Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 64 (2004) 649–661
Journal of Parallel and Distributed Computing
http://www.elsevier.com/locate/jpdc

Concurrent checkpoint initiation and recovery algorithms on asynchronous ring networks

Partha Sarathi Mandal and Krishnendu Mukhopadhyaya*

Advanced Computing and Microelectronics Unit, Indian Statistical Institute, 203 B. T. Road, Kolkata 700108, India

Received 10 February 2003; revised 9 February 2004

Abstract

Checkpointing with rollback recovery is a well-known method for achieving fault-tolerance in distributed systems. In this work, we introduce algorithms for checkpointing and rollback recovery on asynchronous unidirectional and bi-directional ring networks. The proposed checkpointing algorithms can handle multiple concurrent initiations by different processes. While taking checkpoints, processes do not have to take into consideration any application message dependency. The synchronization is achieved by passing control messages among the processes. Application messages are acknowledged. Each process maintains a list of unacknowledged messages. Here we use a *logical checkpoint*, which is a standard checkpoint (i.e., snapshot of the process) plus a list of messages that have been sent by this process but are unacknowledged at the time of taking the checkpoint. The worst case message complexity of the proposed checkpointing algorithm is $O(kn)$ when k initiators initiate concurrently. The time complexity is $O(n)$. For the recovery algorithm, time and message complexities are both $O(n)$.
© 2004 Elsevier Inc. All rights reserved.

Keywords: Distributed system; Ring network; Coordinated checkpointing; Rollback recovery; Logical checkpoint

1. Introduction

Checkpointing is an important feature in distributed computing. It gives fault tolerance without requiring additional efforts from the programmer. A *checkpoint* is a snapshot of the current state of a process. It saves enough information in non-volatile stable storage such that, if the contents of the volatile storage are lost due to process failure, one can reconstruct the process state from the information saved in the non-volatile stable storage. If the processes communicate with each other through messages, rolling back a process may cause some inconsistencies. In the time since its last checkpoint, a process may have sent some messages. If it is rolled back and restarted from the point of its last checkpoint, it may create *orphan messages*, i.e., messages whose receive events are recorded in the states of the destination processes but the send events are lost. Similarly, messages received during the rolled back period, may also cause problem. Their sending processes will have no idea that these messages are to be sent again. Such messages, whose send events are recorded in the state of the sender process but the receive events are lost, are called *missing messages*. In Fig. 1 if process P_3 is rolled back to its last checkpoint, then m_3 would be an orphan message. Similarly, if P_0 is rolled back to its last checkpoint then m_1 would be a missing message.

A set of checkpoints, with one checkpoint for every process, is said to be *consistent global checkpointing state* (*CGS*), if it does not contain any *orphan message* or *missing message*. However, generation of missing messages may be acceptable, if messages are logged by sender.

In a distributed system, each process has to take checkpoints periodically on non-volatile stable storage. In case of a failure, the system rolls back to a consistent set of checkpoints. If all the processes take checkpoints at the same time instant, the set of checkpoints would be consistent. But since globally synchronized clocks are very difficult to implement, processes may take

*Corresponding author.
E-mail addresses: partha_r@isical.ac.in (P.S. Mandal), krishnendu@isical.ac.in (K. Mukhopadhyaya).

0743-7315/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.03.013

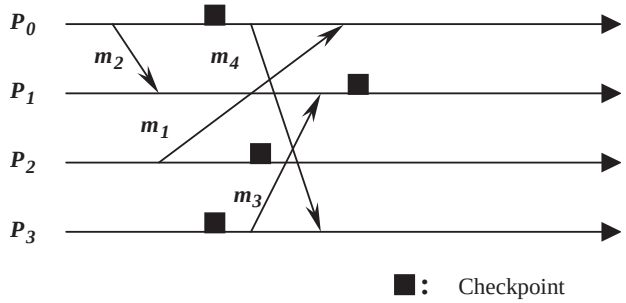

Fig. 1. An example showing orphan message (m_3) and missing message (m_1).

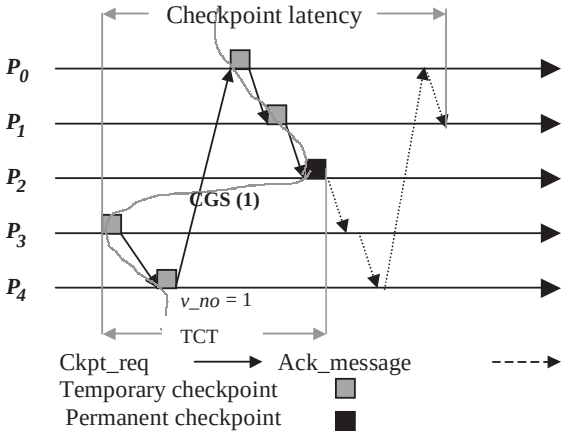

Fig. 2. An example showing the checkpointing for a single initiator (P_3).

checkpoints within an interval. In order to achieve synchronization, sometimes processes take temporary checkpoints. They are made permanent, when all processes agree. The time from the checkpointing initiation to the time of the last process taking its checkpoint (may be temporary) is called the *total checkpointing time (TCT)* (Fig. 2). The time interval from the initiation to completion of the checkpointing process, when all checkpoints are taken and made permanent, is called *checkpointing latency* (Fig. 2).

Checkpointing algorithms may be classified into two broad categories: (a) *coordinated* and (b) *uncoordinated*. In uncoordinated checkpointing [1,11,17] each process takes checkpoints independently, without bothering about other processes. In case of a failure, after recovery, a CGS is found among the existing checkpoints and the system restarts from there. Here, finding a CGS is quite tricky. The choice of checkpoints for the different processes is influenced by their mutual causal dependencies. The common approach is to use *rollback-dependent graph* or *checkpoint graph* [1,2,4,13,20]. In coordinated checkpointing [2,3,7–9,11,13,19], all processes have to synchronize through control messages before taking checkpoints. These synchronization messages contribute to extra overhead. On the other hand, in uncoordinated checkpointing some of the checkpoints taken may not lie on any CGS. Such checkpoints are called *useless checkpoints*. Useless checkpoints degrade system performance. Unlike uncoordinated checkpointing coordinated checkpointing does not generate useless checkpoints.

Coordinated checkpointing algorithms are of two types: (a) *blocking* [6,7] and (b) *non-blocking* [2,3,13,14]. Blocking algorithms force all relevant processes in the system to block their computation during checkpointing latency and hence degrade system performance. In non-blocking algorithms application processes are not blocked when checkpoints are being taken. Some of the main issues addressed in the literature are reducing the number of checkpoints, minimizing the complexity for synchronization, minimizing roll back, etc.

2. Earlier works

Several earlier works [2,3] on snapshot collection algorithms assume that at any point of time only one snapshot collection process is active. Koo and Toueg [7], Spezialetti and Kearns [16] and Prakash and Singhal [12] have proposed methods for handling concurrent initiations of snapshot collection.

According to Koo and Toueg's (K–T) algorithm, once a process takes a local checkpoint, either as an initiator or on request from another process, it becomes *unwilling* to take a checkpoint in response to another initiator's request. The process sends an 'unwilling' response to all subsequence requests, until the checkpoint it has taken, is made permanent or the checkpointing collection is aborted. This algorithm is blocking. Prakash and Singhal have shown that in this algorithm all the initiations may end up aborting, leading to a wastage of effort [12].

Spezialetti and Kearns (S–K) algorithm [16] forces all process to take local checkpoints similar to Chandy–Lamport checkpoint collection algorithm [3]. A process takes local checkpoint for the first request and forwards that request to its neighbors. All subsequent requests are collected in a list called *border_list*. Once a process has received requests along all its incident edges, its checkpointing phase is complete. Then the process sends its *border_list* to the process from which it received the first checkpoint request message. In this way, mutually disjoint sets of processes take their local checkpoints in response to requests from different initiators. Finally, initiators communicate with each other and one checkpoint for each process is selected to build a CGS, which is minimal [10].

On the other hand, the Prakash and Singhal (P–S) algorithm [12] generates a CGS, which is maximal [10]. Unlike S–K algorithm the P–S algorithm permits full propagation of checkpoint requests generated by all the concurrent checkpoint initiations. Thus the P–S

algorithm outputs a CGS with more recent checkpoints than the S–K algorithm. S–K algorithm requires the transmission of $O(n^2)$ messages to take the local checkpoints and $O(m^2n)$ messages for information dissemination phase with message size $O(n/m)$. For m concurrent initiators P–S algorithm requires $O(mn^2)$ messages to take tentative checkpoints. Another $O(m^2n)$ messages of size $O(n)$ are exchanged for establishing a CGS. Although the number of messages required by P–S algorithm is higher, as they are sent concurrently, the time to collect tentative checkpoints is comparable with S–K algorithm.

Suppose n processes are running together. After one or more processes fail, the system recovers by rolling back to a CGS. Many recovery algorithms on distributed computing system have been proposed in the literature [4,5,15,17,18]. Worst case message complexity of the Stron and Yemini's algorithm [17] is $O(2^n)$ for single process failure. Sistea and Welch have proposed an algorithm [15] which requires $O(n^2)$ messages to be exchanged with $O(n)$ information appended to each application message and $O(n^3)$ message exchanges when $O(1)$ extra information is appended to each message. Juang and Venkatesan proposed an algorithm [18] which appends $O(1)$ extra information to each message. When arbitrary number of processes fail, $O(n^2)$ messages are sufficient for general networks and for ring networks $O(n)$ messages are necessary and sufficient for recovery. They also proposed an algorithm [18] on general network where $O(n)$ additional information is appended to each application message; $O(kn)$ messages are required for rolling-back all of the processes to a CGS when k processes fail.

In our proposed asynchronous coordinated checkpointing algorithm processes take checkpoints independent of message pattern. It allows any set of processes in the system to initiate checkpointing. A process need not consider causal dependency generated by the application messages. Due to the special nature of the ring network, the scheme does not need to trace dependence at the time of roll back to get CGS.

3. System model

We consider a distributed system consisting of n processes on a ring network. Processes are numbered $P_0, P_1, P_2, \ldots, P_{n-1}$ sequentially, in the clockwise direction. In case of the unidirectional ring, the direction of the ring is also assumed to be clockwise. There is no common clock or common memory. Message passing is the only mode for communication between any pair of processes. Two types of messages are generated by the processes: *application messages* for underlying distributed application and *control messages* to facilitate checkpointing and roll back of the system. In the unidirectional ring ith process can directly send a message to jth process if and only if $j = (i+1) \bmod n$. In bi-directional ring ith process can directly send a message to jth process if and only if $j = (i \pm 1) \bmod n$. The communication channel is assumed to be FIFO. In a link as well as in intermediate node, a message arriving later does not leave before an earlier message. We assume that there is no link failure, only processes may fail. The computation is asynchronous, i.e., each process runs with its own speed; messages are exchanged with finite but arbitrary delays. Application messages are acknowledgement based, i.e., for each message an acknowledgement is required (this is to let the sender know that the receiver is alive).

In this paper, we consider logical checkpoints [11,19,21], which are slightly different from standard checkpoints. A *logical checkpoint* is a standard checkpoint (i.e., snapshot of the process) plus a list of messages, which have been sent by this process but are unacknowledged at the time of taking the checkpoint. Message lists are updated continuously. After getting acknowledgement for a message, it is deleted from the list. Our algorithm allows the generation of missing messages in case the system has to roll back to its last checkpoint. At the time of restart after a failure, processes retransmit their unacknowledged messages (not all of whom may be missing messages). So there may be duplicate messages after recovery from a failure and that has to be handled using message identifiers.

In our algorithm, for each process, at most two checkpoints may have to be stored in the stable storage when checkpointing procedure is running; otherwise one checkpoint per process is enough to make a system consistent. Checkpoints have a one-bit version numbers (v_no). In the beginning all processes start by taking a permanent checkpoints with $v\_no = 0$.

4. Checkpointing algorithm for unidirectional ring network

A process has complete freedom to take a decision about checkpointing initiation, provided it does not have any temporary checkpoint. Any subset of the n processes, may initiate checkpointing independently. After certain time periods processes initiate checkpointing. It itself takes a logical checkpoint and sends a checkpoint request message *(ckpt_req)* to the next process, with its own id as the initiator. The new checkpoint is marked temporary and stored in the stable storage and set its $initiator\_flag_i = True$. Initially it was *False*. If the v_no of the existing checkpoint is 0 (1) then the v_no of the new checkpoint is 1 (0). On getting a *ckpt_req* message, a process checks whether it has taken any temporary checkpoint or not; if not, then it takes a logical checkpoint with received initiator id as the

initiator id of the checkpoint and then forwards *ckpt_req* to the next process. Within a TCT, each non-initiator process takes at most one temporary checkpoint. The checkpoint is taken in response to the first *ckpt_req* message; other *ckpt_req* messages are either forwarded or discarded depending on the id value of the initiator of the request with respect to the id of the receivers. Each forwarded *ckpt_req* message always contains initiator id. If receiver id is less than the initiator id and if receiver has already taken a temporary checkpoint then that *ckpt_req* will be discarded provided its $initiator\_flag_j$ is *True*.

When a process knows that all other $n-1$ processes have already taken their checkpoints temporarily to the stable storage then that process will take permanent logical checkpoint directly or change the state of the existing checkpoint to permanent, if its state was temporary. Then, it deletes its old permanent checkpoint and sends *ack_message* to the next process, with own id as *ack_message* generator. On receiving *ack_message*, a process changes its temporary checkpoint to permanent, deletes its previous permanent checkpoint and discards or forwards *ack_message* to next the process depending on whether the receiver $(id = i)$ is the immediate predecessor to the *ack_message* initiator $(ack\_initiator\_id = j)$, i.e., if $i = (j-1) \bmod n$ or not respectively. When all processes have made their checkpoints with the new *v_no* permanent then that set of checkpoints will be a CGS. So in our algorithm consistent global checkpointing means each process has a permanent checkpoint in its own stable storage with same *v_no* over all processes.

```
Algorithm. Unidirectional_Checkpointing_Initiator_i
/*This algorithm is executed by process P_i when P_i
decides to initiate a checkpointing */
begin
  if there is no temporary checkpoint then
    take a new temporary checkpoint with new v_no
      /* v_no = 0/1, if the existing checkpoint
      v_no = 1/0*/
    set initiator_flag_i ← True /* process P_i initiating
    checkpoint, initially it was False*/
        set initiator_id ← i
        /* initiator id is attached to the checkpoint
        request;
        it denotes the id of the initiator */
    send a ckpt_req to the next process with initiator_id
  end if
end
```

```
Algorithm. ckpt_req_receiver_j
/* This algorithm is executed by process P_j when it
receives a ckpt_req */
begin
  if temporary checkpoint exists then
    if j is the immediate predecessor of the
    rec_initiator_id then
        /* rec_initiator_id is the initiator_id of the
        ckpt_req generator */
      delete the existing permanent checkpoint
      make the existing temporary checkpoint perma-
      nent
      set ack_initiator_id ← j
      generate and send ack_message to the next
      process with ack_initiator_id
        /* ack_initiator_id is the id of the ack_message
        generator */
    else if j > rec_initiator_id then
      forward the ckpt_req to the next process
      else /* j<rec_initiator_id */
      if initiator_flag_j is False then   /* process has
      not initiated any checkpointing */
        forward the ckpt_req to the next process
      else /* initiator_flag_j is True */
        discard the ckpt_req
      end if
      end if
    else /* permanent checkpoint exists,
            but no temporary checkpoint */
    if j is the immediate predecessor of the
    rec_initiator_id then
      delete old permanent checkpoint
      take a new permanent checkpoint with new v_no
      set ack_initiator_id ← j,
      generate and send ack_message to the next
      process with ack_initiator_id
    else /* process j is not the immediate predecessor
    of the rec_initiator_id */
      take a new temporary checkpoint with new v_no
      forward the ckpt_req to the next process
    end if
  end if
end
```

```
Algorithm. ack_message_receiver_j
/* This algorithm is executed by process P_j when it
receives an ack_message */
begin
  if j is not the immediate predecessor of the initiator of
  this acknowledgement message then
    if temporary checkpoint exists then
      delete the permanent checkpoint
      make the existing temporary checkpoint perma-
      nent
    end if
    forward ack_message to the next process
        else
        /* process j is the immediate predecessor of the
        ack_initiator_id */
```

```
    if temporary checkpoint exists then
      delete the permanent checkpoint
      make the existing temporary checkpoint permanent
    end if
    discard ack_message
  end if
end
```

5. Checkpointing algorithm for bi-directional ring network

Here also, any process, (say P_i), that has not taken a temporary checkpoint may decide to initiate checkpointing. In such a case, the process itself takes a new temporary logical checkpoint. The version number of the new checkpoint is the complement of the version number of the existing permanent checkpoint. Then initiator sends *ckpt_req* to both of its neighboring processes $(=(i\pm 1) \bmod n)$. On receiving a *ckpt_req* message, the conditions for a process taking a new temporary checkpoint or forwarding or discarding the *ckpt_req* are same as those of the previous algorithm. The forwarding is done to the neighbor other than the one from which it is received. When a process receives *ckpt_req* initiated by the same initiator twice, the process changes the existing *ckpt_state* from temporary T to permanent P, deletes its old permanent checkpoint and forwards *ckpt_req* message. No acknowledgement message is required here. The final initiator (one with minimum id among all initiators who have initiated checkpointing within TCT) receives two forwarded *ckpt_req* messages. When it receives the first request, it changes the existing *ckpt_state* from T to P, deletes old permanent checkpoint and stops the *ckpt_req* message propagation. When it receives the other *ckpt_req*, it just discards the message and the checkpointing algorithm terminates.

5.1. Checkpointing. algorithm for bi-directional

```
Algorithm. Bi-directional_Checkpointing_Initiator_i
/* This algorithm is executed by process P_i when P_i decides to initiate a checkpointing */
begin
  if there is no temporary checkpoint then
    take a new temporary checkpoint with new v_no
      /* v_no= 0/1, if the existing checkpoint v_no= 1 / 0 */
    set initiator_id ← i
    send ckpt_req to both adjacent processes with initiator_id
  end if
end
```

```
Algorithm. ckpt_req_receiver_j
/* This algorithm is executed by process P_j when it receives a ckpt_req */
begin
    if there is no temporary checkpoint then
        take a new temporary checkpoint with new v_no
        set initiator_id ← rec_initiator_id
                /* rec_initiator_id is the initiator_id of the ckpt_req */
        forward the ckpt_req to the other adjacent process
                /* other than the process from which the ckpt_req was received */
    else        /* temporary checkpoint exists */
        if initiator_id < rec_initiator_id then
            discard the ckpt_req message
        else if initiator_id > rec_initiator_id then
            set initiator_id ← rec_initiator_id
            forward the ckpt_req to the other adjacent process
        else    /* initiator_id is equal to rec_initiator_id */
            if j = rec_initiator_id then
                delete the existing permanent checkpoint
                make the existing temporary checkpoint permanent
                discard the ckpt_req message
            else /* j ≠ rec_initiator_id */
                delete the existing permanent checkpoint
                make the existing temporary checkpoint permanent
                forward the ckpt_req to the other adjacent process
            end if
        end if
    end if
end
```

6. Recovery algorithm for unidirectional ring network

We assume that, when the faulty process is restored, it initiates recovery process by sending recovery messages to the other processes. The recovery algorithm finds out a *v_no* for which checkpoints exist in all the processes. Processes may fail when distributed application is

running or when checkpointing process is in execution. But a checkpoint is made permanent, only when checkpoints for that *v_no* has already been taken in all other process. Thus the existence of even one permanent checkpoint indicates that checkpoints for this *v_no* are present in all other process. Note that some of them may be temporary, while the rest are permanent. In such a case, the temporary checkpoints are made permanent by the recovery algorithm. Processes resume computation from these checkpoints. At the time of restart, processes resend their messages (in the same order as sent before) which were unacknowledged at the moment of taking the checkpoint. There might be duplicate messages after re-sending messages and these problems have to be resolved using message identifier at the receiver end.

Our recovery algorithm generates two types of messages

(1) *Recovery message*, for synchronization over same checkpointing version no.
(2) *Resume message*, after synchronization, initiator sends this message to all processes. After receiving this message a process resumes computation from the latest checkpoint. If the checkpoint corresponding to this *v_no* happens to be temporary, it is made permanent, deleting the old permanent checkpoint.

When a process initiates recovery process by sending recovery message to its neighbor, it sends own id as the recovery initiator, latest checkpoint *v_no* (whether it was permanent or temporary). On receiving a recovery message a process checks its own *v_no* with the *rec_v_no*. o. If they are identical, it sets own *initiator_id* to *rec_initiator_id* and forwards the recovery message, as it is, to the next process. When this forwarded message reaches its initiator, initiator generates *resume_message* with own id. If *v_no* is not equal to received *v_no*, then process checks its *ckpt_state*. If $ckpt\_state = T$ then it deletes its temporary checkpoint keeping the permanent checkpoint. Then it forwards recovery message to the next process. And if its $ckpt\_state = P$ then this process takes over of the role the initiator. In this case this process sends recovery message to the next process with own id as a recovery initiator id and own *v_no*.

When a process receives a *resume_message* if its $ckpt\_state = T$ it makes $ckpt\_state = P$ and deletes its old permanent checkpoint. If its *ckpt_state* was P then no changes are made. In both the cases it forwards the *resume_message* to the next process unless it knows that all previous $n-1$ processes have already know about this *resume_message*.

6.1. Recovery algorithm for unidirectional

Algorithm. Unidirectional_Recovery_Initiator$_i$
/* This algorithm is executed by process P_i when the faulty process P_i is restored */
Begin
set *recovery_initiator_id* ← *i*
send recovery message to the next process
with latest checkpoint v_no and recovery_initiator_id
/* latest checkpoint may be temporary or permanent */
end

Algorithm. Recovery_message_receiver$_j$
/* This algorithm is executed by process P_j when it receives a recovery message */
Begin
if *process id, j and v_no of the latest checkpoint match with the initiator id and v_no of the recovery message* **then**
if *latest checkpoint is temporary* **then**
delete the permanent checkpoint
make the temporary checkpoint permanent
end if
generate resume message
set *resume_initiator_id* ← *j*
send resume message to the next process with resume initiator id
else if *v_no of the latest checkpoint does not match with v_no of the recovery message but* $j \neq initiator\_id$ **then**
if *temporary checkpoint exists* **then**
delete the temporary checkpoint
rollback to its previous permanent checkpoint
forward recovery message to the next process
else /* temporary checkpoint does not exists */
discard recovery message

Unidirectional_Recovery_Initiator$_j$/* set process *j* as a new recovery initiator */**end ifelse**/* *v_no* of the latest checkpoint match with *v_no* ofthe recovery message but $j \neq initiator\_id$ */*forward recovery message to the next process***end ifend**

Algorithm. Resume_message_receiver$_j$
/* This algorithm is executed by process P_j when it receives a resume message */
Begin
if *process j is not the immediate predecessor of the initiator of this resume message* **then**
if *temporary checkpoint exists* **then**
delete the existing permanent checkpoint
make the existing temporary checkpoint permanent

```
    end if
    forward resume message to the next process
        else /* process j is the immediate predecessor of
        the initiator of this resume message*/
    if temporary checkpoint exists then
      delete the existing permanent checkpoint
      make the existing temporary checkpoint perma-
      nent
    end if
    terminate resume message
  end if
end
```

7. Recovery algorithm for bi-directional ring network

In this algorithm we use two flags, *flag_visit* and *flag_resume* for every process P_i. When P_i comes to know about a fault, it sets both its flags to *False*. As in the unidirectional recovery algorithm, when the faulty process P_i is restored, it initiates recovery process. It sends recovery message (*reco_message*) along with latest checkpoint *v_no* (irrespective of whether it is permanent or temporary) to its two neighbors ($P_{(i+1) \bmod n}$ and $P_{(i-1) \bmod n}$) and sets *flag_visit* = *True*. The algorithm finds out a *v_no* for which checkpoints exist in all processes. The *resume_message* is not required here.

When a process P_i receives a *reco_message* it compares *rec_v_no* with own *v_no*. If they are equal then P_i checks *flag_visit*. If *flag_visit* = *False*, P_i sets *flag_visit* = *True* and forwards the message. If *flag_visit* = *True* and if P_i's *ckpt_state* is T, it deletes its old permanent checkpoint and changes the value of *ckpt_state* to P. If its *ckpt_state* is P then the *ckpt_state* remains unchanged. In both the cases the message is forwarded to the next process, in the direction of travel of the *reco_message* and the process resumes computation from the permanent checkpoint. When a process resumes computation it sets *flag_resume* = *True*. The *reco_message* is forwarded till it reaches a process whose *flag_resume* = *True*.

In case *rec_v_no* is not equal to *v_no*, if *ckpt_state* is T then P_i deletes its current checkpoint (T) and forwards the message, otherwise it sends *reco_message* to the next process with its checkpoint *v_no*. In both the cases P_i sets *flag_visit* = *True*.

```
Algorithm. Bi-directional_Recovery_Initiator_i
/* This algorithm is executed by process P_i when the
faulty process P_i is restored */
Begin
   set flag_visit ← True
   send recovery message to both adjacent processes
   with latest checkpoint v_no
end
```

```
Algorithm. Recovery_message_receiver_j
/* This algorithm is executed by process P_j when it
receives a recovery message */
Begin
    if flag_resume is True then /* recovery for process j
    is complete */
        discard the recovery message
    else  /* flag_resume is False, implies recovery for
        the process is not yet complete */
        if v_no of process j matches with the v_no of
        the recovery message then
            if flag_visit is True then /* process j
            already received a recovery message */
                if temporary checkpoint exists
                then
                    delete the existing perma-
                    nent checkpoint
                    make the temporary check-
                    point permanent
                end if
                set flag_resume ← True
            else /* flag_visit is False, this is the first
            recovery message received by process
            j */
                set flag_visit ← True
            end if
            forward the recovery message to the
            other adjacent process
        else /* v_no of process j does not match the
        v_no of the recovery message */
            if temporary checkpoint exists then
                set flag_visit ← True
                delete the existing temporary
                checkpoint
                rollback to its previous perma-
                nent checkpoint
                forward recovery message to the
                other adjacent process
            else /* permanent checkpoint exist but
            no temporary checkpoint */
                set flag_visit ← True
                send recovery message to the
                other adjacent process with its
                own checkpoint v_no
            end if
        end if
    end if
end
```

8. Correctness of the proposed algorithms

For unidirectional as well as bi-directional rings, in order to show that the proposed algorithm is correct, we first show that, at any point of time, there exists a value

of *v_no*, for which each process has a checkpoint. Then we show that, the set thus obtained is indeed a CGS, i.e., it does not contain any orphan message.

Theorem 1. *In the proposed checkpointing algorithm for unidirectional ring, at any point of time, there exists exactly one value of v_no, for which each process has a checkpoint.*

Proof. If no checkpointing process is in execution, then the checkpoints corresponding to the *v_no* of the last checkpointing process will be available in every process. Now we consider a point of time, say t, within the checkpointing latency. If t is within the TCT, since the new checkpoint has not been made permanent, the permanent checkpoints in all the processes correspond to the previous checkpoint latency and hence have the same *v_no*. For example, in Fig. 3, process P_2 fails before taking its new checkpoint. In this case system has to roll-back to their permanent checkpoint, for each process to get the previous existing CGS(1). If t is after the TCT, here at least one process has taken a new permanent checkpoint. Since permanent checkpoints are taken only when all other processes have taken temporary or permanent checkpoints corresponding to that *v_no*, a checkpoint (temporary or permanent) corresponding to the new *v_no* is available with every process. For example in Fig. 4, process P_2 has failed after taking its permanent checkpoint. In this case, system will not roll-back to CGS(1). Instead, the recovery algorithm goes to CGS(0) using the current existing checkpoints (temporary or permanent) with $v\_no = 0$ for all processes.

Now note that before taking the first permanent checkpoint corresponding to a new checkpointing process, the existing permanent checkpoint is deleted. Thus at any point of time there is only one complete set of checkpoints. □

Our recovery algorithm finds a set of checkpoints corresponding to the same *v_no*. It only remains to show that the set thus obtained is consistent.

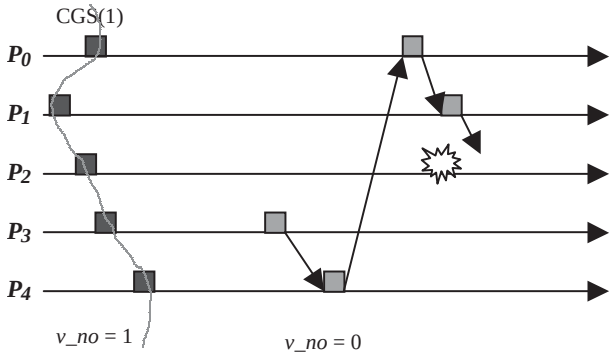

Fig. 3. An example showing the checkpointing and recovery when a failure occurs within TCT.

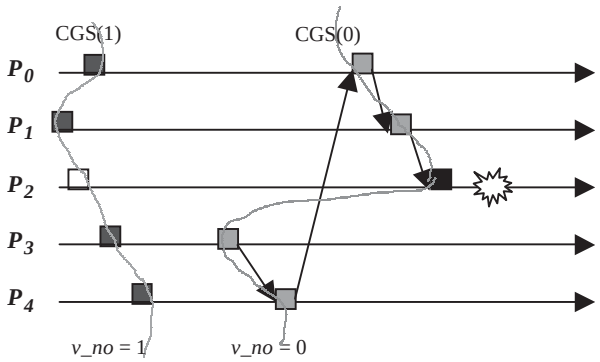

Fig. 4. An example showing the checkpointing and recovery when a failure occurs after TCT but within the checkpointing latency.

Theorem 2. *The set of checkpoints corresponding to the v_no in Theorem* 1, *is consistent.*

Proof. Suppose for an application message, the send event is not recorded in our set of checkpoints. Then, in the sender process, the checkpoint was taken before sending this application message. As the checkpoint was taken earlier, the *ckpt_req* message following the checkpoint will also precede the application message. As we assume that the channel is FIFO, and unidirectional, the *ckpt_req* message will always be received before the application message. All processes take checkpoints before receiving the message. Hence, no checkpoint will show this message being received. □

Theorem 3. *In the proposed checkpointing algorithm for bi-directional ring, at any point of time, there exists exactly one value of v_no, for which each process has a checkpoint.*

Proof. If no checkpointing process is in execution, then the checkpoints corresponding to the *v_no* of the last checkpointing process will be available in every process. Now we consider a point of time, say t, within the checkpoint latency. If t is within the TCT (Fig. 5), since all processes have not taken their temporary checkpoint for the ongoing checkpointing process, the permanent checkpoints in all the processes correspond to the previous checkpoint latency and hence have the same *v_no*. If t is after the TCT but within the checkpoint latency (Fig. 6), here every process has taken a new temporary checkpoint. Some processes may have made these checkpoints permanent also. So a checkpoint corresponding to the new *v_no* is available with every process. □

Theorem 4. *The set of checkpoints corresponding to the v_no in Theorem* 3, *is consistent.*

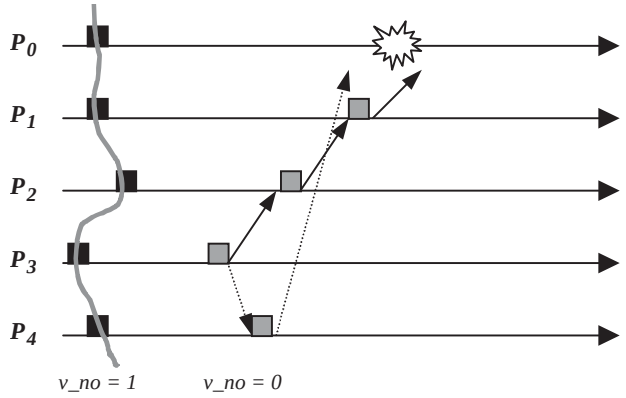

Fig. 5. An example showing the checkpointing and recovery when a failure occurs within TCT.

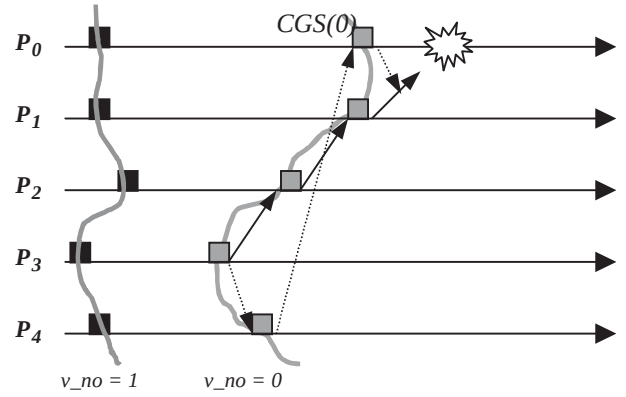

Fig. 6. An example showing the checkpointing and recovery when a failure occurs after TCT but within the checkpointing latency.

Proof. Suppose for an application message the send event is not recorded in our set of checkpoints. Then the checkpoint was taken before sending this application message. But since the checkpoint was taken earlier, the *ckpt_req* message following the checkpoint will also precede the application message. As we assume that the channel is FIFO, the *ckpt_req* message will always be received before the application message. Hence all processes take checkpoints before receiving the message. Hence, no checkpoint will show this message being received. □

9. Complexity analysis

For both unidirectional and bi-directional checkpointing algorithms, in a single checkpointing latency, a process takes exactly one checkpoint. But several checkpointing request messages may be generated because of multiple concurrent initiations. Among those only the request message with minimum *id* (initiator) survives and goes round the ring once. All other request messages are dominated by that message and will be discarded before completing a round. By the time, the surviving message completes the round, all other requests are discarded. One more round may be necessary for the confirmation in case of unidirectional ring. But in case of bi-directional ring, both checkpointing request messages go round the ring just one each; along different directions and no separate confirmation is required. Thus, the checkpointing time is $O(n)$ for both cases.

With respect to message complexity (i.e., the number of control messages), the worst case occurs when all the processes initiate at the same point of time. If such a thing happens in a unidirectional ring, the *ckpt_req* message from P_0 goes to all other processes. For P_i $(i \neq 0)$ it goes up to P_0 and is discarded. Thus a total of $(n-1)+(n-1+n-2+\cdots+2+1) = (n-1)(n+2)/2$ messages are generated. Also $(n-1)$ acknowledgement messages will be generated. Thus, a total of $(n-1)(n+4)/2$ $(= O(n^2))$ control messages will be generated. And for bi-directional ring, the *ckpt_req* message from P_0 goes to all other processes along both directions and comeback. For all other processes packets going in the clockwise direction go up to P_0 and those going in the counter clockwise direction go just one hap each and are discarded. Thus a total of $(2n)+(n+n-1+\cdots+2) = (2n-1)+n(n+1)/2$ $(= O(n^2))$ control messages will be generated.

For rollback recovery algorithms in unidirectional and bi-directional ring, worst case time complexities and message complexities are all $O(n)$.

10. Comparison with existing algorithms

K–T algorithm does not work on a unidirectional ring network when multiple processes initiate checkpoints concurrently. In such a case, all the checkpointing processes end up in aborting [12]. Like S–K algorithm our algorithm takes n temporary checkpoints, one for each process, and this does not depend on the number of concurrent initiations. In P–S algorithm, if all processes are dependent on each other, and k processes initiate checkpointing concurrently, each process takes k temporary checkpoints, i.e., a total of kn checkpoints for the system. Both S–K and P–S algorithms are designed for general network topologies. Their worst case message complexities are $O(n^3)$. But for the simple unidirectional ring, this worst case is achieved. In case of the proposed algorithm message complexity is $O(n^2)$.

Table 1 compares of the proposed algorithms with the S–K algorithm and the P–S algorithm.

Table 1
Performance of the proposed algorithms and other existing algorithms

	S–K algorithm	P–S algorithm	Proposed checkpointing algorithms
Network topology for which applicable	General	General	Ring (unidirectional/ bi-directional)
Worst case message complexity (Ring)	$O(n^3)$	$O(n^3)$	$O(n^2)$
Time complexity (Ring)	$O(n)$	$O(n)$	$O(n)$
Control message size	$O(n/k)$	$O(n)$	$O(1)$
Number of checkpoints stored for k concurrent initiations	One permanent one temporary checkpoint for each process	One permanent and k temporary checkpoints for each process	One permanent and one temporary checkpoint for each process
Number of checkpoints rollback after a failure	At most one temporary checkpoint	At most k temporary checkpoints	At most one temporary checkpoint

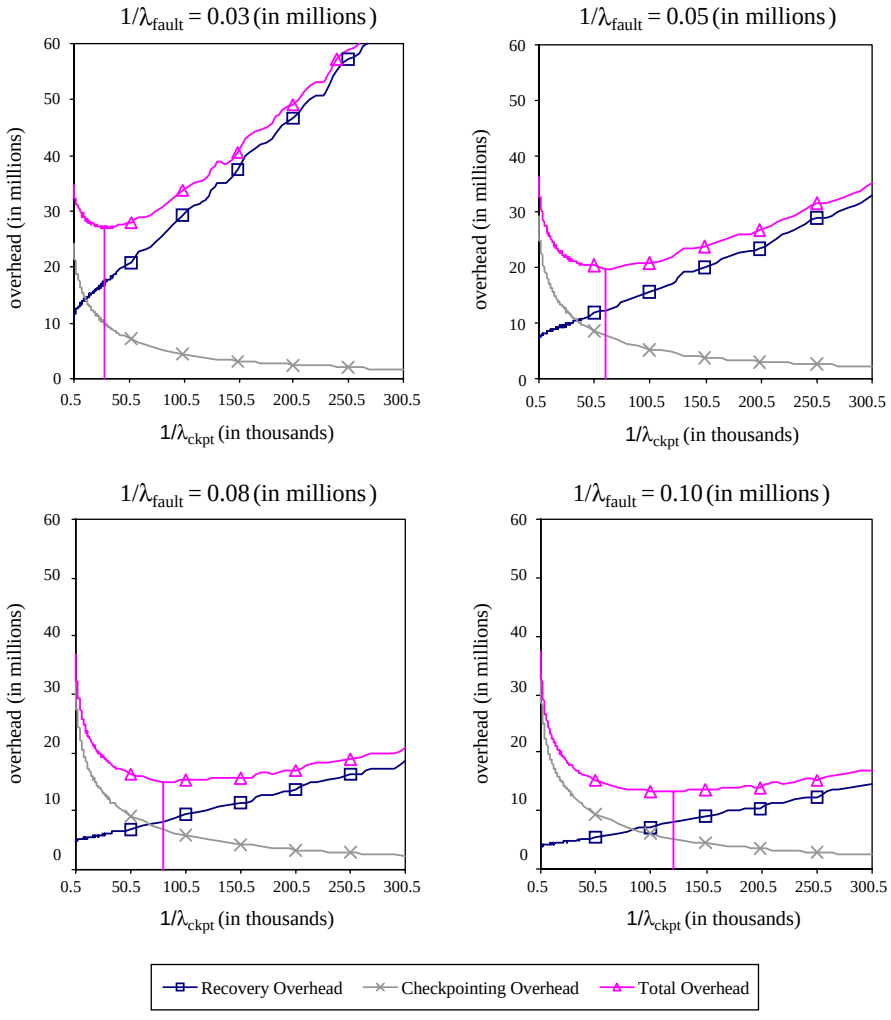

Fig. 7. Simulation results showing rollback recovery, checkpointing, and total overhead costs for the proposed algorithm in a unidirectional ring.

11. Simulation results

Simulation studies were conducted for the behavior of the proposed algorithm when implemented on unidirectional as well as bi-directional ring networks. We assume that inter-fault time, inter-checkpoint time and inter-message send for a process follow exponential distribution with parameters (λ_{fault}), (λ_{ckpt}) and (λ_{send}), respectively. The simulation program takes λ_{fault}, λ_{ckpt} and λ_{send} as input parameters. It is assumed that a message takes one unit of time to travel across one link. The time for taking a checkpoint is assumed to be 5000 units. We varied $\frac{1}{\lambda_{ckpt}}$ between 500 and 300,000 with increments of 100 for a fixed value of $\frac{1}{\lambda_{fault}}$ (between 30,000 and 100,000) whereas λ_{sent} remains fixed. Simulation has been carried out for 20,000,000 units of time. For each set of values of the input parameters, the program was run 20 different times and then the average of the 20 runs is taken. Figs. 7 and 8 show the simulated values of checkpointing overhead, rollback recovery overhead and the total overhead for our proposed uni-directional and bi-directional algorithms, respectively. *Total overhead* is the sum of checkpointing overhead and rollback recovery overhead. As checkpointing rate (λ_{ckpt}) decreases, checkpointing overhead decreases while recovery overhead goes up. Initially the total overhead decreases with decreasing checkpointing rate. At this stage the checkpointing overhead is the dominant cost. After it reaches a minimum value, the rollback cost starts to dominate and the total overhead starts increasing again. In each case we show the optimum value of the checkpointing rate that minimizes the total overhead. As the fault rate (λ_{fault}) goes down (in different graphs), the recovery cost also goes down and the optimum checkpointing rate goes down too. The number of control messages was affected strongly by the number of concurrent initiations. Concurrent initiations abort many control messages without letting them complete the cycle. This explains the variation in the curves.

Table 2 compares the proposed Unidirectional (U) and Bi-directional (B) algorithms with S–K and P–S algorithms in terms number of total control messages. Simulated runs of the four algorithms were carried out. We assume that whenever one process initiates checkpointing, all the processes, take checkpoints. We have simulated systems with 4, 6 and 10 processes each. The value of $\frac{1}{\lambda_{ckpt}}$ was taken 200, 400 or 600. For each algorithm, total control messages was counted as the average of 200 different runs. The results clearly reflect

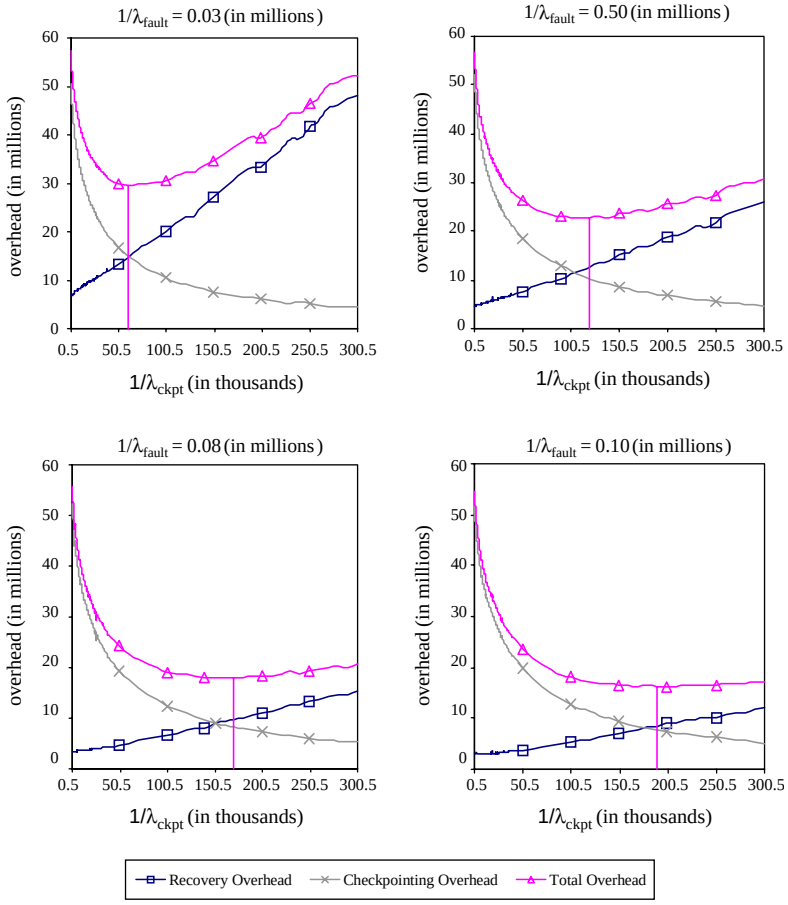

Fig. 8. Simulation results showing rollback recovery, checkpointing, and total overhead costs for the proposed algorithm in a bi-directional ring.

Table 2
Simulation results comparing the number of control messages for the proposed Unidirectional (U) and Bi-directional (B) algorithm with S–K and P–S algorithms

	No. of processes											
$\frac{1}{\lambda_{ckpt}}$ (↓)	4				6				10			
Algorithms (→)	U	B	S–K	P–S	U	B	S–K	P–S	U	B	S–K	P–S
200	16	24	58	188	33	41	239	841	82	80	1502	28,720
400	18	24	38	174	36	45	155	830	88	91	958	43,442
600	19	25	27	158	39	45	104	787	94	98	663	38,899

the superiority of the proposed algorithms. The difference is more pronounced for larger systems with higher number of processes.

12. Conclusion

In this work, we have proposed checkpointing and recovery algorithms, for unidirectional as well as bi-directional ring networks. In our model, processes take logical checkpoints, i.e., snapshot of the process plus the unacknowledged messages. Our algorithm can handle multiple initiations of checkpointing. During recovery each process has to rollback at most one checkpoint. For each process at most two checkpoints (one permanent and other temporary) may be saved in the stable storage. For the checkpointing as well the recovery algorithms, the control message makes two rounds along the unidirectional ring and one round for the bi-directional ring.

Though checkpointing schemes for general network topologies are available in the literature. There is scope for improvement for particular classes of topologies. A more general approach showing the effect of the topologies on the complexities of the checkpointing algorithms may also be considered.

Acknowledgments

The first author is thankful to Council of Scientific and Industrial Research (CSIR), India, for financial support during this work. The authors thank the anonymous reviewers for their constructive criticism and helpful suggestions. The authors are also grateful to Professor Bhabani P. Sinha of ACM Unit, Indian Statistical Institute, Kolkata, for his patient hearing and many suggestions which have improved the organization of the paper.

References

[1] B. Bhargava, S.R. Lian, Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach, in: Proceedings of the Seventh IEEE Symposium on Reliable Distributed System, 1988, pp. 3–12.

[2] G. Cao, M. Singhal, On coordinated checkpointing in distributed systems, IEEE Trans. Parallel Distrib. Systems 9 (12) (1998) 1213–1225.

[3] K.M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems, ACM Trans. Comput. Systems 3 (1) (1985) 63–75.

[4] E.N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, D.B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Comput. Surveys 34 (3) (2002) 375–408.

[5] D. Johnson, W. Zwaenepoel, Recovery in distributed systems using optimistic message logging and checkpointing, J. Algorithms 3 (11) (1990) 462–491.

[6] J.L. Kin, T. Park, An efficient protocol for checkpointing recovery in distributed system, IEEE Trans. Parallel Distrib. Systems 5 (8) (1998) 955–960.

[7] R. Koo, S. Toueg, Checkpointing and rollback-recovery for distributed system, IEEE Trans. Software Eng. 13 (1) (1987) 23–31.

[8] D. Manivannan, M. Singhal, A low-overhead recovery technique using quasi-synchronous checkpointing, in: Proceedings of the IEEE Sixth International Conference on Distributed Computer Systems, May 1996, pp. 100–107.

[9] D. Manivannan, M. Singhal, Quasi-synchronous checkpointing: models, characterization, and classification, IEEE Trans. Parallel Distrib. Systems 10 (7) (1999) 703–713.

[10] F. Mattern, Virtual time and global states of distributed systems, in: M. Cosnard et al. (Ed.), Proceedings of the Workshop on Parallel and Distributed Algorithm, Elsevier Science Publishers B. V., North-Holland, Amsterdam, 1989, pp. 215–226.

[11] K.Z. Meth, W.G. Tuel, Parallel checkpoint/restart without message logging, in: Proceedings of the IEEE 28th International Conference on Parallel Processing (ICPP '00), August 2000, pp. 253–258.

[12] R. Prakash, M. Singhal, Maximal global snapshot with concurrent initiators, in: Proceedings of the Sixth IEEE Symposium of Parallel and Distributed Processing, October 1994, pp. 334–351.

[13] R. Prakash, M. Singhal, Low-cost checkpointing and failure recovery in mobile computing systems, IEEE Trans. Parallel Distrib. Systems 7 (10) (1996) 1035–1048.

[14] L.M. Silva, J.G. Silva, Global checkpointing for distributed programs, in: Proceedings of the 11th Symposium on Reliable Distributed Systems, 1992, pp. 115–162.

[15] A.P. Sistla, J. Welch, Efficient distributed recovery using message logging, in: Proceedings of the ACM Symposium on Principle of Distributed Computing, 1989, pp. 223–238.

[16] M. Spezialetti, P. Kearns, Efficient distributed snapshots, in: Proceedings of the Sixth ICDCS, 1986, pp. 382–388.

[17] R.E. Strom, S. Yemini, Optimistic recovery in distributed systems, ACM Trans. Comput. Systems 3 (3) (1985) 204–226.

[18] T.T-Y. Juang, S. Venkatesan, Efficient algorithms for crash recovery in distributed systems, in: Proceedings of the 10th Conference on FSTTCS, Springer, Berlin, December 1990, pp. 349–361.
[19] N.H. Vidya, Staggered consistent checkpointing, IEEE Trans. Parallel Distrib. Systems 10 (7) (1999) 694–702.
[20] Y.M. Wang, Consistent global checkpoints that contain a given set of local checkpoints, IEEE Trans. Comput. 46 (4) Apr. (1997) 456–468.
[21] Y.M. Wang, Y. Huang, W.K. Fuchs, Progressive retry for software error recovery in distributed systems, in: Proceedings of the IEEE Fault-Tolerant Computing Symposium (FTCS-23), June 1993, pp. 138–144.

Partha Sarathi Mandal received a Bachelor of Science (Hons.) degree in Mathematics from the University of Calcutta, India, a Master of Science degree in Mathematics from Jadavpur University, India, in 1995, and 1997 respectively. He is awarded Junior and Senior Research Fellowship by the Council of Scientific & Industrial Research (CSIR), India. He is currently working towards his Ph.D. degree in Computer Science at the Advanced Computing and Microelectronics Unit of the Indian Statistical Institute, Kolkata. His current research interests include parallel and distributed computing, fault tolerance, mobile agent, performance analysis etc.

Krishnendu Mukhopadhyaya received a Bachelor of Statistics (Hons.), Master of Statistics, Master of Technology in Computer Science, and Ph.D. in Computer Science all from the Indian Statistical Institute, Kolkata, in 1985, 1987, 1989 and 1994 respectively. From 1993 to 1999 he worked as a Lecturer in the Department of Mathematics, Jadavpur University. Since 1999, he is working at the Indian Statistical Institute, Kolkata as an Associate Professor. He was a recipient of the Young Scientist Award of the Indian Science Congress Association and the BOYSCAST Fellowship of the Department of Science and Technology, Government of India. His current research interests include mobile computing, parallel and distributed computing, sensor networks etc.