Annales UMCS Informatica AI 6 (2007) 5-13

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

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

Fault tolerant control for RP* architecture of Scalable Distributed Data Structures

Grzegorz Łukawski[*], Krzysztof Sapiecha[**]

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

Abstract

Scalable Distributed Data Structures consist of two components dynamically spread across a multicomputer: records belonging to a file and a mechanism controlling record placement in the file space. Record (data) faults may lead to invalid computations at most, while record placement faults may bring whole file to crash. In this paper, extended SDDS RP* (Range Partitioning) architecture tolerant to the latter faults is presented and evaluated.

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).

Scalable Distributed Data Structures (SDDS) [2,3] consist of two components dynamically spread across a multicomputer: records belonging to a file and a mechanism controlling record placement in the file space. Record placement mechanism is spread between SDDS servers and their clients.

Two factors are crucial for SDDS dependability: fault-tolerance of data stored in records and fault-tolerant record placement in file space. Methods of making fault-tolerant records, such as LH*$_M$ [4], are developed. Fault-tolerant record placement architectures for SDDS LH* were presented and evaluated in [5,6]. In this paper fault-tolerant record placement architecture for SDDS RP* is introduced.

[*]*E-mail address*: g.lukawski@tu.kielce.pl
[**]*E-mail address*: k.sapiecha@tu.kielce.pl

In section 2 brief outline of SDDS RP* architecture is presented. Possible SDDS RP* failures are analyzed in section 3. A new fault-tolerant SDDS RP* architecture is presented in section 4. The paper ends with conclusions.

2. Scalable Distributed Data Structures

The smallest 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 the 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 an extreme situation all SDDS RP* 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 a specific key range, so records having consecutive key values are usually stored in the same single bucket. Hence, if lucky one may hold all required records for the 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

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

on many multicomputer nodes. There are three RP* subarchitectures, using different message types:

- RP*$_N$ – no bucket index (file image) is used at all. Broadcast/multicast messages are mostly used.
- 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.
- RP*$_S$ – RP*$_C$ plus bucket file image, stored at distinct buckets called the *kernel*. Almost every operation may be executed using unicast messages.

Each bucket stores records ordered by keys usually in ascending order. Each bucket has its *header* with minimal (λ) and maximal (Λ) key the 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.

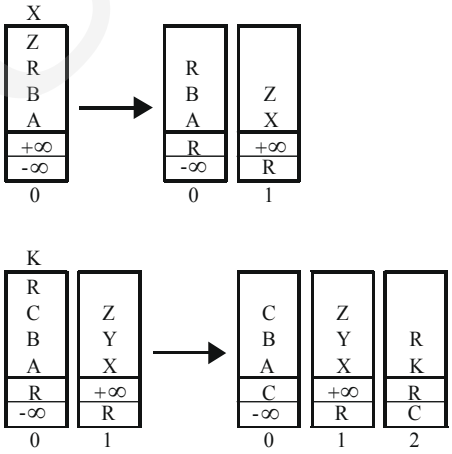

Fig. 1. RP* file expansion

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 the RP* file may be accessed and modified with three kinds of queries sent by the 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 the ascending or descending order. RP*$_N$ architecture uses no file image and all queries are sent using broadcast/multicast messages. Each bucket receives such a message and only the bucket holding the required key range replies. Reply messages are usually unicast type, as the bucket performing operation obviously knows the sender's network address.

RP*$_C$ and RP*$_S$ scheme uses the file image made of bucket addresses and ranges. In RP*$_C$ clients are equipped with such a 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 the RP*$_S$ scheme only. *Kernel buckets* store a bucket file image. The image structure is similar to the client file image and consists of addresses and ranges of other kernel and data buckets. Owing to the kernel, RP*$_S$ file uses unicast messages mostly.

The kernel is organized in a tree-like hierarchical structure with data buckets as leaves. RP*$_S$ file may be multi-level. The RP*$_S$ tree structure may be traversed in all directions, so each bucket header is supplemented with the backward pointer (parent node address). The kernel is used for client addressing error resolving. If such an error is committed, incorrectly addressed data bucket uses its parent pointer for forwarding. A message with an unknown recipient is sent to the parent node, and after the 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 the 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 a situation wrongly addressed messages are forwarded, just as client's messages are. Similarly, a bucket may receive IAM message with a new backward pointer.

More technical details concerning the 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 the SDDS file structure. Client operational faults are following:

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

Because the 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). The RP*$_C$ and RP*$_S$ client may commit such an error, forwarding will be used then, and a message will reach correct bucket at last.

In fact, the berserk client is also not a danger for the RP* file structure, but unfortunately his addressing errors cause many unnecessary messages to be sent through the network, and decrease the RP* file efficiency [8].

3.2. The bucket

A bucket may send a 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 client performing data operations. Bucket faults are following:

- *Deaf bucket* – stops reacting to all messages, its content is practically lost (in the case of persistent fault).
- *Berserk bucket* – sends messages to randomly chosen destinations. Its content is lost, but it is not a danger for the file structure. As for a berserk client, the network efficiency is decreased due to many unnecessary messages sent to correct addressing errors.

– *Invalid recipient* – a client commits an addressing error, the message will be forwarded to the 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 the RP* file is lost and becomes unrecoverable. To prevent from such a loss, the 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], and similar method could be used for RP*. In such a scheme, a damaged bucket with all its content could be recovered and replaced with a new, correctly working instance. For such a 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 a correct message recipient, but not accepting this message, a message may be forwarded many times and will never reach the 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 the IP network protocol.

3.3. RP*$_S$ kernel buckets

Kernel buckets are used for correcting all addressing errors. If such a bucket goes deaf or berserk, some addressing errors could not be resolved. Incorrectly sent query may never reach a correct bucket, so can hold the file evolution. Fortunately, it seems that kernel bucket damage is not a danger for the bucket and the data integrity [8]:

– *Berserk kernel bucket* – forwards badly 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* – does not respond to any message, access to records stored in the file is complicated or even impossible.

3.4. Bucket faults propagation

The 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, a 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 an invalid update message (invalid address and range of a new bucket), thus leading to the berserk state of one or more kernel buckets.

More detailed RP* fault analysis along with the experiment results may be found in [8].

4. Fault tolerant control for RP*

To study SDDS behaviour under fault conditions a specialized Software Implemented Fault Injector called *SDDSim*, was developed [7]. Earlier experiments proved that the client or the bucket in a deaf or berserk state is not really danger for the RP* file structure. Hence, their behaviour will not be analyzed further in this paper. We have focused on the RP*$_S$ kernel buckets instead.

4.1. Client

If a client is turned off, it is not a danger for the file structure. In the case of berserk client, every addressing error made is corrected (if all other components are working correctly). So client faults are not danger for the RP* file structure, network throughput is affected only.

A client may behave unpredictably if his file image was damaged (as a result of internal or propagated fault). Clearing his file image may, in many situations, help and bring him back to correct operation. File image may be cleared in the following situations:

- The client receives no response to some or many of his queries. This could mean that there are no buckets where the file image shows up (queries are being sent beyond the file space). Such a client should clear his file image due to his own decision.
- The Recovery Coordinator receives information sent by a client, reporting broken bucket, but such bucket does not really exist. The RC should send a message 'clear image' to the sender.

After clearing the client's file image, it should be quickly updated by IAM messages, according to the RP* rules, if all other file components work correctly, of course.

4.2. Bucket

Data fault tolerant RP* architectures should work correctly, if a bucket's failure is detected as fast as possible, and reported to the Recovery Coordinator.

In the case of deaf bucket, a failure will be detected if it stops responding to all messages.

In the case of berserk one, message sent from a bucket to a bucket many times should be removed from a network if its TTL (Time To Live) expires. For such a message both its sender and recipient should be reported to the RC, as one of them may be in the berserk state.

If a bucket goes berserk and sends incorrect IAM messages, a client's file image may become invalid, leading to a client's berserk state. In such a situation, this client's file image should be cleared as it was explained in section 4.1, so it may bring this client back to correct operation.

4.3. RP*$_S$ file kernel

The kernel is used for correcting clients' and buckets' addressing errors only, so its failure is not a danger for the file structure. Unfortunately, as it was proved, its failure may lead to serious data access problems. If a client or a bucket detects kernel bucket's failure (deaf or berserk state), effects of such a fault may be tolerated as follows:

- RP*$_S$ structure degradation – kernel buckets are no more used, RP*$_C$ rules are used for correcting client addressing errors. Instead of kernel forwarding, broadcast/multicast messages should be used. Possible to use if the network is capable of sending such message types. May be used to tolerate both persistent and transient faults.
- Kernel bucket recovery (Algorithm 1) – the invalid kernel bucket is replaced with a brand new instance.

Algorithm 1: RP*$_S$ kernel bucket recovery

{New kernel bucket m_0 replaces the failed bucket m}

1. If a parent node for m exists then:
   - send a query to the root node, requesting bucket's m range and its parent node address;
   - fill bucket's m_0 header with received data;
2. else
   - set bucket's m_0 range to $(-\infty, +\infty)$ and its parent node pointer to *null*; m_0 is a new root node.
3. end if
4. Send a broadcast/multicast message to all file buckets, containing address of m, m_0 and actual range.
5. For each bucket whose m is a parent node:
   - set a new parent pointer (address) to m_0;
   - send a message to m_0, reporting this bucket's range.

6. The m_0 bucket receives all information required to rebuild its content, so it does.

4.4. RP* file degradation

Simpler RP* architectures are also more tolerant of control faults. In RP*$_N$ no file images are used at all, so berserk components are not danger for the file structure and even data processing efficiency. The most complicated RP*$_S$ architecture uses two kinds of file images, and is the most volatile for all types of faults.

For successful tolerance of transient or even persistent faults, less complicated RP* rules may be used:

- In the case of kernel bucket faults, RP*$_C$ forwarding rules may be used, as the RP*$_C$ uses no bucket file image. For addressing error correcting broadcast/multicast messages are used.
- In the case of client faults, the RP*$_N$ client rules may be used. In RP*$_N$ there is no client file image, broadcast/multicast messages are used instead, so to help tolerate client file image faults.
- In the case of bucket faults, the RP*$_N$ rules may be applied without the need of any file image, which may be helpful to tolerate bucket-related faults.

5. Conclusions

The RP* version of Scalable Distributed Data Structures uses Range Partitioning for record addressing. It is more resistant to faults than LH* [5,6], especially if control faults are considered. However, control faults still may lead the whole file to crash, just as in the LH* architecture.

The simplest RP*$_N$ architecture uses no special control components and very simple addressing rules, so it may be useful in faulty environments. On the other hand, RP*$_S$ is the most complex RP* version, supplemented with file images, and unfortunately, it is the most vulnerable to control faults.

Considering the RP* architecture's nature, we proposed the RP* structure degradation mechanism, allowing more complex RP* versions to use addressing rules from less complex sub-architectures. As it was shown, it is the most useful for tolerating transient faults.

So called "kernel" buckets are used in RP*$_S$ for correcting addressing errors. Unfortunately, breakdown of one or many kernel buckets may become serious danger for the file structure and operation. We have proposed the RP*$_S$ kernel recovery algorithm, useful especially for permanent faults, similar to mechanisms developed for data fault tolerant LH* architectures [4].

Efficiency of our new, modified RP* architectures is almost not affected by additional fault tolerance rules. Structure degradation and/or kernel bucket recovery takes place only if some problems occur, so normal file activity in correct mode is not affected at all. Moreover, the file can properly function even if some transient or permanent faults arise, what would not be possible for basic RP* architectures.

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 Intl. Conf. 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.* Intl. Conf. on Coope. Inf. Syst. COOPIS-96, Brussels, (1996).

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

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

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

[8] Łukawski G., Sapiecha K., *Cause-effect operational fault analysis for Scalable Distributed Data Structures.* Submitted for publication in Technical Journal (Czasopismo Techniczne) PK, Kraków.