Enhanced N+1 Parity Scheme combined with Message Logging

Ch.D.V. Subba Rao and M.M. Naidu

Abstract— **Checkpointing schemes facilitate fault recovery in distributed systems. The present work extends James S Plank's Diskless checkpointing scheme (N+1 Parity) by introducing *'Timeout'* to checkpoint programs with high locality of [1]reference. This mechanism enables applications with high locality of reference to take checkpoints periodically. The limitation of N+1 Parity scheme is that all the processes freeze their respective computation, while taking synchronous checkpoints. The Enhanced N+1 Parity Scheme solves this problem by introducing a new message logging technique namely *partial message logging* which allows asynchronous checkpointing at both sender and receiver. This paper includes the performance evaluation of proposed scheme by making use of distributed simulator test-bed. The results indicate that proposed scheme outperforms N+1 Parity Scheme.**

Index Terms—**Checkpointing, Fault tolerance, Message Logging, Performance Analysis**

I. INTRODUCTION

Distributed systems are increasingly gaining importance in the present world with the advances in network technology. They provide opportunities for developing high performance parallel and distributed applications. The vast computing potential of these systems is often hampered by their susceptibility to failures. In case of a failure the distributed applications have to be restarted resulting in loss of several hours/ days of computation. Therefore, providing fault tolerance is an important issue in distributed computing. One effective way to recover distributed system failures is to use checkpointing and rollback recovery [5,6]. Checkpointing refers to saving the address space and state of processes periodically to stable storage. On detection of failures, each process rolls back to its latest checkpoint and resumes the execution from that point [1]. Checkpointing schemes are classified into two categories based on the 'storage medium' used for storing the checkpoints. They are i) Disk-based checkpointing ii) Diskless checkpointing.

In diskless checkpointing scheme, each checkpoint is saved to stable storage that is implemented on disk. Though the disk-based checkpointing is most widely used approach, the applications that need most frequent checkpointing lead to several disk accesses which results a performance bottleneck. In diskless checkpointing the stable storage is replaced with main memory and processor redundancy i.e. the checkpoints are taken in the main memory. By eliminating stable storage, diskless checkpointing removes the main source of overhead in checkpointing[3,8,10]. The failure coverage of diskless checkpointing is less than disk-based checkpointing. Moreover, the diskless checkpointing includes the memory, processor and network overheads that are absent in disk-based schemes.

II. DESIGN OF ENHANCED N+1 PARITY SCHEME

In this paper, we present Enhanced N+1 Parity scheme and demonstrate that it can perform better than the N+1 Parity scheme proposed by N. H. Vaidya [12].

In Diskless checkpointing scheme there exist a collection of processors with disjoint memories that coordinates to take a checkpoint of the global system state. A consistent global state consists of checkpoints of each processor in the system plus a log of messages in transit at the time of checkpointing. In the proposed scheme the messages are logged using partial massage logging scheme (discussed in detail in Section III). The message log is part of the checkpoint information of individual processors. The checkpointing and rollback recovery procedures in case of diskless checkpointing are as described:

Let's consider a multicomputer/ distributed system which consists of n+2 processors/ nodes: P_1, P_2, ……..P_n, P_c and P_b. Each node contains its own physical memory and communication devices. Processors P_i, $1 \le i \le n$ are called application processors and the processors P_c and P_b are called the 'checkpoint processor' and 'backup processor' respectively. A computing task is partitioned into 'n' subtasks, such that each subtask is executed on a distinct application processor P_i in an asynchronous manner. These subtasks communicate with each other by passing messages via the underlying interconnection network.

The consistent global state is maintained cooperatively by the application processors P_1, P_2, ……..P_n, checkpoint processor P_c, and backup processor P_b using N+1 parity [4,7]. Specifically, each application processor will have a copy of its own local checkpoint in physical memory. The checkpoint processor will have a copy of the 'parity checkpoint', which is constructed as follows: Let S_i denote the size of main memory containing local checkpoint at processor P_i, $1 \le i \le n$. The checkpoint processor P_c reserves a bank of memory M_c of size S_c which is equal to the maximum of S_i's, i.e. $S_c = \max \{S_1, S_2, \ldots, S_n\}$. Let $b_{i,j}$ be the jth byte of P_i's checkpoint if $j \le S_i$, and 0 otherwise. The jth byte $b_{c,j}$ of checkpoint processor holds the parity as a result of exclusive-oring all of the jth bytes of the n application processors, that is,

$$b_{c,j} = b_{1,j} \oplus b_{2,j} \oplus \ldots \oplus b_{n,j} \text{ for } 1 \le j \le S_c .$$

Mr. Ch.D.V. Subba Rao and Dr. M.M. Naidu are with Dept. of Computer Science and Engineering, S. V. University College of Engineering, Tirupati – 517 502, India.
E-mail:subbarao_chdv@hotmail.com

One copy of the above parity checkpoint which is the exclusive-or of checkpoints of all application processors is stored on backup processor and this backup is useful when the checkpoint processor wants to update its contents.

When any application processor, say P_f fails, each non-failed processor restores its state to its local checkpoint. The failed processor's checkpoint can be obtained by exclusive-oring the checkpoints of non-failed application processors with parity checkpoint present in the checkpoint processor as shown:

$b_{i,j} = b_{1,j} \oplus b_{2,j} \oplus \ldots\ldots \oplus b_{i-1,j} \oplus b_{i+1,j} \oplus \ldots\ldots \oplus b_{n,j} \oplus b_{c,j}$, for $1 \le j \le S_i$.

If the checkpoint processor fails, then it restores its state from the backup processor, or by recalculating the parity checkpoint from scratch. The backup processor may be restored similarly. The above scheme tolerates a single failure with two additional processors (i.e. P_c and P_b). This idea can be extended to simultaneous failure of 'n' nodes, where the system needs '2*n' additional/ replacement processors.

Initially, each application processor P_i, $1 \le i \le n$, takes a checkpoint 0 by simply setting the access modes of all of the memory pages to read-only state. The content of the memory is then sent to checkpoint processor P_c. P_c calculates parities of the pages byte-by-byte, and stores the parity data in the corresponding location of M_c. In addition, each application processor clears its extra memory. This space is split in half, and each half is used as a checkpointing buffer. We will call them the primary and secondary checkpointing buffers as shown in Figure 1.

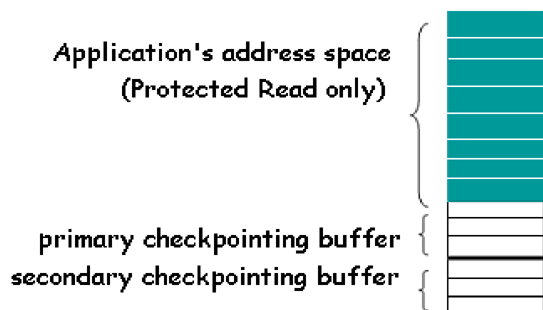

Fig. 1. Address space of the Application processor

Application processors start executing their programs after their initialization phase is completed. Read operations proceed as usual. However, a page fault is generated when a processor attempts to write to a read-only page. Once a page fault occurs, the content of the accessed page is copied to its primary checkpointing buffer, and the page's access mode is set to read-write so that it can be written. If any application processor fails during this time, the system can be restored to the most recent checkpoint by copying the pages back from the buffer, reprotecting them as read only, and then restarting. Obviously, if the checkpoint processor fails during this time, it can be restored from the backup processor, and vice-versa.

Timeout Mechanism

Whenever, the space of a processor's primary checkpointing buffer is used up or when it is time to take a new checkpoint, which we simply call ***Timeout,*** then it must start a new checkpoint. In other words, if the last completed checkpoint was checkpoint number c, then it starts checkpoint c+1. The processor requests the checkpoint processor for authorization to take the checkpoint. The checkpoint processor in turn checks the global checkpoint number; if the difference is not greater than one, it will grant permission; otherwise it holds the request until the global checkpoint number is incremented.

The significance of *Timeout* is that, if a user is executing a program with high locality of reference, the chance of primary buffer getting filled is low or the buffer gets filled after consuming significant amount of computational time. If the processor fails during the execution of the programs with high locality of reference, the lost work will be more as the checkpoint is taken after a longer interval of time and sometimes the application even cannot take checkpoint. Because of *Timeout* mechanism that is added to N+1 parity, the application can take checkpoints periodically, thus reducing maximum work lost is equal to the checkpoint interval.

III. PARTIAL MESSAGE LOGGING SCHEME

Message handling got a significant role in checkpoint and recovery of the distributed system [9]. We can't assume consistent state of the system without proper message handling. The major concern is for the messages that are in transit at the time of checkpointing. The N+1 parity scheme (i.e. diskless checkpointing scheme) proposed by J. S. Plank is based on the assumption of zero message state, where all processors halt/ freeze their computation until the in transit messages are delivered to the destination process. Our scheme proposes partial message logging technique which allows asynchronous checkpointing at both sender and receiver, which won't results into any blocking/ halting overhead. Since, the in-transit message data is saved in the in-memory checkpoint, the proposed message logging technique does not result into any additional overhead. Further, the proposed scheme partially logs the messages by always ensuring the difference among the checkpoint states of individual application processors should never be greater than 1. Therefore, at any time, the message log of each process contains messages that are sent/ received during present and previous checkpoint intervals.

The following example in Figure 2 will show the occurrence of orphan and lost messages when message logging is not done. A system is strongly consistent if and only if there are no orphan and lost messages [15].

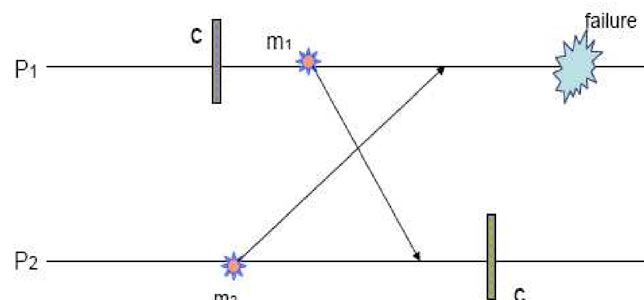

Fig. 2. Lost and Orphan messages

ISBN: 978-988-98671-8-8

Process P_1 and process P_2 both take in-memory checkpoint 'c' asynchronously. Now when the failure occurs they will rollback to checkpoint c. Process P_1 resends message m_1 to process P_2 which is an *orphan* message. Process P_2 won't replay the message m_2 resulting in a *lost* message. Logging of messages introduces memory and computational overhead. Therefore, the proposed scheme partially logs the messages by always ensuring the difference among the checkpoint states of individual application processors should never be greater than 1. At any time, the message log of each process contains messages that are sent/ received during present and previous checkpoint intervals. Each application processor possesses one or more process/es running in its address space.

A. Log Structure

Every process maintains two message log tables that are sent table and receipt table in the address space of an application processor in which it is present.

Sent Table: The sent table looks like the one shown in Table1. Each entry in the sent table is a quadruple where

- Seq number : The process sequentially numbers the sent messages.
- Destination is the process to which the message is sent.
- Message slot holds the message data.
- The messages can have the following acknowledge states.
  0 - Message is sent
  1- Acknowledgement for the sent-message is received
  2 - The Messages sent during previous checkpoint interval. (Archived messages)

The message logging mechanism will modify all the message slots with ack status 1 to 2, at the time of taking in-memory checkpoint. When, the application processor rolls back to the previous checkpoint at the time of failure recovery, all processes will resend the messages in the sent table with ack.2, thus eliminating the *lost-messages*.

TABLE 1
Sent Table

Ack Status	Seq Number	Destination	Message slot

Receipt Table: The information of the received messages is logged in this table. The table looks like the one shown in Table 2 where Seq. number is the sequence number contained by the received message. Source is the process id from where the message is received.

The messages can have two Ack States:
1 - Message is received and acknowledgement is sent
2 - The messages received during previous checkpoint interval.

The message logging mechanism will make entry for all received messages and mark them with ack. 1. At the time of taking in memory checkpoint all processes (of that particular processor) modifies their ack. status 1 to 2 in receipt table. Whenever a process receives any message it checks for its presence in the receipt table. If the entry for that message is already present, the process simply ignores the message, thus avoiding the *orphan-messages*.

TABLE 2
Receipt Table

Ack Status	Seq Number	Source

At the time of taking a checkpoint, the entries in the receipt table and sent table with ack. 2 are removed. This ensures that at any point of time the message log will have information about messages of present and previous checkpoint interval thus reducing the memory and lookup overhead. As the processes maintain the message log information in the address space of the application processes this partial message logging scheme will work for both for the diskless and disk-based checkpointing schemes.

B. An Example of Partial Messaging Logging

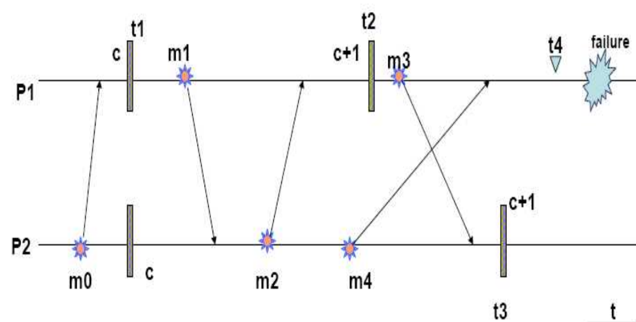

Fig. 3. Message exchanges between processes P_1 and P_2

The Figure 3 shows the message exchange between two processes. For ease of understanding we consider that the two processes P_1 and P_2 are running on separate application processors. First, both the application processors take in-memory checkpoint almost concurrently at time t_1 and the sent table of the process P_2 is as given in Table 3. At time t_2 process P_1 had taken in-memory checkpoint c+1. At this moment the status of sent and receipt tables of the process P_1 are as shown in Tables 4 and 5 respectively.

TABLE 3
Sent-table of process P_2 after checkpoint c

Ack Status	Seq Number	Destination	Message slot
2	m0	p1	Mq34#$62^#

TABLE 4
Sent-table of P_1 after checkpoint $c+1$

Ack Status	Seq Number	Destination	Message slot
2	m1	p2	Mt*hjkvd#$62^#

TABLE 5
Receipt-table of P_1 after checkpoint $c+1$

Ack Status	Seq Number	Source
2	m2	p2

After taking the checkpoint c+1 the process P_1 will send message m_3 to process P_2 and receive message m_4 from process P_2. At time t_3 the process P_2 takes the checkpoint. After taking the checkpoint c+1 the status of sent and receipt tables of process P_2 are given in Tables 6 and 7 respectively. At time t_4 the sent-table and receipt-table of process P_1 is as given in Table 8 and Table 9.

TABLE 6
Sent-table of P_2 after checkpoint $c+1$

Ack Status	Seq Number	Destination	Message slot
2	m2	p1	Mtlw*(0kvd#$62^#
2	m4	p1	Mtlw(dju)62^#

TABLE 7
Receipt –table of P_2 after Checkpoint $c+1$

Ack Status	Seq Number	Source
2	m1	p1
2	m3	p1

TABLE 8
Sent-table of *P1 at* time t_4

Ack Status	Seq Number	Destination	Message slot
2	m1	p2	Mt*hjkvd#$82^#
1	m3	p2	Mtl@sv!54ju)62^#

TABLE 9
Receipt-table of P_1 at time t_4

Ack Status	Seq Number	Source
2	m2	p2
1	m4	p2

After time t_4 a fault occurs in the application processor containing process P_1. Now process P_2 rolls back to its local checkpoint c+1. Process P_1 is recovered by first-level recovery scheme. After rollback the status of sent-table and receipt-table of process P_2 will be same as that of Table 6 and Table 7 respectively. After rollback the sent-table and receipt-table of process P_1 will be equal to that of Table 4 and Table 5 respectively. Now when the process P_1 sends message m_3 to process P_2, it makes a lookup into Table 7 and it founds a match. Therefore process P_2 discards the message as a redundant message, thereby eliminating the orphan message problem. Process P_2 replays the message m_4, as there is a mismatch of ack. status bits of sent-table 6 and receipt-table 9 with respect to message m_4. Thus the lost message problem is also eliminated. Hence proposed partial message logging scheme handles the messages effectively and with utmost reliability.

IV. PERFORMANCE EVALUATION

In order to assess the performance of Enhanced N+1 Parity Scheme, we have used distributed simulator testbed.

A. Distributed Simulator Testbed

The Distributed Simulator testbed facilitates to demonstrate how system reliability can be enhanced as a result of using a particular checkpointing and recovery scheme. The simulator functions by taking a system specification, a task set to be run, and injecting faults to see the performance of the system in the presence of faults. The two important components of Distributed Simulator testbed are

1. Checkpoint simulator
2. Distributed Virtual Machine

The functionality of Checkpoint Simulator is similar to that of Rapids Simulator [14].

Distributed Virtual Machine

DVM (Distributed Virtual Machine) is an integrated set of software tools and libraries that emulates a general-purpose, flexible, computing framework on interconnected computers. The overall objective of the DVM system is to enable such a collection of computers to be used cooperatively for concurrent or parallel computation.

The DVM system is composed of two parts. The first part is a daemon that resides on all the computers making up the virtual machine. The second part of the system is a library of DVM interface routines. It contains a functionally complete repertoire of primitives that are needed for cooperation between tasks of an application. This library contains user-callable routines for message passing, spawning processes, coordinating tasks. The DVM computing model is based on the notion that an application consists of several tasks. Each task is responsible for a part of the application's computational workload.

Distributed Virtual Machine will take care of fault detection, installation and management, reconfiguration mechanism when a new node is added or when already existing node is removed. Distributed Virtual Machine automatically takes decision regarding the recovery using the information of control file and available backup processors.

B. Applications considered for Evaluation

In order to assess the performance of proposed scheme and other checkpointing and recovery schemes, the following applications [15] are run on the distributed simulator test-bed.

- Merge Sort
- Matrix Multiplication
- All Pair Shortest Path Problem

Merge Sort:
The given data/ elements to be sorted are distributed to all the application processors in the system by the coordinator using divide-and-conquer approach. The application processors sort the elements and return the result to the coordinator. If 10,00,000 records need to be sorted, the coordinator which acts as dispatcher distribute these records to application processor in chunks of 1,00,000 records. Each application processor after completion of sorting sends back the data to coordinator which will merge the incoming data in ascending order of their values. This is a computational intensive problem.

Matrix Multiplication:
The matrix multiplication of two square matrices is carried out using cannon's algorithm where elements are floating point numbers. Matrix size of the order 2629 x 2629 is considered.

All Pairs Shortest Path Problem:
The all pairs shortest path problem computes the shortest distance from each vertex to all other vertices. Here we have considered 15 x 15 connected graph. Each application processor is assumed to be a vertex, and shortest path from it to all other application processors is calculated. This problem is communication intensive.

C. Performance Metrics

Apart from the correctness, the performance is the most important aspect of a checkpointer. N.H. Vaidya derived equations 1 to 3 for assessing the performance of an application in the presence of checkpointing and recovery. These equations will take the checkpoint overhead, checkpoint latency, and recovery time as input and are as described below:

$$e^{\lambda\,(T_{opt}+0)}\left(1-\lambda T_{opt}\right)=1 \text{ for } T_{opt}\neq 0 \quad \ldots\ldots(1)$$

$$\Gamma=\lambda^{-1}e^{\lambda\,(L-O+R)}\left(e^{\lambda\,(T_{opt}+0)}-1\right) \quad \ldots\ldots(2)$$

$$r=\frac{\Gamma}{T_{opt}}-1 \quad \ldots\ldots(3)$$

where
λ = the rate of failures (1/MTBF)
T_{opt} = the optimal checkpoint interval
O = the average overhead per checkpoint
L = the average latency per checkpoint
R = the average recovery time from a checkpoint
r = the overhead ratio
Γ = optimal checkpoint interval in the presence of failures, checkpointing and recovery.
From (1), it can be seen that T_{opt} *decreases when overhead O decreases*.

D. Performance Measurements

Before running different applications on distributed simulator testbed, initially consider the basic definitions of various overheads.

Checkpoint overhead: Checkpoint overhead is the time added to the running time of the target program as a result of checkpointing.

Checkpoint latency: Checkpoint latency is the time that it takes for the checkpointer to complete a checkpoint, from start to finish. Checkpoint latency is the duration of time required to save the checkpoint. In many implementations, checkpoint latency is larger than the checkpoint overhead.

Recovery overhead/time: This is the time that it takes the system to restore a checkpointed state following the detection of a failure.

To calculate the above overheads we have considered 4 checkpoint processors and 4 backup processors both incase of N+1 parity and proposed scheme. In this scenario, N+1 Parity supports simultaneous failure of at most four application processors. In order to evaluate the performance of the proposed protocol, the overhead ratio of different checkpointing schemes and proposed scheme is calculated by using the equations 1, 2 and 3 and the results are summarized in Tables 10, 11 and 12 for merge sort, matrix multiplication and all pairs shortest path problem respectively. λ value is assumed as $6.301 * 10^{-6}$

TABLE 10
The results of different checkpointing schemes in case of Merge sort

	O	L	R	r
N+1 Parity	420	43.34	140.2	0.07482
Proposed Scheme	209	32.15	142.5	0.05277

TABLE 11
The results of different checkpointing schemes in case of Matrix multiplication.

	O	L	R	r
N+1 Parity	391	90	190.2	0.07286
Proposed Scheme	183	52	190.2	0.049993

TABLE 12
The results of different checkpointing schemes in case of all pairs shortest path problem

	O	L	R	r
N+1 Parity	391	90	190.2	0.07286
Proposed Scheme	183	52	190.2	0.049993

The above results indicate that Enhanced N+1 Parity Scheme results into minimal *overhead ratio 'r'*. Therefore, we can conclude that the proposed scheme outperforms the N+1 Parity checkpointing scheme.

V. CONCLUSIONS

The N+1 Parity Scheme proposed by James S Plank fails to checkpoint applications with high locality of reference. Our proposed scheme introduced the *Timeout* mechanism to handle this problem efficiently. The proposed message logging technique i.e. partial message logging allows asynchronous checkpointing at both sender and receiver and does not freeze/ halt the state of process. Partial message logging technique provides a better performance even if the communication channels are non-reliable. The garbage collection is made simple and reliable, as deletion of old messages are handled implicitly at the time of checkpointing itself.

Using distributed simulator testbed, we have evaluated and compared the performance of the proposed scheme with N+1 Parity Scheme. Few application programs have been developed and executed in this virtual environment. The overhead ratio is computed for both of these schemes and it is found to be small in case of the proposed scheme. Hence it has been concluded that the Enhanced N+1 Parity scheme combined with message logging outperforms N+1 Parity scheme.

REFERENCES

[1] A. Ralston and E.D. Reily, Encyclopedia of Computer Science, Third Edition, IEEE Press, 1993.
[2] S. Kamal, "An Approach to the Diagnosis of Intermittent Faults," IEEE Transactions on Computers, 24, pp. 461-467, 1975.
[3] James S Plank, Kai Li and Michael A. Puening, "Diskless Checkpointing," IEEE Trans. on Parallel and Distributed Systems, Vol. 9, No. 10, pp. 972-986 October 1998.
[4] J.S. Plank and K. Li, "Faster Checkpointing with N + 1 Parity," Proc. 24th Int'l Symp. Fault-Tolerant Computing, pp. 288-297, Austin,Tex., June 1994.
[5] E.N. Elnozahy, D.B. Johnson, and Y.Wang "A Survey of Rollback Recovery Protocols in Message Passing Systems," Technical Report CMU-CS-99-148, Carnegie Mellon Univ., June 1999.
[6] Ch D V Subba Rao and M M Naidu, "A Survey of Error Recovery Techniques in Distributed Systems," Proc. 28th Annual Convention and Exhibition of IEEE India Council, pp. 284-289, December 2002.
[7] J.S. Plank and K. Li, "Ickp – A Consistent Checkpointer for Multicomputers," IEEE Parallel & Distributed Technology, vol. 2, no. 2, pp. 62-67, 1994.
[8] J.S. Plank, Efficient Checkpointing on MIMD Architectures, Ph.D. Thesis, Department of Computer Science, Princeton University, 1993.
[9] E.N. Elanozahy and W.N. Zwaenepol "On the Use and Implementation of Message Logging," Proc. IEEE International Symposium on Fault Tolerance Computing Systems, pp. 298-307, 1994.
[10] L.M. Silva and J.G. Silva, "An Experimental Study about Diskless Checkpointing," Proc. 24th Euromicro Conference, vol. 1, pp. 395-402, Aug 1998.
[11] L.M. Silva and J.G. Silva, "Using Two-level Stable Storage for Efficient Checkpointing," IEE Proceedings-Software, vol. 145, issue 6, pp. 198-202, Dec. 1998.
[12] Nitin H. Vaidya, "A Case for N+1 Parity Schemes," IEEE Trans. Computers, vol. 47, no. 6, June 1998.
[13] T. Chiueh and P. Deng, "Evaluation of Checkpoint Mechanisms for Massively Parallel Machines," Proc 26th Int'l Symp. Fault-Tolerant Computing, pp. 370-379, Sendai, June 1996.
[14] Rapids-simulator. - www.ecs.umass.edu/ece/realtime/publications/**rapids**_paper.pdf
[15] Jean_Michel Helary, A. Mostefaoui, R.H.B. Netzer, and M. Raynal, "Preventing Useless Checkpoints in Distributed Computation," Proc. Symp. on Reliable Distributed Systems, pp. 183-190, Oct. 1997.
[16] N. H. Vaidya, "Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme," IEEE Trans. Computers, Vol. 46, No. 8, pp. 942-947, August 1997.