High Performance Fault Tolerance Protocol for Mobile Distributed Systems

Gurdev Singh[1], R. K. Verma[2]
[1] **Assistant Professor, Department of CSE, Jind Institute of Engineering & Technology, Jind, Haryana, India.**

[2] **Professor, Department of CSE , KITM, Kurukshetra, Haryana, India.**

Abstract
A mobile computing system consists of mobile and stationary nodes, connected to each other by communication network. The system raises several constraints such as limited battery life, mobility, disconnection of hosts and lack of stable storage. To reduce the lost of computational work during recovery from the node failures periodic collection of a consistent snapshot of the system (checkpoint) is required. This paper presents an efficient coordinated checkpoint protocol which is non-blocking and not forces every node to take local checkpoint. We proposed that collected global snapshot is consistent. Our protocol meet the low energy consumption, reduces storage overhead having low communication and low band width constraints of mobile computing systems.

Keywords: *Mobile Computing Systems, Coordinated checkpointing, Consistent Checkpoints, Global Snapshot, Recovery, Non-blocking.*

1. Introduction

A mobile computing system is a distributed system where some of the nodes are mobile computers (Mobile Hosts (MHs)) [9]. As time passes mobile computers location gets change. To communicate with MHs, mobile support stations (MSSs) are added. An MSS communicate with other MSS by wired networks and with MHs with wireless network. Each of saved state is called snapshot (checkpoint). All the processes in the system take their checkpoints periodically.

The checkpointing techniques do not require user interaction and can be classified into following categories: (a) Uncoordinated checkpointing (b) Coordinated checkpointing (c) Quasi-Synchronous (d) Message – Login based checkpointing [14]. In this paper we concentrate on coordinated checkpointing technique which maintains a consistent snapshot of system all the times. A consistent global snapshot indicates set of N local snapshots (checkpoints) one from each process forming a consistent system state which can be used to restart process execution upon a failure. It is desirable to minimize the amount of lost work by restoring the system to most recent consistent global checkpoint. A good snapshot collection algorithm should be Non-Blocking i.e. which does not force the nodes in the system to stop their computations during snapshot collection. An efficient algorithm keeps minimum effort required for collecting a consistent snapshot to a minimum. The snapshot collection algorithm by Chandy and Lamport forces every node to take its local snapshots but the computation is allowed to continue while the global snapshot is being collected [1]. In Koo and Toueg's algorithm all the nodes are not forced to take their local snapshots [7]. However, the underlying computation is suspended during snapshot collection. We propose a new coordinated checkpoint protocol which is non- blocking and efficient that forces a minimal set of nodes to take their snapshot and underlying computation is not suspended during snapshot collection. The rest of the paper is organized as follows: Section 2 presents the system model. Section 3 presents the basic idea, minimal dependency information at each node and the node mobility management of our protocol. Section 4 presents a Non-Blocking Coordinated checkpoint protocol with data structure used and an example describing our algorithm. Section 5 presents the related work on which our protocol is based. Finally, Section 6 presents conclusion

2. System Model

A message passing system consists of N fixed number of nodes that communicate each other only through messages.

Some of the nodes may change their location with time. They are referred to as mobile hosts or MHs [9]. Static Nodes are referred to as mobile support stations or MSSs are connected to each other by a Static Network. An MH can be directly connected to at most one MSS at any given time and can communicate with other MHs and MSSs only through the MSS to which it is directly connected. The system does not have any shared memory or a global clock. Hence all the communication and synchronization takes place through messages.

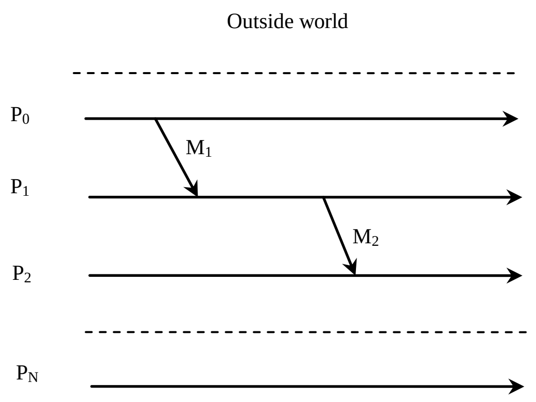

Fig. Message Passing System

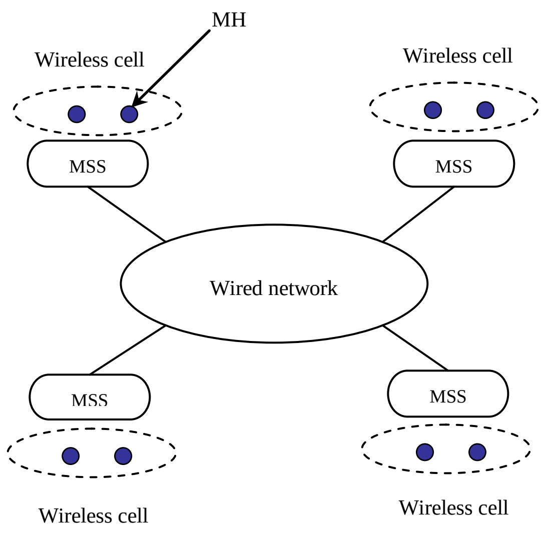

Fig: Mobile Distributed System

The messages generated by underlying distributed application will be referred to as computation messages. Messages generated by the nodes to advance checkpoints, handle failures and for recovery will be referred to as system messages. In this paper the horizontal lines extending towards right hand side represent the execution of each process (MH) and arrows between them represent the messages. Processes have access to a stable storage device that survives failures. The number of tolerated process failures may vary from 1 to N [14].

3. Main Idea behind Our Protocol

Two main reasons behind the design of a checkpoint collection algorithm are:
(1) Its efficiency: An efficient algorithm forces a minimum number of nodes to take their local snapshots.
(2) Its Non-Blocking approach: A Non-Blocking algorithm does not stop the computation at the participating nodes during checkpoint collection. [2]

A. Propagation of minimal dependency information

The dependency is created by means of messages between nodes. Node Pi maintains a Boolean vector Ri of n components. At Pi, the vector initialized as follows:

$$Ri[j] = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

When a node Pi sends a message to Pj it then modifies vector Ri. This informs Pj about the nodes that have affected Pi.
While processing a message M Pj extracts Boolean vector M.R from the message and uses it to update Rj as follows: Rj[k] ← Rj[k] OR M.R[k], where $1 \leq k \leq n$.
Following diagram shows the dependency information through messages: Since P2 was dependent on P1 before sending M2 to P3; P3 becomes transitively dependent on P1 on receiving M2.

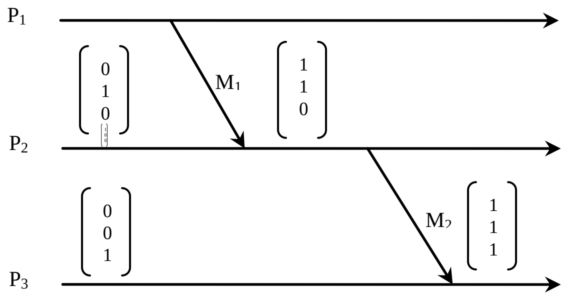

Fig. Propagation of dependency information via R[i] vector

The dependency information is used to minimize the effort required to collect global checkpoint .But there should be avoidance of useless checkpoint in global checkpoint collection. The following figure describes : There are three processes P, Q and R. Let Q initiates checkpoint request to processes P and R. Let P and R take their local checkpoints. If R sends message M to P before receiving checkpoint request then message M will become an orphan message which creates a problem during snapshot collection. To avoid such problem a concept of checkpoint sequence number get arise. We call this ckpt_num in our protocol.

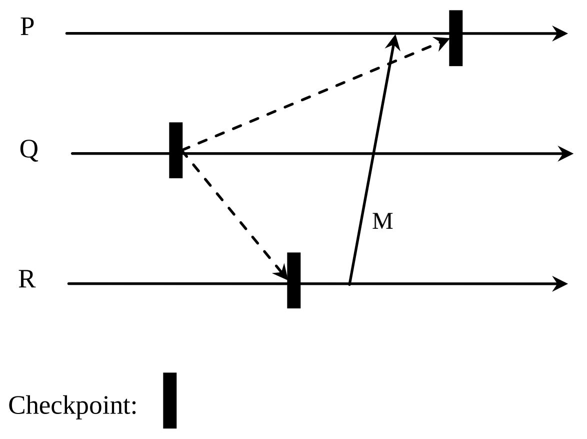

Fig: Inconsistent Checkpoint

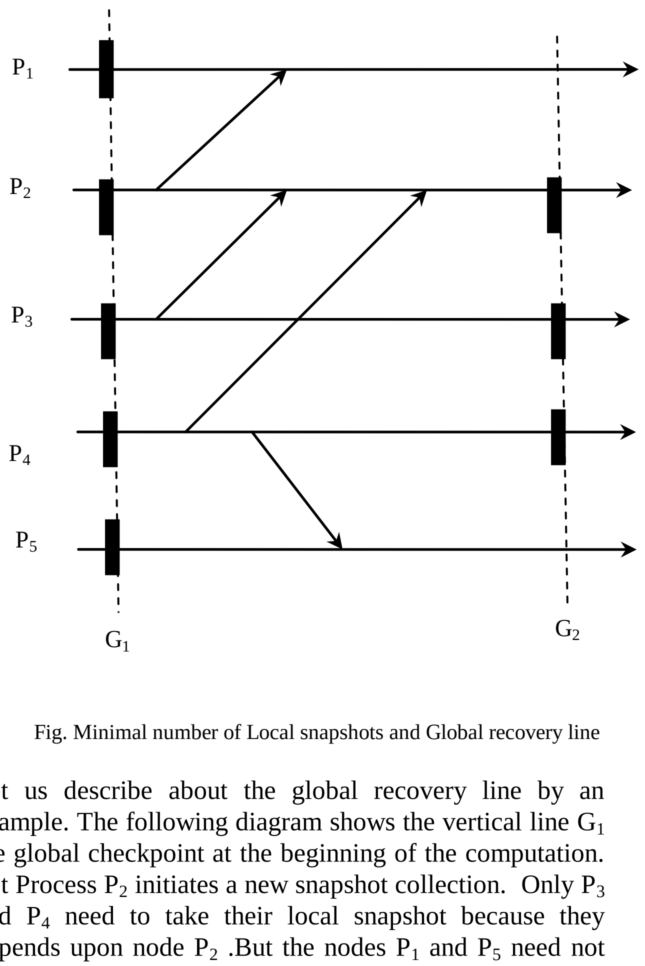

Fig. Minimal number of Local snapshots and Global recovery line

Let us describe about the global recovery line by an example. The following diagram shows the vertical line G_1 the global checkpoint at the beginning of the computation. Let Process P_2 initiates a new snapshot collection. Only P_3 and P_4 need to take their local snapshot because they depends upon node P_2 .But the nodes P_1 and P_5 need not take their snapshot because they do not have dependencies on to process P_2. Where dotted line G_2 shows the global recovery line or current global checkpoint.

B. Managing Node Mobility and Disconnection Title

Let a Mobile Host MH be initially connected to MSS_1.It disconnects from MSS_1. After a finite period of time it connects with MSS_2. In such disconnected period: (a) only local events can take place on MH. (b) There is no message arrive or send events occur during this interval. Hence no any dependency events with respect to another node are created during this interval.

Disconnection

Disconnection of an MH is a voluntary operation and it may take arbitrary period of time. At the time of disconnection from MSS_1:

(a) MH takes its local checkpoint which is stored at MSS_1 as disconnect_checkpoint$_i$ which serves request messages for MH to take checkpoint

(b) Stores its dependency vector R_i at MSS_1.

(c) The Computation messages, for MH arriving at MSS_1 during disconnect interval are stored at MSS_1 until the end of the interval.

(d) Self identity at its stable storage at MSS_1

Reconnection

At the time of reconnection to MSS_2 : MH executes a reconnection protocol. The reconnection protocol sends a message through MSS_2 to MSS_1. On receiving the message MSS_1 executes the following steps:

(1)If MSS_1 had processed request message for MH then disconnect_checkpoint$_i$ and the buffered messages are sent to MH.

(2)If no checkpoint request for MH was received by MSS_1 during disconnect interval only buffered messages are sent.

(3)After that MSS_1 removes the buffered messages, disconnect_checkpoint$_i$ and MH's dependency vector.

When the data sent by MSS_1 arrives at MH, MH executes the following actions:

(1) If the received data contains disconnect_checkpoint$_i$, MH

stores this checkpoint as its local checkpoint and resets all except the ith component of dependency vector R_i before processing the messages.

(2) Process all the received buffered messages.

(3)The dependency vector is updated.

Now this reconnect protocol ends and MH makes normal communication.

MSS_1 removes the disconnect_checkpoint$_i$ at the end of disconnect interval. In such a way mobility and disconnection of MH get managed.

4. Minimal Checkpointing Protocol

In this section, we present a nonblocking snapshot collection protocol for mobile distributed system. The protocol forces a minimum set of nodes to take local checkpoints. Thus overhead of checkpoint collection get minimized. After the coordinated snapshot collection terminates, the nodes that did not participate in snapshot collection can take their local checkpoints in lazy phase approach. When a node initiates a request for snapshot collection to another node then that node takes its local snapshot and propagating the request to neighbouring nodes. A global snapshot is collection of all the local nodes which participates for snapshot initiation. The snapshot thus generated is latest than each of the snapshot thus collected independently. Thus amount of lost work during rollback, after the node failure is minimized. The underlying computation need not have to be suspended during snapshot collection

A. Data structures

R_i : a Boolean vector R_i of n components. At P_i, the vector initialized as follows:
$Ri[i] = 1; Ri[j] = 0$ if $i \neq j$;
When a node P_i sends a message to P_j it then changes vector R_i . This tells Pj about the nodes that are dependent on P_i. While processing a message M Pj extracts Boolean vector M.R from the message and uses it to update Rj as follows: $Rj[k] \leftarrow Rj[k]$ OR $M.R[k]$, where $1 \leq k \leq n$.
ckpt_num: when the node takes its local checkpoint then this integer number is increased.
weight: A nonnegative real variable with maximum value 1 used to detect the termination of snapshot collection or checkpointing algorithm.
transmit: a Boolean array of size n maintained by each node in its stable storage. This array is initialized to all zeros. It is used to keep the trail of those nodes to which checkpoint requests were sent by node. If in this array each element has all 0s then response message is sent to the snapshot initiator with a weight equal to weight received in the request. If in this array some elements are put to 1 then for all i such that transmit[i] = 1, a request is sent to P_i with a non zero segment of weight received in request message and rest part of weight is sent to initiator with a response message.
trigger: A set of 2-tuples (init_id, init_ckptnum) maintained by each node, where init_id indicates the identifier of checkpointing initiator. Where init_ckptnum shows the checkpoint number of the initiator node when it took its own local snapshot on initiating the snapshot collection. trigger is changed for all system messages and the first computation message that a node sends to every other node after taking a local snapshot.
ckpt_array: This is an array of n integer maintained at each node, ckpt_array[i] indicates the ckpt_num of the next message expected from node P_i .
self_trigger: The trigger tuple of a node receiving computation message
msg_trigger: Trigger tuple of computation message
get_weight: The weight received by dependent nodes
forward_weight: The weight sends by the node which further spread checkpoint request.

B. A Checkpointing Protocol

<u>Checkpoint initiation process</u>
Let P_i be the checkpoint initiator. It takes following action: (1)It takes a tentative local checkpoint. (2) Increments its ckpt_num (3) initialized weight to 1 (4) It sets init_id and init_ckptnum in its trigger tuple (5) It sends checkpoint request message to all its dependent nodes. The request message now includes: weight, initiator's trigger and dependency vector $R_{i.}$

<u>Response of a node receiving of checkpoint request</u>
Let node P_i receives checkpoint request.

```
if (reqst_msg.trigger ≠ $P_i$.trigger) then
  {
   $P_i$ takes tentative local checkpoint;
   $P_i$ propagates reqst_msg to all the dependent nodes but not
    M.R; // Explained in further Checkpoint…… module //
    Send portion of the received weight with its reqst_msg;
    Update initiator trigger tuple.
    Send response_msg to the initiator.
   }
else
   {
     $P_i$ does not take the local checkpoint
      if ( transmit[i] = 0 ) then
       {
        $P_i$ send response_msg with weight received with
        reqst_msg to initiator.
        }
 else
  {
   $P_i$ send reqst_msg to nodes for which transmit[j] =1
   with portion of weight;
   $P_i$ sends response_msg with remaining weight to initiator.
  }}
```

<u>Response of a node receiving of computational message</u>
Let a node Pj receives a computation message M from other node P_i then following action occurs:

```
If (ckpt_num$_i$ ≤ ckpt_array[i]) then
   {
     Pj will not take any checkpoint;
```

```
        Restart the computation by processing message M;
    }
 else
    {
      // $P_i$ has already taken a checkpoint before sending M and
         this is the first computation message sent from $P_i$ to Pj.
         M carries a trigger (init_id, init_ckptnum)//
       Set   ckpt_array[i] = ckpt_num$_i$ ;
    }
 If (msg_trigger = self_trigger) then
       // Pi and Pj has taken checkpoints w.r.t same initiator//
       Update dependency vector Ri[j];
  else
     {
        If (msg_trigger.pid ≠ self_trigger.pid)
        {
          If (Pj had processed a message from node $P_k$) then
            Return;
          Pj takes tentative checkpoint;
          Set msg_trigger=self_trigger;
          Propagate snapshot request to dependent processes;
        }
     }
```

Further Checkpoint request propagation

Let node P_i take checkpoint request. It propagates checkpoint request to its dependent processes as follows:

```
Take local checkpoint;
Update ckpt_num$_i$ and transmit[i] ;
 Self_trigger=msg_trigger;
 transmit[i] = $R_i$ – M.R;
 for all k dependent nodes set transmit[k]=1
 {
     get_weight = get_weight/2
 forward_weight  = get_weight;
// Send following module to dependent nodes //
send($P_i$  ,request_msg,chkpt_num,self_trigger,forward_weight);
}
```

Closing Checkpoint collection

When the initiator receives weights from all the response messages then initiator makes the addition of all the weight when this addition becomes equal to 1. It decides that all the nodes involved in snapshot collection have taken local checkpoints. Then it propagates the commit message to all those nodes. The previous permanent local checkpoints at these nodes are discarded. Now if further recovery is required the nodes will rollback to current checkpoint.

C. *Example*

Following example clarifies the concepts used in our algorithm with the help of Fig node P_3 initiates snapshot collection by taking its local checkpoint. The node P_2 and P_4 shows dependencies to P_3. The broken arrows shows request messages sent to P_2 and P_4 to take their snapshots on their timeline. P_4 takes first snapshot and then sends a message M3 to P_2. When M3 reaches P_2 it is the first message reached at P_2 such that msg_trigger.pid ≠ self_trigger.pid. Hence P_2 takes its snapshot before processing M3. Node P_1 takes its local independent snapshot before sending a message M4 to P_2. The interval number of M4 is greater than the value expected by P_2 from P_1. But when M4 reaches P_2 it is not the first computation message received by P_2 with a higher interval number than expected whose msg_trigger.pid is different from P_2's self_trigger.pid. So a snapshot is not taken because it will create inconsistency: The reception of M3 will be recorded if P_2 takes a snapshot just before it processes M4, but the transmission of M3 will not have been recorded by P_4 and now M3 becomes orphan. Also here msg_trigger of M3 = self_trigger of request message to P2 . Hence no need to take further checkpoint

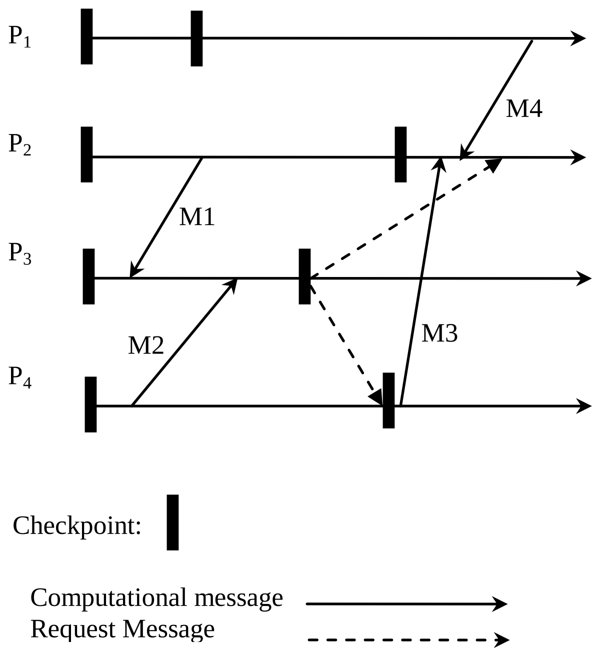

Fig: Global Snapshot collection

D. *RELATED WORK*

In Chandy-Lamport algorithm [1] control messages are sent to all the nodes for consistent global checkpoint. Hence message send overhead is increased along all the channels of network.

Acharya-Badrinath algorithm [9] proposed an uncoordinated checkpointing algorithm for mobile

distributed system because they found the limitations of high cost to receive request messages along every channel in network and absence of local checkpoint of MH during disconnect interval in coordinated checkpoint algorithm.
In Koo-Toueg Algorithm [7] the underlying computation is blocked. There is direct dependency approach is used while global snapshot collection. Such algorithm is not suitable for concurrent initiation.
In Venkatessan and Juang's optimistic failure recovery algorithm [15] no dependency information is send with the computation messages. Hence while recovery process too many rollback occurs.
In [3] Guohong Cao and Mukesh Singhal had proposed an efficient algorithm that neither forces all the processes to take checkpoints nor blocks the underlying computation during checkpointing and which significantly reduces the number of checkpoints. In this paper it is described that there does not exist a nonblocking algorithm that forces only a minimum number of processes to take checkpoints. Their algorithm requires minimum number of processes to take tentative checkpoints and thus minimizes the workload on stable storage server. Their algorithm has three kinds of checkpoints: tentative, permanent and forced. Tentative and permanent checkpoints are saved on stable storage. Forced checkpoints do not need to be saved on stable storage. They can be saved on any where even in the main memory .When a process takes a tentative checkpoint; it forces all dependent processes to take checkpoints. However a process taking a forced checkpoint does not require its dependent processes to take checkpoint. Thus taking a forced checkpoint avoids the cost of transferring large amount of data to stable storage.

4. Conclusions

An efficient recovery mechanism for mobile computing system is required to maintain the continuity of computation in the event of node failures. In this paper we have proposed low-overhead checkpoint collection protocol to meet requirements of node mobility, energy conservation and low communication bandwidth. Dependency information among nodes is used to advance the global checkpoint of the system in coordinated manner. The proposed snapshot collection protocol is Non-Blocking i.e. the participating node does not require to stop their computation during snapshot collection. What actions are carried out when a MH disconnects from MSS and its reconnection to MSS are presented in our paper.

References

[1] K.M. Chandy and L.Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems" ACM Transactions Computer systems vol. 3, no.1.pp.63-75, Feb.1985.
[2] Prakash R. and Singhal M., "Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems" ,IEEE Transaction On Parallel and Distributed Systems, vol. 7, no. 10, pp. 1035-1048, October1996
[3] Guohong Cao and Mukesh Singhal, "On Coordinated Checkpointing in Distributed Systems" IEEE Transaction On Parallel and Distributed Systems, vol. 9, no. 12, pp. 1213-1224, December 1998.
[4] Guohong Cao and Mukesh Singhal, "Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems", IEEE Transaction On Parallel and Distributed Systems, vol. 12, no. 2, pp. 157- 171, February 2001.
[5] Weigang Ni, Susan V. Vrbsky and Sibabrata Ray "Pitfalls in Distributed Non blocking Checkpointing", University of Alabama
[6] Prakash R. and Singhal M. "Maximal Global Snapshot with concurrent initiators," Proc. Sixth IEEE Symp. Parallel and Distributed Processing, pp.344-351, Oct.1994.
[7] Koo. R. and S.Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems" .IEEE Transactions on Software Engineering, SE- 13(1):23-31, January 1987.
[8] Bidyut Gupta, S.Rahimi and Z.Lui. "A New High Performance Checkpointing Approach for Mobile Computing Systems". IJCSNS International Journal of Computer Science and Network Security, Vol.6 No.5B, May 2006.
[9] Acharya A. and Badrinath B. R., "Checkpointing Distributed Applications on Mobile Computers," Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, pp. 73-80, September 1994.
[10] Ch.D.V. Subba Rao and M.M.Naidu. "A New, Efficient Coordinated Checkpointing Protocol Combined with Selective Sender-Based Message Logging"
[11] Nuno Neves and W. Kent Fuchs. "Adaptive Recovery for Mobile Environments",in Proc.IEEE High-Assurance Systems Engineering Workshop,October 21-22,1996,pp.134-141.
[12] Y.Manable. "A Distributed Consistent Global Checkpoint Algorithm With minimum number of Checkpoints". Technical Report of IEICE, COMP97-6(April1997)
[13] J.L.Kim and T.Park. "An efficient protocol for checkpointing recovery in Distributed Systems" IEEE Transaction On Parallel and Distributed Systems,4(8):pp.955-960, Aug 1993.
[14] Elnozahy E.N., Alvisi L., Wang Y.M. and Johnson D.B., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375-408, 2002.
[15] S.Venkatesan and T.T.-Y.Juang , " Low Overhead Optimistic Crash Recovery:", Preliminary version appears in Proc. 11th Int'l Conf. Distributed Computing Systems as "Crash Recovery with Little Overhead,"pp.454- 461, 1991