http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 7 (4): 599-606, 2008
ISSN 1812-5638
© 2008 Asian Network for Scientific Information

Replicated R-Resilient Process Allocation for Load Distribution in Fault Tolerant System

Jian Wang, Jianling Sun, Xinyu Wang and Hang Chen
Computer College in Zhejiang University, Zheda Road 38, Hangzhou, Zhejiang, 310027, China

Abstract: Process allocation for load distribution can improve system performance by utilizing resources efficiently. For primary-backup based fault tolerant system, a classic load-balancing process allocation method (two-stage allocation algorithm) has been proposed that can balance the load before as well as after faults occurrence. But two-stage allocation algorithm has bad scalability since its load-balancing performance reduces dramatically when each primary process is duplicated more than once (i.e., has more than one backup process). In this study, we present an improved algorithm named RSA (R-Stage Allocation algorithm) that can have the load better balanced no matter how many backup processes each primary process owns; Simulations are also used to compare the proposed algorithm with the two-stage allocation algorithm and the experimental results show that when extending to replicated R-Resilient processes, RSA has significantly better load distribution performance than two-stage allocation algorithm.

Key words: Primary-backup, fault-tolerant, load-balancing, replicated processes, process allocation

INTRODUCTION

Process replication is heavily used by the software based fault tolerant system. How to allocate the replicated processes for load distribution is an important research area in the study of fault tolerant system. Bannister and Trivedi (1983) firstly presented an algorithm which evenly distributes the load of the system to all nodes thus improves the system performance by utilizing the resources efficiently. An assumption of their research work is that all the primary processes and the replicas play the same role; the invocations of client process are received and processed by the non-faulty replicas in the same order. This fault tolerant approach is also called Active Replication or State-Machine Approach (Mullender, 1993).

In addition, there is another important fault tolerant approach called Primary-Backup (also known as Passive Replication), in which one of the replicas called the primary plays a special role (Mullender, 1993): it receives the invocations from the client process and sends the response back. The backup replicas interact with the primary and do not interact directly with the client process.

The allocation algorithm presented by Bannister and Trivedi cannot be applied to primary-backup based fault tolerant system because the active replication uses more resources than the primary-backup approach by having all the process replicas work with the same load and execute the same client invocation. In primary-backup approach, only primary process executes the client invocation and then sends the state update message to the backup process; backup process just needs to update their state hence the load is much less than the primary process, i.e., 5~10% of the load of the primary process.

Considering this difference, Kim *et al*. (1995, 1997) consequently proposed another static allocation algorithm which can balance the load before as well as after faults occurrence for the primary-backup based fault tolerant system. They firstly formalized and proved this kind of primary-backup process allocation is an NP-hard problem, then they gave a heuristic algorithm called two-stage allocation algorithm with an assumption that each primary process is duplicated only once (only has one backup process). After that, Guo Hui *et al*. (2005) extended the two-stage allocation algorithm to heterogeneous distributed system.

However, two-stage allocation algorithm pays less consideration to a more prevalent situation when each primary process is duplicated more than once. In this case when a fault occurs, one of the backup replicas is elected to take over the role of the primary process until no more backup process is available. Although the two-stage allocation algorithm was extended (Lee *et al*., 1999) for handling this case, the critical point that the backup replicas of the same primary process start up at different time is ignored.

In this study we present a new process allocation algorithm named R-Stage Allocation algorithm (RSA) that can have the load better balanced after faults occurrence

Corresponding Author: Jian Wang, Computer College in Zhejiang University, Zheda Road 38, Hangzhou, Zhejiang, 310027, China
Tel: +86-13067850475

than the two-stage allocation algorithm no matter how many backup replicas each primary process owns. RSA specifies the backup processes startup sequence by assigning an id to every replica and utilizing the election algorithm to select the one with the smallest id to take over the role of the primary process when fault occurs.

FAULT-TOLERANT PROCESS ALLOCATION

Fault-tolerant system model: In this study, we use the similar system model considered by Lee *et al.* (1999). As shown in Fig. 1, the fault-tolerant system in this model consists of N nodes. To tolerate the fault, each process is replicated R times ($1 \le R < N$) and executed as a group, referred to as a primary-backup process group. By using R backup processes, the system can allow at most R faulty Nodes (Assuming the nodes only have fail-stop failures; the faulty nodes have all the processes running on it become unavailable). Primary processes can be allocated to any node. However, there is one restriction on the placement of backup processes. That is, the primary and backup processes that belong to one primary-backup group cannot be allocated to the same node.

It is assumed that the CPU loads of both the primary processes and the backup processes running in the system are known in advance. The assumption is valid in the computing environment where the occurrence, load and duration can be predicted. Examples of such systems include on-line transaction processing and real-time systems, in which most of the processes are running continuously or repeating in a periodic manner (Kim *et al.*, 1997; Mullender, 1993).

Furthermore, it is also assumed in this model that every backup process is assigned with an id and when a fault occurs, the role of the primary process P_i is taken over by the one with the smallest id until no more backup process for P_i is available (i.e., the backup process startup sequence for P_i is $B_{i\text{-}1}, B_{i\text{-}2},..., B_{i\text{-}r}$). This can be achieved by using some sophisticated leader election algorithms such as bully algorithm or ring-based election algorithm (Garcia-Molina, 1982; Singh and Kurose, 1994; Stoller, 2000).

The primary-backup process model considered in this study has been used heavily in the high available distributed systems such as Grid Service, Network File Server, Application Server Clusters and fault tolerant computer systems such as the Tandem Nonstop System and Delta System (Powell, 1994; Zhang *et al.*, 2004; Liu *et al.*, 2005)

Notation: The following notation is used to formulate the allocation problem:

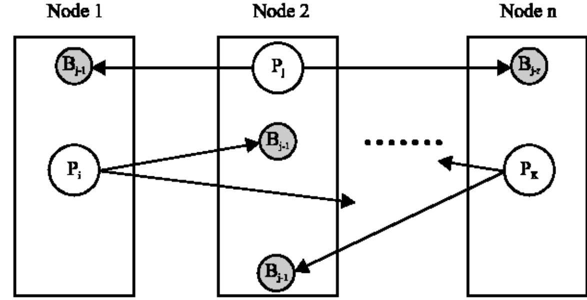

Fig. 1: Primary-backup based fault tolerant process

N	=	No. of all the nodes
M	=	No. of primary processes
a_j	=	0 if node j has failed, 1 otherwise.
V	=	The set of all the possible node status vectors in the form $[a_1a_2a_3...a_n]$
V	=	$\{[a_1a_2a_3...a_n]\}$
R	=	No. of the backup processes for each primary process $1 \le R < N$
F	=	No. of current faulty nodes. $0 \le F \le R$. Thus, $F = N - \sum_{k=1}^{N} a_k$
P_i	=	Load of primary process i
B_i	=	Load of the backup processes of primary process i (Assume every backup process belonging to one primary-backup process group has equal load)
X_{ij}	=	1 if primary process i is allocated to node j, 0 otherwise
Y_{ij}^r	=	1 if rth backup replica of primary process i is allocated to node j, 0 otherwise
T_i^r	=	1 if rth backup replica of process i is available, 0 otherwise. Thus, $T_i^r = \sum_{k=1}^{N} a_k Y_{ik}^r$
τ_j	=	Load increment to be added to node j when the fault occurs
μ_j	=	The node set that process j can be allocated on
P(j)	=	Total load of node j
$\phi(V_i)$	=	Difference between the maximum and the minimum load when given a node status vector V_i ($V_i \in V$)

Formal problem description: In this part of research, we formally describe the load balancing allocation for replicated R-Resilient fault tolerant processes.

The load-balancing process allocation problem is represented as a constrained optimization problem, which aims to minimize an objective function subject to constraints on the possible values of the independent

variable. The constraints and objective function are described below.

Let us assume that there are N nodes and M primary processes in the system. Because each primary process has R backup processes, the total number of primary and backup processes is (R+1)M. Primary processes can be allocated to any node. However the primary and backup processes that belong to the same primary-backup group cannot be allocated to the same node. Thus,

$$\sum_{i=1}^{M}\sum_{j=1}^{N} X_{ij} = M, \quad \sum_{i=1}^{M}\sum_{j=1}^{N}\sum_{r=1}^{R} Y_{ij}^{r} = RM$$
$$\sum_{i=1}^{M}\sum_{j=1}^{N}\sum_{r=1}^{R} X_{ij}Y_{ij}^{r} = 0, \quad \sum_{i=1}^{M}\sum_{j=1}^{N}\sum_{r=1}^{R}\sum_{k=1,k!=r}^{R} Y_{ij}^{k}Y_{ij}^{r} = 0 \tag{1}$$

The load increment to be added to node j in the event of a fault in node k can be represented as

$$\sum_{i=1}^{m} X_{ik}Y_{ij}^{f(i)}(P_i - B_i),$$

where, f(i) represents the id of the elected backup process replica to take over the role of the primary process i (i.e., f(i) equals 2 if the backup replica with id 1 is unavailable and the backup replica with id 2 is available). Thus,

$$f(i) = T_i^1 + \sum_{r=2}^{F}\prod_{k=1}^{r-1}\left(1 - T_i^k\right)\cdot T_i^r\, r \tag{2}$$

Hence, the total load increment for node j is:

$$\tau_j = \sum_{k=1}^{N}(1 - a_k)\left[\sum_{i=1}^{M} X_{ik}Y_{ij}^{f(i)}(P_i - B_i)\right] \tag{3}$$

The load of node j, denoted as P(j), is the sum of the load before the fault occurrence and the load increment τ_j to be incurred upon the occurrence of a fault.

$$P(j) = \tau_j + \sum_{i=1}^{M}(X_{ij}P_i + \sum_{r=1}^{N} Y_{ij}^{r}B_i) \tag{4}$$

There are two metrics for evaluating load balance between nodes. One is the standard deviation of the processor loads used in Mullender (1993). Another one is the load difference between the node with the heaviest load and the node with the lightest load (Kim *et al.*, 1997). We use the second one in order to keep the metric same as the two-stage allocation algorithm.

The objective function of load balance evaluation is formally described as below.

For a given node status vector V_i (V_i = $[a_1a_2a_3...a_n]$($V_i \in V$)), the load difference denoted as $\phi(V_i)$, can be represented as follows:

$$\phi(V_i) = \max\left(P_{(1)}, P_{(2)}, ..., P_{(N)}\right) - \min\left(P_{(1)}, P_{(2)}, ..., P_{(N)}\right) \tag{5}$$

The node status vector V_i is divided into V_{non} and V_{faulty} according to whether a faulty node exits. Moreover, V_{faulty} is divided into $V_{f\,1}$,$V_{f\,1}$, ... *et al.*, according to the number of faulty nodes exit.

The objective function φ is defined as:

$$\varphi = \sum_{V_i \in V} W_i \cdot \phi(V_i)$$
$$= \sum (W_1 \cdot \phi(V_{non}) + W_2 \cdot \phi(V_{f1}) + ... + W_i \cdot \phi(V_{fF})) \tag{6}$$

W_i (0≤i≤size of (V)) denotes the relative weights of importance before and after faults occurrence, respectively. The value for each W_i depends on the possibility of the pertinent node status occurrence in reality. However, to give an obvious comparison of the load distribution performance for the process allocation algorithms, we assume that the weights have the same value, i.e., $W_0 = W_1 = W_2 = ... = W_i = 1$. Thus, the objective function becomes:

$$\varphi = \sum_{V_i \in V} \phi(V_i) \tag{7}$$

Hence, the load balancing process allocation is the problem of finding values of X_{ij} and Y_{ij}^{k} for all possible i (1≤i≤M), j(1≤j≤N), k(1≤k≤R) that minimize the objective functions φ in our fault tolerant system model with the constraint given in Eq. 1.

HEURISTIC PROCESS ALLOCATION ALGORITHMS

In this part of study, two-stage allocation algorithm is firstly introduced with an example and then R-stage allocation algorithm is presented using the same example. The result shows that RSA improve the load distribution performance than two-stage allocation algorithm when primary processes are replicated more than once. Then the time complexity of the RSA is analyzed.

Two-stage allocation: Here we present a simple example. Assuming there are 20 primary processes (M = 20) running on 4 nodes (N = 4). Each primary process has 2 backup processes (R = 2). The processes loads are shown in Table 1.

The two-stage allocation algorithm works as follows.

In the first stage, a greedy method is used to allocate the primary process with the highest load to the node with the lowest load. Table 2 shows the allocation results after the first stage.

Table 1: Process and their loads

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
P_i	27	21	8	17	22	10	23	10	11	12	13	22	10	23	15	17	19	13	8	10
B_{i-1}	2.5	1.0	0.5	1.0	2.0	1.0	2.0	2.5	1.0	0.5	1.0	2.0	1.0	2.0	1.0	1.0	2.0	2.0	0.5	1.0
B_{i-2}	2.5	1.0	0.5	1.0	2.0	1.0	2.0	2.5	1.0	0.5	1.0	2.0	1.0	2.0	1.0	1.0	2.0	2.0	0.5	1.0

Table 2: Allocated primary process

Node 1	Node 2	Node 3	Node 4
P_1 (27)	P_2 (21)	P_3 (8)	P_{12} (22)
P_{16} (17)	P_{10} (21)	P_4 (8)	P_5 (22)
P_{15} (15)	P_{14} (23)	P_7 (8)	P_{11} (22)
P_{20} (10)	P_{18} (13)	P_{13} (10)	P_9 (11)
P_6 (10)	P_{19} (8)	P_{17} (19)	P_8 (10)
79	77	77	78

Table 3: Grouping backup processes on node 1

Node	Group	Processes	Differences (load)
	B_{11}	$(P_1-B_{1-2}) = 24.5$	49.5
		$(P_{16}-B_{16-2}) = 16$	
		$(P_{20}-B_{20-2}) = 9$	
Node 1	B_{12}	$(P_1-B_{1-2}) = 24.5$	47.5
		$(P_{15}-B_{15-2}) = 14$	
		$(P_6-B_{6-2}) = 9$	
	B_{13}	$(P_{16}-B_{16-2}) = 16$	48
		$(P_{15}-B_{15-2}) = 14$	
		$(P_6-B_{6-2}) = 9$	
		$(P_{20}-B_{20-2}) = 9$	

In the second stage, for each node, the backup processes are firstly sorted in descending order using the load difference between the primary process and the backup. Secondly, the backup processes of the primary processes on each node are divided into (N-1) groups having approximately equal incremental load by assigning each r backup processes to the r group with the smallest load, in the order of the sorted list in the previous step. This load difference is the amount of load increment to be incurred upon the occurrence of a fault. The total number of backup groups generated in this step is N(N-1). Thirdly, the algorithm computes the actual load of each group using the actual load of the backup processes. Greedy method is used again to allocate the group with the highest actual load to the node with the lowest load. However, when allocating each backup group, the algorithm checks whether there is a pre-allocated backup group that comes from the same node as the to-be allocated backup group. In such a case, the node with the next-to-the-minimum load is selected (Lee *et al*., 1999).

Table 3 shows the grouping results on node 1 in the example and Table 4 shows the final allocation results computed by the two-stage allocation algorithm.

In the two-stage allocation algorithm, the purpose of dividing the backup processes into (N-1) groups for each node is to guarantee that each node has an approximately equal amount of load increment. Hence, the system will have a balanced load when a fault occurs. The purpose of computing the actual load of each group and assigning groups based on their actual loads is to guarantee that each processor's load is balanced before the occurrence of a fault. Therefore, the algorithm can balance the processor load before as well as after the occurrence of a fault (Kim *et al*., 1997).

Table 4: Final results of two-stage allocation

Node 1	Node 2	Node 3	Node 4
P_1 P_{16}	P_2 P_{10}	P_3 P_4 P_7	P_{12} P_5
P_{15}	P_{14}	P_{13} P_{17}	P_{11} P_9 P_8
P_{20} P_6	P_{18} P_{19}	B_{11-2} B_{5-2}	B_{1-2} B_{15-2}
B_{7-2} B_{17-1}	B_{5-2} B_{12-1}	B_{19-2} B_{8-2}	B_{6-1} B_{14-1}
B_{3-2} B_{12-2}	B_{8-2} B_{1-2}	B_{16-2} B_{15-2}	B_{10-1} B_{18-1}
B_{11-1} B_{9-1}	B_{16-1} B_{20-1}	B_{6-2} B_{20-2}	B_{17-2} B_{4-2}
B_{2-2} B_{10-2}	B_{7-1} B_{4-1}	B_{9-2} B_{14-2}	B_{13-2} B_{3-1}
B_{18-2} B_{19-1}	B_{13-1}	B2-1	
91.5	92	91	91.5

However, for a more prevalent situation when each primary process is duplicated more than once, simply applying the two-stage allocation algorithm would reveal two notable defects:

- Allocating all the backup processes in one stage decreases the load-balancing performance since the point that backup processes start at different time according to their assigned id is ignored. In other words, comparing to B_{i-2}, B_{i-1} has higher election priority but is now treated equally. In the previous example as shown in Table 3 and 4, when Node 1 is failed, those preferential backup processes (B_{1-1}, B_{6-1}, B_{15-1}, B_{16-1}, B_{20-1}) are only started on Node 2 and 4 while no load is added to Node 3. It is more reasonable to allocate B_{i-1}s and B_{i-2}s in different stages since for any fixed i, B_{i-1} and B_{i-2} cannot start with each other at the same time although they are both the backup replicas for P_2.
- Grouping backup processes only based on each node cannot well balance the load when multiple faults happen. For example, as shown in Table 4, P19, B_{20-1}, B_{16-1} are allocated on Node 2 while P20, P16 and B_{19-1} are allocated on Node 1. Consider when Node 1 and Node 2 both fail, P20, P16, P19 would lose their primary and first-preference backup processes which lead to require start up their second-preference backup processes. Thus, B_{20-2}, B_{16-2}, B_{19-2} need to be considered grouping together although their primary processes and first-preference backup processes are not allocated on the same node.

Therefore, the two-stage allocation algorithm lacks scalability due to it pays little consideration to the backup

replicas startup sequence and only groups the backup processes on each node. Its load-balancing performance decreases when the number of backup processes for each primary process (R) increases, as its grouping algorithm cannot guarantee that each node has an approximately equal amount of load increment when faults occur.

A new allocation algorithm named R-stage allocation that solves the above problems is presented as follows.

R-stage allocation: To solve the defects of two-stage allocation algorithm described earlier, the new allocation algorithm has to consider two issues. First, since B_{i-1} and B_{i-2} need to be allocated in different stages separately, which one should be allocated prior than the other? Second, to group the backup processes on all the nodes rather than on each node, what grouping rule should the new allocation algorithm to use? The following lemmas which form the core of the R-stage allocation algorithm address these questions one by one.

LEMMA 1: If $1 \leq x < y \leq R$, then $\varphi(allocate(allocate(\text{nodes}, B_{i-x}), B_{i-y})) \leq \varphi(allocate(allocate(\text{nodes}, B_{i-y}), B_{i-x})$.

The function $allocate(\text{nodes}, B_{i-x})$ represents allocating B_{i-x} to the proper nodes for all i such that $1 \leq i \leq M$.

Lemma 1 implies that the backup processes with smaller id should be allocated in the stage prior to those with bigger id to minimize φ (In present model, the backup processes with smaller id have higher priority to take over the role of the primary processes).

LEMMA 2: If size of $(\mu_i \cap \mu_j) \geq$ size of$(\mu_i \cap \mu_k)$, then $\varphi(\text{group}(\text{group}(i, j), k)) \leq \varphi(\text{group}(\text{group}(i, j), j)$.

The function group (i, j) represents dividing processes i and j into groups having approximately equal incremental load.

Lemma 2 implies that one process should be considered grouping with the processes having the maximum μ intersection with it to minimize φ. This is because the bigger the μ intersection of processes is, the more possible that these processes would start up at the same time. For example, processes with μ intersection as {2, 3, 4} means they would start up if one node (Node 1) fails (Assuming there are only 4 nodes). Thus, they have more possibility to start up at the same time than the processes with μ intersection as {1, 3} which means the processes would start up if two nodes (Node 2 and 4) both fail.

According to the two lemmas described above, RSA firstly allocates primary processes using the same way as the two-stage allocation algorithm. But for the backup processes allocation, RSA needs R rounds to accomplish (R is the number of the backup process replica). The main principal in RSA is to allocate the backup process replica groups one by one in their election sequence. The algorithm is formally described below.

R-stage algorithm

Stage-0: Allocate primary processes:

- Sort primary processes in descending order of CPU load.
- Allocate each primary process to the node with the minimum load from the highest load to the lowest.

According to Lemma 1, the backup processes are allocated based on their election sequence (B_{i-1}, B_{i-2},..., B_{i-r}) in different stages. For example, Stage-2 allocates B_{i-2}, Stage-R allocates B_{i-r}. The stage-1 which allocates B_{i-1} is presented below. The following stages after stage-1 use the similar steps as it.

Stage-1 Allocate B_{i-1}: Firstly, backup groups are generated by the steps below:

- Compute the load difference between each primary process and its corresponding B_{i-1}.
- Sort B_{i-1} in descending order using the load difference.

According to Lemma 2, the backup processes with maximum μ intersection are grouped first; a hashmap is constructed in the next step to facilitate the grouping operations.

- Construct a hashmap whose key is B_{i-1} and value is a list including intersections section of μ (Each intersection has at least one element).

$$\text{Hashmap} = [\text{key} = B_{i-1}, \text{value} = \{\mu_{B_{i-1}} \cap \mu_{B_{k-1}}\}] \quad (i < k, \text{size of}(\mu_{B_{i-1}} \cap \mu_{B_{k-1}}) \geq 1)$$

- Go to step (7) if the hashmap value set is empty, otherwise, merge the same μ intersection for the value of each key in the hashmap, the fake code is described below,

$$\text{If } (\mu_{B_{i-1}} \cap \mu_{B_{k-1}} = \mu_{B_{a-1}} \cap \mu_{B_{b-1}}) \text{ Then } (\mu_{B_{i-1}} \cap \mu_{B_{k-1}}), (\mu_{B_{a-1}} \cap \mu_{B_{b-1}}) = (\mu_{B_{i-1}} \cap \mu_{B_{k-1}} \cap \mu_{B_{a-1}} \cap \mu_{B_{b-1}})$$

- Sort the μ intersections in the value set of the hashmap in descending order of the μ intersection size.

- Divide B_{i-1} into groups according to the μ intersections in the order of the sorted list in the previous step. The fake code of this step is described below:

While (sorted list size is not empty) {

Let σ be the first μ intersection in the sorted list generated in the previous step

Let Partn be a temporary partition. Select out all the B_{i-1} appeared in σ into it.

Divide the backup processes in Partn into min (size of σ, size of Partn) groups having approximately equal incremental load by assigning each backup process with the maximum load difference to the group with the smallest difference load.

Remove all the intersections in the sorted list if they include one of the B_{i-1} in Partn. (This can be quickly done by utilizing the previous constructed hashmap).

}

- After the previous steps, the left non-grouped B_{i-1}s only have the empty intersection of μ with each other; hence, each of them forms a group.

Secondly, do the following for the backup groups that are generated by the steps above.

- Sort all the backup groups using the actual loads in descending order.
- Sort the N nodes using their current loads in ascending order.
- Allocate each backup group to the node with the minimum load. However, if the node does not belong to the μ intersections of the backup processes in this group or if one of the backup groups that has already been allocated to this node comes from the same Partnas the backup group to be allocated, choose the node with the next-to-the-minimum load.

The left backup processes are allocated using the same algorithm described in the stage-1. Stage-2 allocates B_{i-2}, Stage-R allocates B_{i-r}.

The same example described previously in the two-stage algorithm is used here. Since each primary process has two backup replicas, two stages are used to allocate the backup process replicas.

After primary processes have been allocated, RSA allocates all the B_{i-1} on the nodes according to their μ intersections. Table 5 shows the grouping results for the μ intersection {2, 3, 4} during stage-1. Table 6 shows the allocation result in stage-1.

Table 5: Grouping backup processes in stage one

μ	Group	Processes	Differences (load)
	B_{11}	$(P_{16}-B_{16-1}) = 16$	25.0
		$(P_{20}-B_{20-1}) = 9$	
2, 3, 4	B_{12}	$(P_1-B_{1-1}) = 24.5$	24.5
	B_{13}	$(P_{15}-B_{15-1}) = 14$	23.0
		$(P_6-B_{6-1}) = 9$	

Table 6: Allocation results in stage one

Node 1	Node 2	Node 3	Node 4
P_1 P_{16}	P_2 P_{10}	P_3 P_4 P_7	P_{12} P_5
P_{15}	P_{18} P_{19}	P_{13} P_{17}	P_{11} P_9
P_{20} P_6	B_{5-1}	B_{14-1}	P_8 B_{17-1}
B_{4-1} B_{13-1}	B_{8-1}	B_{1-1} B_{11-1}	B_{3-1}
B_{10-1}	B_{7-1}	B_{9-1}	B_{15-1}
B_{18-1}	B_{20-1}		B_{2-1}
B_{12-1}	B_{16-1} P_{14}		B_{19-1}
			B_{6-1}

Table 7: Grouping backup processes in stage two

μ	Group	Processes	Differences (load)
2,3	1	$(P_{15}-B_{15-2}) = 14$	23.0
		$(P_6-B_{6-2}) = 9$	
	2	$(P_{12}-B_{12-2}) = 20$	20.0
1,3	3	$(P_2-B_{2-2}) = 20$	27.5
		$(P_{19}-B_{19-2}) = 7.5$	
	4	$(P_5-B_{5-2}) = 20$	27.5
		$(P_8-B_{8-2}) = 7.5$	
2,4	5	$(P_1-B_{1-2}) = 24.5$	24.5
	6	$(P_4-B_{4-2}) = 16$	25.0
		$(P_{13}-B_{13-2}) = 9$	
1,4	7	$(P_{14}-B_{14-2}) = 21$	21.0
	8	$(P_7-B_{7-2}) = 21$	21.0
3,4	9	$(P_{16}-B_{16-2}) = 16$	25.0
		$(P_{20}-B_{20-2}) = 9$	
	10	$(P_{19}-B_{19-2}) = 11.5$	22.5
		$(P_{18}-B_{18-2}) = 11$	
1,2	11	$(P_{17}-B_{17-2}) = 17$	24.5
		$(P_3-B_{3-2}) = 7.5$	
	12	$(P_9-B_{9-2}) = 10$	22.0
		$(P_{11}-B_{11-2}) = 12$	

Table 8: Final results of r-stage allocation

Node 1	Node 2	Node 3	Node 4
P_1 P_{16}	P_2 P_{10}	P_3 P_4 P_7	P_{12} P_5
P_{15}	P_{14}	P_{13} P_{17}	P_{11} P_9 P_8
P_{20} P_6	P_{18} P_{19}	B_{1-1} B_{11-1}	B_{17-1} B_{3-1}
B_{10-1} B_{18-1}	B_{5-1} B_{8-1}	B_{9-1} B_{14-1}	B_{15-1} B_{6-1}
B_{12-1} B_{4-1}	B_{16-1} B_{20-1}	B_{5-2} B_{8-2}	B_{2-1} B_{19-1}
B_{13-1} B_{17-2}	B_{7-1} B_{1-2}	B_{16-2} B_{20-2}	B_{10-2} B_{18-2}
B_{3-2} B_{7-2}	B_{11-2} B_{9-2}	B_{6-2} B_{15-2}	B_{4-2} B_{14-2}
B_{2-2} B_{19-1}	B_{12-2}		B_{13-2}
91.5	92	92	90.5

In stage-2, B_{i-2} are grouped as shown in Table 7. After allocating each group to the nodes, RSA gets the final result as shown in Table 8.

Let us compare the load distribution in Table 4 (two-stage allocation results) with Table 8 (RSA results). When Node 1 fails, the preferential backup processes (B_{i-1}, B_{6-1}, B_{15-1} B_{16-1}, B_{20-1}) are only started on Node 2 and Node 4 in Table 4 while in Table 8, these backup processes are started on the left available nodes; hence φ is minimized in Table 8. Apparently, RSA balances the load better after faults occurrence.

Algorithm complexity:

- **Primary process allocation:** A sorting algorithm whose running time is O(M lg M) is used to sort M primary processes. Allocating M primary processes to N nodes requires O(M lg N) time.
- **Backup process allocation:** Let us first consider the time complexity for one stage.
  Sorting M backup processes takes O(M lg M) time.

  It requires

$$\frac{M(M-1)}{2}$$

time to select arbitrarily two backup processes and takes at most 2N time to compute their μ intersections. Thus, the hashmap construction requires $O(M^2N)$. Merging and sorting its value set in descending order of the size of the intersection requires $O(M^2 \lg M)$.

The worst case for step 5 and step 6 is that every two backup processes forms a separate Partn, thus requires O(M.

There are at most M groups generated by the previous steps. Allocating these groups to N nodes requires o(M lg N) time.

So the worst time complexity for the backup process allocation in one stage is $O(M \lg M+M^2N+M^2\lg M+M+M \lg N)$.

The above steps described are repeated in R stages, which together require $O(R(M \lg M+M^2N+M^2 \lg M+M+M \lg N)$.

Hence, plus the primary process allocation and R Stages of backup process allocation, the total time complexity of this algorithm is

$$\begin{aligned} &O(M\lg M + M\lg N + R(M\lg M + M^2N + M^2\lg M + M + M\lg N)) \\ &= O(M^2NR + M^2R\lg M) \end{aligned} \quad (8)$$

Assuming that the number of processes is much larger than the number of nodes ($M>N^2$), the execution time is bounded by $O(M^2R \lg M)$.

Comparing Eq. 8 with the time complexity of two stage allocation algorithm $O(NM \lg NM+N^3 \lg N)$, we can see the time overload of RSA is acceptable and it is due to using more allocation stages (R stage instead of two stage), hashmap construction and its value set sorting.

PERFORMANCE COMPARISON

In this part of study, we compare the load distribution performance of RSA with the two-stage algorithm using simulations.

The environment parameters used in the simulation are as follows. To keep the total load of each node below 100%, the load of the primary processes is chosen randomly in the range of 1 to 100(N-R)/M based on a uniform distribution. The loads of the backup processes are also chosen randomly between 5~10% of the load of their primaries, also based on a uniform distribution.

Figure 2 shows the simulation results on how φ in Eq. 10 is affected by R in two-stage allocation algorithm and RSA algorithm. It is assumed that the number of nodes (N) is eight. The Y-axis represents the value of φ while the X-axis represents R (the number of the backup processes replica). In two-stage algorithm, φ increases dramatically when R increases especially when R reaches N-1 due to the fact that the algorithm does not group the backup replicas on each node well. RSA algorithm minimizes φ hence it has much better scalability and load distribution performance than two-stage allocation algorithm.

Assuming the number of process is 400 and the node number is eight, Fig. 3 shows the maximum and minimum

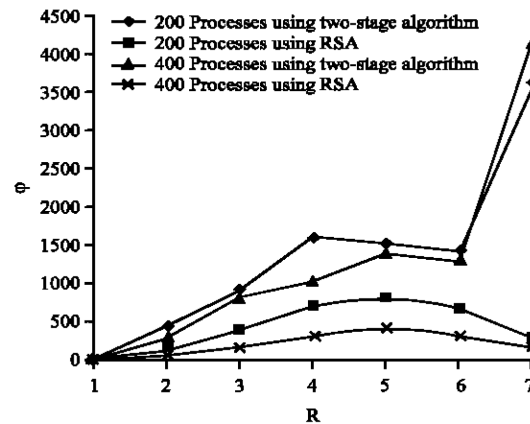

Fig. 2: φ affected by R (No. of backup process)

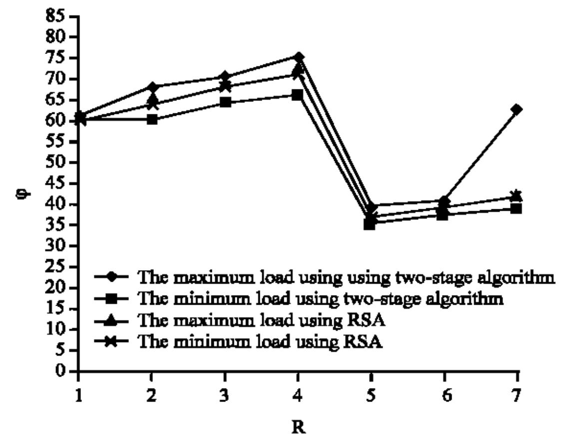

Fig. 3: Maximum and minimum load when one fault occurs

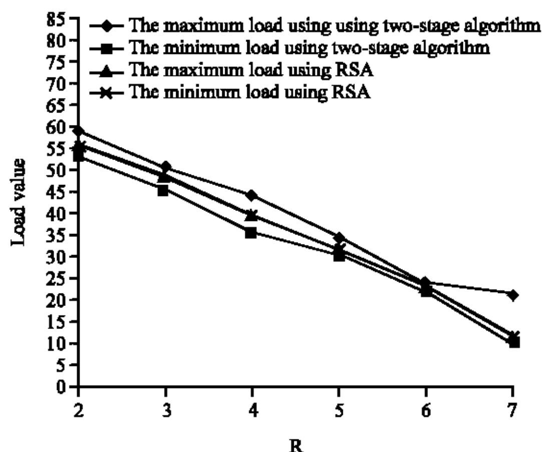

Fig. 4: Maximum and minimum load when two faults occur

load using the two-stage algorithm and RSA after one node fails. The X-axis represents R while the Y-axis represents the CPU load. The difference between maximum load and minimum load in RSA is smaller than the two-stage algorithm especially when R increases. Figure 4 shows the simulation results after two nodes fail based on another generated random test data but with the same parameters as Fig. 3.

CONCLUSION

In this study, we considered the static process allocation for load distribution in the primary-backup based Fault-Tolerant system.

In the primary-backup fault tolerant model, only primary process receives the invocations from the client process and sends the response back. Hence the load of primary process is much bigger than its backup processes. And when a fault occurs, one of the backup processes will be elected to take over the role of the primary process. Therefore, the load of this elected backup process varies before and after the occurrence of a fault. Previous research shows it is an NP-hard problem to find out a static process allocation algorithm to balance the system load before as well as after faults occur.

The main contribution of this study is the presentation and analysis of a new heuristic approximation static load-balancing algorithm for replicated R-Resilient process in the primary-backup based Fault Tolerant System. The proposed algorithm has better scalability and load distribution performance than the previous allocation algorithms in this area. In this study, we assume there is no difference between each node. We are currently working on extending this algorithm to handle heterogeneous distributed systems. Also, we plan to study the process allocation in the dynamic situation in the future.

REFERENCES

Bannister, J.A. and K.S. Trivedi, 1983. Task allocation in fault-tolerant distributed systems. Acta Inform. J., 20 (3): 261-281.

Garcia-Molina, H., 1982. Elections in a distributed computing system. IEEE Trans. Comput. J., 31 (1): 48 - 59.

Guo Hui, J.L. Zhou and Z.G. Wang, 2005. Load balancing based process scheduling with fault-tolerance in heterogeneous distributed system. Chin. J. Comput., 28 (11): 1807-1816.

Kim, J., H. Lee and S. Lee, 1995. Process allocation for load distribution in fault-tolerant multicomputers. Proceedings of 25th International Symposium on Fault-Tolerant Computing.

Kim, J., H. Lee and S. Lee, 1997. Replicated process allocation for load distribution in fault-tolerant multicomputers. IEEE Trans. Comput. J., 46 (4): 499-505.

Lee, H., J. Kim and S. Hong, 1999. Evaluation of two load-balancing primary-backup process allocation schemes. IEICE Trans. Inform. Syst. J., E82-D (12): 1535-1544.

Liu, A.F., Z.G. Chen and G.P. Long, 2005. Research on fault tolerant scheduling algorithms of web cluster based on probability. Wuhan University J. Natural Sci. J., 10 (1): 70-74.

Mullender, S.J., 1993. Distributed Systems. ACM Press Publishing.

Powell, D., 1994. Distributed fault tolerance: Lessons from delta-4. IEEE Microb. J., 14 (1): 36-47.

Singh, S. and J.F. Kurose, 1994. Electing good leaders. Parallel and Distributed Comput., J., 21 (2): 184-201.

Stoller, S.D., 2000. Leader election in asynchronous distributed systems. IEEE Trans. Comput. J., 49 (3): 283-284.

Zhang, X.N., D. Zagorodnov and M. Hiltunen, 2004. Fault-tolerant grid services using primary-backup: Feasibility and performance. Proceedings of IEEE International Conference Cluster Computing.