Scalable Computing: Practice and Experience
Volume 8, Number 3, pp. 241–248. http://www.scpe.org
ISSN 1895-1767
© 2007 SWPS

HOW TO ACHIEVE HIGH THROUGHPUT WITH DYNAMIC TREE-STRUCTURED COTERIE [*]

IVAN FRAIN, ABDELAZIZ M'ZOUGHI, JEAN-PAUL BAHSOUN[†]

Abstract. Data replication permits a better network bandwidth utilization and minimizes the effect of latency in large-scale systems such as computing grids. However, the cost of maintaining the data consistent between replicas may become difficult if the read/write system has to ensure sequential consistency. In this paper, we limit the overhead due to the data consistency protocols by introducing a new dynamic quorum protocol called the *elementary permutation protocol.*This protocol permits the dynamic reconfiguration of a tree-structured coterie [2] in function of the load of the machines that possess the data replicas. It applies a tree transformation in order to obtain a new less loaded coterie.This permutation is based on the load information of a small group of machines possessing the copies. The implementation and the evaluation of our algorithm have been based on the existing atomic read/write service of [14]. We demonstrate that the elementary permutation ameliorates the system's throughput upto 50% in the best case. The results of our simulation show that the tree reconfiguration based on the elementary permutation is more efficient for a relatively small number of copies.

Key words. data replication, dynamic quorums, tree coterie, grid

1. Introduction.

Replication permits a better bandwidth usage of the network by avoiding unnecessary data transfers between the sites. Nevertheless, high latency time exposes the replica management protocols to potential performance degradations. Among the existing replica management protocols, the quorum ones are well suitable because of their ability of diminishing the number of exchanged messages for the Read/Write operations applied to the copies. To perform an operation, copies of a quorum (read or write) must be contacted to insure consistency among the replicas. The set of all the possible quorums is called a coterie [11]. We can make a distinction between majority-based quorum systems [12, 19] and structured quorum systems [2, 13, 8, 15]. The former uses the majority of the replicas (possibly weighted) to construct the quorums. The latter uses a logical organization of the copies to diminish the quorum's cardinality and thus the number of exchanged messages of an operation.

Many works focus on quorum systems' performance improvement. They usually concentrated on the latency between processors that maintain the copies to construct adapted structured-coterie [20, 10, 6]. The authors of [7] proposed an algorithm for the creation of geographic quorums. They created a coterie in such a way that the distance between any client and any quorum is optimal. Their solutions are based on the distance between the sites, which is a static value and is related to the used accessing media's physical time latency and not to the loads of the machines or the network. There is a dynamic characteristic that must be taken into more consideration than the static one, which is the load of the processors [5]. We generally associate this load to the service response time of an operation. As the load increases, the service response time becomes longer.

In distributed environments like computing grids [9], the grid scheduler can not have total control over the nodes to which it delegates tasks. In fact, a shared machine in the grid is not always dedicated to the task that the scheduler has granted to it. The local user of the machine is its only master and hence he can ask it to realize tasks of which the grid system has no knowledge about them and that it can not quantify (in terms of the load) in advance. Moreover, computing grids are characterized by certain common properties such as weak bandwidth and high latency between the sites, distinct administrative domains and strong heterogeneity among the resources. Therefore such an environment is the perfect context to manage replicated data and use structured quorum consensus protocols based on the processor's load.

The problem to which this paper addresses is the dynamic reconfiguration of a tree-structured coterie in function to the load of its processors. A processor can be a node, a personal computer, or a single storage resource in a grid environment. What is important is the fact that a processor has a quantity of work to fulfill that we characterize it as its load and possesses a replica.

In this paper, we present a new tree-based coterie reconfiguration scheme used in a multi-reader/writer fault-tolerant algorithm. Our main contributions are:

1. The definition of quorum and coterie loads in order to construct a coterie based on the processors' loads.

[*]This work is part of the french RNTL project ViSaGe which is supported by the french ministry for research n°04K459.

[†]Institut de Recherche en Informatique de Toulouse, Université Paul Sabatier 118, route de Narbonne, 31062 Toulouse Cedex 9, France {frain,mzoughi,bahsoun}@irit.fr

2. The introduction of a new reconfiguration scheme to the tree-structured coterie of [2]. This reconfiguration is based on the processors' loads and permits to diminish the coterie's overall load.
3. The extension of the algorithm of [14] to embed our elementary permutation. The extension is made to take into account the following three policies: an information policy, a selection policy and a reconfiguration policy. The information policy is used to collect the processor's load. The selection policy is used to choose the right moment of reconfiguration. The reconfiguration policy applies one or several above mentioned elementary permutation.
4. The implementation of this extended algorithm in the neko simulator [21] to demonstrate the performance improvement of our solutions. We show that throughput is improved of 50% by the elementary permutation under certain circumstances.

Related Work. There exists extensions of the different quorum protocols that permit the reconfiguration of a coterie when one of the nodes crashes (crash-stop) [1, 17, 3, 16]. When the failure of one node is detected, a new coterie is constructed with n-1 nodes. In our solution, in addition to being active or not, we take into account the availability of a node on its load basis. In [4], they focus on byzantine quorum systems and discuss on the quorum's load. However, their load definition of quorums differs from that of ours which will be given in the coming sections. In fact, the authors consider the inherent load of the coterie by taking into account the structure of the coterie and the accessing probability of a quorum but not the workload of each node.

In [18], the ViSaGe project was presented. This grid level software's objective is to provide to the grid community a flexible storage virtualization service. ViSaGe will permit to share storage resources in a transparent manner and with some levels of quality of services. An administrator of such a service can choose to plug any consistency management protocol such as the protocols we introduce in this paper.

The rest of the paper is organized as follows. In the following section, we present our load model. Sections 3 introduces the elementary permutation scheme. Section 4 presents the extension of the used read/write algorithm as well as the three policies that are introduced to integrate our permutation to this algorithm. Section 5 presents the implementation and the evaluation of performance of our proposal. The conclusion is the subject of section 6.

2. Model.

We consider P_r as the set of all processors such that $P_r = \{P\, is\, a\, processor\}$. To each processor P, a working load is associated which will be denoted by x_p. Each processor possesses a copy of the data item d. In the remaining of this paper, we will reason about only a single data item, without losing generality.

Whatever is the quorum protocol type, either a majority quorum or a structured one, all of these types are subject to two properties : the intersection and minimality properties whose definitions are given hereafter [11].

Definition 2.1. ***Coterie and quorum***
Let C be a set of groups of Pr, then C is called a ***coterie*** *if it satisfies the following condition:*

$$C = \{Q \in \mathcal{P}(Pr) | \forall Q' : Q' \in \mathcal{P}(Pr) \land Q' \neq Q \rightarrow Q \cap Q' \neq \emptyset \land Q \nsubseteq Q'\}$$

The $Q \cap Q'$ property is called the intersection property and the $Q \nsubseteq Q'$ property is called the minimality property. Each element Q of a coterie C is called a ***quorum.*** The dynamic reconfiguration algorithm of a coterie that we present in the following sections is based on the load level of the processors to decide whether to perform a reconfiguration or not. One of the most important property that we take into account is the load of a quorum.

Definition 2.2. ***Load of a quorum***
The load Y_Q of a quorum Q is the maximum *of the loads x_P of the processors P that constitute this quorum.*

$$Y_Q = Max(x_P : P \in Q)$$

We consider that the accesses to different quorums of a coterie are fairly distributed among the quorums. We define the fairness access as such:

Definition 2.3. ***Fairness access***
Let m be the number of quorums Q of a coterie C. Let R_Q be the accessing probability to a quorum Q for a Read or Write operation. Then we consider the following:

$$\forall Q \in C : R_Q = \frac{1}{m}$$

We define the load of the coterie below. It will permit us to evaluate the efficiency of a coterie with respect to another, for the same number of quorums and for the same loaded nodes.

DEFINITION 2.4. ***Load of a coterie***
We denote the load of the coterie by δ_C which is equivalent to the sum of the loads of all the quorums of C.

$$\delta_C = \sum_{Q \in C} Y_Q$$

The quorum protocol that we use in this work is the one that was presented in [2]. In the remaining of this paper, when we use the word coterie, we mean a binary tree-structured coterie.

DEFINITION 2.5. ***Binary tree-structured coterie***
The processors are logically organized in the form of a binary tree. The processors are the nodes or the leaves of the tree. A Read or Write operation is carried out on a quorum of the coterie. A quorum is obtained by taking all the processors located on any path that starts from the root and terminates at the leaves of a binary-tree.

The binary tree protocol is classified as one of the structured quorum protocols. Intersection and minimality properties are well respected by this protocol. Figure 3.1 presents an example of a binary tree-structured coterie. In this figure, there are 15 processors that contain the replicas. The in-circle numbers represent the load of the processors and the out-circle numbers represent the identity of the processors. For example, P_1 is the root of the tree and its load is 2. The light gray-colored processors $\{P_1, P_3, P_6, P_{13}\}$, form one of the eight possible quorums. In the original paper [2], the tree quorum protocol was presented with a recursive definition of quorum which take care of faulty-processes. For example, if the root of the tree crashes, a quorum will be composed of two paths from the root to the any leaves in the right **and** the left sub-trees. This definition of quorum does not degrade gracefully when processors failed so we use the extension of this protocol presented in [16] which only use the paths from the root to the leaves to be a valid quorum. If a processor fails, a new coterie is constructed.

The Problem. is to minimize the tree-structured coterie's load. This can be achieved by applying a reconfiguration to a given coterie to obtain a less loaded coterie. Next, we propose the elementary permutation and we show it diminishes the coterie's load.

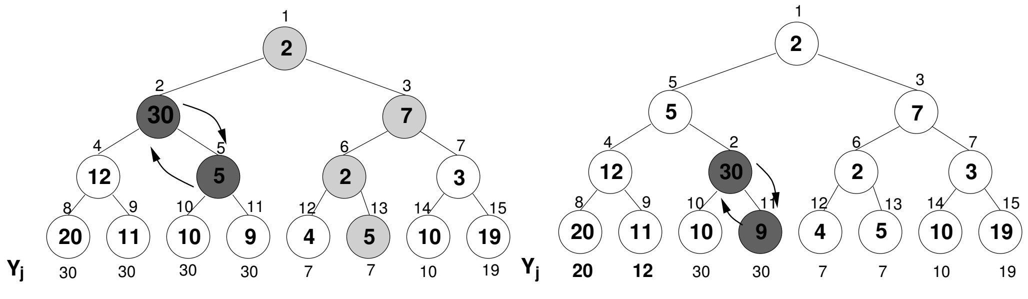
3. The Elementary Permutation.

In this section, we define the notion of an *elementary permutation* that can be used to reconfigure a tree-structured coterie. In fact, during the dynamic reconfiguration of a coterie with partial knowledge of the load of the processors, a new coterie is constructed by applying one or several elementary permutations to the previous one (see section 4.2.1).

3.1. Principle and Algorithm.

The principle of an elementary permutation algorithm is made up two steps:

1. finding a particular pattern in the tree of the form (P_a, P_b) such that P_b is the son of P_a and P_a's load is greater than P_b's load ($x_{P_a} > x_{P_b}$, see Figure 3.1).
2. if such a pattern was found, transforming it into another one (by permuting the two nodes thus P_a becomes the son of P_b) in such a way that it ameliorates the performance.



(a) $\delta_C = 163$ (b) $\delta_C = 135$

FIG. 3.1. Elementary permutation sequence

Figure 3.1 illustrates the application of several elementary permutations to a coterie. The algorithm 1 presents a more precise definition of an elementary permutation. In this algorithm, we also introduce a node and a binary tree coterie data types.

Algorithm 1: The Elementary Permutation Algorithm

```
type Node is record (name:String;load:Int)
type TreeCoterie is array(1..N) of Node
/* The first item of a TreeCoterie tc is tc[1] and corresponds to the root of the tree.
The left child of tc[i] corresponds to tc[2i] and the right child to tc[2i+1] */
Input: c:TreeCoterie,child:Int
Output: c':TreeCoterie
Data: nodeTemp:Node
begin
    c'←c
    if c'[child].load < c'[⌊ child/2 ⌋].load then
        /* Permutation between the parent and the child */
        nodeTemp←c'[child]
        c'[child]← c'[⌊ child/2 ⌋]
        c'[⌊ child/2 ⌋]← nodeTemp
    return c'
end
```

3.2. About Coterie's Load. By applying an elementary permutation to the tree, the performance must be ameliorated. The metric that we have taken to measure the gain in performance is the overall load of the coterie (Definition 2.4). An elementary permutation must at best diminish this load and at worst must not increase it.

Given two coterie configurations C and C' such that C' is obtained by applying an elementary permutation to C. We consider δ_C and $\delta_{C'}$ as the loads of the coteries C and C' respectively. If we consider the definition of the elementary permutation to be the same as defined previously (algorithm 1), then we must have $\delta_{C'} \leqslant \delta_C$.

Let us consider the levels of the nodes of the tree in the following manner: the nodes at the leaves are at level 0 and the root's node is at the highest possible level. According to the tree-structured coterie definition, we deduce that a node at level i belongs to 2^i quorums. An elementary permutation is applied to two nodes, the parent P_a^{i+1} at level $i+1$ and its child P_b^i at level i, if and only if $x_{P_a^{i+1}} > x_{P_b^i}$. Thus after the permutation, the more loaded node P_a^{i+1} will be at level i, hence we denote it by P_a^i whereas the less loaded node P_b^i, will be at level $i+1$, hence we denote it by P_b^{i+1}. So P_a^i will be contained in 2^i quorums whose loads remain the same and P_b^{i+1} will be in 2^{i+1} quorums distributed in the following manner:

- $(2^{i+1} - 2^i)$ quorums whose loads may have diminished because $x_{P_a^{i+1}} > x_{P_b^i}$ (the dark-gray colored left sub-tree of Figure 3.2)
- the other 2^i quorums of P_a^i (the light-gray colored sub-tree of Figure 3.2)

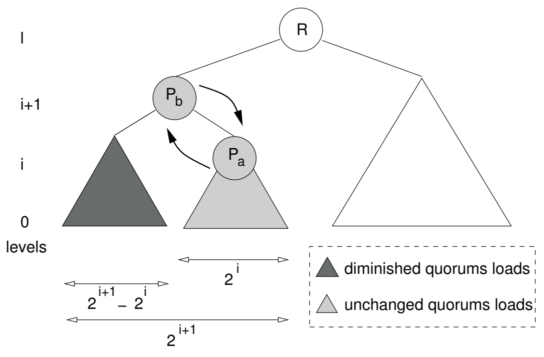
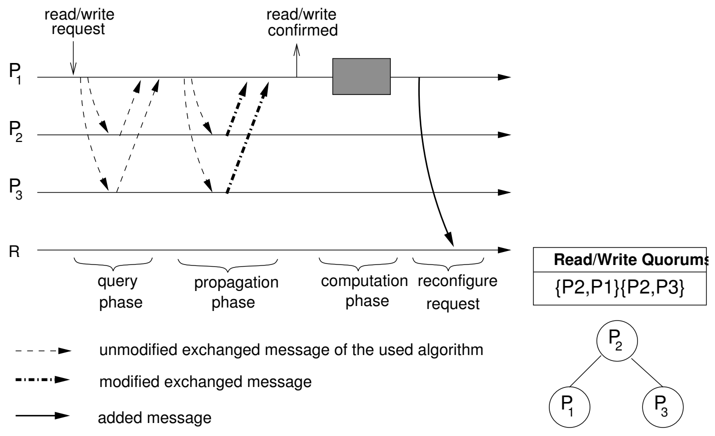


FIG. 3.2. *After an elementary permutation between P_a and P_b*

4. The Read/Write Algorithm. Our elementary permutation of the tree-structured coterie must be embedded in a suitable read/write algorithm. This algorithm must take care of concurrent accesses as well



(a) Read/Write operation sequence (b) 3 processors of a tree coterie

FIG. 4.1. *Quorum based Read/Write Atomic Service*

as dynamic reconfiguration of the tree-structured coterie with our permutation scheme. We present next an existing algorithm chosen from the literature of distributed systems. This algorithm is the one presented in [14].

4.1. The Used Atomic Read/Write Service. This algorithm is composed of two interfaces: the user interface that permits to access data by the well known read/write operations and the management interface which is used to reconfigure the current coterie.

User Interface. The read/write operations of a data item consist of two phases: a query phase and a propagation phase. At the time of the query phase, a read quorum is contacted and each processor of the quorum returns the value and the version of their local replica as well as the value and the version of their current coterie. Once all the answers are collected, the most recent version of the data is extracted. According to the operation, this recent version is either incremented and propagated (write) or simply propagated (read). In this propagation phase, the new value and the new version of the data is assigned to a write quorum. These two phases are carried out systematically for a read or write operation, that makes it possible to update the obsolete copies even when a data is read.

Figure 4.1 illustrates these two phases. Sub-figure *(a)* shows us the read/write protocol. Sub-figure (b) gives us the used three processors of a coterie and its corresponding read/write quorums. We emphasize that in the tree-structured coterie, read and write quorums are identical.

Management Interface. The service also possesses a management interface which makes it possible the dynamic reconfiguration of the used coterie. A reconfiguration can be carried out as the read or write operations are being performed. A reconfigurer is in charge of the reconfiguration process. It can be either an elected or a dedicated processor. The reconfiguration protocol is composed of three phases. During the installation phase, the reconfigurer contacts a minimal group of processors. The contacted group is the union of a read quorum and a write quorum to which the new configuration is send by the reconfigurer. The processors return to the reconfigurer the value and the version of their local replica. When all the answers are arrived, the reconfigurer enters the propagation phase. During this phase, a write quorum of this new coterie is contacted which guarantees the consistency among the replicas. Finally, the confirmation phase confirms the installation of the new configuration by sending it to a write quorum of this new coterie.

4.2. Extensions of the Atomic Read/Write Service. We propose to extend the atomic read/write service by adding three functionalities which are beyond the scope of [14] and that permit to realize our elementary permutation.

These functionalities are as follows:

1. **an information policy**: to gather information concerning the load of each processor,
2. **a selection policy**: to define the possible and convenient moment of reconfiguration,
3. **a reconfiguration policy**: to choose one of the previously defined permutations to apply if a reconfiguration can be carried out.

Next, we introduce the extensions which correspond to the elementary permutation.

4.2.1. Elementary Permutation Based Extension. Here we describe our elementary permutation based extension of the read/write atomic service. This extension consists of the following three policies.

The information policy. An elementary permutation can be carried out by having the load information of the processors that must be permuted. Hence, the major role of the information policy is to acquire, during the propagation phase of the read/write operations, the load of the processors of a quorum. The collected loads are enough to apply one or more elementary permutations within only one quorum: a path from the root to a leaf.

The selection policy. The choice of when to reconfigure is the major role of the selection policy. The question here is when to apply one or more elementary permutations. This choice is made naturally at the time of each operation, once the propagation phase is completed and the operation is confirmed. Each operation leads to contact a quorum. If this quorum contains a pattern where a parent is more loaded than a child then an elementary permutation can be carried out.

The reconfiguration policy. After each propagation phase, once the loads are known and the patterns are identified in the used read/write quorum, all possible elementary permutations can be applied. So after the reconfiguration, the path from the root to a leaf contains the processors in descending order of loads. The less loaded processor of the initial quorum is at the root and the more loaded one is at the leaf.

In Figure 4.1 the propagation phase's bold lines correspond to our information policy. Just after the propagation phase, the processor performs the reconfiguration policy by computing all the permutations that can be achieved. We call this phase a *computation phase.* If there exists one or several permutations to be applied, the new configuration is sent to the reconfigurer in order to perform the actual reconfiguration which is depicted as the *reconfiguration request.*

5. Performance Evaluation.

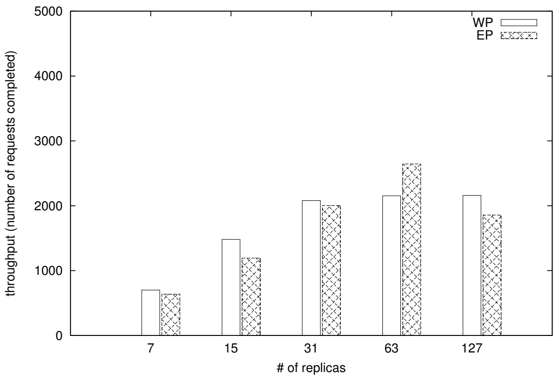
In order to evaluate our algorithm, we implemented it in the Neko simulation environment [21]. We then proceeded to a simulation campaign where we studied several different characteristics such as throughput and scalability.

We realized each simulation by taking into consideration the following two cases: without reconfiguration (**WP**) and with elementary permutation (**EP**). For each case, we used different numbers of replicas: 7, 15, 31, 63 and 127, each corresponding to a number of processors. Each processor has its own load that can evolve randomly during the simulation. The time during which the load remains constant is called the session time. The session time follows the Poisson law that permits a long enough session. The number of read/write requests executed by each processor is also a parameter of the simulation. What we first found out is the fact that there is a strong relationship between the session time and the number of requests in our simulation results. So we took into account different number of requests per session to present our simulation results. The simulation time was fixed so that we can compare the number of confirmed requests of our different cases.

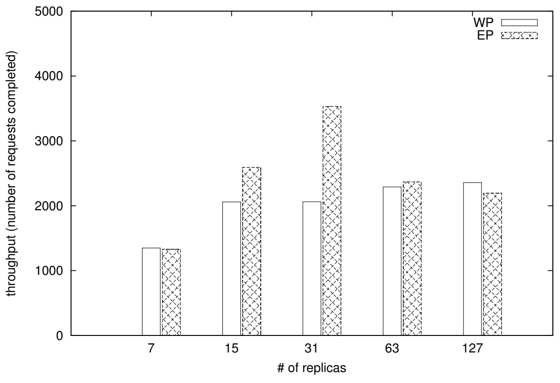
5.1. The Impact of the Number of Requests. Figure 5.1 represents the throughput as a function of the number of replicas. Each sub-figure corresponds to a specific number of requests per session. The first observation we can make is that the differences between **WP** and **EP** are more significative when the number of requests per session is high. At the beginning of a new session a well loaded coterie is naturally configured using one or several elementary permutations. The more requests occurs before the next session, the more important is the impact of the first permutation. To sum up, the ratio between the number of requests and the number of reconfigurations is higher when there are more requests per session. The results shown in Figure 5.1 illustrate that **EP** is better when there are up to 50 requests per session.

5.2. Low Scalability of the Elementary Permutation Algorithm. Even if there is a high number of requests: up to 50 requests, there are cases where **EP** have a lower throughput than **WP**. Figure 5.1(b) and 5.1(c), the case with 127 replicas performs better throughput without permutation than with elementary permutation. If there is a large number of replicas, there are too much reconfigurations and the benefits of using elementary permutations are lost.

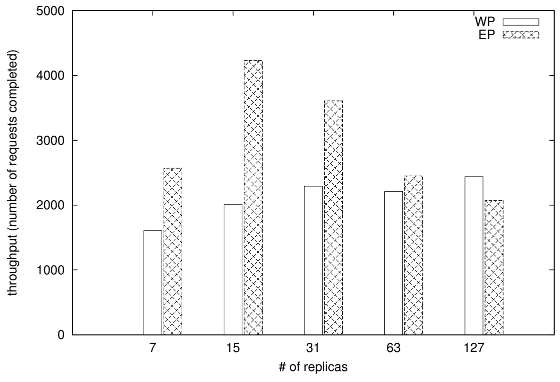
5.3. Increased Throughput with Elementary Permutation. In the others cases, when there aren't a lots of replicas and with a high number of requests per session, **EP** permits to have better throughput than **WP**. In several cases, Figure 5.1(b) for 15,31 and 63 replicas and Figure 5.1(c) for 7,15,31 and 63 replicas, we notice an enhanced throughput which is between 20% and 50% with respect to a non permuted system.



(a) 25 requests per session



(b) 50 requests per session



(c) 100 requests per session

FIG. 5.1. The throughput as a function of the number of replicas. There are two different represented series: **WP** (Without Permutation) and **EP** (Elementary Permutation)

6. Conclusion and Future Work. We have linked the construction of a coterie to the loads of its processors. We have defined the notion of a quorum's load as well as coterie's load. These definitions helped us to propose a new reconfiguration protocol to apply to a tree-structured coterie: the principle of *elementary permutation*. We have shown that this permutation protocol permits to ameliorate the load of a coterie. We have extended an atomic read/write algorithm that permits dynamic reconfigurations of a coterie so that we can embed our permutation protocol. The simulation campaigns that we have carried out thanks to the Neko simulator, showed us the benefits of our simulations. The elementary permutation allows to improve the throughput by 50% for a small number of processors and a large number of requests per session. But it is subject to the scale. If there is an important number of nodes, do not apply any permutation seems to be better.

We claim that our algorithm can be used in a grid computing environment. It could be difficult to use this protocol in peer-to-peer grid environment because of the high number of replicas but the *elementary permutation* protocol can be used in collaborative computing if there is no need to have a large number of replicas. Elementary permutation protocol can also be used in data intensive read/write applications if the number of requests is high and if it is needed to maintain sequential consistency between data copies.

In this paper we show that the elementary permutation algorithm does not scale well. If the number of replicas is large, do not apply any permutation is better. One of our major concern is to find an algorithm that can resolve this scalability problem. Another issue that our algorithm does not address is the network delay. This is done in [20] where the authors map quorum onto a physical network with a fixed topology. However they left as an open problem to take into account the service time. We do the opposite and we didn't use the network delay in our load model. It would be interesting to construct coteries taking into account the network latency and the processor's load to have a more precise load model and to find a more suitable solution.

REFERENCES

[1] A. E. Abbadi, D. Skeen, and F. Cristian, *An efficient, fault-tolerant protocol for replicated data management*, in Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems, ACM Press, 1985, pp. 215–229.

[2] D. Agrawal and A. E. Abbadi, *An efficient and fault-tolerant solution for distributed mutual exclusion*, ACM Transactions on Computer Systems, 9 (1991), pp. 1–20.

[3] ———, *Using reconfiguration for efficient management of replicated data*, IEEE Transactions on Knowledge and Data Engineering, 8 (1996), pp. 786–801.

[4] L. Alvisi, E. T. Pierce, D. Malkhi, M. K. Reiter, and R. N. Wright, *Dynamic byzantine quorum systems*, in DSN'00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8), Washington, DC, USA, 2000, IEEE Computer Society, p. 283.

[5] P. Barford and M. Crovella, *Critical path analysis of TCP transactions*, in ACM SIGCOMM Internet Measurement Workshop, June 2001.

[6] G. Cao and M. Singhal, *A delay-optimal quorum-based mutual exclusion algorithm for distributed systems*, IEEE Trans. Parallel Distrib. Syst., 12 (2001), pp. 1256–1268.

[7] P. Carmi, S. Dolev, S. Har-Peled, M. J. Katz, and M. Segal, *Geographic quorum systems approximations*, in to appear in algorithmica, December 2003.

[8] S. Cheung, M. Ammar, and A. Ahamad, *The grid protocol: A high performance scheme for maintaining replicated data*, IEEE Transactions on Knowledge and Data Engineering, 4 (1992), pp. 582–592.

[9] I. Foster, C. Kesselman, and S. Tueke, *The anatomy of the grid - enabling scalable virtual organizations*, Intl J. Supercomputer Applications, (2001).

[10] A. W. Fu, *Delay-optimal quorum consensus for distributed systems*, IEEE Transactions on Parallel and Distributed Systems, 8 (1997), pp. 59–69.

[11] H. Garcia-Molina and D. Barbara, *How to assign votes in a distributed system*, Journal of the ACM, 32 (1985), pp. 841–860.

[12] D. K. Gifford, *Weighted voting for replicated data*, in Proceedings of the seventh ACM symposium on Operating systems principles, ACM Press, 1979, pp. 150–162.

[13] A. Kumar, *Hierarchical quorum consensus: A new algorithm for managing replicated data*, IEEE Transactions on Computers, 40 (1991), pp. 996–1004.

[14] N. A. Lynch and A. A. Shvartsman, *Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts*, in FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97), IEEE Computer Society, 1997.

[15] D. Peleg and A. Wool, *Crumbling walls: a class of practical and efficient quorum systems*, Distributed Computing, 10 (1997), pp. 87–97.

[16] M. Rabinovich and E. D. Lazowska, *The dynamic tree protocol: avoiding "graceful degradation" in the tree protocol for distributed mutual exclusion*, in Conference Proceedings., Eleventh Annual International Phoenix Conference on Computers and Communications, Scottsdale, AZ, USA, April 1992, pp. 101–107.

[17] ———, *Improving fault tolerance and supporting partial writes in structured coterie protocols for replicated objects*, in Proceedings of the 1992 ACM SIGMOD international conference on Management of data, ACM Press, 1992, pp. 226–235.

[18] F. Thibolt, I. Frain, and A. M'Zoughi, *Virtualisation du stockage dans les grilles informatiques* , in 16me Rencontres Francophones du Paralllisme, (Renpar'05) , Croisic, France, ASF/ACM/Sigops, 6-8 avril 2005, pp. 219–224.

[19] R. Thomas, *A majority consensus approach to concurrency control for multiple copy databases*, ACM Transactions on Database Systems, 4 (1979), pp. 180–209.

[20] T. Tsuchiya, M. Yamaguchi, and T. Kikuno, *Minimizing the maximum delay for reaching consensus in quorum-based mutual exclusion schemes*, IEEE Transactions on parallel and distributed systems, 10 (1999), pp. 337–345.

[21] P. Urban, X. Dfago, and A. Schiper, *Neko: A single environment to simulate and prototype distributed algorithms*, in 15th Int'l Confereence on Information Networking (ICOIN-15), 2001, pp. 503–511.

Edited by: Dana Petcu
Received: May 31, 2007
Accepted: June 18, 2007