CZASOPISMO TECHNICZNE
WYDAWNICTWO POLITECHNIKI KRAKOWSKIEJ

z. 1-I/2007
ISSN 0011-4561
ISSN 1897-631X

GRZEGORZ ŁUKAWSKI*, KRZYSZTOF SAPIECHA**

CAUSE-EFFECT OPERATIONAL FAULT ANALYSIS FOR RANGE PARTITIONING SCALABLE DISTRIBUTED DATA STRUCTURES

PRZYCZYNOWO-SKUTKOWA ANALIZA BŁĘDÓW OPERACYJNYCH DLA SKALOWALNYCH ROZPROSZONYCH STRUKTUR DANYCH Z PODZIAŁEM ZAKRESÓW

Streszczenie

Skalowalne Rozproszone Struktury Danych (SDDS) składają się z dwóch komponentów rozproszonych dynamicznie w obrębie multikomputera: danych należących do pliku oraz mechanizmu kontroli położenia rekordów w pliku. Błędy rekordów (danych) mogą co najwyżej doprowadzić do błędnych obliczeń, podczas gdy błędy związane z mechanizmem rozmieszczania rekordów mogą doprowadzić cały plik SDDS do zniszczenia. W niniejszym artykule dokonano analizy przyczynowo-skutkowej błędów rozmieszczania rekordów dotyczących struktur SDDS typu RP* (z podziałem na zakresy).

Słowa kluczowe: Skalowalne Rozproszone Struktury Danych, multikomputery, odporność na błędy

Abstract

Scalable Distributed Data Structures (SDDS) consists of two components dynamically spread across a multicomputer: data records belonging to a file and a mechanism controlling record placement in file space. Record (data) faults may lead to invalid computations at most, while faults concerning record placement mechanisms may lead whole SDDS file to crash. In this paper, cause-effect analysis of record placement faults concerning SDDS RP* (Range Partitioning) file is given.

Keywords: Scalable Distributed Data Structures, multicomputers, fault tolerance

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

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

1. Introduction

High Performance Computing (HPC) can be achieved in different ways [1]. The cheapest one consists in multicomputing. For example, a multicomputer may be built from desktop or server PCs, connected through fast Ethernet and supervised by Linux-based operating system (Linux supplemented with cluster controllers).

As computer systems have become more and more complex, system designers have to deal with more different hardware and software failures. In a multicomputer there are a lot of possibilities for operational faults, as there are more CPUs, memories and another hardware and software components working together. A very useful way for cause-effect fault analysis for computer system is to use a Software Implemented Fault Injector (SIFI) [8].

Scalable Distributed Data Structures (SDDS) [2, 3] consists of two components dynamically spread across a multicomputer: records belonging to a file and a mechanism controlling record placement in file space. Record placement mechanism is spread between SDDS servers and their clients. To study SDDS behavior under fault conditions a specialized SIFI called *SDDSim*, was developed [8].

In section 2 brief outline of SDDS RP* architecture is presented. Possible SDDS RP* failures are analyzed in section 3. In section 4 results of injecting faults into SDDS RP* are given. The paper ends with conclusions.

2. Scalable Distributed Data Structures

The least SDDS component is a *record*. Each record is equipped with a unique *key*. Records with keys are stored in *buckets*[1]. Each bucket's capacity is limited. If a bucket's load reaches some critical level, it performs a *split*. A new bucket is created and a half of data from the splitting bucket is moved into a new one.

A *client* is another SDDS file component. It is a front-end for accessing data stored in SDDS file. The client may be a part of an application. There may be one or more clients operating the file simultaneously. The client may be equipped with so called *file image* (index) used for bucket addressing. Such 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 he will never commit the same addressing error again.

All the SDDS file components are connected through a network. Usually, one multicomputer node maintains single SDDS bucket or a client, but there may be more components maintained by single node. In extreme situation all SDDS buckets, clients and additional components may be run on single PC.

2.1. SDDS RP* architecture

RP* (Range Partitioning) [2] is a family of record order preserving SDDS architectures. Each bucket holds records with keys in specific key range, so records having consecutive

[1] As data storage, multicomputer nodes' local memories are usually used, so SDDS offers outstanding data processing performance.

key values are stored in the same single bucket usually. Hence, if lucky one may hold all required records for corresponding RP* file in only single bucket.

For communication between RP* components, three kinds of messages are used as follows:

– *Unicast* – point to point connection,
– *Multicast* – message is sent to given address group,
– *Broadcast* – message is sent to all machines in a network segment.

Usually, multicast and broadcast messages allow more data to be sent at once. Such messages are the most effective for operations performed simultaneously on many multicomputer nodes. There are three RP* subarchitectures, using different message types:

1. RP*$_N$ – no bucket index (file image) is used at all. Broadcast/multicast messages are mostly used.
2. RP*$_C$ – RP*$_N$ plus client file image, IAM messages are sent if a client commits addressing error. Unicast messages are used mostly and broadcast/multicast for message forwarding. Each client has its own file image.
3. RP*$_S$ – RP*$_C$ plus bucket file image, stored at distinct buckets called the *kernel*. Almost every operation may be executed using unicast messages.

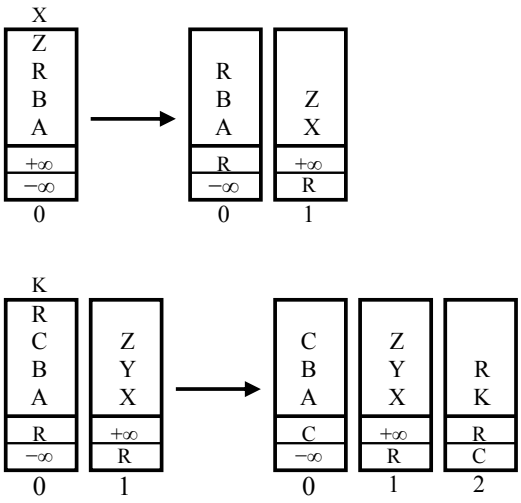

Fig. 1. RP* file expansion

Ryc. 1. Rozwój pliku RP*

Each bucket stores records ordered by keys in ascending order usually. Each bucket has its *header* with minimal (λ) and maximal (Λ) key values. The range (λ, Λ] is so called *bucket range*. A bucket may store records with keys *c* fitting its range only

$$\lambda < c \leq \Lambda$$

2.2. RP* file expansion

Newly created RP* file consists of a single bucket only (Fig. 1) with number 0 (logical address) and infinite range $\lambda = -\infty$ and $\Lambda = +\infty$. All queries are sent to this lonely bucket.

Just as this bucket load reaches some critical level, a split is performed and new bucket with number 1 is created. Generally, newly created bucket gets number (logical address) M, if the file consists of M buckets while split is initiated.

Each split partitions the file into ranges and record key order in each bucket is preserved. At any moment of file evolution, each bucket stores records with keys in its range.

2.3. RP* file access

The data stored in RP* file may be accessed and modified with three kind of queries send by RP* file clients:

– *Single key queries* – concerning a record.
– *Range queries* – concerning records with keys in given range $c_1 < c_2$.
– *General queries* – send to all records in the file.

A client may perform record traversal operation on the whole RP* file or some part of the file in ascending or descending order. RP*$_N$ architecture uses no file image and all queries are send using broadcast/multicast messages. Each bucket receives such message and only the bucket holding required key range replies. Reply messages are unicast type usually, as the bucket performing operation obviously knows the sender's network address.

RP*$_C$ and RP*$_S$ scheme uses file image made of bucket addresses and ranges. In RP*$_C$ clients are equipped with such file image, in RP*$_S$ another file image is maintained for servers (buckets), so each forwarding is done with unicast.

2.4. RP*$_S$ file kernel

Kernel is a set of specific buckets, used in RP*$_S$ scheme only. *Kernel buckets* stores bucket file image. The image structure is similar to client file image and consists of addresses and ranges of another kernel and data buckets. Owing to the kernel, RP*$_S$ file uses unicast messages mostly.

The kernel is organized in tree-like hierarchical structure with data buckets as leaves. RP*$_S$ file may be multi-level. RP*$_S$ tree structure may be traversed in all directions, so each bucket header is supplemented with backward pointer (parent node address). The kernel is used for client addressing error resolving. If such error is committed, incorrectly addressed data bucket uses its parent pointer for forwarding. A message with unknown recipient is sent to the parent node, and after simple range and address comparison, and maybe some more forwarding, reaches proper bucket at last. The correct buckets send an IAM message to the client, with information (address and range) about every node visited by forwarded message. Kernel buckets are updated after each successful split operation. Kernel bucket may also be overloaded and split, parent pointers in some buckets may then become invalid. In such situation wrongly addressed messages are forwarded, just as client's messages are. Similarly, a bucket may receive IAM message with new backward pointer.

More technical details concerning RP* file operation may be found in [2].

3. Range partitioning fault model

3.1. The client

A client failure is not a big danger for SDDS file structure. Client operational faults are the following:

1. *Empty file image* – every new RP* client's image is empty and consists of only one bucket (number 0) with infinite range, without respect to actual file state. It is a normal situation for SDDS scheme, such client's file image will be updated with IAM messages, just as the client will start to operate RP* file.
2. *Deaf client* – stops sending and receiving any messages. Such client is not a danger for file structure, because he stops to function just as turned off client. A client is not necessary for correct RP* file operation.
3. *Berserk client* – client sends his queries to incorrectly chosen buckets (wrong destination address calculation or damaged file image) in random moments. Such client's reaction for any message is unpredictable. A message send by a berserk client may have correct structure, so it could be properly processed by a bucket and correct reply message could be sent then.

Because RP*$_N$ client send all his queries with broadcast/multicast messages, he will never commit any addressing error (as long as the network is working properly). RP*$_C$ and RP*$_S$ client may commit such error, forwarding will be used then, and a message will reach correct bucket, at last.

In fact, berserk client is also not a danger for RP* file structure, but unfortunately his addressing errors cause many unneeded messages to be send through the network, and decrease RP* file efficiency.

3.2. The bucket

A bucket may send message to another bucket only when an addressing error is committed (forwarding) or some bucket overloads and is making a split. In fact, any bucket from another bucket's point of view may be treated as a client performing data operations. Bucket faults are the following:

- *Deaf bucket* – stops reacting to all messages, its content is practically lost (in case of persistent fault).
- *Berserk bucket* – sends messages to randomly chosen destinations. Its content is lost, but it is not a danger for file structure. As for a berserk client, the network efficiency is decreased due to many unnecessary messages send to correct addressing errors.
- *Invalid recipient* – a client commits addressing error, the message will be forwarded to correct bucket. Normal RP* file operation.
- *Overload (collision)* – again, normal SDDS RP* file event, leading to bucket split and file scaling.

If a bucket is damaged and its content is lost (for persistent faults mostly) then a part of data stored in RP* file is lost and becomes unrecoverable. For preventing such loss, data fault tolerant scheme with some kind of redundancy should be used. Data fault tolerant schemes for SDDS LH* have been developed, such as LH*$_M$ [4] or LH*$_{SA}$ [5], and similar schemes could be used for RP*. In such scheme, damaged bucket with all its content could

be recovered and replaced with a new, correctly working instance. For such data fault tolerant scheme another file component for recovery coordination is required. This component will be called Recovery Coordinator (RC). In LH*, recovery management is done by the Split Coordinator (used in centralized LH* scheme).

If a berserk bucket is correct message recipient, but it is not accepting this message, a message may be forwarded many times and will never reach correct bucket. Unfortunately, it leads to significant efficiency decrease, but could be easily tolerated with some TTL (Time To Live) parameter added to each message, just as the one defined for IP network protocol.

3.3. RP*$_S$ kernel buckets

Kernel buckets are used for correcting all addressing errors. If such bucket goes deaf or berserk, some addressing errors could not be resolved. Incorrectly send query may never reach correct bucket, so can hold the file evolution. Fortunately, it seems that kernel bucket damage is not a danger for bucket and data integrity:

- *Berserk kernel bucket* – forwards bad addressed queries to incorrect buckets. It causes more messages to be sent and may lead to removing the query in question from the network (if message's Time To Live expires). The sender (client or another bucket) will probably never receive any reply message.
- *Deaf kernel bucket* – is not responding to any message, access to records stored in the file is complicated or even impossible.

3.4. Bucket faults propagation

Berserk bucket (kernel bucket) may become a source of many incorrect IAM messages. Such messages, if received and processed by other components, may lead to client's file image disruption. Buckets' parent/child pointers (addresses) may be set to incorrect values also. This way, faulty bucket may cause many other components' bad operation. A client with invalid file image should be treated as a berserk one.

A kernel bucket may receive invalid update message (invalid address and range of a new bucket), so leads to berserk state of one or more kernel buckets.

4. Experimental results

Because both the client and the bucket in deaf or berserk state are not really a danger for RP* file structure, what was proved by our recent experiments, here their faulty behavior are not taken into consideration. Faulty RP*$_S$ kernel buckets are the focus of attention of this work.

4.1. SDDS software implemented fault injector

To study SDDS behavior under fault conditions a specialized Software Implemented Fault Injector called *SDDSim*, was developed [8]. It is an application using SDDS file for storing integer key values. The file is able to expand only. Main differences between *SDDSim* and a multicomputer SDDS are the following:

– the buckets are stored in local PC's RAM instead of multicomputer global RAM,
– there is no real data stored in each of the buckets.

Such a solution leads to the following limitations:

– only SDDS functionality is tested, time related problems are neglected,
– each element (process) performance is almost equal to every other element one. On the contrary a multicomputer may be built from different machines having different amount of RAM memory, different network interfaces, etc.

Maximum number of buckets in a file simulated with *SDDSim* is 128, what is usually enough to acquire reliable results. Because of this relatively small maximum number of buckets in RP*$_S$ file, only one kernel bucket storing whole file image is simulated.

The application allows monitoring all SDDS file activities interactively. All operations are logged into a log file and can be analyzed later. Another application called *SDDSlog* was developed for this purpose. Graphs presented below were generated with this application. The vertical axis should be interpreted differently for various curve types (Fig. 2):

– *messages* – shows how much of the network throughput is taken by message of given type (0–100%),
– *the number of buckets* – actual number of buckets is shown (0–128),
– *average file load* – computed for all active buckets. 150% is the maximum, because each bucket can hold additional 50% of record capacity.

Fig. 2. Correctly working RP*$_S$ file with bucket capacity 64 (4500 insert operations)

Ryc. 2. Poprawnie działający plik typu RP*$_S$ z pojemnością wiaderek równą 64 rekordy (4500 operacji wstawiania)

4.2. RP*$_S$ file kernel fault injection

The kernel is used for client addressing error correcting, so its failure may be dangerous for correctness of RP* file operation. To investigate this problem a simulation experiment was executed.

Firstly, RP*$_S$ file with bucket capacity 64 records was created, 1500 random records were inserted, all operations finished successfully (see Fig. 2). The lonely kernel bucket was turned deaf then and another 3000 records were inserted (Fig. 3). As it was expected, many operations failed as the file was growing. The kernel wasn't able to correct client errors and the client's file image was becoming less and less accurate with each split. It is important to notice that the kernel isn't used directly for record addressing and the file structure was intact all the time, according to basic RP* rules.

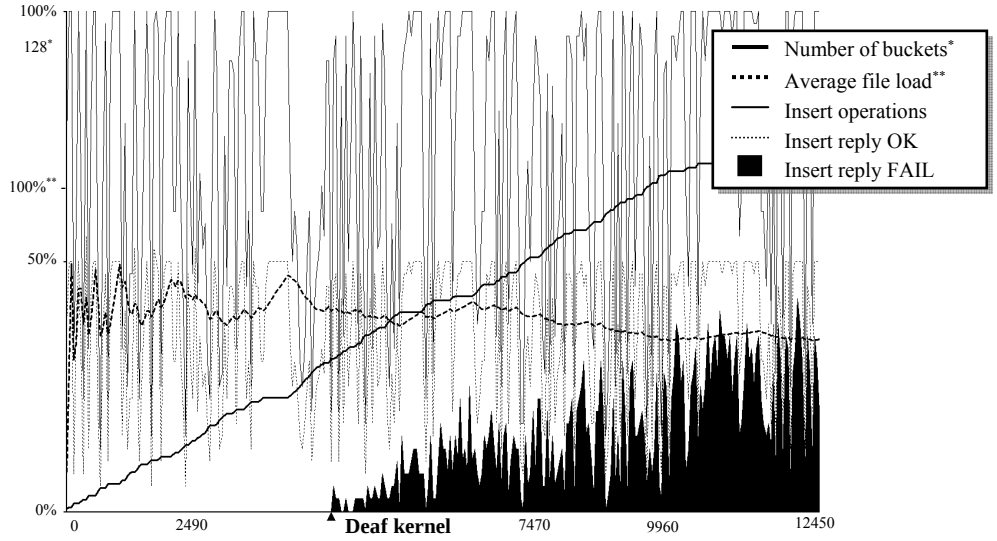

Fig. 3. Deaf RP*$_S$ kernel bucket

Ryc. 3. Wiaderko kernela RP*$_S$ w stanie głuchym

In another experiment, after insertion of 1500 records, the lonely kernel bucket was turned berserk (Fig. 4). Most of the next 1500 insert operations finished successfully.

To check the client's image convergence, this image was cleared before taking another 1500 inserts. The number of failed operations is constant and isn't increasing significantly as the file evolves. Wrongly addressed messages were received by different buckets, these buckets was sending correct IAM messages, updating client's image to nearly actual state.

In case of real multicomputer SDDS RP* file the kernel may occupy more than one kernel bucket. However, such multikernel bucket scheme behaves in very similar way as single kernel bucket scheme (analyzed here) does.

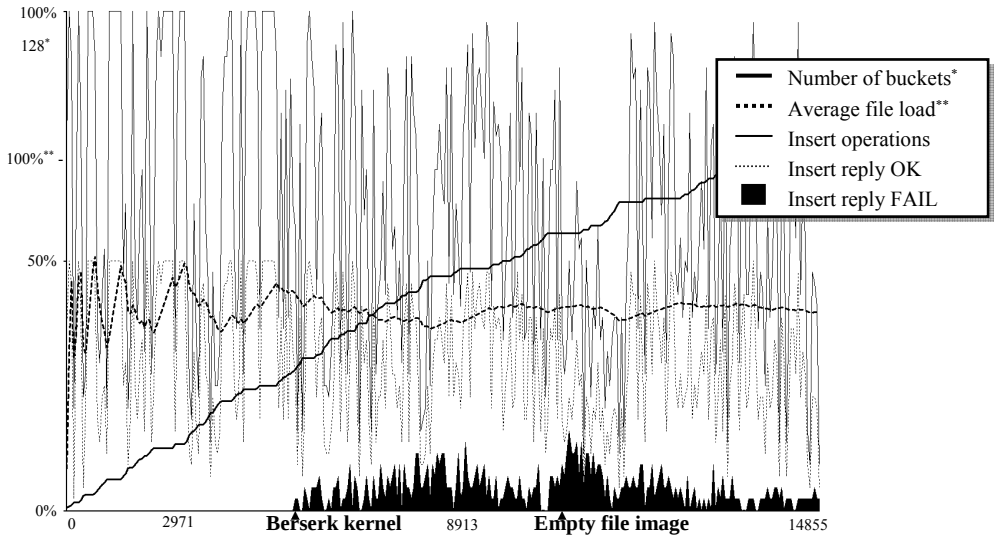

Fig. 4. Berserk RP*$_S$ kernel bucket and empty file image

Ryc. 4. Wiaderko kernela RP*$_S$ w stanie szału i pusty obraz pliku

5. Conclusions

SDDS RP* is scalable and efficient structure applicable for storing a large amount of data in distributed multicomputer RAM. One of the three RP* sub-architectures could be used to fit a multicomputer. If a network connection in the multicomputer is efficient in sending multicast/broadcast messages, more simple RP*$_N$ may be used. If such messages should not be used (or there is no possibility to use them at all) then advanced RP*$_S$ version can be applied.

The Range Partitioning itself is more robust (of any kind) than the Linear Hashing (LH*) architecture is [6]. For tolerating data faults, an architecture similar to data fault tolerant LH* may be used [4, 5].

As it was shown, control faults concerning Range Partitioning SDDS may become a real danger for correct file operation and data access. Control faults usually do not result in data loss. The data stored in the buckets is still intact even if many control faults arise. Unfortunately, control faults may lead to problems in accessing data stored in the buckets, so from client's point of view data is inaccessible and maybe lost. Therefore, RP* schemes fault tolerant for all kind of operational faults are needed. They are under development and evaluation now.

The SDDSim software fault injector was tailored for SDDS control fault vulnerability analysis. Therefore, it uses very simple multicomputer architecture representation but still giving dependable results. On the other side no conclusions concerning hardware or operating system faults may be drawn. Using SDDS implementation with MPI/PVM interface would result in more interesting analysis. However, this would take much more time but not enrich our knowledge about SDDS fault vulnerability.

References

[1] Dongarra J., Sterling T., Simon H., Strohmaier E., *High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions*, IEEE Computing in Science and Engineering, 2005.

[2] Litwin W., Neimat M.-A., Schneider D., *RP*: A Family of Order-Preserving Scalable Distributed Data Structures*, 20th International Conference on Very Large Data Bases (VLDB), 1994.

[3] Litwin W., Neimat M.-A., Schneider D., *LH*: A Scalable Distributed Data Structure*, ACM Transactions on Database Systems ACM-TODS, 1996.

[4] Litwin W., Neimat M.-A., *High-Availability LH* Schemes with Mirroring*, International Conference on Cooperative Information Systems COOPIS-96, Bruksela 1996.

[5] Litwin W., Menon J., Risch T., *LH* Schemes with Scalable Availability*, IBM Almaden Research Rep., 1998.

[6] Sapiecha K., Łukawski G., *Fault-tolerant Control for Scalable Distributed Data Structures*, Annales Universitatis Mariae Curie-Skłodowska, Informatica, 2005.

[7] Sapiecha K., Łukawski G., *Fault-tolerant Protocols for Scalable Distributed Data Structures*, Springer-Verlag, LNCS 3911, 2006.

[8] Łukawski G., Sapiecha K., *Software Functional Fault Injector for SDDS*, GI-Edition Lecture Notes in Informatics (LNI), ARCS'06 Workshop Proceedings, 2006.