Experimental Performance Analysis of LH$_{RS}$ Parity Management

RIM MOUSSA
CERIA Lab.-Université Paris dauphine, France

WITOLD LITWIN
CERIA Lab.-Université Paris dauphine, France

Abstract
We present an implementation of LH$_{RS}$: a high-availability Scalable Distributed Data Structure on Windows 2000. We show experimental determination of the time for file creation, key search, record recovery, parity bucket creation, and data bucket recovery. The measures prove the efficiency of the LH$_{RS}$ scheme, the scalability especially.

Keywords
High Availability, Fault-tolerance, Scalability, Distributed Data Structures, Reed Solomon codes, Multicomputers

1 Introduction
Scalable 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 is becoming highly popular. In spite of the advantages given by the data distribution layout, failures remain the problem, and the vulnerability 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. The latter approach uses erasure correcting codes to guard against failures. The simplest codes, e.g. in RAID systems, 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].
The Scalable and Distributed Data Structure (SDDS) schema called [LS00] provides high-availability server nodes of data buckets by using Reed Solomon codes. We present our contribution to the design and implementation of a prototype system for LH\textsuperscript{*}\textsubscript{RS} files on Windows 2000 multicomputers. The goal of the prototype is to tune the system and experimentally determine the LH\textsuperscript{*}\textsubscript{RS} performance. Our LH\textsuperscript{*}\textsubscript{RS} implementation completes and improves the one presented in [L00]. The entire prototype reuses an existing LH\textsuperscript{*} implementation that does not provide the high-availability [B00]. Our contribution to the prototype design consists in the solutions to the problems encountered by the first prototype, namely, (1) a more reliable data bucket split, (2) a more efficient data buckets' recovery, and (3) a parity bucket creation algorithm increasing the availability of a data bucket group. We show experimental determination of the time for file creation, key search, record recovery, parity bucket creation, and data bucket recovery.

Below, Section 2 recalls the LH\textsuperscript{*}\textsubscript{RS} file structure. Section 3 overviews our contribution to the implementation. Further sections present the performance analysis. Section 8 concludes the article.

The hardware testbed consists of six machines; each has 128 MB of RAM, with a 731 MHz Pentium III processor, and runs Windows 2000. All the machines are connected by a 100 Mbps Ethernet. For the experiments, the record size is set to 100 bytes and the configuration tested consists of one client, four data buckets and \( k \) parity buckets; \( k \in \{0, 1, 2\} \).

2 LH\textsuperscript{*}\textsubscript{RS} Data Structure

LH\textsuperscript{*}\textsubscript{RS} scheme is described in detail in [LS00]. An LH\textsuperscript{*}\textsubscript{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, while the parity buckets store the parity ones. 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 Fig. 1a. 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. 1b-c, shows each the structure of a data record and a parity record. The parity calculus is done using Reed Solomon codes.

The file starts with one data bucket and one 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 advertises a special entity called the split coordinator. The latter designates a data bucket to split.
3 File Creation

3.1 Bucket Splitting

During a data bucket split, half of the splitting data bucket move to a newly created data bucket. As a consequence to the data transfer, the records remaining in the data bucket and those moving get new ranks. The parity buckets belonging to $g_1$, the splitting data bucket group, and to $g_2$, the new data bucket’s group, have to be updated. A first update scenario is proposed in [L00]. It consists in sending update messages to parity buckets. Each parity bucket belonging to $g_1$ receives one message per record to delete, i.e., $b$ delete messages in total and one message per record to insert, i.e., about $b/2$ insert messages, a total of $1.5b$ update messages. Same for each parity bucket belonging to $g_2$, it receives $b/2$ insert messages. Those messages are sent through an unreliable protocol that is UDP without acknowledgements. The parity buckets may be inconsistent if the messages are lost, besides the communication is inefficient since many small messages are sent.

We have implemented a splitting scenario overcoming disadvantages of the above one. We have replaced UDP by TCP/IP, sending a single message with all parity records to update. The update buffer is a collection of the structure $S$:

- Record’s key;
- Record’s Data field;
- Old Rank: the rank occupied by the record; and it has to be deleted from it;
- New rank: the rank of the record after the split, that it has to be inserted on it.

We now give the details of this scenario at the level of the involved buckets:

---The splitting data bucket:
The splitting data bucket £Is in each structure $S$ as follows: If the data record is moving: then it is to be deleted from parity records of rank oldrank in the parity buckets, belonging to the same group as
the splitting data bucket. And, $newrank$ is set to -1. Else, if the data record is staying: the data record is to be first deleted from parity records of rank $oldrank$, and then reinserted at rank $newrank$.

$\rightarrow$ New data Bucket:
Following a data bucket split, the new data bucket receives a buffer, containing about $b/2$ records to insert. While inserting, it prepares a buffer to update the content of its parity buckets. The parity bucket uses the structure $S$, the field $oldrank$ is set to –1, and the field $newrank$ indicates the rank occupied by the record just inserted.

$\rightarrow$ Parity bucket:
Each parity bucket, receiving an update buffer, processes each structure $S$ as follows: (1) If $S.oldrank \neq 1$, delete the data record $\{S.key, S.datafield\}$ from parity record of rank $oldrank$. (2) Else if $S.newrank \neq 1$, insert the data record $\{S.key, S.datafield\}$ in the parity record of rank $newrank$.

3.2 Experimentation description

First we create a file of 25000 records. The file spread on four data buckets, each containing 6250 records. During the file creation, three splits occurred. The first split is of the data bucket numbered 0. Then, data buckets 0 and 1 split approximately the same time. Hereafter, Figure 2 reports on file creation performance.

![Figure 2: Performance Measurements of the creation of a k-available LH*RS File, that spread on 4 data buckets (25000 data records) k ∈ {0, 1, 2}](image)

Following a data bucket split, there is an increase in the insert average time of 40% from $k = 0$ to $k = 1$, and of 7% from $k = 1$ to $k = 2$. The increase is mainly due to the preparation and sending of TCP/IP update buffers (case of $k \geq 1$) from the splitting data bucket and new data bucket to their group reliability. We
estimate that the splitting process may become cumbersome when \( k \) increases, and a more efficient parity buckets’ update strategy is then required. One can release the splitting data bucket and the new data bucket from the task of communicating the updates to all parity buckets of its group.

To prove the scalability of the file creation scenario, we conduct experiments first on the creation of an LH*RS file, that spread on 8 data buckets, and containing 50000 records (see Figure 3), and another file containing 25000 records, that spread on 4 data buckets, but has group size 2. We obtain an insert time per record approximately equal to \{0.40 ms, 0.44 ms, 0.48 ms\} respectively to \( k \in \{0, 1, 2\} \). We notice that the insert time per record is invariant of group size and of the number of records inserted.

![Figure 3: Performance Measurements of the creation of a k-available LH*RS File, that spread on 8 data buckets (50000 data records) k ∈ \{0, 1, 2\}](image)

### 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, then we report the time to create a parity bucket.

#### 4.1 Creation Scenario

The new parity bucket is created in at most \( x \leq m \) steps, such that \( x \) is the number of not dummy data buckets in the group \( g \). At each step, the parity bucket establishes a TCP/IP connection with one data bucket, and the latter sends its contents. The received buffer is so processed for insert/update.

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.
4.2 Parity Bucket Creation Performances

In each experiment, we create a file of \(2.5 \times \text{Bucket Size}\) records. The file spreads on up to four data buckets, each of \(0.625 \times \text{Bucket Size}\) records. The performance results are in Table 1. We highlight *The Communication Time*, i.e. the time spent on establishing TCP/IP connections to receive each data bucket content; and *The Process Time*, i.e. time spent on processing the buffer.

<table>
<thead>
<tr>
<th>BucketSize</th>
<th>Com.Time (sec)</th>
<th>ProcessTime (sec)</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>2.211</td>
<td>0.302</td>
<td>2.523</td>
</tr>
<tr>
<td>10000</td>
<td>2.433</td>
<td>0.611</td>
<td>3.044</td>
</tr>
<tr>
<td>25000</td>
<td>3.185</td>
<td>1.652</td>
<td>4.847</td>
</tr>
<tr>
<td>50000</td>
<td>4.667</td>
<td>3.395</td>
<td>8.062</td>
</tr>
</tbody>
</table>

Table 1: Parity Bucket Creation Performance Results

We notice that the time spent on establishing TCP/IP connections with the four data buckets is relatively high compared to process time. Indeed, the communication time is \(\{87.63\%, 79.93\%, 65.71\%, 57.89\%\}\) for buckets sizes \(\{5000, 10000, 25000, 50000\}\). It appears interesting to choose a larger bucket to amortize TCP/IP connection cost.

5 Basic Key Search Performance

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. Figure 4 presents 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. The synchronized search time measures the server access time. Let’s notice that it is 30 times faster than that to a file on a typical local disk. The parallel search time measures the maximal client speed. Notice also that this search time is about 100 times faster than that to a local disk.
6 Data Record Recovery

6.1 Recovery Algorithm

The routine “recover one record” follows the following steps:

1. Look for the data record’s key inside the parity bucket structure,
2. Send search queries to alive buckets,
3. Wait until receiving replies to sent queries,
4. Compute the missing data record through RS-decoding,
5. Send the recovered record to the client.

6.2 Performance Analysis

We create an LH\textsuperscript{rs}, 1-available file of 125000 records. Then, we simulate the failure of one data bucket. Figure 5 summarizes the performance results of recovering \( x \) records of the failed data bucket.

The time to look for a data record’s key inside a parity bucket structure, of 32150 records, is approximately equal to 0.7\( ms \). It is 56\% of the average time to recover a data record.

Notice that while the record recovery time is 15 times higher than that of a normal key search, it is still 8 times faster than a normal key search on a local disk.

7 Data Bucket Recovery

We describe and we report the performance results of two scenarios devised.
7.1 UDP-based Data Buckets’ recovery Scenario

The data buckets’ recovery scenario starts at the coordinator level. The coordinator probes all buckets belonging to the group, waits a timeout, then either notifies the application of the impossibility of 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 the case of recovery, the elected parity bucket is advertised of the bucket’s states and addresses. It chooses \( m \) buckets among surviving ones, parity buckets being preferred. This is to let the data buckets being available for the application requests, while the recovery is in progress.

To measure the performance of the scenario, we create a \( k \)-available LH*\( _{RS} \) files, with \( k \in \{1, 2, 3\} \). The created file spreads over four data buckets. At each experiment, we simulate the failure of \( k \) data buckets, and then we recover them.

Table 2 reports decoding performance for the recovery of \( k \) data buckets, such that \( k \in \{1, 2, 3\} \) for different bucket sizes.

<table>
<thead>
<tr>
<th>Bucket Size</th>
<th>Bucket contents</th>
<th>1 DB recovery (s)</th>
<th>2 DBs recovery (s)</th>
<th>3 DBs recovery (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>3125</td>
<td>1.742</td>
<td>2.043</td>
<td>2.564</td>
</tr>
<tr>
<td>10000</td>
<td>6250</td>
<td>3.465</td>
<td>4.076</td>
<td>4.797</td>
</tr>
<tr>
<td>25000</td>
<td>15625</td>
<td>8.662</td>
<td>10.185</td>
<td>11.928</td>
</tr>
<tr>
<td>50000</td>
<td>31250</td>
<td>17.375</td>
<td>20.349</td>
<td>23.895</td>
</tr>
</tbody>
</table>

Table 2: Performance Measurements of \( k \) Data Buckets Recovery using UDP protocol, \( k \in \{1, 2, 3\} \).

Figure 6 proves the scalability of the algorithm. Independently of bucket size parameter, we get same recovery time to recover \( k \) data buckets. Indeed, the time
to recover $k$ data records per iteration is approximately 0.557 ms for $k = 1$, 0.652 ms for $k = 2$, and 0.763 ms for $k = 3$. The obtained results show that independently of the value of bucket size, there is constant time needed to compute a record per iteration, which is approximately equal to 0.10 ms for a record size set to 100 bytes. We notice that the time to recover a record decreases when $k$ increases. This is due to the querying messages being factorized. Since, in each iteration, we query $m-1$ buckets, to recover $k$ data records.

Despite the scalability and the linearity in the results obtained for different buckets sizes, we estimate that the UDP-based recovery scenario becomes unsuitable for higher values of bucket size. Thus, we have sketched a new scenario based on TCP/IP.

### 7.2 TCP/IP-based Data Buckets’ recovery Scenario

At each iteration, the recovery manager asks participating buckets to search a slice of $s$ records, corresponding to ranks: $r, \ldots, r+s-1$. Then, $r$ is incremented by $s$. 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, $k = 1, 2, 3$. The created file spreads over four data buckets. 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. Table 3 reports on the performance results of recovering $k$ data buckets, $k \in \{1, 2, 3\}$, using TCP/IP-based recovery.
Recovery of 1 Data Bucket

<table>
<thead>
<tr>
<th>Slice</th>
<th>Tot. Time (sec)</th>
<th>Com. Time (sec)</th>
<th>Process Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1250</td>
<td>50.352</td>
<td>46.968</td>
<td>1.161</td>
</tr>
<tr>
<td>3125</td>
<td>23.073</td>
<td>20.008</td>
<td>1.052</td>
</tr>
<tr>
<td>6250</td>
<td>13.650</td>
<td>10.523</td>
<td>1.043</td>
</tr>
<tr>
<td>15625</td>
<td>8.292</td>
<td>1.986</td>
<td>1.042</td>
</tr>
<tr>
<td>31250</td>
<td>6.700</td>
<td>3.134</td>
<td>1.042</td>
</tr>
</tbody>
</table>

Recovery of 2 Data Buckets

<table>
<thead>
<tr>
<th>Slice</th>
<th>Tot. Time (sec)</th>
<th>Com. Time (sec)</th>
<th>Process Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1250</td>
<td>62.620</td>
<td>57.083</td>
<td>2.495</td>
</tr>
<tr>
<td>3125</td>
<td>29.893</td>
<td>24.081</td>
<td>2.403</td>
</tr>
<tr>
<td>6250</td>
<td>19.529</td>
<td>14.243</td>
<td>2.402</td>
</tr>
<tr>
<td>15625</td>
<td>14.510</td>
<td>8.556</td>
<td>2.364</td>
</tr>
<tr>
<td>31250</td>
<td>12.718</td>
<td>7.580</td>
<td>2.354</td>
</tr>
</tbody>
</table>

Recovery of 3 Data Buckets

<table>
<thead>
<tr>
<th>Slice</th>
<th>Tot. Time (sec)</th>
<th>Com. Time (sec)</th>
<th>Process Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1250</td>
<td>82.619</td>
<td>74.227</td>
<td>3.996</td>
</tr>
<tr>
<td>3125</td>
<td>38.675</td>
<td>30.824</td>
<td>3.815</td>
</tr>
<tr>
<td>6250</td>
<td>26.028</td>
<td>18.387</td>
<td>3.775</td>
</tr>
<tr>
<td>15625</td>
<td>19.638</td>
<td>11.436</td>
<td>3.756</td>
</tr>
<tr>
<td>31250</td>
<td>17.825</td>
<td>9.233</td>
<td>3.735</td>
</tr>
</tbody>
</table>

Table 3: Performance Measurements of $k$ Data Buckets Recovery, $k \in \{1, 2, 3\}$ using TCP/IP protocol.

### 7.3 Performance comparison

The cost of recovery of 31250 records using UDP lasts 17.375 sec. We get better performance results using TCP/IP when slices are either a fifth of the bucket: 13.65 sec, a half: 8.292 sec, or the entire bucket: 6.7 sec. The best result is when we recover the entire bucket. The cost of recovery using UDP is 2.6 times greater than TCP/IP, so the improvement is 260%. Similar remarks hold for multiple data buckets recovery. Indeed, we get better performance results for slice in the range $[b/5, b]$, where $b$ is the bucket size.
8 Conclusion

We have evaluated the LH*$_{RS}$ file creation, parity bucket creation, data retrieval in both normal mode and degraded mode, and finally the recovery of more than one data bucket. Experiments prove the efficiency of the proposed scenarios. They also prove the scalability of the LH*$_{RS}$ scheme. The curves of the total insert time are about linear function of the file size. Hence should remain so for further inserts and files scaling to much larger sizes. Likewise, the record processing times of a normal search or of a recovery remain constant. They should remain so for much further scaling files.

Further work concerns in depth performance analysis and implementation improvements to our parity calculus [S02]. It will also concern the full integration of our system under SDDS-2002 prototype. Finally, we plan to investigate other encoding and decoding techniques.

References


**Rim Moussa** is a PhD student with CERIA Lab., Dauphine University, Pl. du Mal de Lattre, Paris 75016, France. E-mail: Rim.Moussa@dauphine.fr

**Witold Litwin** is the director of CERIA Lab., and professor at Dauphine University, Pl. du Mal de Lattre, Paris 75016, France. E-mail: Witold.Litwin@dauphine.fr

**Acknowledgements** We thank Prof. Thomas Schwarz and Dr. Jim Gray for helpful discussions and comments. This work was partly supported by the research grants from Microsoft Research, and by the European Commission project ICONS project no. IST-2001-32429 as well as by the scholarship from Tunisian government.