

GRZEGORZ ŁUKAWSKI*, KRZYSZTOF SAPIECHA**

EFFICIENT DATA MANAGEMENT
ON A MULTICOMPUTEREFEKTYWNE ZARZĄDZANIE DANymi
W OBREBIE MULTIKOMPUTERA

Abstract

High performance, fault tolerance and scalability are usual requirements for an application running on a multicomputer. The paper presents different variants of centralized SDDS LH* architecture in the light of all the requirements. Hence, the paper briefly summarizes already published features of SDDS that concern data scalability and fault tolerance, and then introduces a new option for SDDS called throughput scalability that can balance workload of nodes of a multicomputer. Finally, having met all the requirements for efficient management of data on a multicomputer the SDDS schemes are estimated as for the time and memory overhead.

Keywords: multicomputer, scalability, fault tolerance, SDDS

Streszczenie

Wysoka wydajność, odporność na błędy i skalowalność to typowe wymagania aplikacji dla multikomputerów. W artykule zaprezentowano różne odmiany struktur SDDS LH* o architekturze scentralizowanej w świetle wszystkich tych wymagań. Podsumowano znane już możliwości struktur SDDS dotyczące skalowalności danych i odporności na błędy oraz przedstawiono nową funkcjonalność SDDS nazwaną skalowalnością przepustowości, pozwalającą na zrównoważenie obciążenia węzłów multikomputera. Ostatecznie, po spełnieniu wszystkich wymagań w kwestii efektywnego zarządzania danymi w obrębie multikomputera, struktury SDDS są analizowane pod względem kosztów czasowych i pamięciowych.

Słowa kluczowe: multikomputer, skalowalność, odporność na błędy, SDDS

* Dr inż. Grzegorz Łukawski, Katedra Informatyki, Wydział Elektrotechniki, Automatyki i Informatyki, Politechnika Świętokrzyska.

** Prof. dr hab. inż. Krzysztof Sapiecha, Katedra Informatyki Technicznej, Wydział Inżynierii Elektrycznej i Komputerowej, Politechnika Krakowska.

Paper submitted for publication: 29.04.2010

1. Introduction

Efficient management of distributed resources is one of the most important factors deciding about performance of an application running on a multicomputer. Physical memory is one of the resources. The application stores data in the memory. Accessing the data located either in distributed RAM or on disks should be fast. Moreover, the data should be scalable. Results of computations involving the data should be dependable. 'Natural' redundancy of a multicomputer might be used for implementation of fault tolerance. Sometimes as a side effect of improvement of the performance and data scalability. This explains why all these three factors should be considered simultaneously when efficient data management is required.

High performance, fault tolerance or scalability may be accomplished at different levels of operation of the multicomputer (hardware, middleware, programming model, application level, etc.). There are low-level software libraries and programming models, helpful in building scalable applications. Distributed Shared Memory (DSM) is a well known solution, implemented in hardware and/or software [1]. DSM simplifies the memory management for multicomputer applications and has been used for years as an alternative to typical 'message-passing' multicomputer programming model. DSM makes the development of the applications much easier compared to their development using typical message-passing schemes based on MPI or PVM [2]. There are programming schemes designed especially for multicomputers where the throughput scalability (performance) is considered. For example, Google MapReduce [3], allows for utilization of a large number of nodes and processors. It divides a problem recursively into sub-problems (map stage) and distributes among multicomputer nodes. Next, results are put together to form the global solution (reduce stage). Unfortunately, MapReduce may be applied for specific, divide-and-conquer type problems and algorithms only. At higher level (an application), memory and throughput scalability is frequently present in enterprise-class software. Scalable Oracle Coherence [4] approaches the problem of multicomputer memory management. RAM distributed over a multicomputer is used as a cache memory for application's working data set. Oracle Coherence uses peer-to-peer architecture along with load balancing for optimal RAM utilization. IBM WebSphere [5] is another example of throughput scalable application. In WebSphere, web servers are multiplied by 'cloning' (cloned servers are identical). The cloning may concern single server where multiple instances of WebSphere are run. This is called 'vertical cloning'. It helps to fully utilize the CPU power. Another possibility is the 'horizontal cloning' where identical WebSphere instances are run on different machines. Beside the throughput scalability, horizontal cloning introduces fault tolerance, as there are multiple instances of identical web server. While distributed RAM of a multicomputer can offer gigabytes of memory, hard disks distributed among multicomputer nodes can offer terabytes of space for data. Although much slower than RAM, hard disks have some obvious advantages, such as much less cost and the persistence of data. However, applying multiple independent disks as a data store is ineffective (in terms of organization of data structures and fault tolerance). So some RAID schemes, in a form of NAS (Network Attached Storage), could be an option. When the scalability is in focus, already developed filesystems, such as Lustre [6] may be used.

From among the various approaches to data management on a multicomputer the one namely Scalable Distributed Data Structures (SDDS) [7] seems to be particularly

interesting because of their fast access to data. SDDS is more than a programming model but is not enough to build a stand-alone application. SDDS is a middleware and may be applied for different purposes, where scalable memory is required. SDDS stores a file of records using so called buckets maintained by multicomputer nodes. There are two basic variants of addressing records in SDDS file: RP* where Range Partitioning [8] is applied and LH* where Linear Hashing [7] is used. The latter one may use decentralized split control with so called token, or centralized with Split Coordinator [7]. The idea was very interesting and useful. However, in [7] only scalability of data stored in a SDDS file was introduced and evaluated. This caused that numerous variants of basic architecture of SDDS file were developed. They concern different aspects of data processing and fault tolerance. Architectures of SDDS file with fault tolerant data stored in buckets were presented in [9–13], but with fault tolerant control mechanisms in [14–17]. Different hashing functions were considered in [18–20]. SDDS files with multi-key addressing, especially useful in databases, were discussed in [21–23]. Also peer-to-peer networks were adapted to work with SDDS [24]. Furthermore, the SDDS file was used as a block device for Linux data storage [25] and some adoption of the idea of SDDS file were used for implementation of scalable store of objects [26, 27]. However, up to now the schemes lack performance scalability for nodes of a multicomputer.

The aim of the paper is to present different variants of centralized SDDS LH* scheme in the light of all the three requirements for applications that are to run on a multicomputer. Hence, the paper briefly summarizes already published features of SDDS that concern data scalability and fault tolerance, and then introduces a new option for SDDS called throughput scalability that can balance workload of the nodes. Finally, having met all the requirements for efficient management of data on a multicomputer the SDDS schemes are estimated as for the time and memory overhead.

Summarizing, an effective data management scheme for a multicomputer should meet the following requirements:

- data is stored in RAM (fast access),
- size of the store could grow as more and more space for data would be required (data scalability),
- access to data is fast even if more and more clients operate on the data (throughput scalability),
- data is safe in case of a software/hardware faults (fault tolerance),
- the scheme is simple and easy for implementation in a programming language.

The paper is organized as follows: in chapter 2 data scalability, along with basic SDDS architecture, is presented, chapter 3 presents novel idea of throughput scalability, fault tolerant SDDS variants are described in chapter 4, chapter 5 contains overhead analysis. The paper ends with conclusions.

2. Data scalability

SDDS stores a file of *records* using so called *buckets* maintained by multicomputer nodes. Each record is equipped with an unique *key*. Each bucket's capacity is limited. If a bucket's load reaches some critical level, it performs a split. A new bucket is created

then and a half of data from the splitting bucket is moved into a new one. This is so called *data scalability*.

For accessing data stored in SDDS file, another file component called a *client* is used. It may be a part of an application. There may be one or more clients operating the file simultaneously. The client is supplied with so called *file image* used for bucket addressing. This file image not always reflects actual file state, so client may commit *addressing error*. Incorrectly addressed bucket *forwards* such message to the correct one, and sends *Image Adjustment Message (IAM)* to the client, updating his file image, so it will never commit the same addressing error again [7]. All the SDDS file components are connected through a *network*. For record addressing, the following modified Linear Hashing (LH*) [7] is used

$$h_i(C) = C \bmod 2^i$$

where:

C – a record key,
 i – a file/bucket level.

Because the number of buckets in SDDS file is usually not a power of 2, for bucket and record addressing two hashing functions with levels i and $i+1$ are used simultaneously. Value i is called *file level*, each bucket has its own *bucket level* denoted j (Fig. 1). After successful bucket split, its level j increases.

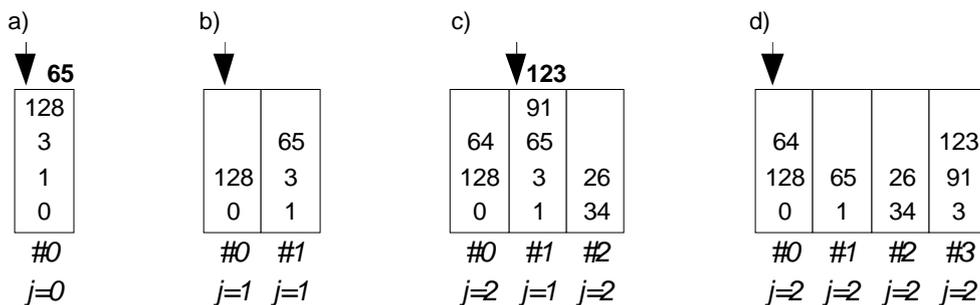


Fig. 1. Example SDDS LH* file evolution, an arrow shows split pointer (n) location

Rys. 1. Przykład ewolucji pliku SDDS LH*, strzałka wskazuje kolejne wiaderko do podziału (n)

LH* bucket splits must be performed in specific order. So called *split pointer* (n) specifies next bucket to split (Fig. 1). In centralized SDDS LH* for split synchronization an extra file component, called *Split Coordinator* (SC) is used. For basic SDDS LH* single SC is enough for the whole file, no matter if it consists of ten or million buckets (single SC is unfortunately a danger for the data integrity in faulty environments [14]). Overloaded bucket sends collision message to the SC. The SC decides which bucket to split and sends split message to the corresponding bucket pointed by n . SC is the only file component, holding actual values of n and i ¹.

¹ The client's file image consists of two local values denoted i' and n' , usually different than global i and n .

In SDDS, data access efficiency is very high, as the buckets are usually stored in RAM memory of each node. Another important advantage of the SDDS is outstanding data processing speed. Instead of a single server holding and processing all the data, many servers are used simultaneously. If a SDDS file consists of N servers (buckets), the data may be processed even N -time faster compared to a single server, as the query is processed by every bucket in parallel [7].

3. Throughput scalability

The backbone of the SDDS is fast communication network. The number of messages coming up to a bucket may be very high. Moreover, some records of SDDS file may be more frequently accessed than the other ones, such as even keys of records are more often used than the odd ones. In such a case, some buckets (and obviously servers storing these buckets) may be overactive because of very frequent access.

In SDDS an access of a client to the proper bucket is very fast. Next task is to ensure fast access to the proper record inside of the bucket. One of the most efficient data structures, which may be used for single bucket management, is the B+ tree. To offer maximum “key search” efficiency, the records are organized into a search tree. B+ trees are commonly used in databases, where “index” may have a form of a B+ tree, especially for the master key of each record. This is actually very fast.

Unfortunately, even if B+ trees are used and all the operations are “key search”, a bucket performance may drop down. In case of many clients operating the same bucket, data processing speed decreases linearly, along with the number of simultaneously connected clients. A test for 1Gbit Ethernet and 1 to 8 clients was run. Clients were operating simultaneously the same single bucket. The results show that the time spent on retrieving 20'000 of 60KiB records increases almost linearly along with the number of clients. Lonely client finished his request in about 20 seconds, while for 8 clients the processing time reached 90 seconds.

Performance of the server (and hence the bucket) does not rely only on a speed of the access to data. For some applications, there is no way to operate only on keys and indexes of stored data. For example, image matching/recognition (e.g. for astronomical photos) or voice/speech recognition relies on pattern matching and similar algorithms. Moreover, databases used for such purposes may consist of thousands gigabytes, should be quickly accessible and offer high data processing performance. Data set is often read in such a case, and the database may even be read-only. A test was run for a database containing 1,27 GiB of data. For “key search” the time of data processing was not measurable. The requests were processed in time shorter than 1ms. However, for “fixed record searches”, where the query concerned a part of a non-indexed value (such as a few middle numbers from the whole phone number) average speed of data processing was 12.73 records per second. For “full record search” where the query required to search the whole record/value using pattern matching algorithm (such as a particular letter to be found in a name) average speed of data processing was even slower and counted 0.64 records per second.

Summarizing, too active buckets may be the source of bottlenecks in SDDS file. To avoid the bottlenecks an overactive bucket should perform a split, even if it still has enough space for new records.

3.1. Reference counter

In addition to typical SDDS data-driven splits, query-driven splits are introduced for the extended SDDS architecture. For determining which bucket should split, due to high activity, each bucket is supplemented with so called *reference counter (RC)*.

The idea of virtual memory page reference counter is adopted for SDDS. Instead of every page/record having independent counter, single counter is used for the whole SDDS bucket. This assumption simplifies the bucket structure and saves memory. Moreover, as it will be proven, single bucket counter is enough to perform query-driven splits in SDDS.

The reference counter is just a single integer value, initiated with 0 just as the bucket is created. With every operation performed by a particular bucket (data insert, delete, find etc.) *RC* value is incremented. To keep track of each bucket activity in the mean of time, every *RC* should be zeroed after some predefined time window, 10 minutes for example. The *RC* value after each time-out reflects every bucket activity in the last 10 minutes, and if previous *RC* values are recorded, during longer period of time.

Depending on the particular SDDS implementation, in case of a bucket split, one of the three following strategies may be applied:

- *RC* may be set to zero for the splitting bucket and the new bucket,
- the splitting bucket and the new bucket gets half of the *RC* value, because each split moves about half of the bucket capacity into a new one,
- *RC* value may be divided accordingly to the real number of moved records.

3.2. Split decision

Too high current *RC* value is another reason for performing a split, in addition to bucket overload (data scalability). The decision whether to split may use some predefined *RC* threshold value (maximum *RC*), or may be taken after dynamic evaluation, taking into consideration other bucket's *RC* values, server resources, daytime, etc. (Fig. 2).

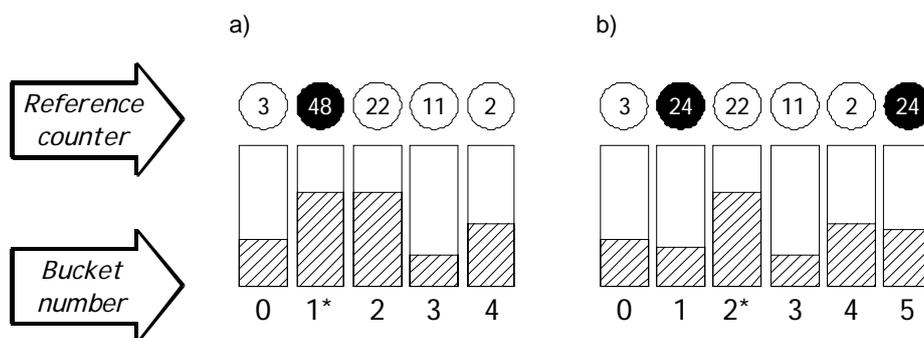


Fig. 2. Overactive bucket split, a bucket with number marked with a star denotes next bucket to split (*n* pointer), the file before (a) and after split (b)

Rys. 2. Podział nadaktywnego wiaderka, wiaderko z numerem oznaczonym gwiazdką to kolejne wiaderko do podziału (wskaźnik *n*), plik przed podziałem (a) i po podziale (b)

Especially, the number of time-consuming data operations should be analyzed, before computing particular *RC* value. More active buckets should be selected and split, even if they have available space for new data.

3.3. The scaling strategies

Classic SDDS LH* file scales along with incoming data, as it was described in chapter 2. This will be called *across scaling* later in this paper. Unfortunately, if some buckets are split due to higher activity, this way of scaling may lead to creation of nearly empty or even totally empty buckets (Fig. 3).

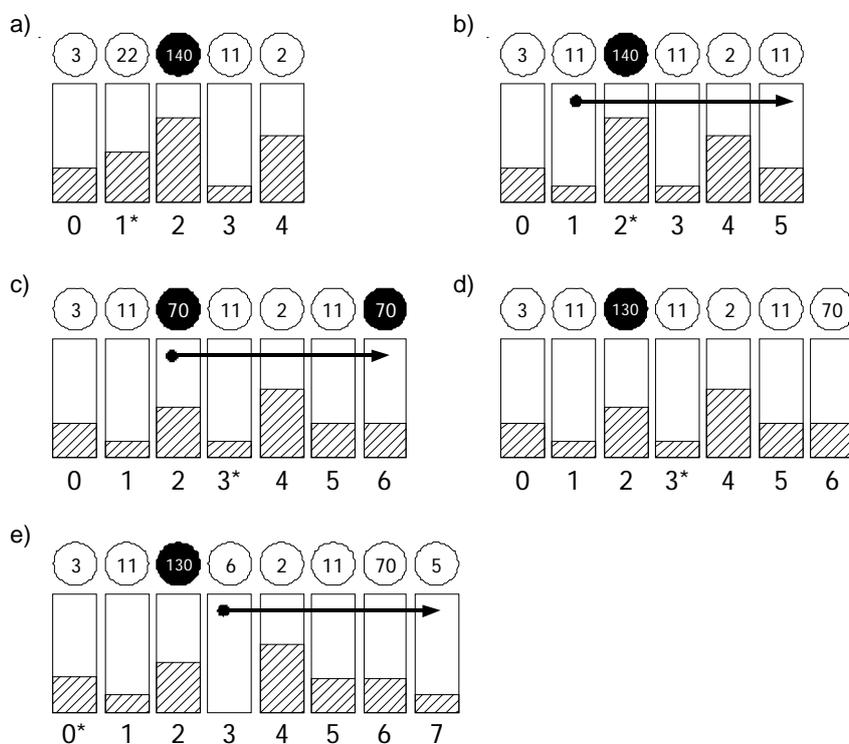


Fig. 3. Overactive bucket #2 and file splits leading to empty and nearly empty buckets

Rys. 3. Nadaktywne wiaderko nr 2 i podziały pliku prowadzące do powstania pustych lub prawie pustych wiader

As it is shown on Fig. 3a, overactive bucket #2 sends 'collision' message to the Split Coordinator. According to the basic SDDS LH* rules (chapter 2), next bucket to split is not the overactive one, but bucket number 1. Data scalability in SDDS requires that splits are made in specific order, otherwise the LH* hashing would fail. On Fig. 3 next bucket to split, with respect to the basic SDDS LH* rules, is marked with a star.

The bucket with number 1 splits (Fig. 3b), because the *RC* value for bucket 2 is still high, another split is performed, concerning the overactive bucket number 2 (Fig. 3c). Unfortunately, the bucket number 2 gets some more requests and becomes overactive again (Fig. 3d). The file splits again, almost empty bucket number 3 splits and becomes empty (Fig. 3e). Because the bucket 2 is not the one which was split, the process continues. As a result, there is a lot of wasted bucket space.

Another possibility is so called *up scaling*. Overactive bucket (called *primary bucket* later in this paper) may create its own replica, and half of all active clients may be redirected there (Fig. 4a-e). This way, data processing performance of a bucket may be theoretically doubled, similarly to 'horizontal cloning' present in IBM WebSphere [5]. In case of high activity, more replicas of a single bucket may be created, similarly to SDDS LH^*_M architecture [9].

Scaling strategy should use both scaling methods. Both are good enough for increasing the data processing performance. Across scaling should be a priority, but if there is a possibility to create empty bucket (Fig. 3), up scaling should be applied. If a bucket is overactive because of insert operations, across scaling should always be applied. Insert operations increase bucket load, so overload will happen, leading to classic SDDS split, at this moment high bucket activity should be ignored. If other kinds of queries are processed by the bucket in question, especially search operations, up scaling would give the best results.

This way or another, the Reference Counter should be distributed along splitting or replicating buckets, according to rules given in section 3.1.

The bucket replica, in case of a need to across split, is a perfect candidate for becoming a new regular file bucket. After updating the level *j* and deleting about half of the unneeded records, the replica may join the SDDS file and become a regular bucket (Fig. 4c-e). Moreover, in case of failures the bucket replica may be used for data recovery, as it contains all records of the original bucket (rules defined for LH^*_M architecture may be used in this case [9]).

3.4. Client redirection

After successful splitting an overactive bucket should redirect some of the clients to the newly created bucket. In case of an *across split*, clients use the new bucket according to basic SDDS rules, as described in chapter 2, so no further actions are required. In case of an *up-split*, the bucket should choose which clients should be redirected to the new bucket. The client should simply update its address tables and replace bucket address with the replica address, just as it was defined for the LH^*_M architecture [9].

Because every new client's address table contains address of original bucket it is easy to control client redirection and balance the load between two or more mirrored buckets. Each new client contacts the original bucket firstly, so decision whether redirect a new client to the replica is considered by the original bucket, with respect to his current *RC* value. For taking the best decision, the original bucket should know his replica(s) current Reference Counter(s). This may be obtained with a special query send to the replica, which will supply his current *RC* value with the reply message. Another possibility is to attach current *RC* value to some, or maybe all, propagated messages sent between the bucket and its replica(s).

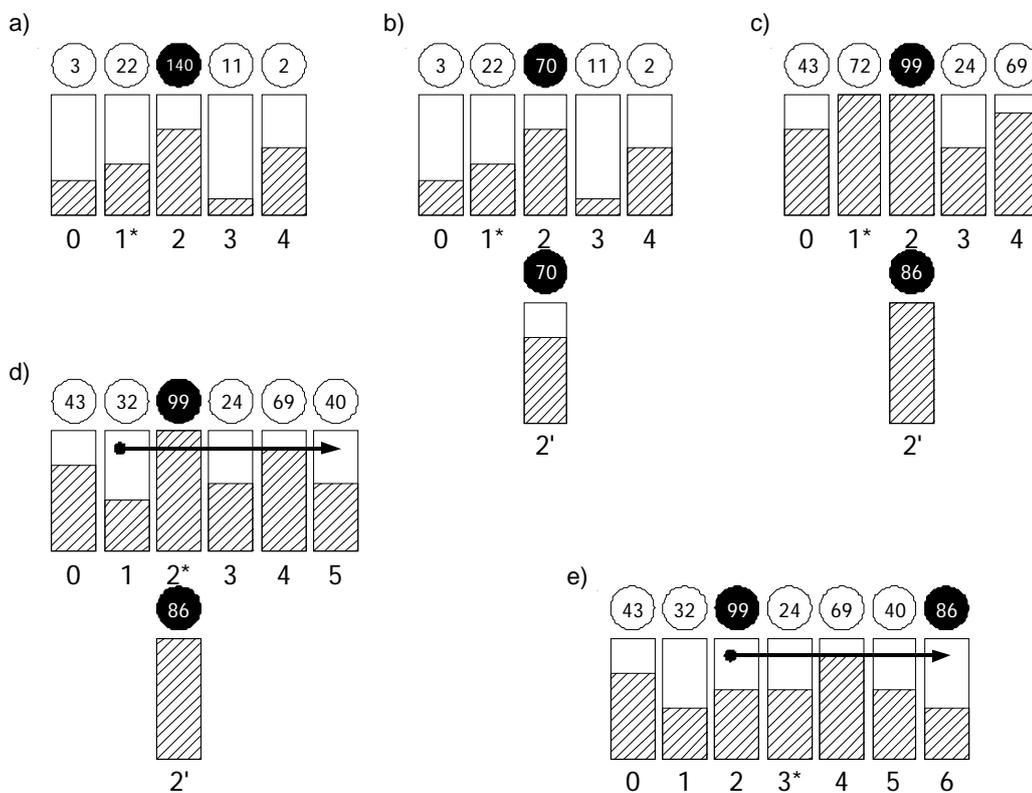


Fig. 4. Up scaling for overactive bucket (a–b) and replica used as a new ordinary bucket after data overload (c–e), replica 2' becomes bucket 6

Rys. 4. Skalowanie wzwyż nadaktywnego wiaderka (a–b) i replika użyta jako nowe zwykłe wiaderko po przepełnieniu danymi (c–e), replika 2' staje się wiaderkiem numer 6

4. Fault tolerance

There are numerous SDDS variants with different types of fault tolerance. All of them can be divided into two groups: concerning data fault tolerance and control fault tolerance.

Data fault tolerance techniques focuses on keeping the data intact. In case of a data loss (bucket failure), redundant buckets holding copy of the data or parity data may be used for full recovering the bucket. Data fault tolerance uses many techniques similar to RAID schemes for hard disks.

Control fault tolerance focuses on making the control mechanisms tolerant to faults. If the data stored in the buckets is intact, but addressing rules fail, the data may be impossible to retrieve and in fact should be considered as lost. Control fault tolerance uses different redundancy schemes and protocols for masking faults concerning addressing and split control of SDDS.

4.1. Data fault tolerance

The problem of data fault tolerance is exhaustively discussed in [9–13]. The following solutions were proposed:

- LH^*_M (LH^* with mirroring) – the whole file has a copy – a mirror, and every file operation is doubled. Each client and each server (bucket) is able to access every machine in the first and second file limits.
- LH^*_S (LH^* with segments) – every record is striped at bit or attribute level into $k > 1$ segments. Each segment is put into distinct LH^* file and goes into bucket stored on different server. There is another segment containing parity bits for each record used to recover lost record's segment.
- LH^*_g (LH^* with records grouping) – the group is a structure made of at most k records (where k is a predefined file parameter). Group members always are stored in different buckets, regardless of file size and splits. Each group is armed with a parity record, which may be used to recover one group member (in case of bucket failure). It is possible to prepare more than one parity record for every group (each in different bucket) for recovering more than one group member or lost parity record. Groups are organized with grouping function, such as a simple division modulo k function

$$g = a \bmod k$$

where:

- g – the record's group number,
- a – the record's first insertion bucket address.
- LH^*_{SA} (LH^* with scalable availability), LH^*_{RS} (LH^* using Reed Solomon codes) – group organization is slightly different from previous LH^* variant. The group number is computed based on actual record's bucket address (in LH^*_g it was the record's first insertion bucket address). If some splits are performed, group composition changes, so the parity records must be updated then. As the LH^*_g , the LH^*_{RS} scheme may store more than one parity record for one record group. In LH^*_{SA} scheme, each group may be equipped with only one parity record. High-availability is achieved by adding one record to several record groups.

4.2. Control fault tolerance

The tolerance to control faults, concerning wrong record placement in file space, is achieved by adding several functions or redundancy in different SDDS components [14–17]. The following mechanisms are proposed for centralized SDDS LH^* :

- *Backwarding* is an addition to *forwarding* defined for basic SDDS LH^* [7]. If a client commits addressing error, backwarding along with forwarding ensures that the misplaced record reaches correct bucket at last, no matter which bucket was initially addressed.
- *Double Split Coordinator* along with *JCT* (Job Comparison Technique), forces each of the two SCs to compare every decision with the other SC's decision (Fig. 5a). This ensures that SC fault are detected. Moreover, a bucket may perform a split, only if it receives two identical messages from these two SCs, so the SDDS file structure may not be damaged by faulty SC [14].

- *Triple Split Coordinator with TMR* (Triple Modular Redundancy), uses three identical SCs (Fig. 5b). As for JCT, every decision is compared with every other SC's decision. Faults are detected, and the invalid SC may be selected by voting. As for JCT, a bucket performs a split only if it receives at least two identical messages, so single SC faults may be successfully masked [14].

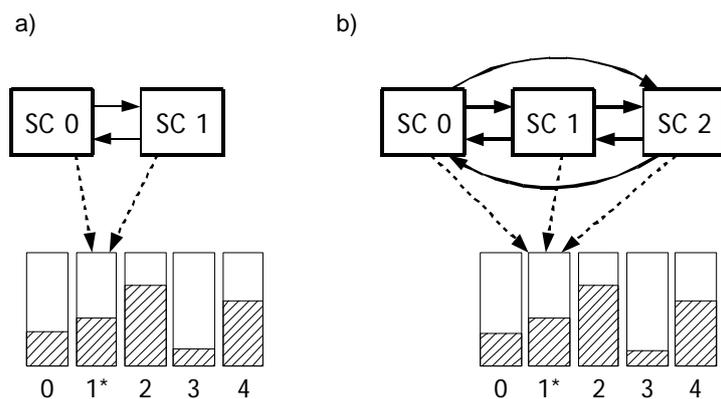


Fig. 5. Double with JCT (a) and triple with TMR (b) Split Coordinator communicating with each other

Rys. 5. Podwójny Koordynator Podziałów korzystający z JCT (a) i potrójny korzystający z TMR (b) komunikujący się z pozostałymi Koordynatorami

- *Variable number of Split Coordinators* includes dynamic reconfiguration. The system consisting of two SCs (JCT) may switch to triple SC (TMR) in case of faults [17]. This may be a cheaper solution for masking single SC faults.
- *SC purging* – faulty SCs (in case of permanent faults) may be permanently removed from SDDS file and replaced with a fresh SC instance. In case of further faults the procedure may be repeated, so the SC set always consists of correctly working SCs only [17].
- *Stand-by Split Coordinator(s)* – a stand-by SC working along with other SC instances, may be especially useful for tolerating transient faults [17]. In case of permanent faults and dynamic reconfiguration, stand-by SC may be quickly plugged into the SDDS file.
- *Multiple Split Coordinator with NMR* (N-Modular Redundancy) – the idea of TMR may be expanded to 5, 7 and more SCs, for tolerating multiple faults.

There are some problems with multiple SC schemes, concerning SC directory required for buckets and SC parameters recovery. These problems are exhaustively analyzed in [17] and are not raised in this paper.

5. Comparison of time and memory overhead

Let a file consists of *records* each of them equal in size and identified with the help of a unique *key* (keys are integers ascending from 0), and a multicomputer is built of N *nodes*. Then the simplest way of distributing the file onto the nodes is to split it into N parts and

put each of them into RAM memory of another node, forming so called *buckets*. Keeping buckets in RAM ensures efficient access to data and high data processing speed.

As the data is distributed between N nodes, a simple hashing function may be used for bucket addressing, such as *modulo*

$$a = C \bmod N$$

where:

- a – the destination bucket number,
- C – the record key,
- N – the number of multicomputer nodes (and so buckets).

Table 1

Overhead of different data management strategies

Variant	Memory overhead	Throughput overhead
The basis	None	None
Basic SDDS LH*	None	Up to additional 50% for splits and data scalability
Throughput scalable LH*	Space for additional Reference Counters (one for each bucket). Mirrored buckets after <i>up scaling</i>	Additional split operations for overactive buckets ¹⁾
LH* _M	Additional 100% space for each mirror file	Write/delete operations must be propagated to every mirror ¹⁾
LH* _S	Up to 30% additional space for parity records	Every record must be split and parity bits computed for write. Every record must be reconstructed for read ¹⁾
LH* _g	Up to 30% additional space for parity records	Parity bits must be computed for write ¹⁾
LH* _{SA} /LH* _{RS}	Up to 50% additional space for parity records (depends on the chosen file structure and the number of parity files)	Parity records must be computed for write. Parity records must be relocated after split ¹⁾
Basic LH* with double SC	None	Additional messages for multiple SC (Fig. 5a), up to 0.2% (depends on the bucket capacity) ¹⁾
Basic LH* with triple SC	None	Additional messages for multiple SC (Fig. 5b), up to 0.4% (depends on the bucket capacity) ¹⁾

¹⁾ Includes throughput overhead for basic SDDS LH*.

Though simple such scheme of data distribution is very inflexible. Capacity of such a file is equal to the sum of capacities of all nodes. No file growth is available. When the file shrink some of the buckets may become empty what makes that their corresponding nodes become idle. If one of the buckets holds extensively accessed records, it could

become a bottleneck. Also, if some of the buckets fail the data will be lost without a trace as the data are stored in RAM.

Simple scheme described above, even though impractical for intensive software systems, can be used as the basis for evaluation of overhead for different SDDS schemes. Costs of data scalability or fault tolerance of software might be decisive for most of engineers when considering implementation. Table 1 contains estimations of time and memory overhead for different SDDS file management strategies. 'Memory overhead' means additional memory required for a particular variant of SDDS. 'Throughput overhead' represents additional computing power/number of messages required for a variant to work. There are possible additional mixed variants, such as LH^*_s with triple SC and so on.

6. Conclusions

The idea of Scalable Distributed Data Structures is an interesting solution to the multicomputer memory management problem applied in many large-scale applications. Its main advantages are the outstanding data processing speed (the data may be processed concurrently by many multicomputer nodes) and scalability. There are many SDDS variants supplemented with different fault tolerance mechanisms, ensuring high availability of the data, while keeping the throughput high (Table 1).

The paper summarizes all known features of SDDS and supplements it with throughput scalability. The last one makes it possible to better balance the performance of SDDS file still keeping all its advantages. As scalability, high performance and fault tolerance should come together, time and memory overhead for each of these features are estimated.

SDDS-based structures, which are discussed in the paper, allows for efficient management of a set of data on a multicomputer. As the data are (usually) stored in RAM, the access to the data is fast. The file is data-scalable, and with the help of simple algorithms it may become a throughput-scalable one. Even if the data resides in RAM, the structure may be fault-tolerant.

Main disadvantages of the SDDS are the necessity for having unique key value for each record (hard to achieve in some applications), constant bucket capacity and problems with choosing proper LH^* hashing function and key value for optimal bucket load (otherwise the SDDS file may hold many empty buckets and waste space). There is also a minor problem with shrinking, and the fact that there are only a few practical SDDS implementations.

References

- [1] Nitzberg B., Lo V., *Distributed shared memory: A survey of issues and algorithms*, Computer, 1991.
- [2] Gropp W., Lusk E., *PVM and MPI are Completely Different*, Argonne National Lab, 1998.
- [3] Dean J., Ghemawat S., *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004.

- [4] *Oracle Coherence Knowledge Base Home*: <http://coherence.oracle.com>
- [5] Ueno K., Alcott T., Blight J., Dekelver J., Julin D., Pfannkuch C., Shieh T., *WebSphere Scalability: WLM and Clustering*, IBM Redbooks, September 2000.
- [6] *Lustre a Network Clustering FileSystem*: <http://www.lustre.org>
- [7] Litwin W., Neimat M.-A., Schneider D., *LH*: A Scalable Distributed Data Structure*, ACM Transactions on Database Systems ACM-TODS, December 1996.
- [8] Litwin W., Neimat M.-A., Schneider D., *RP*: A Family of Order-Preserving Scalable Distributed Data Structures*, 20th Intl. Conf on Very Large Data Bases (VLDB), 1994.
- [9] Litwin W., Neimat M.-A., *High-Availability LH* Schemes with Mirroring*, Intl. Conf. on Coope. Inf. Syst. COOPIS-96, Brussels 1996.
- [10] Litwin W., Neimat M.-A., *LH*s: a High-availability and High-security Scalable Distributed Data Structure*, IEEE Workshop on Research Issues in Data Engineering, IEEE Press, 1997.
- [11] Litwin W., Risch T., *LH*g: a High-availability Scalable Distributed Data Structure through record grouping*, U-Paris 9 Tech. Rep., May 1997.
- [12] Litwin W., Menon J., Risch T., *LH* Schemes with Scalable Availability*, IBM Almaden Research Rep., May 1998.
- [13] Litwin W., Schwarz T., *LH*RS: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes*, CERIA Res. Rep. & ACM-SIGMOD 2000, Dallas.
- [14] Sapięcha K., Łukawski G., *Fault-tolerant Protocols for Scalable Distributed Data Structures*, Springer-Verlag LNCS 3911, 2006.
- [15] Łukawski G., Sapięcha K., *Fault tolerant record placement for decentralized SDDS LH**, Springer-Verlag LNCS 4967, 2008.
- [16] Łukawski G., Sapięcha K., *Software Functional Fault Injector for SDDS*, GI-Edition Lecture Notes in Informatics (LNI), ARCS06 Workshop Proceedings, 2006.
- [17] Łukawski G., Sapięcha K., *Multiple Split Coordinator for fault tolerant SDDS*, Proceedings of the 3-rd International Conference: ACSN 2007.
- [18] Gupta V., Modi M., Pimentel A.D., *Performance Evaluation of the LH*lh Scalable, Distributed Data Structure for a Cluster of Workstations*, SAC 2001.
- [19] Devine R., *Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm*, Springer-Verlag LNCS 730, 1993.
- [20] Zegour D.E., *Scalable distributed compact trie hashing (CTH*)*, Information Software Technology, 2004, 923-935.
- [21] Litwin W., Neimat M.-A., *k-RP*s: A Scalable Distributed Data Structure for High-Performance Multi-Attribute Access*, Proceedings of the fourth international conference on Parallel and distributed information systems (DIS 1996).
- [22] du Mouza C., Litwin W., Rigaux P., *SD-Rtree: A Scalable Distributed Rtree*, IEEE 23rd International Conference on Data Engineering (ICDE 2007).
- [23] Boukhelef D., Zegour D.-E., *IH*: A New Hash-Based Multidimensional SDDS*, Distributed Data Structures 4, Records of the 4th International Meeting (WDAS 2002).

- [24] Zhang Z., Mahalingam M., Xu Z., Tang W., *Scalable, Structured Data Placement over P2P Storage Utilities*, 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04).
- [25] Chrobot A., Łukawski G., Sapiecha K., *Scalable Distributed Data Structures for Linux-based Multicomputer*, 7th International Symposium on Parallel and Distributed Computing, (ISPDC'08).
- [26] Hidouci W.K., Zegour D.E., *Actor oriented databases*, WSEAS Transactions on Computers, Vol. 3, No. 3, 2004, 653-660.
- [27] Bedla M., Sapiecha K., *Scalable Store of Java Objects Using Range Partitioning*, Proceedings of the 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009.