Annales UMCS Informatica AI 3 (2005) 273-283

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Fault-tolerant control for Scalable Distributed Data Structures

Krzysztof Sapiecha, Grzegorz Łukawski[*]

Department of Computer Science, Kielce University of Technology,
Al. 1000-lecia Państwa Polskiego 7, 25-314 Kielce, Poland

Abstract

Scalable Distributed Data Structures (SDDS) can be applied for multicomputers. Multicomputers were developed as a response to market demand for scalable and dependable but not expensive systems. SDDS consists of two components dynamically spread across a multicomputer: records belonging to a file and a mechanism controlling record placement in the file. Methods of making records of the file more or less fault-tolerant have already been given. Methods of making the mechanism controlling record placement in the file fault-tolerant have not been studied, yet, although it seems that this is more important for the system dependability than record fault tolerance. Faults in control may lead an application to crash, while record data faults may cause invalid computations at most. In the paper a fault-tolerant control for SDDS is given. It is based on an application of Job Comparison Technique along with TMR. Time overhead due to redundancy introduced is estimated, too.

1. Introduction

For over 10 years the computer market has been dominated by PC class computers [1]. The most effective PCs based on SMP architecture reached performance compared to mainframes from the beginning of 90s [2, 3]. But now, after years of pursuing maximum performance, computer designers focused their efforts on providing scalability and dependability of computer systems based on PCs [1, 2].

Pushed to the limits of computing power, SMP systems do not meet modern requirements, providing limited solutions for scalability and dependability for more demanding applications. NUMA, COMA architectures [4] and multicomputers were developed as a response to market demand for scalable and dependable but not expensive systems (with cost located between PCs and mainframes). In such distributed systems, effective data and process relocating is a crucial task. This relocation may be caused by the system reconfiguration

[*]Corresponding author: *e-mail address*: g.lukawski@tu.kielce.pl

procedure after operating fault detection or by an application scaling to meet users requirements. SDDS was developed for the latter case [5].

SDDS consists of two components dynamically spread across the multicomputer: records belonging to the file and mechanism controlling the record placement in the file space. Methods of making fault-tolerant records in the file were discussed in [6-11]. Methods of making fault-tolerant mechanism controlling record placement in the file have not been studied yet, although it seems that this problem is very important for system dependability. Faults causing wrong record placement can lead whole application to crash, while faults concerning record data can cause invalid computations at most. Moreover, methods of data protection against operational faults have evolved for years and there is a lot of literature on this subject [12].

Controlling record placement in a file is the SDDS specific mechanism. It is spread between SDDS servers, their clients and dedicated process called the split coordinator, running on any of the SDDS servers.

In this paper extended SDDS architecture with the fault-tolerant record placement is introduced and estimated. Section 2 introduces principles of basic SDDS scheme. Section 3 presents what has already been done in making SDDS fault-tolerant. Section 4 describes a new fault-tolerant SDDS architecture. Efficiency of the new SDDS architecture is estimated in Section 5. The paper ends with conclusions.

2. Scalable Distributed Data Structure (SDDS)

A *record* is the least SDDS element. Every record is identified with a *key* unique for a file. Records equipped with keys are loaded into *buckets*. The buckets are stored on machines (computers, multicomputer nodes) called the *servers*. Every server stores one or more buckets (depending on supplies and system strategy). All the servers can communicate with each other. If a bucket capacity is exceeded it performs a *split*, moving about half of the records to a new bucket.

Any number of machines called *clients* can simultaneously operate SDDS file. Each client does not know anything about other clients' existence. Every client has its own *file_image* (information about an arrangement and a number of buckets), not exactly reflecting actual file state. A client may commit *addressing error*, sending a query to an inappropriate server (bucket). If that happens, the server *forwards* the query to the appropriate bucket and the client receives *Image Adjustment Message* (IAM), updating this client's file image near to the actual state. A client never is going to commit the same addressing error twice.

There is a single dedicated process called the *split coordinator* (SC), controlling bucket splits, communicating with servers only. The clients, servers and the split coordinator are connected together with a *net*.

SDDS file grows as the amount of required storage space increases[1]. There is no need for central directory. Splitting and addressing rules are based on a modified linear hashing method (LH*) [5].

According to LH* rules, the set of buckets is addressed with a pair of *hashing functions* h_i and h_{i+1} (i = 0, 1, 2...). Function h_i distributes keys into 2^i bucket addresses. An example of such a function is a division modulo *x* function:

$$h_i(C) = C \bmod 2^i \,. \tag{1}$$

As the records are inserted, the file grows and splits overloaded buckets into two buckets, then the h_{i+1} function replaces h_i. A special pointer n specifies which of the hashing functions should be used (h_i or h_{i+1}) and points next bucket to be split.

Real values of *n* and *i* may be stored in two ways: centralized and decentralized. In the former case the values are computed and stored by a specified machine (process) known to all servers – the split coordinator. The latter case means that there is no such coordinator, the servers send each other a *token*, which allows for splitting a bucket actually holding it (an analogy to *n* pointer).

More detailed description of LH* and its variants may be found in [5-11].

3. SDDS and fault-tolerance – previous work

Two factors are crucial for SDDS dependability: fault-tolerance of the data stored in records and fault-tolerant record placement in the file space. The former problem is exhaustively discussed in [6-11]. The following solutions were proposed:

- *LH*$_M$ *(LH* with mirroring)* – the whole file has a copy – a *mirror*, and every file operation is doubled. Every client and every server are able to access every machine in the first and the second file limits.
- *LH*$_S$ *(LH* with segments)* – every record is striped at bit level into k>1 *segments*. Each segment is put into distinct LH* file and goes into the bucket stored on a different server. There is another segment containing *parity bits* for each record used to recover lost record segment.
- LH*$_G$ (LH* *with records grouping*) – the group is a structure made of at most *k* records (*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 *x* function:

[1]If all the buckets are stored in *RAM memory* then SDDS highly improves data access efficiency [5].

$$g = a \bmod k\,, \tag{2}$$

where g is the record group number, a is the record first insertion bucket address.

– LH*$_{SA}$ (LH* *with scalable availability*), LH*$_{RS}$ (LH* *using Reed Solomon codes*) – group organization is slightly different from the previous LH* variant. The group number is computed based on actual record bucket address (in LH*$_{G}$ it was the record 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. Fault tolerant control for SDDS

Fault-tolerant record placement mechanism is crucial for proper SDDS functioning. Data fault may lead to some invalid computations, while controlling fault may bring SDDS application crash.

4.1. LH* controlling mechanism specification

The LH* controlling mechanism is distributed between all the SDDS clients and servers.

The client functions are as follows:

– Computing destination bucket address (may be wrong) for a given key. No matter what kind of operation is performed (insert, delete, read, modify, search), client's file image is used to compute a bucket address for the current record.
– Computing physical address of a server maintaining a required bucket. The message is sent then.
– Receiving a response message.
– Updating a client's file image based on the received IAM message, if there was any.

The server functions are as follows:

– Receiving incoming messages (sent by clients or other servers).
– Verifying the destination bucket address – to check if the current server is maintaining a required record:
  – if yes, the given operation is performed and the results are sent back to the client;
  – if no, a probable destination address is calculated and the message is forwarded. The message is extended with information about the client's invalid file image (there is a need to send IAM message to client).

– IAM message sending – if there was information about client's wrong file image.
– Sending a collision message to the split coordinator if any of the bucket's capacity reached critical level after record insertion,
– Performing a split if there was a message from the split coordinator. When split operation is complete, committing the split to the coordinator.

Split coordinator functions are as follows:
– Maintaining real LH* file parameters.
– Collecting information about overloaded buckets, sent by the servers.
– Controlling file splits – sending a split initialization message to the bucket pointed with *n* pointer.
– Receiving split committing messages from the servers, updating real file parameters.
– Implementing file growing strategy – cascade splits, concurrent splits, etc.

4.2. Possible faults and their operational effects

Client faults:
– The client breakdown (broken communication, deaf to commands) – off-line client is not a danger for file structure and file control mechanisms.
– Wrong bucket address calculated by the client (client has gone berserk) – the query would be sent to the wrong server. This kind of fault may be caused by client's wrong file image, invalid address computation, etc. Such a fault may lead to file corruption, as it is explained below.

Server faults:
– Invalid recipient – the server received a message concerning a bucket stored on a different server. There are two possibilities:
  – The message is received by a server having too small number – such a fault is not a danger – the message will be forwarded properly. The destination server will send a response to the client;
  – The message is received by a server having too big number – basic SDDS LH* scheme for growing file would never forward a query to a server having a number smaller than the current server's number. This may lead to a situation that the client receives wrong information about the record presence and there may appear more than one record having the same key value (if there were some insertions performed).
– Collision – a bucket overload. This happens if any bucket stored on the current server is full (or reached critical load level) and an insert message is received. In this situation the server sends collision message to the split coordinator.
– Token damage (for LH* variant without coordinator) – if there was a bucket damage the token might be lost. If this occurred the file expansion

is no longer possible. If there was a token recovery procedure performed, there might be several tokens in LH* file. Such a situation might in short time lead the file to crash.

Split coordinator faults:

– Deaf coordinator – collision messages do not initiate bucket splits and file expansion. This may lead in short time to file overload.
– Coordinator sending invalid messages (coordinator has gone berserk) – such messages may cause splits incompatible with the LH* scheme. This situation may lead to file structure crash and massive data access problems.

4.3. Fault-tolerant centralized control for LH*

A multicomputer means a net of computers. Hence, two rather obvious assumptions are taken. First, that at least a three-server multicomputer is considered. Next, that all single transient or permanent faults of LH* control should be tolerated.

The proposed mechanism uses Job Comparison Technique (JCT) and Triple Modular Redundancy (TMR) [12] to arrange fault tolerant split coordination. Split coordinator processes are started according to the following procedure:

Step1:

Just as the first bucket is created, *first and second coordinators* are generated on the first and second servers respectively.

Step 2:

From now on, both coordinators work in parallel and every final result of their work is compared (JCT). The doubled split coordinator works as long as the results are identical. If a difference is detected then the client is asked to repeat its query (retry operation). This retry operation is repeated until results agree or retry limit expires. In that latter case.

Step 3:

The third coordinator is created on the third server. It starts to process the query. Then all three results are compared and the faulty coordinator is identified (TMR). Finally after voting the correct decision is taken.

Additional rules:

– Every server sends a collision message to every active split coordinator.
– Each of the split coordinators sends his own decision to the remaining coordinators. The comparison is made by every coordinator concurrently.
– The coordinators decide which one sends a single response message to the server.
– This extended scheme allows invalid coordinators be replaced with new instances.

As it was explained above, the client may commit addressing fault, sending a query to the server not maintaining the required bucket. The basic LH* scheme without file shrinking would forward such a message to a proper server if the message was originally sent to the server (bucket) having too small number. We must define backwarding, if we want LH* to tolerate every addressing fault. Hence, the destination address *a'* should be calculated as follows:

```
a′ = $h_j$(C);
if a′ != a then
   a″ = $h_{j-1}$(C)
   if a < a″ and a″ < a′ then a′ = a″;
   if a″ < a and a < a′ then a′ = a″;
```

The last (added) phase does not allow the message to be sent beyond the LH* file space. If the result address *a'* is smaller than current bucket address *a* then it is valid and the message would be sent there.

4.4. Fault-tolerant distributed control for LH*

When LH* is distributed the servers send each other a *token*, which allows for splitting a bucket actually holding it. The token lost is the most dangerous fault of distributed LH* because it stops SDDS growing at all.

In fault tolerant distributed architecture of LH* servers are organized in a ring with a circulating token. The server after sending a token (allowing next server to make a split) periodically checks the next server. The procedure for checking the current token owner (i+1 server), performed by the previous token owner (i server) is as follows:

Step1:
The i server waits a predefined time-out, then
Step 2:
The i server sends a message to *i+1* server, asking about its current status and token presence;
Step 3:
The i server waits for response from *i+1* server for predefined time (depending on the net parameters) and then

- if positive response (the *i+1* is in good condition and is holding a token) go to step 1;
- if negative response (the *i+1* server is no longer in possession of a token it was sent to *i+2* server) the *i+1* server takes care of checking the next server, *end of the procedure*;
- if no response (the *i+1* server is probably faulty) depending on its current state and system's strategy, one of the following recovery steps is undertaken:

Step 4:

Send another token to the *i+1* server (retry - temporary fault is assumed, first), go to step 1;
Step 5:
Send a new token to the *i+2* server (skip the *i+1* server – permanent fault is assumed, then), go to the beginning and watch server *i+2*;
Step 6:
Recover lost buckets using a spare server. The new server gets *i+1* number, go to the beginning of the procedure (bucket recovery is a well defined operation for the extended LH* schemes described in section 3).

5. Comparison consideration

The split coordinator process is woken up only if some collision messages were received and there is a need to make a split. How often the coordinator is making a decision, depends on single bucket capacity and amount of incoming data.

Our analysis was prepared for different streams of incoming data D. It includes only new data inserted into the LH* file, because newly inserted data increase buckets load and initiate splits. According to what was told above, the frequency of LH* file splits may be calculated as follows:

$$P = \left(D / \left(N * 1024\right)\right) * 3600, \tag{3}$$

where: N – the single bucket capacity, presented as the number of records *1024, D – the incoming data stream intensity in the number of records per second, P – a number of splits made in the period of an hour (the frequency of coordinator's decisions) for three different incoming data streams (Fig.1).

Practically, the bucket size depends on the available server RAM memory, the record size and the number of records in a bucket. For modern PC machines, those parameters may look like this (assuming that we have 512MB of free RAM):

– One bucket, 512 * 1024 = 524'288 records, 1024 bytes each;
– 16 buckets having 8 * 1024 = 8'192 records, 4096 bytes each, etc.

The number of messages transporting data is constant, no matter how big the bucket capacity is. Assuming that every insertion needs two messages:

1. The message sent by the client, carrying the record and its key (size of a record may vary on a large scale and is not considered here);
2. Commit a message, sent by the server to the client.

During a split, overloaded server sends about half of its capacity to a new server. We assume that the number of messages for transferring data between servers is identical to the number of messages needed to transfer data between a client and a server.

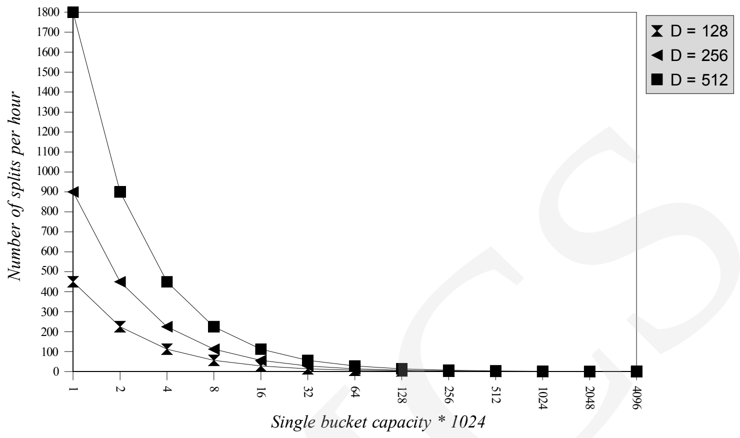

Fig. 1. The average number of bucket splits per hour considered for different bucket sizes and three data streams D

A number of IAM messages is not considered, it depends on a number of clients and their activity. As it is written in [5], a number of additional IAM messages is relatively small and for larger bucket capacities the number of addressing errors decreases.

With respect to given rules, the number of data carrying messages per hour is constant and independent of bucket capacity:

$$DD = D * 3600 * 2 + 2 * P * \left(N * 1024/2\right), \tag{4}$$

where: DD – the number of messages and commits per hour, D – the input stream intensity in number of records per second, P – the number of splits per hour, N – a single bucket capacity * 1024.

After the first equation consideration:

$$DD = D * 3600 * 2 + D * 3600 = D * 3600 * 3\,. \tag{5}$$

The number of messages is independent of bucket capacity. If we assume the input stream intensity D=128, we have:

$$DD = 128 * 3600 * \ 3 = 1'382'400\,.$$

The SDDS comparison is considered for three split coordinator variants. Each variant needs different number of messages:

1. Single coordinator: 1 collision message + 1 split decision + 1 commit, DC1 = 3;
2. Double coordinator with Job Comparison Technique: 2 collision messages + 2 result comparison + 1 decision + 2 commits, DC2 = 7;
3. Triple coordinator with Triple Modular Redundancy: 3 collision messages + 3 result comparison + 1 decision + 3 commits, DC3 = 10;

The total number of coordinator's messages and data carrying messages for each variant may be computed as follows:

$$\sum DCn = DD + P * DCn, \quad n = 1,2,3, \tag{7}$$

where: DD – the number of messages and commits per hour, P – the number of splits per hour.

It is easy now to calculate efficiency decrease caused by additional coordinator's messages for the JCT and TMR variants (Fig. 2):

$$\eta DC2 = \frac{\sum DC1}{\sum DC2} * 100\%, \tag{8}$$

$$\eta DC3 = \frac{\sum DC1}{\sum DC3} * 100\%.$$

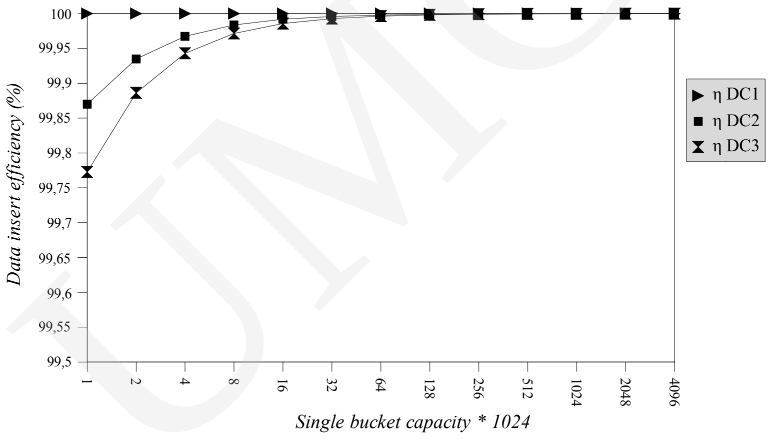

Fig.2 The efficiency decrease caused by additional coordinator's messages (in comparison with single coordinator SDDS)

6. Conclusions

Split coordination is a crucial SDDS mechanism. Its breakdown may lead to SDDS file damage and data corruption. The extensions of SDDS architecture, we have proposed here, increase SDDS dependability.

The application of Job Comparison Technique and Triple Modular Redundancy allows the SDDS to work correctly in spite of any single coordinator fault. The proposed method for querying the server holding a token ensures that it would never be lost. Such a situation might lead to SDDS faults analogous to the coordinator faults.

As it was shown, the number of additional messages required for double or triple split coordinator is so small that it has no meaning compared to the number of messages transferred during the basic SDDS scheme activity. Decreasing SDDS efficiency due to the extended split coordinator becomes smaller as the bucket capacity increases. It is worth mentioning that our analysis was focused on new record inserts only. In a real system, where more kinds of operations are performed (search, delete, modify), the decreasing SDDS efficiency seems to be even smaller.

References

[1] Hennessy J., *The Future of Systems Research*, Computer, August (1999).
[2] Lewis T., *Mainframes are Dead, Long Live Mainframes*, Computer, August (1999).
[3] Austin T. et al., *Mobile Supercomputers*, Computer, May (2004).
[4] Dahlgren F., Torrellas J., *Cache-Only Memory Architectures*, Computer, June (1999).
[5] Litwin W., Neimat M-A., Schneider D., *LH*: A Scalable Distributed Data Structure*, ACM Transactions on Database Systems ACM-TODS, December (1996).
[6] Litwin W., Neimat M-A., *High-Availability LH* Schemes with Mirroring*, Intl. Conf. on Coope. Inf. Syst. COOPIS-96, Brussels (1996).
[7] 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).
[8] Litwin W., Risch T., *LH*g: a High-availability Scalable Distributed Data Structure through record grouping*, U-Paris 9 Tech. Rep., May (1997).
[9] Litwin W., Menon J., Risch T., *LH* Schemes with Scalable Availability*, IBM Almaden Research Rep., May (1998).
[10] Litwin W., Menon J., Risch T., Schwarz T., *Design Issues for Scalable Availability LH* Schemes with Record Grouping*, DIMACS 99, Princeton U. & IBM-Almaden Res. Rep., (1999).
[11] Litwin W., Schwarz T., *LH*RS: A High-Availability Scalable Distributed Data Structure Using Reed Solomon Codes*, CERIA Res. Rep. & ACM-SIGMOD Dallas, (2000).
[12] Dhiraj K. Pradhan, *Fault-Tolerant Computing: Theory and Techniques*, Prentice-Hall, Inc., May (1986).