How to Improve the Scalability of Read/Write Operations with Dynamic Reconfiguration of a Tree-Structured Coterie

Ivan Frain Robert Basmadjian Jean-Paul Bahsoun
Abdelaziz M'zoughi
Institut de Recherche en Informatique de Toulouse - IRIT
Université Paul Sabatier - Toulouse III
{frain,basmadji,bahsoun,mzoughi@irit.fr}

Abstract

In large-scaled environments such as Computing Grids, data replication is used to permit a better bandwidth usage of the network. Nevertheless, high latency time exposes the replica management protocols to potential performance degradations. Two new protocols are presented in this paper[1] *in order to avoid this degradation. Their goal is to ameliorate the access performance of replicated data in computing grids. The two protocols permit the dynamic reconfiguration of a tree-structured coterie [2] in function of the load of the machines possessing the data replicas. Each of the protocols permits to apply a tree transformation. The* elementary permutation *can be performed by having the load information of a small group of machines possessing the copies, whereas a* global permutation *must have the load information of all the machines that posses the copies. The implementation and the evaluation of our algorithms have been based on the existing atomic read/write service of [14]. We demonstrate that the permutations permit to ameliorate the system's throughput. The results of our simulation show that tree reconfiguration based on the elementary permutation is more efficient for a small number of copies. The global permutation scales well and is more efficient when there is a large number of replicas.*

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 consitency among the replicas. The set of all the possible quorums is called a coterie [11]. We can make a distinction between majority-based quorum [12] systems and structured ones [2, 13, 7]. 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 [10]. The authors of [6] 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, 18]. 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 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) them 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

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

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.

Hence we target to the problem to which this paper addresses, that is the dynamic reconfiguration of a tree-structured coterie in function to the load of its processors. A processor can be a node, a personnal 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.

Contributions In this paper, we present two new tree-based coterie reconfiguration schemes 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.

2. The introduction of two reconfiguration schemes to the tree-structured coterie of [2]. These reconfigurations are based on the processors' loads and permit to diminish the coterie's overall load. The first reconfiguration scheme is called *elementary permutation* of the tree coterie and is based only on the load of any quorum of a coterie. The second is called *global permutation* and is based on the whole coterie's load information.

3. The extension of the algorithm of [14] to embed our elementary and global permutations. 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 mentionned permutations.

4. The implementation of this extended algorithm in the neko simulator [20] to demonstrate the performance improvement of our solutions. We show that throughput is improved by both permutations : the elementary and the optimal. Our simulations leverage interesting results on scalability issues. So we noticed that the global permutation based algorithm performs better than the elementary one when there are plenty of processors. It's the opposite when the number of processors decreases.

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. For example, if a node belongs to all quorums, then it will be more loaded than a node which belongs only to one quorum. On the contrary, we took into account the workload of a node to construct a new coterie whose load is less than the previous one.

These different previous works focalise on static systems in constract of mobile, dynamic systems. More recent works focalize on mobile computing where client are volatile and move across different sites. In [8] the authors present an algorithm for the construction of a geographic quorum system which is optimal in terms of distance between a client and the nearest quorum. They use computational geometry to achieve their goal. The problem exposed in [15], is a bit different then the previous one because the authors do not suppose that servers are static. They want to form a quorum of mobile nodes. They introduced the dynamic path quorum system as a good candidate to resolve the problem of node dynamicity.

In [19], 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 ressources 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 will introduced later.

The rest of the paper is organized as follows. In the following section, we present our load model. Sections 3 and 4 introduce our two permutation schemes, the elementary and the optimal permutations respectively. Section 5 presents the extension of the used read/write algorithm as well as the three policies that are introduced to integrate our permutations to this algorithm. Section 6 presents the implementation and the evaluation of performance of our proposals. The conclusion is the subject of section 7.

2 The Coterie's Load Model and the Problem

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 [11] whose definitions are given hereafter.

Definition 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:

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

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 algorithms of a coterie that we present in the following sections are 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 that we define it as such.

Definition 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 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 the configuration of a coterie with respect to another, for the same number of quorums and for the same loaded nodes.

Definition 4. Load of a coterie
Let m be the number of quorums of a coterie C. 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 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.

This protocol is classified as one of the structured quorum protocols. Intersection and minimality properties are well respected by this protocol. Figure 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.

The Problem *The problem is to minimize the tree-structured coterie's load. This can be achieved by applying a reconfiguration to a given coterie. Next, we propose two possible transformations called elementary permutation and global permutation. We show that both of them diminish the coterie's load.*

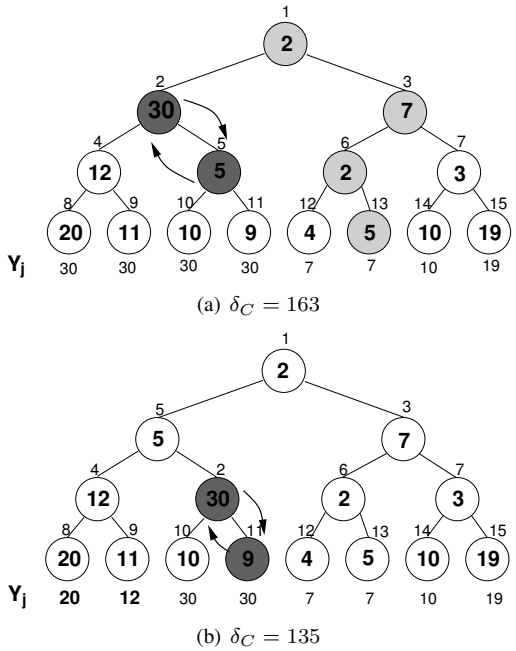


Figure 1. Elementary permutation sequence

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 5.2.1).

3.1 Principle and Algorithm

The principle of an elementary permutation is to find a particular pattern in the tree of the form (P, P') such that P' is the son of P and $x_P > x_{P'}$(see Figure 1), and to transform it into another pattern (by permuting the two nodes thus P becomes the son of P') in such a way that it ameliorates the performance.

Figure 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. 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 subtree of Figure 2)
- the other 2^i quorums of P_a^i (the light-gray colored subtree of Figure 2)

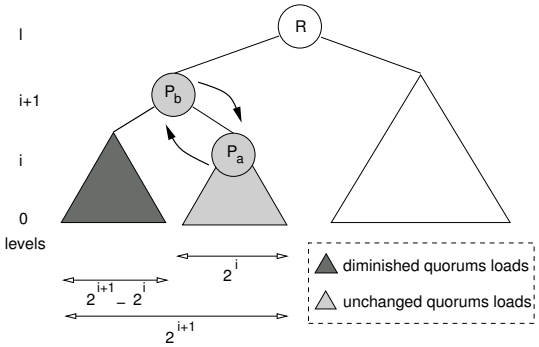


Figure 2. After an elementary permutation between P_a and P_b

3.3 Limitations of an Elementary Permutation

The application of several consecutive elementary permutations do not lead to an optimal configuration. In fact, after a certain number of these elementary permutations, all the heavily loaded processors will be located at the leaves. In such a case, an elementary permutation can no more be

applied and the performance might be degraded. For example, if the load of the processors at the leaves continues to augment then the elementary permutation becomes unsuitable. Hence, we can find a more convenient configuration where the most heavily loaded node are in the same subtree, for example. In the following section, we introduce the notion of the global permutation that permits to resolve this problem.

4 The Global Permutation

By *global permutation*, we mean another binary-tree structured coterie reconfiguration based again on the load of its processors. Such a permutation permits to obtain an optimal configuration in terms of the overall load of the coterie. In contrast to the elementary permutation, this permutation needs to know the load of every processor of a coterie.

4.1 Principle and Algorithm

By having the list of all processors' loads, we can construct an optimal coterie in terms of its overall load. To achieve this, the transformation constructs a binary tree where the processors are sorted in their decreasing load order. The construction of such a tree is performed by using the suffix first depth method[2] beginning with the most heavily loaded processor (this will be located at the left-most leaf of the tree) until the least loaded processor (this will be at the root of the tree).

Algorithm 2: The Global Permutation Algorithm

```
An array of nodes sorted by descending load:
Input: st:array(1..N) of Node
The new Coterie obtained by global permutation
Output: c':TreeCoterie

begin
    pos←globalPermutation(c',1,st,1)
    return c'
end

function globalPermutation(c:TreeCoterie,
nodePosition:Int, sortedNodes:array(1..N) of
Nodes,pos:Int):Int
begin
    if nodePosition*2>N then
        –It's a leaf level
        c[nodePosition]←sortedNodes[pos]
        return pos+1
    else
        –It's a node level
        pos←globalPermutation(c, 2*nodePosition,
        sortedNodes, pos)
        pos←globalPermutation(c,
        2*nodePosition+1, sortedNodes, pos)
        c[nodePosition]←sortedNodes[pos]
        return pos+1
end
```

4.2 About Coterie's Load

Here, we discuss the impact which can have a global permutation on the load of a coterie. We have the intuition that the new coterie's load obtained after the application of a global permutation is optimal , i.e., if δ_C is the load of the new coterie C then $\nexists C', \delta_{C'} < \delta_C$. In fact, after a global permutation, all the most loaded nodes are in the left subtree of the coterie (subtree A in Figure 3), the least loaded node is at the root of the tree and the other nodes are in the right subtree (subtree B in Figure 3). Thus one can easily notice that the root and the right subtree contain all the less loaded quorums. These quorums are of optimal load because the load of the nodes is in decreasing order from the root to a leaf. On the other side of the tree, the root and the left subtree contain the other quorums of the coterie which are the most loaded quorums. For example, in Figure 3, the two most heavily loaded nodes of the coterie are located at the leaves of this left subtree. Moreover, the third most loaded node is also contained in subtree A and its load is hidden by the two other nodes. It is the fact of hiding the third node in the subtree A which makes it possible to have an optimal load with a global permutation.

[2]Such a method is organized in the following order : left child, right child and parent.

4.3 Discussion

The difference between the elementary and the global permutations is that the latter takes into account the load information of all the processors of a coterie whereas the former only needs the load information of all the processors of a quorum. As a conclusion, we dispose two permutation protocols to ameliorate the performance of a quorum system. More precisely, the elementary permutation ameliorates the performance with a minimum cost: the load of the processors of the quorum is necessary. On the other hand, the global permutation permits to obtain an optimal configuration with a little bit higher cost because it is necessary to know the loads of all the processors of the coterie. In section 6, we present the evaluation of these two permutation protocols.

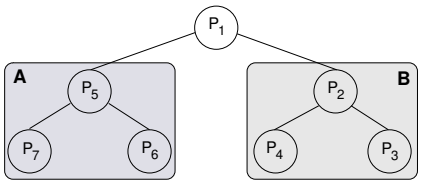


Figure 3. After a global permutation, the heavily loaded nodes are in subtree A, the least loaded node correspond to the root and the others are in subtree B, i.e., $x_{P_7} \geq x_{P_6} \geq x_{P_5} \geq ... \geq x_{P_1}$

5 Reconfiguration Algorithms

Our elementary and global permutations of the tree-structured coterie must be embedded in a suitable read/write algorithm. This algorithm must take care of concurrent accesses as well as dynamic reconfiguration of the tree-structured coterie with our permutation schemes. We present next an existing algorithm chosen from the literature of distributed systems. This algorithm is the one presented in [14].

5.1 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 illustrates these two phases. Subfigure *(a)* shows us the read/write protocol. Subfigure (b) gives us the used three processors of a coterie and its corresponding read/write quorums. 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.

Figure 5 illustrates the exchanged messages during the reconfiguration process. On Figure *5(a)*, we added the reconfigurer which initiates the reconfiguration. The new coterie obtained after this procedure is shown in Figure *5(b)*.

5.2 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 and global permutations.

These functionalities are as follows:

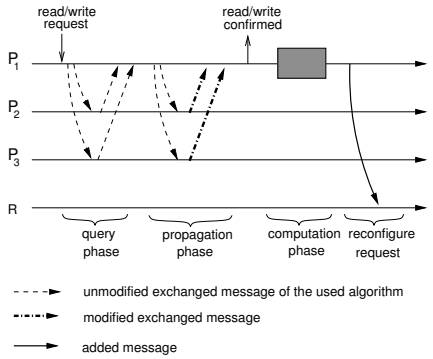
1. **an information policy**: to gather information concerning the load of each processor,
2. **a selection policy**: to define the possible and convenient moment of reconfiguration,
3. **a reconfiguration policy**: to choose one of the previously defined permutations to apply if a reconfiguration can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

Next, we introduce the two extensions which correspond respectively to the elementary permutation and the global permutation presented in the preceding sections by specifying the different policies of each extension.

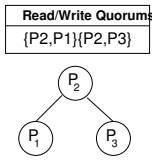
5.2.1 Elementary Permutation Based Extension

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



(a) Read/Write operation sequence



(b) 3 processors of a tree coterie

Figure 4. Quorum based Read/Write Atomic Service

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 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

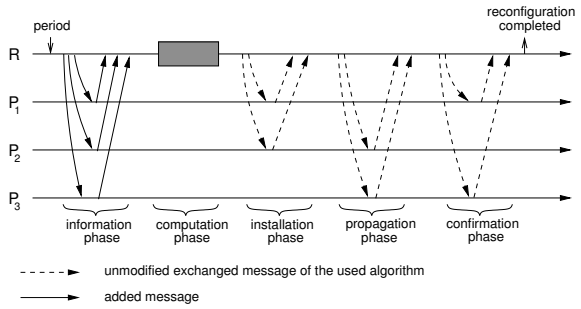


Figure 5. Dynamic coterie Reconfiguration of the Read/Write Atomic Service

sent to the reconfigurer in order to perform the actual reconfiguration which is depicted as the "reconfiguration request".

5.2.2 Global Permutation Based Extension

Like the preceding extension, we introduce here the three policies needed by the global permutation based algorithm.

The information policy A global permutation must know the loads of all the processors of the coterie in order to be applied. The information policy of this extension cannot be satisfied with the information of loads of only one quorum. We add an information phase in the reconfiguration process (see Figure 5). This information phase is realized before the three phases of the reconfiguration process (installation, propagatoion and confirmation). This permits to acquire the load information of all the processors of the coterie.

The selection policy The choice to process a reconfiguration is periodic. If the period is reached, the reconfigurer processor starts the reconfiguration phase by first executing the information phase. The periodicity of verification is a parameter of the selection policy. We discuss it in the evalaution section.

The reconfiguration policy The reconfiguration policy of this extension is obvious. The new configuration is obtained by applying a global permutation on the current tree coterie.

6 Implementation and Evaluation

In order to evaluate our both algorithms, we implemented them in the Neko simulation environment [20]. We then proceeded to a simulation campaign where we studied several different characteristics such as throughput and scalability. We also studied the impact of the global permutation's periodicity parameter. We present these different

results in the subsequent subsections but we first introduce the chosen simulation process.

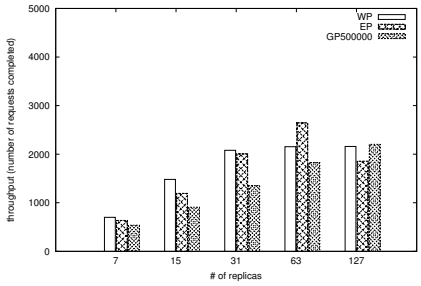
We realized each simulation by taking into consideration the following three cases: without reconfiguration (**WP**), with elementary permutation (**EP**) and with global permutation (**GP**). For each case, we used different numbers of replicas: 7, 15, 31, 63 and 127, each corresponding to a number 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 diiferent 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.

The first obtained results indicated us that the fact of applying the elementary and global permutations permit to augment significantly the number of performed read/write operations whenever the number of requests per session is high. Figure 6, represents the throughput as a function of the number of replicas. Each subfigure corresponds to a specific number of requests per session. From 50 requests per session and on (subfigures 6*(b)*,6*(c)*,6*(d)*), one of the two permutation algorithms is always better than the one with no permutation. For a low number of requests per session, performing no permutation at all seems the most convenient solution (6*(a)*).

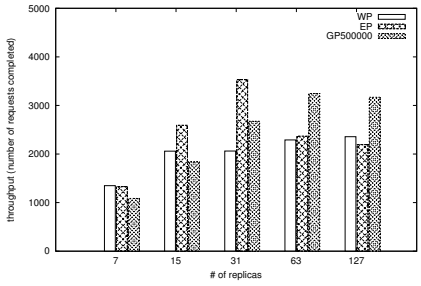
Next, we precisely define the differences between the elementary and global permutations' simulation results by taking into account the cases where the number of requests per session exceeds 50 (subfigures 6*(b)*,6*(c)*,6*(d)* and Figure 7).

6.1 Increased Throughput with Elementary Permutation

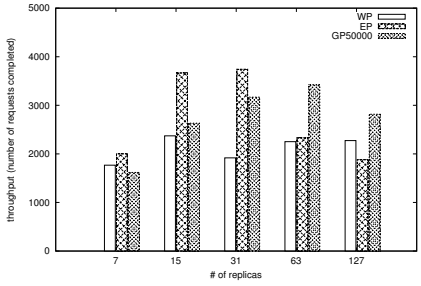
The performed simulations showed that the elementary permutation based algorithm is the one that permits to obtain the best throughput. In fact, if the number of replicas to be managed is not large, then this algorithm performs better than the others. We note that the maximum throughput is always achieved by the elementary permutations : subfigure 6*(b) and (c)* for 31 replicas and,6*(d)* for 15 replicas. For example, in this last case we notice a throughput amelioration of more than 50% with respect to a non permuted system and to 25% with respect to the global permutation based solution. We noticed that the elementary permutation algorithm permits to construct quickly a quasi-optimal



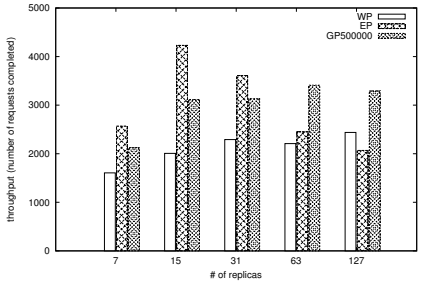
(a) 25 requests per session



(b) 50 requests per session



(c) 75 requests per session



(d) 100 requests per session

Figure 6. The throughput as a function of the number of replicas. There are three different represented series : *WP (Without Permutation), **EP** (Elementary Permutation) and **GP500000** (Global Permutation with a period of 500000 simulation steps)*

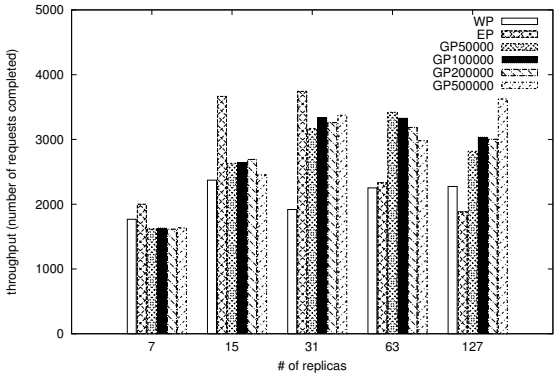


Figure 7. Various Global Permutation (*GP#*) results with different reconfiguration periods compared to Elementary Permutation (*EP*) and Without Permutation (*WP*). GP50000 is for Global Permutation with a reconfiguration period of 50000 simulation unit time. The number of requests per session is equal to 75. The fixed simulation time we used is 2M.

tree in terms of the charge of the processors, having in mind that all the quorums are fairly accessed (see definition 3). We will obtain most of the time a tree with the least loaded node at the root and the heavily loaded nodes at the leaves.

Nevertheless, when the number of nodes increases, we noticed the application of too many reconfigurations. It is in this case that a global permutation algorithm becomes more convenient. The extreme cases show that an algorithm with no permutation premits to obtain better results when the number of nodes increases. Now we treat this scalability problem with the results obtained from global permutations.

6.2 High Scalability of the Global Permutation

We can notice in Figure 7 that the global permutation algorithm is best suited for a system possessing a large number of replicas. Whatever is the number of requests per session, whenever there are 127 nodes, the throughput becomes higher than that of the other algorithms. For example, for 75 requests per session, global permutation permits to perform 50% more requests than the elementary permutations and 40% more requests than non-permuted systems. On the other hand, when the number of replicas is small, this solution is often less convenient. The main objective of the global permutation is to regroup the heavily loaded nodes into the same path in such a way that they hide each other. In fact, the greater is the size of the quorum, the more we can regroup together the nodes of equivalent loads. Hence it is reasonable to obtain the best results when the number of nodes is important.

6.2.1 The Impact of the Periodicity

Figure 7 exhibits the various global permutations. Each series corresponds to a different reconfiguration period. The **GP50000** series correponds to a global permutation of a 50K period. Knowing that the simulation time is 2M, this simulation has undergone about 40 reconfigurations. The other series such as GP100000, GP200000 and GP500000 correspond to the periods of 100K, 200K and 500K time units respectively. For 127 nodes, we can note the strong impact of the periodicity on the throughput. In fact, by choosing a large period with respect to the simulation time, we obtain throughtput ameliorations at a magnitude of 15% between GP500000 and GP200000. This is due to the number of abandoned requests during the reconfiguration phases. As the number of nodes increases, the number of abandoned requests during these phases becomes larger, so a reconfiguration becomes more expensive.

7 Conclusion

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 two types of reconfigurations applied to a tree-structured coterie : the principle of elementary and global permutations. We have shown that these two permutation protocols permit 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 schemes. The simulation campaigns that we have carried out thanks to the Neko simulator, showed us the benefits of our simulations. The elementary permutation permits to ameliorate the throughput by 25% for a small number of processors and a large number of requests per session. On the other hand, as the number of nodes increases, the global permutation produces better results hence improving the scalability.

Perspectives Even if part of this work is achieved, we want to continue working on scalability and evaluate our solutions with larger number of nodes. Peer to peer platforms may help us in realizing this task. Moreover, once the ViSaGe project releases its first version (middle of 2006), we will confirm the results presented here in a more realistic environment than a simulator. In addition, we hope to couple the load of the processors with the transfer time in order to have a finner criteria in constructing the coteries. The fact of coupling these two criterias is an interesting issue.

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*, pages 215–229. ACM Press, 1985.

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

[3] D. Agrawal and A. E. Abbadi. Using reconfiguration for efficient management of replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):786–801, October 1996.

[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)*, page 283, Washington, DC, USA, 2000. IEEE Computer Society.

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

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

[7] 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(6):582–592, December 1992.

[8] S. Dolev, S. Gilbert, N. A. Lynch, A. Shvartsman, and J. Welch. Geoquorum: Implementing atomic memory in ad hoc networks. In S.-V. LNCS, editor, *17th International Conference on Principles of DIStributed Computing*, volume 2848, pages 306–320, 2003.

[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(1):59–69, January 1997.

[11] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

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

[13] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, September 1991.

[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] M. Naor and U. Wieder. Scalable and dynamic quorum systems. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 114–122, New York, NY, USA, 2003. ACM Press.

[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*, pages 101–107, Scottsdale, AZ, USA, April 1992.

[17] M. Rabinovich and E. D. Lazowska. 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*, pages 226–235. ACM Press, 1992.

[18] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[19] F. Thiébolt, I. Frain, and A. M'Zoughi. Virtualisation du stockage dans les grilles informatiques . In *16ème Rencontres Francophones du Parallélisme, (Renpar'05) , Croisic, France*, pages 219–224. ASF/ACM/Sigops, 6-8 avril 2005.

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