of communication time of one DB recovery through XOR decoding in GF[2^{16}].

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {91.19%, 95.76%, 92.63%, 87.03%, 83.64%} for slices in {1250, 3125, 6250, 15625, 31250}.

Comparison between communication times obtained using GF[2^8] and GF[2^{16}]
In both configuration 1 and configuration 2, there's an overhead due to the use of GF[2^{16}].
For instance,

- In the 2nd configuration the overhead is of {5.12%, 1.64%, 1.37%, 2.30%, 2.09%} respectively for slice in {1250, 3125, 6250, 15625, 31250}.

5.3 Two DBs recovery

5.3.1 First Configuration

slice	Total time (sec)	Process time(sec)	Com Time (ms)	Min Com (sec)	Max Com (sec)
1250	8,719	2,398	6,244	5,769	7,962
3125	6,664	2,402	7,217	5,758	11,474
6250	10,477	2,373	8,079	5,709	9,815
15625	8,536	2,368	5,372	5,678	6,589
31250	8,230	2,424	5,788	5,778	5,798
Avg. PT :		2,393			

Table 27: Two Data Buckets Recovery using RS decoding in GF[2^8].

Slice	Total time (sec)	Process time(sec)	Com Time (ms)	Min Com (sec)	Max Com (sec)
1250	10,255	1,642	8,545	4,507	6,759
3125	7,191	1,647	7,096	5,499	8,960
6250	7,666	1,627	6,027	4,476	7,864
15625	6,369	1,630	4,732	4,426	5,568
31250	6,121	1,635	4,463	4,457	4,466
Avg. PT:		1,636			

Table 28: Two Data Buckets Recovery using RS decoding in GF[2^{16}].

The gain performance in decoding done for recovery of two data buckets using GF[2^{16}] instead of GF[2^8] is equal to 31.63%.

5.3.2 Second Configuration

slice	Total time (sec)	Process time (sec)	Com Time (ms)
1250	1,865	1,248	0,472
3125	1,791	1,235	0,364
6250	1,844	1,261	0,380
15625	1,781	1,255	0,401
31250	1,776	1,250	0,422
Avg. PT:		1,250	

Table 29: Two Data Buckets Recovery using RS decoding in GF[2^8].

slice	Total time (sec)	Process time (sec)	Com Time (ms)
1250	1,234	0,590	0,519
3125	1,172	0,599	0,400
6250	1,172	0,598	0,365
15625	1,146	0,609	0,443
31250	1,088	0,599	0,442
Avg. PT:		0,599	

Table 30: Two Data Buckets Recovery using RS decoding in GF[2^{16}].

The gain performance in decoding for recovery of two data buckets using GF[2^{16}] instead of GF[2^8] is equal to 51.49%.

5.3.3 Comparison 1st /2nd configuration

We'll compare the obtained results respectively to two components, first CPU component and second network component.

5.3.3.1 CPU component

1st config.	2nd config.	Improvement (%)
2,393	1,250	37.27

Table 31: Comparison of process time of two DBs recovery in GF[2^8].

1st config.	2nd config.	Improvement (%)
1,636	0,599	57.33

Table 32: Comparison of process time of two DBs recovery in GF[2^{16}].

The 2nd configuration improves the process time needed for two DBs recovery (through RS decoding), against the 1st configuration by 37.27% using GF[2^8], and by 57.33% using GF[2^{16}].

5.3.3.2 Network component

----------	----1st Config ----	----------	------ 2nd config.--	Improvement (%)
Com. Time (sec)	Min Com (sec)	Max Com (sec)	Com. Time (sec)	
6,244	5,769	7,962	0,472	64.70
7,217	5,758	11,474	0,364	69.42
8,079	5,709	9,815	0,380	67.70
5,372	5,678	6,589	0,401	63.67
5,788	5,778	5,798	0,422	62.77

Table 33: Comparison of communication time of two DBs recovery in GF[2^8].

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {67.39%, 75.60%, 99.97%, 61.60%, 62.84%} for slices in {1250, 3125, 6250, 15625, 31250}.

----------	----1st Config ----	----------	------ 2nd config.--	Improvement (%)
Com. Time (sec)	Min Com (sec)	Max Com (sec)	Com. Time (sec)	
8,545	4,507	6,759	0,519	69.05
7,096	5,499	8,960	0,400	77.69
6,027	4,476	7,864	0,365	70.84
4,732	4,426	5,568	0,443	65.41
4,463	4,457	4,466	0,442	64.06

Table 34: Comparison of communication time of two DBs recovery in GF[2^{16}].

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {83.67%, 82.71%, 78.35%, 67.65%, 64.10%} for slices in {1250, 3125, 6250, 15625, 31250}.

We notice that in both configurations, for the recovery of two DBS the use of GF[2^{16}] improves the communication time.

For instance,

- In the 2nd configuration the improvement is of {31.48%, 30.32%, 29.23%, 25.79%, 25.52%} respectively for slice in {1250, 3125, 6250, 15625, 31250}.

5.4 Three DBs recovery

5.4.1 First Configuration

slice	Total time (sec)	Process time(sec)	Com Time (ms)	Min Com (sec)	Max Com (sec)
1250	14,277	3,766	10,795	6,929	13,810
3125	13,517	3,780	9,799	6,970	14,991
6250	14,631	3,762	10,773	8,030	13,220
15625	11,646	3,798	7,834	6,899	8,773
31250	10,823	3,760	7,032	6,919	7,360
Avg. PT :		3,773			

Table 35: Three Data Buckets Recovery using RS decoding in GF[2^8].

slice	Total time (sec)	Process time(sec)	Com Time (ms)	Min Com (sec)	Max Com (sec)
1250	12,480	2,592	9,811	7,811	11,237
3125	11,017	2,588	8,373	6,279	9,793
6250	10,063	2,598	7,437	5,407	9,895
15625	8,021	2,590	5,421	5,367	5,467
31250	8,615	2,577	6,019	5,388	8,432
Avg. PT :		2,589			

Table 36: Three Data Buckets Recovery using RS decoding in GF[2^{16}].

The gain performance in decoding for recovery of three data buckets using GF[2^{16}] instead of GF[2^8] is equal to 31.38%.

Let's notice that we get an improvement of the same scale of size when recovering two DBs (31.63%) (see §5.3.1).

5.4.2 Second Configuration

Slice	Total time (sec)	Process time(sec)	Com Time (ms)
1250	2,457	1,874	0,459
3125	2,495	1,864	0,407
6250	2,448	1,875	0,380
15625	2,443	1,860	0,442
31250	2,464	1,865	0,459
Avg. PT :		1,868	

Table 37: Three Data Buckets Recovery using RS decoding in GF[2^8].

Slice	Total time (sec)	Process time(sec)	Com Time (ms)
1250	1,589	0,922	0,522
3125	1,599	0,928	0,383
6250	1,541	0,907	0,401
15625	1,578	0,891	0,520
31250	1,468	0,906	0,495
Avg. PT :		0,911	

Table 38: Three Data Buckets Recovery using RS decoding in GF[2^{16}].

The gain performance in decoding for recovery of three data buckets using GF[2^{16}] instead of GF[2^8] is equal to 51.23%.

Let's notice that we get an improvement of the same scale of size when recovering two DBs (52.07%) (see §5.3.2).

5.4.3 Comparison 1st /2nd configuration

We'll compare the obtained results respectively to two components, first CPU component and second network component.

5.4.3.1 CPU component

1st config.	2nd config.	Improvement (%)
3,773	1,868	37.77

Table 39: Comparison of process time of three DBs recovery in GF[2^8].

1st config.	2nd config.	Improvement (%)
2,589	0,911	56.01

Table 40: Comparison of process time of three DBs recovery in GF[2^{16}].

The 2nd configuration improves the process time needed for three DBs recovery (through RS decoding), against the 1st configuration by 37.77% using GF[2^8], and by 56.01% using GF[2^{16}].

Let's notice that we get similar results when comparing 2 DBs recovery using the Galois fields in the two configurations (see §5.3.3.1).

5.4.3.2 Network component

----------	----1st Config ----	----------	------ 2nd config.--	Improvement (%)
Com. Time (sec)	Min Com (sec)	Max Com (sec)	Com. Time (sec)	
10,795	6,929	13,810	0,459	67.20
9,799	6,970	14,991	0,407	71.84
10,773	8,030	13,220	0,380	73.92
7,834	6,899	8,773	0,442	66.10
7,032	6,919	7,360	0,459	64.08

Table 41: Comparison of communication time of three DBs recovery in GF[2^8].

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {78.94%, 79.97%, 80.56%, 70.14%, 64.66%} for slices in {1250, 3125, 6250, 15625, 31250}.

----------	----1st Config ----	----------	------ 2nd config.--	Improvement (%)
Com. Time (sec)	Min Com (sec)	Max Com (sec)	Com. Time (sec)	
9,811	7,811	11,237	0,522	78.25
8,373	6,279	9,793	0,383	78.16
7,437	5,407	9,895	0,401	72.48
5,421	5,367	5,467	0,520	68.85
6,019	5,388	8,432	0,495	79.05

Table 42: Comparison of communication time of three DBs recovery in GF[2^{16}].

If we compute the improvement based on the average communication time we get in the first configuration, we'll obtain {82.68%, 83.63%, 79.99%, 69.16%, 81.24%} for slices in {1250, 3125, 6250, 15625, 31250}.

We notice that in both configurations, for the recovery of two DBS the use of GF[2^{16}] improves the communication time.

For instance,

- In the 2nd configuration the improvement is of {25.25%, 30.16%, 28.94%, 28.52%, 54.57%} respectively for slice in {1250, 3125, 6250, 15625, 31250}.

5.5 Recover 1, 2, 3 Data Buckets, Is the time linear?

5.5.1 Scenario analysis

The parity bucket doing recovery get at most *m-1* slices per iteration (less if some data buckets are dummy), from alive buckets, and that's to recover *f* failed data buckets.
Recovering one more data bucket has a process cost relative to decoding, and a communication cost relative to sending one more buffer per iteration to a spare bucket.
Let:

cc be the communication cost of recovery of one data bucket,
dc be the decoding cost of recovery of one data bucket,

Decoding cost of recovery of *f* data buckets should be equal to *f*dc*, except if we use a different decoding scheme (in particular the case of recovering of 1 DB through XOR decoding, and recovery of 2 DBs through RS decoding)

Communication cost of recovery of *f* data buckets is lower than *f*cc*. Indeed, the communication time spent on receiving buffers from alive buckets is common to recovery of one data bucket and *f* data buckets.

5.5.2 First Configuration

We'll compare the obtained results respectively to two components, first CPU component and second network component.

5.5.2.1 CPU component

1 DB .	*2 DBs.*	*3 DBs*
0.688	2,393	3,773

Table 43: Process time to recover *k* DBs in seconds, *k* ∈ {1, 2, 3}, symbols belong to GF[2^8].

1 DB .	*2 DBs.*	*3 DBs*
0.532	1,636	2,589

Table 44: Process time to recover *k* DBs in seconds, *k* ∈ {1, 2, 3}, symbols belong to GF[2^{16}].

5.5.2.2 Network component

The two tables below show the min values of communication time we obtained.

Slice	1 DB (s)	2 DBs (s)	3 DBs (s)
1250	1.631	5.769	6.929
3125	1.430	5.758	6.970
6250	1.382	5.709	8.030
15625	1.422	5.678	6.899
31250	1.352	5.778	6.919

Table 45: Min Communication time to recover *k* DBs, $k \in \{1, 2, 3\}$, symbols belong to GF[2^8].

Slice	1 DB (s)	2 DBs (s)	3 DBs (s)
1250	1.522	4.507	7.811
3125	4.797	5.499	6.279
6250	1.422	4.476	5.407
15625	1.382	4.426	5.367
31250	1.352	4.457	5.388

Table 46: Min Communication time to recover *k* DBs, $k \in \{1, 2, 3\}$, symbols belong to GF[2^{16}].

The two tables below show the average values of communication time we obtained.

Slice	1 DB (s)	2 DBs (s)	3 DBs (s)
1250	3.882	6.244	10.795
3125	4.288	7.212	9.799
6250	3.575	8.079	10.773
15625	1.790	5.372	7.834
31250	1.630	5.788	7.032

Table 47: Average Communication time to recover *k* DBs, $k \in \{1, 2, 3\}$, symbols belong to GF[2^8].

Slice	1 DB (s)	2 DBs (s)	3 DBs (s)
1250	4.438	8.545	9.811
3125	7.188	7.096	8.373
6250	3.976	6.027	7.437
15625	2.351	4.732	5.421
31250	2.048	4.463	6.019

Table 48: Average Communication time to recover *k* DBs, $k \in \{1, 2, 3\}$, symbols belong to GF[2^{16}].

5.5.3 Second Configuration

We'll compare the obtained results respectively to two components, first CPU component and second network component. The performances are shown to scale.

5.5.3.1 CPU component

1 DB .	2 DBs.	3 DBs

0.278	1,250	1,868

Table 49: Process time to recover *k* DBs, *k* ∈ {1, 2, 3}, symbols belong to GF[2^8].

1 DB .	*2 DBs.*	*3 DBs*
0,250	0,599	0,911

Table 50: Process time to recover *k* DBs, *k* ∈ {1, 2, 3}, symbols belong to GF[2^{16}].

5.5.3.2 Network component

Slice	*1 DB (s)*	*2 DBs (s)*	*3 DBs (s)*
1250	0.371	0,472	0,459
3125	0.300	0,364	0,407
6250	0.289	0,380	0,380
15625	0.298	0,401	0,442
31250	0.328	0,422	0,459

Table 51: Communication time to recover *k* DBs, *k* ∈ {1, 2, 3}, symbols belong to GF[2^8].

Slice	*1 DB (s)*	*2 DBs (s)*	*3 DBs (s)*
1250	0,391	0,519	0,522
3125	0,305	0,400	0,383
6250	0,293	0,365	0,401
15625	0,305	0,443	0,520
31250	0,335	0,442	0,495

Table 52: Communication time to recover *k* DBs, *k* ∈ {1, 2, 3}, symbols belong to GF[2^{16}].

6 Data Records' Recovery

6.1 Data records recovery scenario

The routine "recover one record" follows the steps hereafter:

1. Looks for the data record's key inside the parity bucket structure,
2. Sends search queries to alive buckets,
3. Waits until receipt replies to sent queries,
4. Computes the missing data record,
5. Sends the recovered record to the client.

For performance analysis, we create an LH*$_{RS}$, 1-available file of 125000 records. Then, we simulate the failure of one data bucket. Tables below summarize the performance results of recovering *x* records from the failed data bucket.

The results announced estimate the time needed for a parity bucket (performing the 5 steps) to recover one data record. But, don't take into account the client time-out, that when elapsed the client would advert the coordinator of a possible failure.

6.2 First Configuration

Number of Records	Total Time RS (sec)*	/record RS (ms)	Total Time XOR (sec)	/record XOR (ms)
1000	1,622	1,622	1,587	1,587
5000	8,157	1,631	7,967	1,593
10000	16,434	1,643	16,023	1,602
15000	24,590	1,639	24,060	1,604
20000	32,867	1,643	32,076	1,604
25000	41,079	1,643	40,138	1,606
Average		*1,637*		*1,599*

Table 53: Data Records' recovery using XOR and RS decoding in GF[2^8].

Number of Records	Total Time RS (sec)*	/record RS (ms)	Total Time XOR (sec)	/record XOR (ms)
1000	1,622	1,622	1,587	1,587
5000	8,157	1,631	7,967	1,593
10000	16,434	1,643	16,023	1,602
15000	24,590	1,639	24,060	1,604
20000	32,867	1,643	32,076	1,604
25000	41,079	1,643	40,138	1,606
Average		*1,637*		*1,599*

Table 54: Data Records' recovery using XOR and RS decoding in GF[2^{16}].

	GF[2^8]	GF[2^{16}]	Improvement (GF[2^8]/GF[2^{16}]) (%)
RS	1,612	1,637	1,527
XOR	1,582	1,599	1,063
Improvement (XOR/ RS) (%)	1,861	2,321	

Table 55: Time to recover a record using 1st configuration test-bed.

The degradation of performance due to the use of GF[2^{16}] instead of GF[2^8] is of 1.55% for RS decoding and of 1.07 % for XOR decoding.

6.3 Second Configuration

Number of Records	Total Time RS (sec)*	/record RS (ms)	Total Time XOR (sec)	/record XOR (ms)
1000	2,032	1,305	1,281	1,281
5000	6,492	1,298	6,383	1,277
10000	13,109	1,311	12,890	1,289
15000	19,648	1,310	19,273	1,285
20000	26,257	1,313	25,796	1,290
25000	32,812	1,312	32,281	1,291
Average		*1,308*		*1,285*

Table 56: Data Records' recovery using XOR and RS decoding in GF[2^8].

Number of Records	Total Time RS (sec)*	/record RS (ms)	Total Time XOR (sec)	/record XOR (ms)
1000	1,328	1,328	1,289	1,289
5000	6,750	1,350	6,422	1,284
10000	13,203	1,320	12,968	1,297
15000	19,734	1,316	19,453	1,297
20000	26,367	1,318	26,242	1,312
25000	33,266	1,331	32,516	1,301
Average		*1,327*		*1,297*

Table 57: Data Records' recovery using XOR and RS decoding in GF[2^{16}].

	GF[2^8]	GF[2^{16}]	Improvement (GF[2^8]/GF[2^{16}]) (%)
RS	1,308	1,327	1,432
XOR	1,285	1,297	0,925
Improvement (XOR/ RS) (%)	1,758	2,261	

Table 58: Time to recover a record using 2nd configuration test-bed.

The degradation of performance due to the use of GF[2^{16}] instead of GF[2^8] is of 1.43% for RS decoding and of 0.93 % for XOR decoding.

.

6.4 Comparison 1st /2nd configuration

We notice that, in both configurations, the record recovery time using GF[2^8] is better than when using GF[2^{16}]. At the implementation point, there is only one difference, which is converting the GFElement string to character string, before sending the buffer. The latter point is done at the level of the participating buckets in the recovery process. The parity bucket doing recovery since the receipt of a reply does the conversion to GFElement string. And finally conversion to character is done before sending the recovered record to the client. So the additional overhead is due to data conversion.

	1st config.	*2nd config.*	*Improvement (%)*
XOR + GF[2^8]	1,582	1,285	18,774
RS + GF[2^8]	1,612	1,308	18,859
XOR + GF[2^{16}]	1,599	1,297	18,887
RS + GF[2^{16}]	1,637	1,327	18,937

Table 59: Comparison between the 1st configuration and the 2nd one.

The 2nd configuration improves the time needed for one record recovery, against the 1st configuration by about 19%.

7 Search Query

We measure both parallel searches, during which a client sends a flow of search messages in parallel to the four data buckets, and synchronized search, where the client

waits for a reply before issuing another request. The tables below present the times for a file of size 125000 distributed over 4 buckets. The search times per record are essentially independent of the number of searches.

7.1 First Configuration

Ack Key	Parallel Search total time(ms)	/record(ms)	Synchronized Search total time(ms)	/record(ms)
10000	881	0,0881	3475	0,3475
20000	1893	0,0947	6980	0,3490
30000	2664	0,0888	10475	0,3492
40000	3455	0,0864	14030	0,3508
50000	4186	0,0837	17605	0,3521
60000	4947	0,0825	21190	0,3532
70000	5718	0,0817	24785	0,3541
80000	6479	0,0810	28371	0,3546
90000	7280	0,0809	31966	0,3552
100000	7941	0,0794	35561	0,3556
Nr replies	92353		100000	
Average		0,0847		0,3521

Table 60: Search performances in the 1st configuration.

7.2 Second Configuration

Ack Key	Parallel Search total time(sec)	/record(ms)	Synchronized Search total time(sec)	/record(ms)
10000	0,547	0,0547	2,391	0,2391
20000	1,125	0,0563	4,781	0,2391
30000	1,687	0,0562	7,203	0,2401
40000	2,250	0,0563	9,625	0,2406
50000	2,813	0,0563	12,078	0,2416
60000	3,375	0,0563	14,531	0,2422
70000	3,937	0,0562	17,016	0,2431
80000	4,453	0,0557	19,500	0,2438
90000	5,000	0,0556	22,000	0,2444
100000	5,578	0,0558	24,516	0,2452
Nr replies	99919		100000	
Average		0,0559		0,2419

Table 61: Search Performances in the 2nd configuration..

7.3 Comparison 1st /2nd configuration

	1st config.	2nd config.	Improvement (%)
Parallel Search	0,0847	0,0559	34,002
Synchronized Search	0,3521	0,2419	31,298
Improvement (%)	75,944	76,891	

Table 62: Comparison of search performances

The 2nd configuration improves the search queries performances, against the 1st configuration by about 33%.

8 File Creation

We report the time needed to create an LH*$_{RS}$, k-available file ($k \in \{0, 1, 2\}$). We insert 25000 records, that will be distributed on 4 data buckets along LH* scheme. Each record is 100 bytes.

8.1 First Configuration

	Average |	Improvement (%) New Matrix	Improvement (%) GF[2^8] -> GF[2^{16}]
k = 0, GF[2^8]	13,720		
k = 0, GF[2^{16}]	13,845		0,902853016
k = 1,RS, GF[2^8]	15,773		
k = 1,XOR, GF[2^8]	15,703	0,443796361	
k = 1,RS, GF[2^{16}]	15,883		0,692564377
k = 1,XOR, GF[2^{16}]	15,938	-0,346282189	1,474463546
k = 2, GF[2^8]	17,315		
k = 2, GF[2^{16}]	17,020		-1.733254994

Table 63: Time to create a k-available LH*$_{RS}$ file, $k \in \{0, 1, 2\}$.

There's an insignificant improvement (0.05 %) using the new matrix instead of the old matrix for the schema k = 1. We notice also that this improvement is not permanent from one experiment to another.

The overhead due to the use of GF[2^{16}] instead of GF[2^8] is estimated to 0.33%.

	GF[2^8]	Overhead of 1 PB (%)	GF[2^{16}]	Overhead of 1 PB (%)
k = 0	13,720		13,845	
k = 1	15,703	12,62816022	15,938	13,13213703
k = 2	17,315	10,26555435	17.020	6.357

Table 64: Cost due to a supplement parity bucket.

The Table above shows the cost of using an additional parity bucket. Passing from 0-availability scheme to 1-availability scheme there's an overhead of almost 13%. Updating an additional parity bucket has an overhead of 10%. This is reasonable, cause this overhead should be inferior than the overhead comparing k = 1 to k = 0. Indeed,

- From k = 0 to k = 1, update buffer preparation (at level of the splitting bucket and the new bucket) + send to 1 PB.
- From k = 1 to k = 2, one additional update buffer is sent to 1 PB.

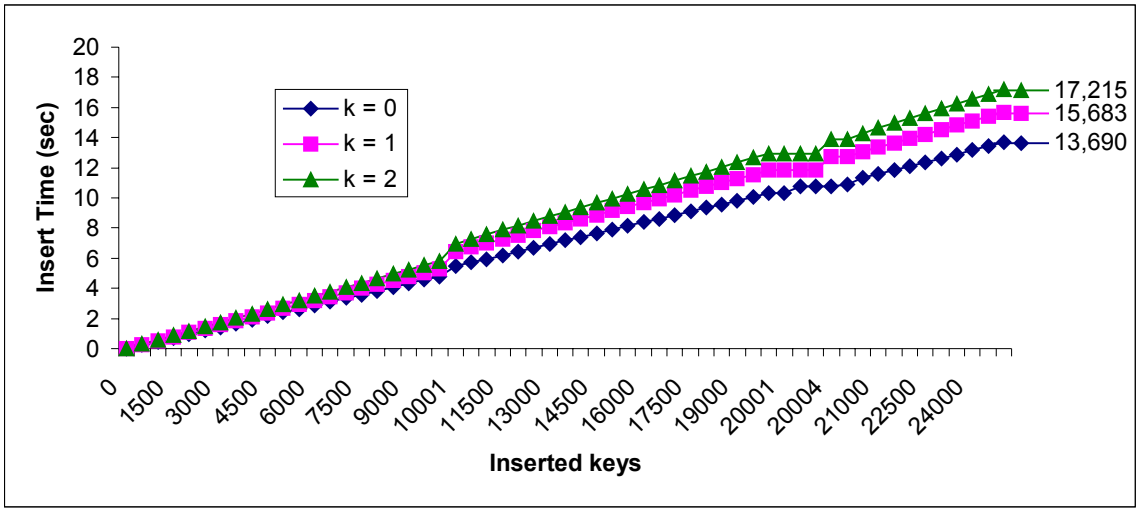


Figure 2: Look of file creation curves using GF[2^8].

Ack Key	k = 0	k = 1	k = 2
500	0,460	0,520	0,582
9000	0,480	0,520	0,580
9500	0,482	0,540	0,580
10000	0,460	0,542	0,582
10001	1,322	1131,000	1161,000
10500	0,520	0,563	0,623
11000	0,482	0,540	0,600
11500	0,480	0,522	0,602
20000	0,482	0,540	0,580
20001	0,000	0,000	0,000
20002	0,880	0,000	10,000
20003	0,000	0,000	0,000
20004	0,000	931,000	961,000
20005	0,322	0,000	0,000
20500	0,800	0,608	0,770
21000	0,542	0,580	0,702
21500	0,520	0,582	0,640
22000	0,520	0,600	0,640
24500	0,520	0,580	0,642
25000	0,522	0,582	0,640

Table 65: Time to insert a record expressed in ms, for a *k*-available file, $k \in \{0, 1, 2\}$.

We notice that the time needed to insert a record is almost constant, except the case when the insert record query causes a data bucket split (cause the data bucket is locked during the split).

8.2 Second Configuration

	Average |	*Improvement (%)* New Matrix	*Improvement (%)* GF[2^8] -> GF[2^{16}]
k = 0, GF[2^8]	7,896		
k = 0, GF[2^{16}]	7,985		1,115
k = 1,RS, GF[2^8]	10,011		
k = 1,XOR, GF[2^8]	9,990	0,209776239	
k = 1,RS, GF[2^{16}]	10,151		1,379
k = 1,XOR, GF[2^{16}]	10,125	0,256140812	1,333
k = 2, GF[2^8]	10,963		
k = 2, GF[2^{16}]	10,974		0,100236924

Table 66: Time to create a *k*-available LH*$_{RS}$ file, $k \in \{0, 1, 2\}$.

There's an insignificant improvement (0.23 %) using the new matrix instead of the old matrix for the schema k = 1, in both schemes GF[2^8] and GF[2^16]. We notice also that this improvement is not permanent from one experiment to another.

The overhead due to the use of GF[2^16] instead of GF[2^8] is estimated to 1%.

	GF[2^8]	Overhead of 1 PB (%)	GF[2^{16}]	Overhead of 1 PB (%)
k = 0	7,896		7,985	
k = 1	9,990	20,96096096	10,125	21,13580247
k = 2	10,963	8,875307854	10,974	7,736468015

Table 67: Cost due to a supplement parity bucket.

The Table below shows the cost of using an additional parity bucket. Passing from 0 availability scheme to 1 availability scheme there's an overhead of almost 21%. Updating an additional parity bucket has an overhead of 8.30%. This is reasonable, cause this overhead should be inferior than the overhead comparing k = 1 to k = 0. Indeed,

- From k = 0 to k = 1, update buffer preparation (at level of the splitting bucket and the new bucket) + send to 1 PB.
- From k = 1 to k = 2, one additional update buffer is sent to 1 PB.

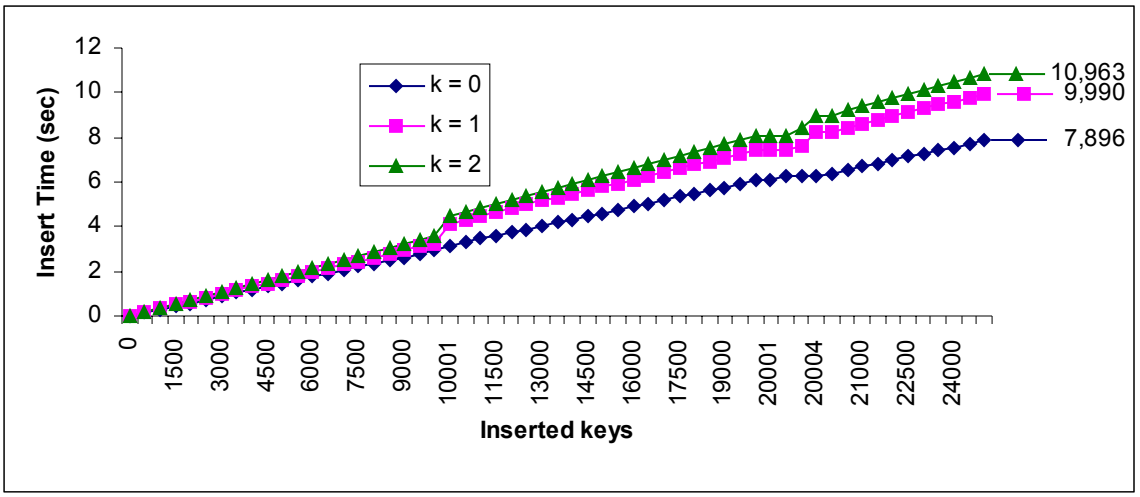


Figure 3: Look of file creation curves using GF[2^8].

Ack Key	*k = 0*	*k = 1*	*k = 2*
500	0,282	0,344	0,342
9000	0,294	0,314	0,376
9500	0,282	0,342	0,344
10000	0,312	0,314	0,374
10001	219,000	890,000	907,000
10500	0,313	0,315	0,375
11000	0,282	0,312	0,344
11500	0,280	0,344	0,374
20000	0,290	0,312	0,376
20002	219,000	0,000	0,000
20003	0,000	250,000	297,000
20004	0,000	625,000	594,000
20005	93,000	0,000	0,000
20500	0,317	0,347	0,378
21000	0,312	0,344	0,376
21500	0,282	0,312	0,374
22000	0,312	0,344	0,376
24500	0,294	0,344	0,344
25000	0,312	0,376	0,374

Table 68: Time to insert a record expressed in ms, for a *k*-available file, $k \in \{0, 1, 2\}$.

We notice that the time needed to insert a record is almost constant, except the case when the insert record query causes a data bucket split (cause the data bucket is locked during the split).

8.3 Comparison 1st /2nd configuration

	1st Config.	2nd Config.	Improvement (%)
k = 0	13,720	7,896	42,44897959
k = 1	15,703	9,990	36,38158314
k = 2	17,315	10,963	36,68495524

Table 69: Comparison file creation (using GF[2^8]) performances in the two configurations.

	1st Config.	2nd Config.	Improvement (%)
k = 0	13,845	7,985	42,32574937
k = 1	15,938	10,125	36,47258125
k = 2	17.020	10,974	35.52291421

Table 70: Comparison file creation (using GF[2^{16}]) performances in the two configurations.

The 2nd configuration improves the file creation performances, compared to the 1st configuration, by about 38.64%.

8.4 Bulk Insert

The client performs bulk inserts when it sends insert queries to data servers, with out waiting for acknowledgements. The file is created quickly, but we expect a message lost rate.
The file creation scenario described is changed in a way that the coordinator don't allow simultaneous data buckets split to occur. From the client side, the file is created in few seconds, exactly the time needed to formulate and send insert queries.

In fact, we have two constraints that affect this scenario:

1. First, an overloaded data bucket adverts its overload one time to the coordinator, and not every time it processes an insert query while it's overloaded. This strategy is adopted to avoid specific process of overload messages incoming to the coordinator.
2. Second, creating a file while acknowledging insert queries allows the coordinator to send orders to split to different data buckets at the same time. This is impossible in bulk insert scenario, cause the coordinator could send an order to split to a data bucket not yet created, for sure in the beginning of the creation of a file, for instance the case of bucket one.

We notice that the record's repartition among the data buckets is different.

9 Conclusion

→ When we map scenarios devised in [M00] for LH*$_{RS}$ scheme, to the new architecture, where a TCP/IP connections Handler was embedded in SDDS2000, the communication performances were incontestably improved. Indeed, if we compare the new implementation results to the one's reported in [ML02], we notice 80% of improvement for TCP/IP based scenarios, namely bucket recovery scenario and parity bucket creation scenario.

→ We compared performances results of encoding/ decoding in two Galois Fields GF[2^8] and GF[2^{16}]. It's turned out that GF[2^{16}] performs better than GF[2^8] in most of the scenarios, despite the light overhead observed in UDP-based scenarios.
→ We conducted experiments in two hardware configurations. The new hardware configuration improves performances of CPU component of 30%, and Network component of 80%.

References

[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.

[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.