A STORE OF JAVA OBJECTS ON A MULTICOMPUTER

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

Krzysztof Sapiecha
Department of Computer Science, Cracow University of Technology
ul. Warszawska 24, 31 - 155 Kraków, Poland

Keywords: Object store, Java, Scalable distributed Data structures.

Abstract: The research deals with Object Oriented versions of Scalable Distributed Data Structures (OOSDDS) to store Java objects in serialized form. OOSDDS can be used as a part of distributed object store. In the paper an architecture for object version of RP* is introduced and its implementation for Java objects is given. Finally performance of the new architecture is evaluated.

1 INTRODUCTION

Object-Oriented Database Management System (OODBMS) relies on Object Store (OS) which implements system and transaction related functions, particularly manages physical objects (Lobry et al., 1997). Future OS should allow for storing and maintaining huge number of objects. As such the OS requires powerful and scalable computer platform. To this end a multicomputer could be used as it may connect many PCs through fast network. It implements parallel architecture shared-nothing (Stonebraker, 1986) in not expensive way. Parallel architecture shared-nothing is the most scalable for very large databases (Lo et al., 2001). In such architecture every computer owns memory and disk, and acts as a server for data (DeWitt and Gray, 1992). Communication is based on message passing.

Scalable Distributed Data Structure (SDDS) (Litwin et al., 1996) is one of the most promising ideas for implementation of distributed and scalable data storage. SDDS is a file of records, a file which expands to computers of a multicomputer. Data are stored in so called buckets localized in main memories of the computers called servers.

The research concerns the development of Object-Oriented versions of SDDS (OOSDDS) to store Java objects in serialized form. OOSDDS can be used as a part of distributed OS. A brief description of SDDS is given in the next section. In section 3 main objectives of the research are stated. An architecture of object version of RP* (OORP*) is introduced in section 4. In section 5 its implementation for Java objects is given. In section 6 a performance of the new architecture is evaluated. The paper ends with conclusions and directions of further research.

2 SCALABLE DISTRIBUTED DATA STRUCTURES

A *record* is the least of components of SDDS. Each record is equipped with a unique *key*. Records with keys are stored in *buckets*. 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 component of SDDS file. 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 on the file simultaneously. The client may be equipped with so called *file image* (index) used for addressing data. 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

Bedla M. and Sapiecha K. (2008).
A STORE OF JAVA OBJECTS ON A MULTICOMPUTER.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - DISI*, pages 374-379
DOI: 10.5220/0001726003740379
Copyright © SciTePress

same addressing error again.

All components of SDDS file are connected through a *network*. Usually, one node of the multicomputer maintains single bucket or a client, but there may be more components maintained by single node, too. If all the buckets are stored in RAM memory then SDDS highly improves data access efficiency.

There are numerous architectures of SDDS (Litwin et al., 1996; Litwin et al., 1994; Devine, 1993). In general they fall into two categories: RP* and LH*. In RP* data are partitioned according to their ranges (Litwin et al., 1994). In LH* modified linear hashing is used for addressing data in distributed file (Litwin et al., 1996).

3 PROBLEM STATEMENT

SDDS has many advantages like scalability, high speed of operation and fault tolerance (Litwin et al., 1996; Litwin et al., 1994; Litwin and Neimat, 1997; Sapiecha and Łukawski, 2006). Using OOSDDS as a part of distributed OS should increase performance of OODBMS. Developing an architecture preserving all advantages of SDDS but applicable to object-oriented model would be very useful.

In principle SDDS architectures assume constant size of records and constant number of records in a bucket (though some of the architectures could be adopted for records of various sizes). Objects of the same class can contain tables of different sizes what makes their sizes different. Hence, the buckets should store various numbers of objects. Split algorithms for all OOSDDS are the first objective of the research. Next is what architectures of OOSDDS should be. There are at least two feasible solutions:

1. An architecture for regular objects. It could be based on remote invocation of methods and could be similar to RMI (Remote Method Invocation). Objects are stored in buckets on servers. All methods of an object are invoked remotely. Arguments are transmitted to the object and a method is invoked. An result of an execution of the method is transmitted back to the client. An access to fields of remote objects could be done by invocation of auto-generated methods.
2. Objects in serialized form are stored in buckets. When a client adds or updates an object in OOSDDS the object is serialized and then transmitted and stored in a bucket. The bucket splits using the same algorithm as during insertion of an object if there is not enough free space in the bucket. When a client invokes a method of the object the bucket finds a serialized form of the object and transmits it to the client. Next the object is deserialized and the method is invoked.

Finally, SDDS stores all data in main memory, which is not durable. Persistence of objects in a store is required. Hence, in OOSDDS a backup of the store on hard drive should be available.

4 RP* FOR OBJECTS

In OORP* addressing algorithm is same as in RP*. The address of a bucket where a record should be stored is calculated on the basis of ranges of the buckets (Litwin et al., 1994). However, the buckets should store various numbers of objects as the objects differ in sizes. In original RP* architecture every record has the same size and bucket's load factor is calculated on the basis of the number of records. During a split half of records are moved to a new bucket. In OOPR* every object may have different size and bucket's load factor is calculated on the basis of total size of all objects stored in the bucket. During a split objects which total size is approximately equal to half size of the bucket are moved to a new bucket. To calculate a load factor a bucket should know total size of all objects stored in the bucket. The total size of all objects must be updated when an object is added, updated or deleted.

OORP* split algorithm is like original RP* one. However, in OORP* middle key is calculated in different way (see Algorithm 1).

Algorithm 1 Middle key calculation algorithm for OORP*.

$size \leftarrow 0$;
$halfBucketSize \leftarrow bucketSize/2$;
while $size < halfBucketSize$ **do**
  $object \leftarrow nextObject(B)$;
  $c_m \leftarrow c(object)$;
end while
where:

- *bucketSize* denotes sum of all objects in the bucket B,
- c and c_m denote key and middle key respectively.

In SDDS used for storing records a size of the record can be chosen to place a record in a single datagram. A client sends a request in one datagram and the bucket answers in one datagram. If the bucket is overloaded it splits. During the split the record may

be moved to another bucket. One record is stored in one bucket.

In OOSDDS an object may be bigger than a size of the datagram. Hence, an overflow of a bucket may occur when n-th slice of the object is transmitted. It complicates a control of the bucket and the split algorithm. There are at least two feasible solutions of this problem. These are as follows:

1. To allow for storing slices of an object in many buckets. Object ID must be supplemented by slice ID. A bucket should control a pair of slice ID's identifying the first and to the last slice of stored objects. The objects might be placed at the top and the bottom of the bucket. Client image and a kernel must be modified: for every bucket maximal value of slice ID must be added to maximal value of object ID. This solution can be applied only when objects are stored in buckets in serialized form.
2. To allow for storing all slices of an object only in one bucket. Overloading of the bucket might be checked after transmission of the first or of the last slice of the object. However, when more (than one) clients add objects concurrently and the checking is done after transmission of the last slice of the object then the bucket can be overloaded too much. This does not happen when the checking is done after transmission of the first slice of the object. If the bucket would be overloaded after addition of the whole object this is postponed. The bucket splits and then the object is added to the proper bucket.

OOSDDS should be transparent for users. It should work properly no respect whether an object moves from one server to another or not. Therefore, the architecture for serialized objects is chosen in OOSDDS. For the other architecture a transparency is unfeasible because the results might rely on a server. Serialization can also be used to ensure persistence.

5 IMPLEMENTATION OF RP* FOR JAVA OBJECTS

An implementation of OOSDDSRP architecture for Java objects should allow for storing, updating, retrieving and deleting individual objects of a class defined by a user, with such restrictions on classes which might be easy accepted (i.e. classes must be serializable).

Objects stored in OOSDDS are distributed among many servers. Hence, they should have some extra features. This can be achieved in different ways. These are as follows:

- modification of a source code of a class, what requires an access to this source code,
- modification of Java Virtual Machine (JVM), but not all licenses allow to do that, additionally permanent modification of JVM affects other applications,
- modification of compiled byte code to gain required features, what means that the source code is not necessary and is left unchanged.

To store objects of some class in OOSDDS the class must contain its own mandatory methods and attributes. A solution based on inheritance can not be applied here as Java class can only extend one base class. A programmer could probably add these method and attributes manually but it excludes transparency. For these reasons the last from the above options that is modification of a byte code is chosen.

The program called SDDSModifier modifies compiled class and adds all required features. Objects are stored in serialized form. They are converted into tables of bytes, transmitted and then stored in the buckets on servers. Because of its popularity and universality Java collection is chosen as a method of accessing objects. The collections, besides tables, are probably the second primary method of arranging objects. All collections implement one of two interfaces: Collection which is basis for lists and sets, and Map which is used to map keys to values. Interface SDDSCollection, which extends Collection, and class SDDSFile, which implements it are then developed.

Summarizing, the development of scalable, distributed store of Java objects consists of two steps. First a programmer develops an application which uses SDDSFile to store objects. The application may use classes which are stored in OOSDDS and other classes not related with OOSDDS. Next, the classes are compiled using a standard Java compiler and then modified by SDDSModifier. SDDSModifier does what follows:

- add an implementation of required interface,
- add or modify an implementation of required methods,
- add an implementation of required attributes,
- modify the way of accessing specific attributes (access by references is replaced by invocation of auto-generated static methods).

Finally, after starting servers the application may be launched. Every server may work in textual or graphical mode.

A kernel in RP*S architecture is deployed on the first server of a multicomputer, where the bucket 0 is placed.

6 PERFORMANCE EVALUATION

In the experiment three OOSDDS architectures: RP*N, RP*C and RP*S were evaluated. OOSDDS file was stored on eight servers. The only one client of OOSDDS file was assumed. Objects were added (A in the Figures) and retrieved (R in the Figures).

Performances of these operations were measured and compared. During the experiment a class containing tables of bytes (512 kB) was used. The number of stored objects was from 1000 up to 6000 (from about 512MB up to about 3GB). Every test was repeated three times on the same servers (Athlon 3.0GHz, 1,5GB RAM and hard disk - ST380211AS) connected through gigabit Ethernet network. Then an average value was calculated.

For the first part of the experiment a distinguished segment of the network was used (the multicomputer was separated from remaining part of the network - S in the Figures). Results of the experiment are shown on Figures 1 and 2.

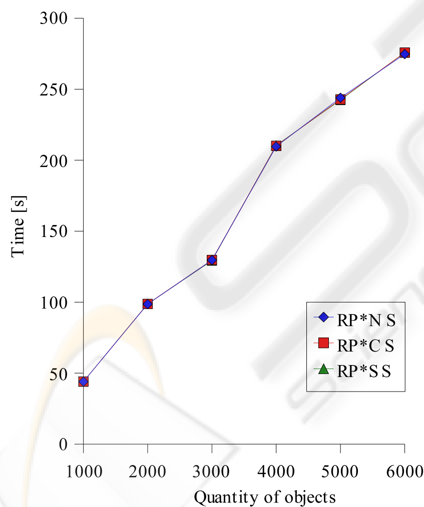

Figure 1: Total time of adding and retrieving all objects in separated segment of the network.

When the servers and the client operated in distinguished segment of the network then times of adding and retrieving of single objects for RP*N, RP*C and RP*S were almost equal. Only RP*N had longer time of adding (about 2.6%) and shorter time of retrieving (about 4.3%). Performances of RP*C and RP*S were very similar.

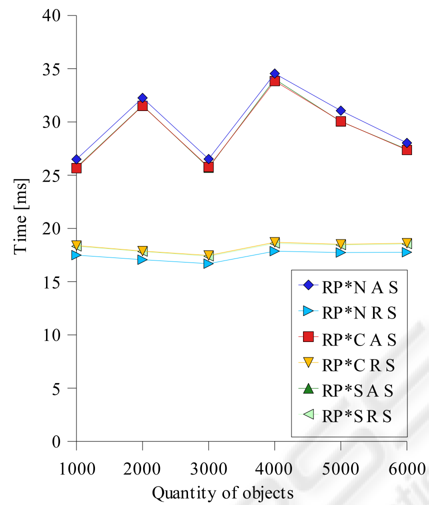

Figure 2: Average time of adding and retrieving of single objects in separated segment of the network.

For the second part of the experiment there was no distinguished segment of the network (the multicomputer was embedded into the network - N in the Figures). Results of the experiment are shown on Figures 3, 4 and 5.

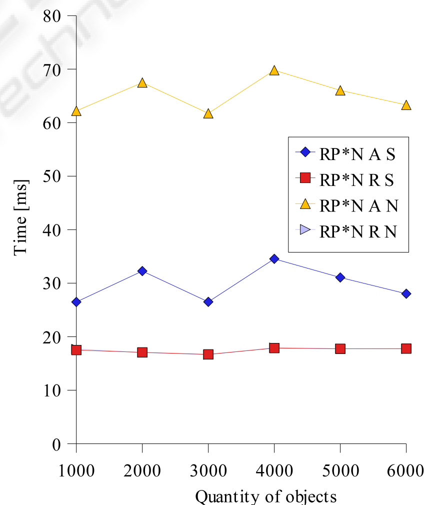

Figure 3: Average time of adding and retrieving of single objects for RP*N.

When the servers and the client were embedded into the network then RP*N differed significantly from the others. Multicast messages were sent to other segments of the network, what took considerable amount of time. The only average time of adding objects was approximately twice longer. The most similar values had RP*C and RP*S as in the first part of the experiment. Any significant difference in RP*S and RP*C

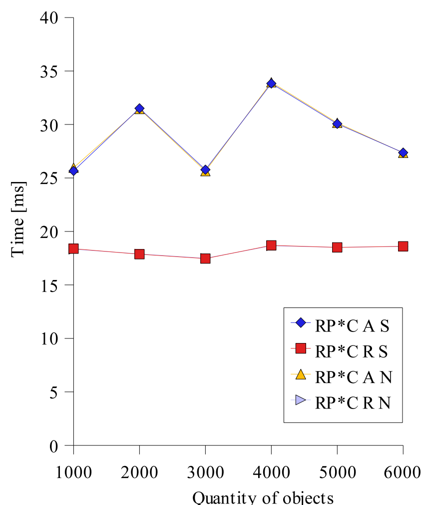

Figure 4: Average time of adding and retrieving of single objects for RP*C.

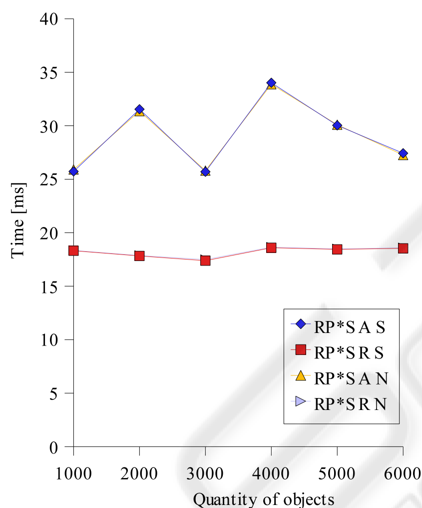

Figure 5: Average time of adding and retrieving of single objects for RP*S.

operated on separated or not separated segment of the network was noticed. These architectures do not use multicast transmission at all or very rarely.

7 CONCLUSIONS

The paper presents partial results of the research concerning development of object-oriented versions of Scalable Distributed Data Structures, namely OORP* and OOLH*. It concentrates on main aspects of OORP*. In the research Java was chosen as a platform for implementation of OORP*. Java is safe, portable and well known programming language. It allows for developing distributed applications easily and rapidly. Java makes it also possible to integrate an application with another ones and distribute it in the Internet.

The OOSDDS can be used as a part of distributed object store. It can store Java objects in serialized form. In OORP* addressing algorithm is same as in RP*. However, buckets can store various numbers of objects. The split algorithm is then modified and based not on the number of objects but on the size of objects.

As far as performances is concerned all OORP* architectures are similar when operate in separated segments of a network. However, RP*N works significantly slower than RP*S and RP*C when all operate in not separated segments of a network. Moreover, not always architectures using multicast transmission can be applied here because not all software and hardware configurations support correctly multicast transmission. Not separated segments of a network may be more difficult to control what could affect a performance of OOSDDS.

Evaluation of the other version of OOSDDS, namely OOLH* which is under development now, is the subject of further research.

REFERENCES

Devine, R. (1993). Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. In *4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 101–114.

DeWitt, D. and Gray, J. (1992). Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98.

Litwin, W. and Neimat, M.-A. (1997). LH*s: A high-availability and high-security scalable distributed data structure. In *7th International Workshop on Research Issues in Data Engineering*.

Litwin, W., Neimat, M.-A., and Schneider, D. (1994). RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Twentieth International Conference on Very Large Databases*.

Litwin, W., Neimat, M.-A., and Schneider, D. (1996). LH* - a scalable, distributed data structure. *ACM Transactions on Data Base Systems*, 21(4):480–525.

Lo, Y.-L., Hua, K., and Young, H. (2001). GeMDA: A multidimensional data partitioning technique for multiprocessor database systems. *Distributed and Parallel Databases*, 9(3):211–236.

Lobry, O., Collet, C., and Déchamboux, P. (1997). The VIRTUOSE Distributed Object Store. In *DEXA workshop*, pages 482–487.

Sapiecha, K. and Łukawski, G. (2006). Fault-tolerant Protocols for Scalable Distributed Data Structures. In *6-th International Conference on Parallel Processing and Applied Mathematics PPAM*, pages 1018–1025.

Stonebraker, M. (1986). The Case for Shared Nothing. *Database Engineering Bulletin*, 9(1):4–9.